2016-12-01

Algorithms for Learning the Structure of Monotone and Nonmonotone Sum-Product Networks

Aaron W. Dennis
Brigham Young University

Algorithms for Learning the Structure of Monotone and Nonmonotone Sum-Product Networks

Aaron W. Dennis

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Dan Ventura, Chair
Kevin Seppi
Bryan Morse
Sean Warnick
Quinn Snell

Department of Computer Science

Brigham Young University

ABSTRACT

Algorithms for Learning the Structure of
Monotone and Nonmonotone
Sum-Product Networks

Aaron W. Dennis
Department of Computer Science, BYU
Doctor of Philosophy

The sum-product network (SPN) is a recently-proposed generative, probabilistic model that is guaranteed to compute any joint or any marginal probability in time linear in the size of the model. An SPN is represented as a directed, acyclic graph (DAG) of sum and product nodes, with univariate probability distributions at the leaves. It is important to learn the structure of this DAG since the set of distributions representable by an SPN is constrained by it. We present the first batch structure learning algorithm for SPNs and show its advantage over learning the parameters of an SPN with fixed architecture. We propose a search algorithm for learning the structure of an SPN and show that its ability to learn a DAG-structured SPN makes it better for some tasks than algorithms that only learn tree-structured SPNs. We adapt the structure search algorithm to learn the structure of an SPN in the online setting and show that two other methods for online SPN structure learning are slower or learn models with lower likelihood. We also propose to combine SPNs with an autoencoder to model image data; this application of SPN structure learning shows that both models benefit from being combined.

We are also the first to propose a distinction between nonmonotone and monotone SPNs, or SPNs with negative edge-weights and those without, respectively. We prove several important properties of nonmonotone SPNs, propose algorithms for learning a special class of nonmonotone SPN called the twin SPNs, and show that allowing negative edge-weights can help twin SPNs model some distributions more compactly than monotone SPNs.

Keywords: Sum-product networks, structure learning, structure search, online structure search, nonmonotone sum-product networks

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Dan Ventura, for his help in completing this dissertation. In particular, his optimism and patience has been a needed balancing force during this process. I have also appreciated his availability; his willingness to be interrupted and take time for discussions; and his prompt, careful feedback of my work. I also thank my committee for taking the time to work with me and offer feedback, including a special thanks to Dr. Seppi for answering questions and actively reaching out to help my research efforts.

I have had many great labmates. Thanks for the discussions, the CS 611 study group, the technical interview preparation group, the deep-learning study group, and for your friendship. I have also benefited from discussions with and encouragement from other friends outside the lab.

I also need to thank my family, especially my wife Maren. She has been willing to endure living in a small-windowed basement for years and several serious fights with the budget, among other things. She has been supportive, patient, willing to listen to my ramblings, and has made our life happy. Thank you. My kids Annie and Scott have been a delight to watch as they have grown up. Thank you for reminding me what is really important.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Probabilistic modeling is used in many fields including engineering, science, economics, and medicine. Instead of ignoring the inevitable uncertainties that arise in the complex systems studied in these fields, probabilistic modeling provides a formal mechanism for explicitly modeling those uncertainties. Generative models and discriminative models are two general types of probabilistic model that are often used. A generative model represents a joint distribution over the variables in question, while a discriminative model represents a conditional distribution. For example, an image classifier may use a discriminative model to represent the distribution over the possible labels, Y, given the image X, or $p(Y|X)$. On the other hand, a generative model could be used to represent the distribution over image-label pairs, or $p(X, Y)$. Which type of model is used often depends on the application at hand; the focus here is on generative models.

A probabilistic model is used to answer probabilistic questions, or queries. The process of answering these queries is called inference. For example, given $X = x$ and/or $Y = y$ we may want to compute the value that the model assigns to the probabilities $p(y|x)$, $p(x|y)$, $p(x)$, or $p(y)$. Taking the image-label example, these queries can help answer, respectively, the following questions: is the label y likely given image x; is the image x likely given label y; is the image x likely, period; and does the label y occur frequently? A generative model, or joint distribution, can help answer queries that discriminative models cannot; in principle, each query above can be answered using just $p(X, Y)$, but only the first can be answered using $p(Y|X)$ alone. Other example queries for which $p(X, Y)$ is sufficient but $p(Y|X)$ is not

include: what is the probability of the left half of image x given the right half, and what is the probability that a particular pixel is black, without taking into account the value of any other pixel?

Probabilistic graphical models (PGMs), which include both Bayesian networks (BNs) and Markov networks, can be used to model $p(X, Y)$ or $p(Y|X)$ [25, 33]. They are widely used and well-studied. One issue with PGMs in practice, however, is that exact inference is not guaranteed to be tractable. Whether exact inference can be performed or not is heavily dependent on the particular model being used; in the intractable cases, approximate techniques such as MCMC or variational inference are often used.

Sum-product networks (SPNs) offer another method for representing joint and conditional distributions, or $p(X, Y)$ and $p(Y|X)$. SPNs are directed, acyclic graphs (DAGs) in which internal nodes are sums or products and in which leaf nodes are univariate distributions over random variables. Having been studied only more recently, they are of interest in large part because of the guarantees that can be made regarding the time complexity of inference. In particular, exact inference is guaranteed to take time linear in the size of the network.

This dissertation expands the usefulness of SPNs by addressing the problem of learning the structure, or DAG, of an SPN from data. Two novel methods for doing so are introduced and one of them is further adapted to the online learning setting. We also introduce what we call nonmonotone SPNs and show how to use them in practice. In the next section we informally describe the mathematics behind SPNs. Then we present a more detailed overview of the contributions of this dissertation. Finally, a very brief survey of the literature is laid out.

1.1 Background

Perhaps the most straightforward way to represent a joint distribution $p(X, Y)$ is to make a list of probabilities aligned to a list of the possible assignments to X and Y. This is only possible to do for discrete variables, but even then it is usually not practical since the

D, S	(0,0)	(0,1)	(1,0)	(1,1)
$p(D, S)$	0.81	0.09	0.005	0.095

Figure 1.1: A joint distribution over binary variables D (disease) and S (symptom). A more realistic distribution, accounting for many more diseases and symptoms, could be of aid in medical diagnosis tasks.

number of possible variable assignments grows exponentially with the number of variables. Nevertheless, this is essentially the approach taken by the network polynomial [11], a method of representing a distribution as a polynomial. The coefficients in the network polynomial form the list of probabilities, and the variables are used to select the appropriate coefficient(s). Interestingly, Darwiche [11] shows that certain partial derivatives of this polynomial have probabilistic interpretations.

As an example, consider the simple medical diagnosis model in which two binary random variables D and S indicate the presence or absence of, respectively, a particular disease and a particular symptom. Figure 1.1 shows $p(D, S)$ represented as a list of probabilities. Using the shorthand $p_{ij} \triangleq p(D{=}i, S{=}j)$, the network polynomial for this distribution is

$$\begin{aligned} &f(\lambda_{D=0},\lambda_{D=1}, \lambda_{S=0}, \lambda_{S=1}) \\ &\qquad = p_{00}\lambda_{D=0}\lambda_{S=0} + p_{01}\lambda_{D=0}\lambda_{S=1} + p_{10}\lambda_{D=1}\lambda_{S=0} + p_{11}\lambda_{D=1}\lambda_{S=1} \\ &\qquad = 0.81\lambda_{D=0}\lambda_{S=0} + 0.09\lambda_{D=0}\lambda_{S=1} + 0.005\lambda_{D=1}\lambda_{S=0} + 0.095\lambda_{D=1}\lambda_{S=1} \end{aligned} \tag{1.1}$$

where the variables $\lambda_{D=0}, \lambda_{D=1}, \lambda_{S=0}$, and $\lambda_{S=1}$ are indicator variables. An indicator $\lambda_{X=x}$ takes the value 1 when the variable X takes the value x, and is 0 otherwise. To compute $p(D{=}0, S{=}0)$ we set $\lambda_{D=0} = 1$ and $\lambda_{S=0} = 1$; the other indicators are set to 0, giving

$$\begin{aligned} f(1, 0, 1, 0) &= p_{00}(1)(1) + p_{01}(1)(0) + p_{10}(0)(1) + p_{11}(0)(0) \\ &= p_{00} = 0.81 = p(D{=}0, S{=}0). \end{aligned}$$

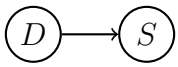
To support computing marginal probabilities we modify the definition of an indicator variable; $\lambda_{X=x}$ now takes the value 1 when $X = x$ *or* when we decide to sum out X, and takes the value 0 otherwise. To compute the marginal $p(D = 0)$, then, $\lambda_{D=0}$ gets set to 1 and, since we are summing out S, both $\lambda_{S=0}$ and $\lambda_{S=1}$ get set to 1; thus we have

$$\begin{aligned} f(1, 0, 1, 1) &= p_{00}(1)(1) + p_{01}(1)(1) + p_{10}(0)(1) + p_{11}(0)(1) \\ &= p_{00} + p_{01} = 0.9 = p(D{=}0). \end{aligned}$$

Any joint or marginal probability can be computed by setting the indicators appropriately and evaluating f.

A network polynomial for a distribution over n binary random variables would have 2^n monomial terms, 2^n corresponding coefficients, and $2n$ indicator variables. If the space and time requirements for using a network polynomial grow exponentially with n, why use them at all? The answer is that network polynomials are not used directly. Instead, some network polynomials can be computed using arithmetic circuits (ACs) that have polynomial size in n. Network polynomials that only have exponential-sized arithmetic circuits computing them are not used in practice at all. In a polynomial-sized AC there are still $2n$ indicator variables, acting now as input nodes, but there is no longer a $O(2^n)$ space requirement.

We now show how the example AC in Figure 1.2b computes the network polynomial for the medical diagnosis distribution. Ordering the bottom row of product nodes from left to right, the polynomials they compute are, respectively, $p_{0|0}\lambda_{S=0}$, $p_{0|1}\lambda_{S=0}$, $p_{1|0}\lambda_{S=1}$, and $p_{1|1}\lambda_{S=1}$. Ordering the two sum nodes in the middle from left to right, the polynomials they compute are, respectively, $p_{0|0}\lambda_{S=0} + p_{1|0}\lambda_{S=1}$ and $p_{0|1}\lambda_{S=0} + p_{1|1}\lambda_{S=1}$. Ordering the two top product nodes from left to right, the polynomials they compute are, respectively, $p(D{=}0)\lambda_{D=0}(p_{0|0}\lambda_{S=0} + p_{1|0}\lambda_{S=1})$ and $p(D{=}1)\lambda_{D=1}(p_{0|1}\lambda_{S=0} + p_{1|1}\lambda_{S=1})$. Finally we see that

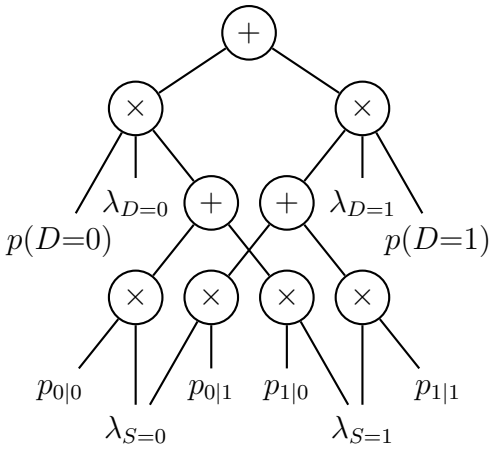


$$p(D, S) = p(D)p(S|D)$$

D	0	1
$p(D)$	0.9	0.1

D	$p(S{=}0\|D)$	$p(S{=}1\|D)$
0	0.9	0.1
1	0.05	0.95

(a) Bayesian network.



(b) Arithmetic circuit.

Figure 1.2: A Bayesian network and arithmetic circuit, each representing $p(D, S)$, the medical diagnosis distribution. In the AC we use the shorthand $p_{i|j} \triangleq p(S{=}i|D{=}j)$.

the root node computes

$$p(D{=}0)\lambda_{D=0}(p_{0|0}\lambda_{S=0} + p_{1|0}\lambda_{S=1}) + p(D{=}1)\lambda_{D=1}(p_{0|1}\lambda_{S=0} + p_{1|1}\lambda_{S=1}) =$$
$$p(D{=}0)p_{0|0}\lambda_{D=0}\lambda_{S=0} + p(D{=}0)p_{1|0}\lambda_{D=0}\lambda_{S=1} +$$
$$p(D{=}1)p_{0|1}\lambda_{D=1}\lambda_{S=0} + p(D{=}1)p_{1|1}\lambda_{D=1}\lambda_{S=1}.$$

If we replace the probability expressions in this polynomial with the appropriate values from the tables in Figure 1.2a, we see that the root node computes the network polynomial

$$f(\lambda_{D=0}, \lambda_{D=1}, \lambda_{S=0}, \lambda_{S=1})$$
$$= 0.81\lambda_{D=0}\lambda_{S=0} + 0.09\lambda_{D=0}\lambda_{S=1} + 0.005\lambda_{D=1}\lambda_{S=0} + 0.095\lambda_{D=1}\lambda_{S=1} \quad (1.2)$$

which is identical to the network polynomial in Equation 1.1.

Bayesian networks over discrete variables can be converted into equivalent ACs, which can then be used as inference engines for the original BNs [7, 8]. Since inference takes time linear in the size of the AC, the goal is to convert the BN into as small an AC as possible. For some BNs, such as those that exhibit determinism or context-specific independence in their

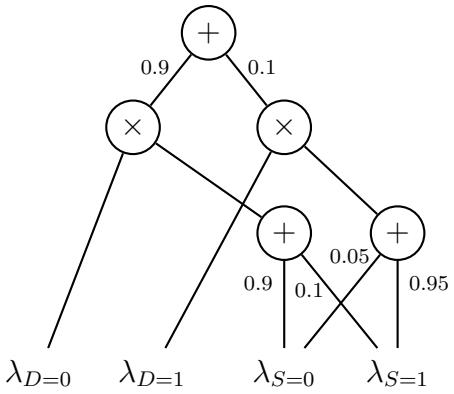


Figure 1.3: A sum-product network representing $p(D, S)$, the medical diagnosis distribution.

conditional probability tables, conversion to a small AC can be done because ACs can take advantage of this local structure. On the other hand, standard exact-inference techniques such as the jointree method [48], which does not take advantage of local structure, can be slow on these same BNs. Figure 1.2a shows a BN for the medical diagnosis distribution and Figure 1.2b an equivalent AC.

Sum-product networks are similar to ACs, but there are important differences. Both are DAGs whose internal nodes are sums and products. In an SPN, leaves can be indicators just like in ACs, but they can also be any univariate distribution; this allows SPNs to handle continuous variables. A sum node in an SPN computes a weighted sum of its inputs, instead of a simple sum as in ACs; weights are indicated by edge labels. Figure 1.3 shows an SPN that represents the medical-diagnosis distribution. The polynomial computed by its root node is

$$\begin{aligned} & f(\lambda_{D=0},\lambda_{D=1},\lambda_{S=0},\lambda_{S=1}) \\ & \quad = 0.9(\lambda_{D=0}(0.9\lambda_{S=0} + 0.1\lambda_{S=1})) + 0.1(\lambda_{D=1}(0.05\lambda_{S=0} + 0.95\lambda_{S=1})) \\ & \quad = 0.81\lambda_{D=0}\lambda_{S=0} + 0.09\lambda_{D=0}\lambda_{S=1} + 0.005\lambda_{D=1}\lambda_{S=0} + 0.095\lambda_{D=1}\lambda_{S=1} \end{aligned} \tag{1.3}$$

which, again, is identical to Equation 1.1.

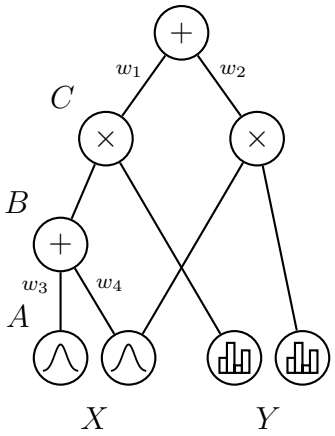


Figure 1.4: An SPN representing a distribution over continuous variable X and discrete variable Y. Node B computes a mixture model and node C computes an independence model.

In the literature, SPNs are viewed as standalone probability models instead of as a method for achieving exact inference in BNs. Thus the SPN literature contains algorithms for learning the parameters and the structure of an SPN from data; none of this is necessary when the goal is to create a BN inference engine. Like ACs, an SPN can compute any joint or marginal probability in time linear in its size. Consider what this means for an SPN modeling n binary variables. There are 2^n subsets of these n variables, and for each subset of size k there are 2^k possible value settings for the k variables. Since each value setting corresponds to a potential probabilistic query, there are $\sum_{k=1}^{n} 2^k \binom{n}{k}$ queries and a single evaluation of the SPN can answer any one of them. This takes into account only joint and marginal queries. Conditional probability queries can also be answered using two evaluations of the SPN. This is so since conditional probabilities are the ratio of two joint/marginal probabilities. For example, $p(y|x) = p(x, y)/p(x)$.

Now we offer another view on SPNs which sheds more light on how they work and how they can be constructed. In short, an SPN is a recursive combination of mixture models and what we call independence models. A mixture model is the weighted sum, using non-negative weights, of several distributions, where each one is over the same set of variables. An independence model is the product of several distributions, where each one is over a

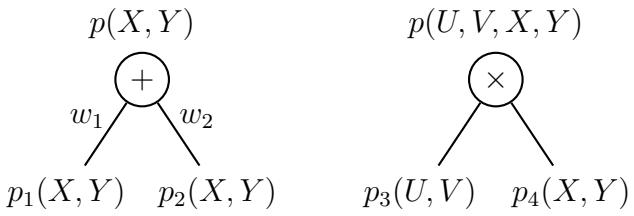


Figure 1.5: On the left distributions p_1 and p_2 are the components of a mixture model with mixing coefficients w_1 and w_2. On the right distributions p_3 and p_4 are combined in an independence model. Note that the scopes of p_1 and p_2 are identical and the scopes of p_3 and p_4 are disjoint.

disjoint set of variables. Mixtures are computed using sum nodes, independence models are computed using product nodes, and leaf nodes are univariate distributions; the leaves form the base case in the recursive combination of models. Figure 1.4 shows an SPN that represents a distribution over continuous random variable X and discrete random variable Y. The leaf nodes consist of two normal distributions over X and two categorical distributions over Y. Denote the distributions represented by the labeled nodes A, B, and C as p_A, p_B, and p_C, respectively. The distribution $p_A(X)$ is a normal over X, $p_B(X)$ is a mixture of p_A and the other normal, and $p_C(X,Y)$ is an independence model that combines p_B and one of the categorical distributions over Y.

For some distribution p let $\text{sc}(p)$ denote its scope, or the set of random variables that p models. An SPN is complete if $\text{sc}(p_1) = \text{sc}(p_2) = \cdots = \text{sc}(p_k)$ for every mixture of distributions $p_1, \ldots, p_k$. An SPN is decomposable if every independence model combining distributions $p_1, \ldots, p_k$ is such that for all pairs $(i,j), i \neq j$ we have $\text{sc}(p_i) \cap \text{sc}(p_j) = \emptyset$. If an SPN is complete and decomposable then a single pass through its network can compute any marginal probability in the distribution that it represents [38]. The following examples demonstrate why this is so. First we mix $p_1(X,Y)$ and $p_2(X,Y)$ and then we combine $p_3(U,V)$ and $p_4(X,Y)$ using an independence model. See Figure 1.5. In both cases we show how to compute a marginal probability in the resulting distribution p, assuming that any

marginal in the distributions $p_1, \ldots, p_4$ can be computed. For the mixture we have

$$\begin{aligned} p(x) &= \sum_y p(x, y) \\ &= \sum_y \left(w_1 p_1(x, y) + w_2 p_2(x, y) \right) \\ &= w_1 \sum_y p_1(x, y) + w_2 \sum_y p_2(x, y) \\ &= w_1 p_1(x) + w_2 p_2(x) \end{aligned}$$

where w_1, w_2 are the mixing coefficients and the last equality is true due to our assumption. For the independence model we have

$$\begin{aligned} p(x) &= \sum_u \sum_v \sum_y p(u, v, x, y) \\ &= \sum_u \sum_v \sum_y p_3(u, v) p_4(x, y) \\ &= \sum_u \sum_v p_3(u, v) \sum_y p_4(x, y) \\ &= p_4(x) \end{aligned}$$

where the last line holds due to our assumption and the fact that $\sum_u \sum_v p_1(u, v) = 1$. Viewing an SPN as a recursive combination of mixture and independence models, it is only left to show that marginalization can be done at the base case, the univariate distributions at the leaf nodes. This can be done, of course, since summing out the variable at a leaf node is equivalent to setting the leaf node to 1.

1.2 Overview

The set of distributions that can be represented by an SPN is constrained by its architecture, i.e., the structure of its DAG. For example, the SPN on the left in Figure 1.6 can only

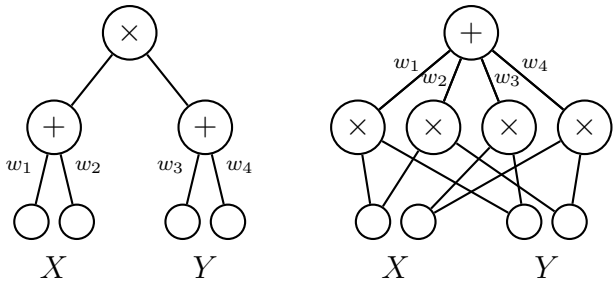


Figure 1.6: The structure of an SPN constrains the set of distributions representable by that SPN. The SPN on the left cannot represent all the distributions that the SPN on the right can represent. Note here that the hollow-circle leaf nodes represent indicator variables.

represent distributions in which X and Y are independent of each other, while the SPN on the right can represent any distribution over X and Y. The SPN on the right can do this since it is essentially representing its distribution as a list of probabilities, similar to Figure 1.1. If we expand this structure to n binary variables, adding a product node for every possible setting of the variables and mixing these with a single sum node, then the size of the SPN will grow exponentially. The problem of SPN structure learning is to find a DAG that is complex and large enough to model the target distribution but not much larger; this can also help avoid overfitting. The seminal paper [38] introducing SPNs describes a parameter-learning algorithm, but does not address the structure-learning problem; its experiments use a fixed-structure SPN.

Chapter 2, which was presented at NIPS 2012, describes the first published SPN structure-learning algorithm, BuildSPN. The algorithm works by recursively constructing a tree of "region" and "partition" nodes which is then transformed into an SPN, with region nodes corresponding to sets of sum nodes and partiton nodes corresponding to sets of product nodes. The root of the tree is fixed as a region node; then the tree is expanded by recursively adding children to the current leaves of the tree. The expansion is guided by clustering the rows and columns of slices of the training data.

A short time later Gens and Domingos [19] developed the current most widely-used structure learning algorithm, LearnSPN. It resembles BuildSPN in that it constructs a tree-structured SPN using recursive partitioning of the rows and columns of the dataset.

However, its column-partitioning strategy uses pair-wise independence tests, an approach that, from a theoretical perspective, is much better-justified than our column-clustering approach. Importantly, unlike BuildSPN, LearnSPN also combines the structure-learning and parameter-learning steps into one.

Chapter 3, which was presented at IJCAI 2015, describes another algorithm, SearchSPN, for learning the structure of an SPN. It takes advantage of the column-partitioning strategy from LearnSPN and is also able to learn the structure and parameters simultaneously. Unlike LearnSPN, which is limited to learning tree-structured SPNs, SearchSPN is able to learn DAG-structured SPNs. We show that SearchSPN is competetive with LearnSPN on a set of twenty real-world datasets and is far superior on a set of synthetic datasets. Another contribution of this paper is the first proof that the function computed by an SPN is the sum of all the embedded trees within the SPN. Zhao et al. [59] also arrived at this same result independently.

In Chapter 4 we describe the OnlineSearchSPN algorithm, an adaptation of SearchSPN to the online setting. This algorithm can change the SPN structure over time to adapt to a changing input distribution.

Chapter 5 shows how autoencoders and SPNs can be combined to model image data. SPNs provide a mechanism that helps the autoencoder generate images. The autoencoder helps improve the quality of images generated when drawing samples from the distribution represented by the SPN.

In Chapter 6 we generalize the definition of an SPN to allow negative-valued mixing coefficients in its mixture models. If an SPN has negative edge-weights we call it a non-monotone SPN. With negative weights there is the possibility that a nonmonotone SPN may compute negative values for some input. This assignment of "negative" probability ruins the probabilistic interpretation of the SPN. So we classify SPNs as being positive if they always compute non-negative probabilities and as being negative otherwise. We prove that, as long as a nonmonotone SPN is positive, computing marginals in it works in the same way it does

in a monotone SPN. We also introduce twin SPNs, a class of nonmonotone SPN that can easily be guaranteed to be positive. An algorithm for learning the structure and parameters of a twin SPN, LearnTwin, is also described and tested on synthetic data.

In summary, our major contributions include the first structure-learning algorithm for SPNs, a structure-learning algorithm that has some of the nice properties of LearnSPN while not being limited to tree structures, and an online SPN structure-learning algorithm. We also prove that every SPN is equivalent to a summation of all embedded trees in the SPN, where typically there are an exponential number of these trees. Lastly, we generalize SPNs to allow negative edge-weights, show how to compute marginals even in the presence of negative weights, and introduce a learning algorithm for a practical class of these generalized SPNs.

1.3 Brief History

The following short overview of the SPN literature, while incomplete, touches on many of the major papers in the field. As shown earlier, the foundational work for SPNs actually lies in work on the network polynomial and arithmetic circuit representations of Bayesian networks [11]. Poon and Domingos [38] builds on this work, introducing the SPN as a standalone model whose structure and parameters can be learned. They also present criteria which, if met, guarantee that an SPN does compute a network polynomial.

The introduction of SPNs was followed by several structure-learning algorithms. Most construct the SPN in a top-down fashion, starting at the root and working to the leaves [14, 19, 44]. We propose a structure-search algorithm [15], Peharz et al. [35] construct SPNs using a bottom-up approach, Lee et al. [27] propose an online structure-learning algorithm, and Vergari et al. [55] make improvements on LearnSPN.

Other work has expanded on techniques for learning the parameters of SPNs discriminatively [18], using moment matching [40], and using expectation-maximization [16, 37, 59]. Melibari et al. [32] show how to adapt SPNs to handle temporal data and several more papers

expand on more theoretical issues [37], including showing the power of deep SPNs [12] and how SPNs relate to Bayesian networks [58].

Chapter 2

Learning the Architecture of Sum-Product Networks Using Clustering on Variables

Published in the proceedings of NIPS 2012

Abstract

The sum-product network (SPN) is a recently-proposed deep model consisting of a network of sum and product nodes, and has been shown to be competitive with state-of-the-art deep models on certain difficult tasks such as image completion. Designing an SPN network architecture that is suitable for the task at hand is an open question. We propose an algorithm for learning the SPN architecture from data. The idea is to cluster variables (as opposed to data instances) in order to identify variable subsets that strongly interact with one another. Nodes in the SPN network are then allocated towards explaining these interactions. Experimental evidence shows that learning the SPN architecture significantly improves its performance compared to using a previously-proposed static architecture.

2.1 Introduction

The number of parameters in a textbook probabilistic graphical model (PGM) is an exponential function of the number of parents of the nodes in the graph. Latent variables can often be introduced such that the number of parents is reduced while still allowing the probability distribution to be represented. Figure 2.1 shows an example of modeling the relationship between symptoms of a set of diseases. The PGM at the left has no latent variables and the PGM at the right has an appropriately added "disease" variable. The model is able to be simplified because the symptoms are statistically independent of one another given the disease. The middle PGM shows a model in which the latent variable is introduced to no simplifying effect, demonstrating the need to be intelligent about what latent variables are added and how they are added.

Deep models can be interpreted as PGMs that introduce multiple layers of latent variables over a layer of observed variables [22]. The architecture of these latent variables (the size of the layers, the number of variables, the connections between variables) can dramatically affect the performance of these models. Selecting a reasonable architecture is often done by hand.

This paper proposes an algorithm that automatically learns a deep architecture from data for a sum-product network (SPN), a recently-proposed deep model that takes advantage of the simplifying effect of latent variables [38]. Learning the appropriate architecture for a

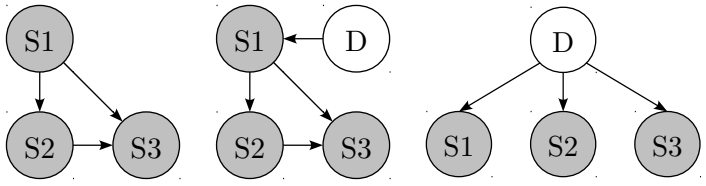


Figure 2.1: Introducing a latent variable. The PGM in (a) has no latent variables. The PGM in (b) has a latent variable introduced to no beneficial effect. The PGM in (c) has a latent variable that simplifies the model.

traditional deep model can be challenging [1, 57], but the nature of SPNs lend themselves to a remarkably simple, fast, and effective architecture-learning algorithm.

In proposing SPNs, Poon & Domingos introduce a general scheme for building an initial SPN architecture; the experiments they run all use one particular instantiation of this scheme to build an initial "fixed" architecture that is suitable for image data. We will refer to this architecture as the Poon architecture. Training is done by learning the parameters of an initial SPN; after training is complete, parts of the SPN may be pruned to produce a final SPN architecture. In this way both the weights and architecture are learned from data.

We take this a step further by also learning the initial SPN architecture from data. Our algorithm works by finding subsets of variables (and sets of subsets of variables) that are highly dependent and then effectively combining these together under a set of latent variables. This encourages the latent variables to act as mediators between the variables, capturing and representing the dependencies between them. Our experiments show that learning the initial SPN architecture in this way improves its performance.

2.2 Sum-Product Networks

Sum-product networks are rooted, directed acyclic graphs (DAGs) of sum, product, and leaf nodes. Edges connecting sum nodes to their children are weighted using non-negative weights. The value of a sum node is computed as the dot product of its weights with the values of its child nodes. The value of a product node is computed by multiplying the values of its child nodes. A simple SPN is shown in Figure 2.2.

Leaf node values are determined by the input to the SPN. Each input variable has an associated set of leaf nodes, one for each value the variable can take. For example, a binary variable would have two associated leaf nodes. The leaf nodes act as indicator functions, taking the value 1 when the variable takes on the value that the leaf node is responsible for and 0 otherwise.

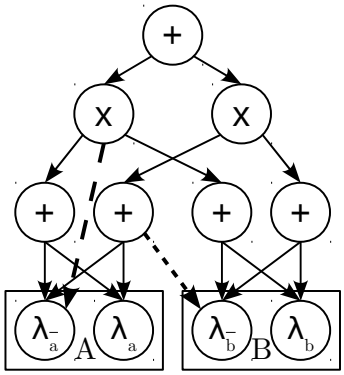


Figure 2.2: A simple SPN over two binary variables A and B. The leaf node $\lambda_{\bar{a}}$ takes value 1 if $A = 0$ and 0 otherwise while leaf node λ_a takes value 1 if $A = 1$ and 0 otherwise. If the value of A is not known then both leaf nodes take value 1. Leaf nodes $\lambda_{\bar{b}}$ and λ_b behave similarly. Weights on the edges connecting sum nodes with their children are not shown. The short-dashed edge causes the SPN to be incomplete. The long-dashed edge causes the SPN to be inconsistent.

An SPN can be constructed such that it is a representation of some probability distribution, with the value of its root node and certain partial derivatives with respect to the root node having probabilistic meaning. In particular, all marginal probabilities and many conditional probabilities can be computed [11]. Consequently an SPN can perform exact inference and does so efficiently when the size of the SPN is polynomial in the number of variables.

If an SPN does represent a probability distribution then we call it a valid SPN; of course, not all SPNs are valid, nor do they all facilitate efficient, exact inference. However, Poon & Domingos proved that if the architecture of an SPN follows two simple rules then it *will* be valid. (Note that this relationship does not go both ways; an SPN may be valid and violate one or both of these rules.) This, along with showing that SPNs can represent a broader class of distributions than other models that allow for efficient and exact inference are the key contributions made by Poon & Domingos.

To understand these rules it will help to know what the "scope of an SPN node" means. The scope of an SPN node n is a subset of the input variables. This subset can be

determined by looking at the leaf nodes of the subgraph rooted at n. All input variables that have one or more of their associated leaf nodes in this subgraph are included in the scope of the node. We will denote the scope of n as $scope(n)$.

The first rule is that all children of a sum node must have the same scope. Such an SPN is called *complete*. The second rule is that for every pair of children, (c_i, c_j), of a product node, there must not be contradictory leaf nodes in the subgraphs rooted at c_i and c_j. For example, if the leaf node corresponding to the variable X taking on value x is in the subgraph rooted at c_i, then the leaf nodes corresponding to the variable X taking on any other value may not appear in the subgraph rooted at c_j. An SPN following this rule is called *consistent*. The SPN in Figure 2.2 violates completeness (due to the short-dashed arrow) and it violates consistency (due to the long-dashed arrow).

An SPN may also be *decomposable*, which is a property similar to, but somewhat more restrictive than consistency. A decomposable SPN is one in which the scopes of the children of each product node are disjoint. All of the architectures described in this paper are decomposable.

Very deep SPNs can be built using these rules as a guide. The number of layers in an SPN can be on the order of tens of layers, whereas the typical deep model has three to five layers. Recently it was shown that deep SPNs can compute some functions using exponentially fewer resources than shallow SPNs would need [12].

The Poon architecture is suited for modeling probability distributions over images, or other domains with local dependencies among variables. It is constructed as follows. For every possible axis-aligned rectangular region in the image, the Poon architecture includes a set of m sum nodes, all of whose scope is the set of variables associated with the pixels in that region. Each of these (non-single-pixel) regions are conceptually split vertically and horizontally in all possible ways to form pairs of rectangular subregions. For each pair of subregions, and for every possible pairing of sum nodes (one taken from each subregion), a product node is introduced and made the parent of the pair of sum nodes. The product node

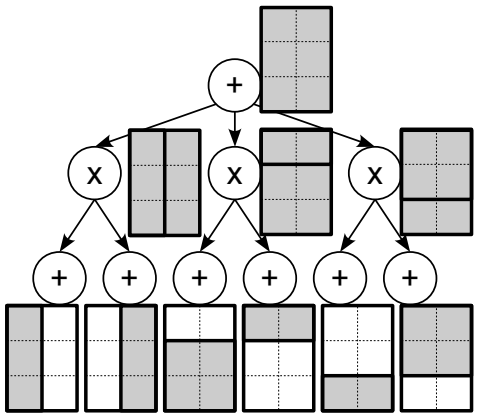


Figure 2.3: The Poon architecture with $m = 1$ sum nodes per region. Three product nodes are introduced because the 2×3-pixel image patch can be split vertically and horizontally in three different ways. In general the Poon architecture has number-of-splits times m^2 product nodes per region.

is also added as a child to all of the top region's sum nodes. Figure 2.3 shows a fragment of a Poon architecture SPN modeling a 2×3 image patch.

2.3 Cluster Architecture

As mentioned earlier, care needs to be taken when introducing latent variables into a model. Since the effect of a latent variable is to help explain the interactions between its child variables [25], it makes little sense to add a latent variable as the parent of two statistically independent variables.

In the example in Figure 2.4, variables W and X strongly interact and variables Y and Z do as well. But the relationship between all other pairs of variables is weak. The PGM in (a), therefore, allows latent variable A to take account of the interaction between W and X. On the other hand, variable A does little in the PGM in (b) since W and Y are nearly independent. A similar argument can be made about variable B. Consequently, variable C in the PGM in (a) can be used to explain the weak interactions between variables, whereas in the PGM in (b), variable C essentially has the task of explaining the interaction between all the variables.

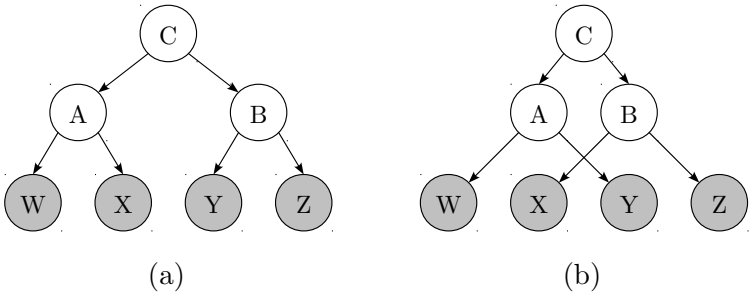


Figure 2.4: Latent variables explain the interaction between child variables, causing the children to be independent given the latent variable parent. If variable pairs (W, X) and (Y, Z) strongly interact and other variable pairs do not, then the PGM in (a) is a more suitable model than the PGM in (b).

In the probabilistic interpretation of an SPN, sum nodes are associated with latent variables. (The evaluation of a sum node is equivalent to summing out its associated latent variable.) Each latent variable helps the SPN explain interactions between variables in the scope of the sum nodes. Just as in the example, then, we would like to place sum nodes over sets of variables with strong interactions.

The Poon architecture takes this principle into account. Images exhibit strong interactions between pixels in local spatial neighborhoods. Taking advantage of this prior knowledge, the Poon architecture chooses to place sum nodes over local spatial neighborhoods that are rectangular in shape.

There are a few potential problems with this approach, however. One is that the Poon architecture includes many rectangular regions that are long and skinny. This means that the pixels at each end of these regions are grouped together even though they probably have only weak interactions. Some grouping of weakly-interacting pixels is inevitable, but the Poon architecture probably does this more than is needed. Another problem is that the Poon architecture has no way of explaining strongly-interacting, non-rectangular local spatial regions. This is a major problem because such regions are very common in images. Additionally, if the data does not exhibit strong spatially-local interactions then the Poon architecture could perform poorly.

Our proposed architecture (we will call it the cluster architecture) avoids these problems. Large regions containing non-interacting pixels are avoided. Sum nodes can be placed over spatially-local, non-rectangular regions; we are not restricted to rectangular regions, but can explain arbitrarily-shaped blob-like regions. In fact, the regions found by the cluster architecture are not required to exhibit spatial locality. This makes our architecture suitable for modeling data that does not exhibit strong spatially-local interactions between variables.

2.3.1 Building a Cluster Architecture

As was described earlier, a sum node s in an SPN has the task of explaining the interactions between all the variables in its scope. Let $scope(s) = \{V_1, \cdots, V_n\}$. If n is large, then this task will likely be very difficult. SPNs have a mechanism for making it easier, however. Essentially, s delegates part of its responsibilities to another set of sum nodes. This is done by first forming a partition of $scope(s)$, where $\{S_1, \cdots, S_k\}$ is a partition of $scope(s)$ if and only if $\bigcup_i S_i = scope(s)$ and $\forall i, j(S_i \cap S_j = \emptyset)$. Then, for each subset S_i in the partition, an additional sum node s_i is introduced into the SPN and is given the task of explaining the interactions between all the variables in S_i. The original sum node s is then given a new child product node p and the product node becomes the parent of each sum node s_i.

In this example the node s is analogous to the variable C in Figure 2.4 and the nodes s_i are analogous to the variables A and B. So this partitioning process allows s to focus on explaining the interactions between the nodes s_i and frees it from needing to explain everything about the interactions between the variables $\{V_1, \cdots, V_n\}$. And, of course, the partitioning process can be repeated recursively, with any of the nodes s_i taking the place of s.

This is the main idea behind the algorithm for building a cluster architecture (see Algorithm 1 and Algorithm 2). However, due to the architectural flexibility of an SPN,

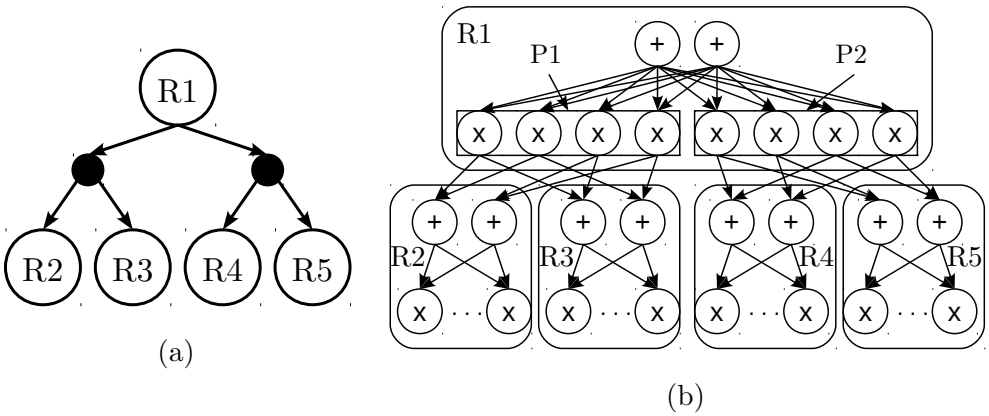


Figure 2.5: Subfigure (a) shows a region graph fragment consisting of region nodes R_1, R_2, R_3, R_4, and R_5. R_1 has two parition nodes (the smaller, filled-in nodes). Subfigure (b) shows the region graph converted to an SPN. In the SPN each region is allotted two sum nodes. The product nodes in R_1 are surrounded by two rectangles labeled P_1 and P_2; they correspond to the partition nodes in the region graph.

discussing this algorithm in terms of sum and product nodes quickly becomes tedious and confusing. The following definition will help in this regard.

Definition 2.1. A **region graph** is a rooted DAG consisting of region nodes and partition nodes. The root node is a region node. Partition nodes are restricted to being the children of region nodes and vice versa. Region and partition nodes have scopes just like nodes in an SPN. The scope of a node n in a region graph is denoted $scope(n)$.

Region nodes can be thought of as playing the role of sum nodes (explaining interactions among variables) and partition nodes can be thought of as playing the role of product nodes (delegating responsibilities). Using the definition of the region graph may not appear to have made things any simpler, but its benefits will become more clear when discussing the conversion of region graphs to SPNs (see Figure 2.5).

At a high level the algorithm for building a cluster architecture is simple: build a region graph (Algorithm 1 and Algorithm 2), then convert it to an SPN (Algorithm 3). These steps are described below.

Algorithm 1 BuildRegionGraph

```
1: Input: training data D
2: C' ← Cluster(D, 1)
3: for k = 2, 3, ..., ∞ do
4:     C ← Cluster(D, k)
5:     r ← Quality(C)/Quality(C')
6:     if r < 1 + δ then
7:         break
8:     else
9:         C' ← C
10:    end if
11: end for
12: G ← CreateRegionGraph()
13: n ← AddRegionNodeTo(G)
14: for i = 1, 2, ..., k do
15:    ExpandRegionGraph(G, n, C_i)
16: end for
```

Algorithm 1 builds a region graph using training data to guide the construction. In lines 2 through 9 the algorithm clusters the training instances into k clusters $C = \{C_1, \cdots, C_k\}$. Our implementation uses the scikit-learn [34] implementation of k-means to cluster the data instances, but any clustering method could be used. The value for k is chosen automatically; larger values of k are tried until increasing the value does not substantially improve a cluster-quality score. The remainder of the algorithm creates a single-node region graph G and then adds nodes and edges to G using k calls to Algorithm 2 (ExpandRegionGraph). To encourage the expansion of G in different ways, a different subset of the training data, C_i, is passed to ExpandRegionGraph on each call.

At a high level, Algorithm 2 partitions scopes into sub-scopes recursively, adding region and partition nodes to G along the way. The initial call to ExpandRegionGraph partitions the scope of the root region node. A corresponding partition node is added as a child of the root node. Two sub-region nodes (whose scopes form the partition) are then added as children to the partition node. Algorithm 2 is then called recursively with each of these sub-region nodes as arguments (unless the scope of the sub-region node is too small).

Algorithm 2 ExpandRegionGraph

```
1: Input: region graph G,
2:       region node n in G, training data D
3: S_n ← scope(n)
4: {S_1, S_2} ← PartitionScope(S_n, D)
5: S ← ScopesOfAllRegionNodesIn(G)
6: for all S_r ∈ S s.t. S_r ⊂ S_n do
7:     p_1 ← |S_1 ∩ S_r|/|S_1 ∪ S_r|
8:     p_2 ← |S_2 ∩ S_r|/|S_2 ∪ S_r|
9:     if max{p_1, p_2} > threshold then
10:        S_1 ← S_r
11:        S_2 ← S_n \ S_r
12:        break
13:    end if
14: end for
15: n_1 ← GetOrCreateRegionNode(G, S_1)
16: n_2 ← GetOrCreateRegionNode(G, S_2)
17: if PartitionDoesNotExist(G, n, n_1, n_2) then
18:    p ← NewPartitionNode()
19:    AddChildToRegionNode(n, p)
20:    AddChildToPartitionNode(p, n_1)
21:    AddChildToPartitionNode(p, n_2)
22: end if
23: if S_1 ∉ S ∧ |S_1| > 1 then
24:    ExpandRegionGraph(G, n_1)
25: end if
26: if S_2 ∉ S ∧ |S_2| > 1 then
27:    ExpandRegionGraph(G, n_2)
28: end if
```

In line 4 of Algorithm 2 the PartitionScope function in our implementation uses the k-means algorithm in an unusual way. Instead of partitioning the instances of the training dataset D into k instance-clusters, it partitions variables into k variable-clusters as follows. D is encoded as a matrix, each row being a data instance and each column corresponding to a variable. Then k-means is run on D^T, causing it to partition the variables into k clusters. Actually, the PartitionScope function is only supposed to partition the variables in $scope(n)$, not all the variables (note its input parameter). So before calling k-means we build a new

matrix D_n by removing columns from D, keeping only those columns that correspond to variables in $scope(n)$. *Then* k-means is run on D_n^T and the resulting variable partition is returned. The k-means algorithm serves the purpose of detecting subsets of variables that strongly interact with one another. Other methods (including other clustering algorithms) could be used in its place.

After the scope S_n of a node n has been partitioned into S_1 and S_2, Algorithm 2 (lines 5 through 14) looks for region nodes in G whose scope is similar to S_1 or S_2; if region node r with scope S_r is such a node, then S_1 and S_2 are adjusted so that $S_1 = S_r$ and $\{S_1, S_2\}$ is still a partition of S_n. Lines 15 through 22 expand the region graph based on the partition of S_n. If node n does not already have a child partition node representing the partition $\{S_1, S_2\}$ then one is created (p in line 18); p is then connected to child region nodes n_1 and n_2, whose scopes are S_1 and S_2, respectively.

Note that n_1 and n_2 may be newly-created region nodes or they may be nodes that were created during a previous call to Algorithm 2. We recursively call ExpandRegionGraph only on newly-created nodes; the recursive call is also not made if the node is a leaf node ($|S_i| = 1$) since partitioning a leaf node is not helpful (see lines 23 through 28).

After the k calls to Algorithm 2 have been made, the resulting region graph must be converted to an SPN. Figure 2.5 shows a small subgraph from a region graph and its conversion into an SPN; this example demonstrates the basic pattern that can be applied to all region nodes in G in order to generate an SPN. A more precise description of this conversion is given in Algorithm 3. In this algorithm the assumption is made (noted in the comments) that certain sum nodes are inserted before others. This assumption can be guaranteed if the algorithm performs a postorder traversal of the region nodes in G in the outermost loop. Also note that the ConnectProductsToSums method connects product nodes of the current region with sum nodes from its subregions; the children of a product node consist of a single node drawn from each subregion, and there is a product node for every possible combination of such sum nodes.

Algorithm 3 BuildSPN

```
Input: region graph G, sums per region m
Output: SPN S
R ← RegionNodesIn(G)
for all r ∈ R do
    if IsRootNode(r) then
        N ← AddSumNodesToSPN(S, 1)
    else
        N ← AddSumNodesToSPN(S, m)
    end if
    P ← ChildPartitionNodesOf(r)
    for all p ∈ P do
        C ← ChildrenOf(p)
        O ← AddProductNodesToSPN(S, m^|C|)
        for all n ∈ N do
            AddChildrenToSumNode(n, O)
        end for
        Q ← empty list
        for all c ∈ C do
            //We assume the sum nodes associated
            //with c have already been created.
            U ← SumNodesAssociatedWith(c)
            AppendToList(Q, U)
        end for
        ConnectProductsToSums(O, Q)
    end for
end for
return S
```

2.4 Experiments and Results

Poon showed that SPNs can outperform deep belief networks (DBNs), deep Boltzman machines (DBMs), principle component analysis (PCA), and a nearest- neighbors algorithm (NN) on a difficult image completion task. The task is the following: given the right/top half of an image, paint in the left/bottom half of it. The completion results of these models were compared qualitatively by inspection and quantitatively using mean squared error (MSE). SPNs produced the best results; our experiments show that the cluster architecture significantly improves SPN performance.

We matched the experimental set-up reported in [38] in order to isolate the effect of changing the initial SPN architecture and to make their reported results directly comparable to several of our results. They add 20 sum nodes for each non-unit and non-root region. The root region has one sum node and the unit regions have four sum nodes, each of which function as a Gaussian over pixel values. The Gaussians means are calculated using the training data for each pixel, with one Gaussian covering each quartile of the pixel-values histogram. Each training image is normalized such that its mean pixel value is zero with a standard deviation of one. Hard expectation maximization (EM) is used to train the SPNs; mini-batches of 50 training instances are used to calculate each weight update. All sum node weights are initialized to zero; weight values are decreased after each training epoch using an L_0 prior; add-one smoothing on sum node weights is used during network evaluation.

We test the cluster and Poon architectures by learning on the Olivetti dataset [46], the faces from the Caltech-101 dataset [17], an artificial dataset that we generated, and the shuffled-Olivetti dataset, which is the Olivetti dataset with the pixels randomly shuffled (all images are shuffled in the same way). The Caltech-101 faces were preprocessed as described by Poon & Domingos. The cluster architecture is compared to the Poon architectures using the negative log-likelihood (LLH) of the training and test sets as well as the MSE of the image completion results for the left half and bottom half of the images. We train ten cluster architecture SPNs and ten Poon architecture SPNs. Average results across the ten SPNs along with the standard deviation are given for each measurement.

On the Olivetti and Caltech-101 Faces datasets the Poon architecture resulted in better training set LLH, but the cluster architecture generalized better, getting a better test set LLH (see Table 2.1). The cluster architecture was also clearly better at the image completion tasks as measured by MSE.

The difference between the two architectures is most pronounced on the artificial dataset. The images in this dataset are created by pasting randomly-shaded circle- and diamond-shaped image patches on top of one another (see Figure 2.6), ensuring that various

Table 2.1: Results of experiments on the Olivetti, Caltech 101 Faces, artificial, and shuffled-Olivetti datasets comparing the Poon and cluster architectures. Negative log-likelihood (LLH) of the training set and test set is reported along with the MSE for the image completion results (both left-half and bottom-half completion results).

Dataset	**Measurement**	**Poon**	**Cluster**
Olivetti	Train LLH	318 ± 1	433 ± 17
	Test LLH	863 ± 9	715 ± 31
	MSE (left)	996 ± 42	814 ± 35
	MSE (bottom)	963 ± 42	820 ± 38
Caltech Faces	Train LLH	289 ± 4	379 ± 8
	Test LLH	674 ± 15	557 ± 11
	MSE (left)	1968 ± 89	1746 ± 87
	MSE (bottom)	1925 ± 82	1561 ± 44
Artificial	Train LLH	195	169
	Test LLH	266 ± 4	223 ± 6
	MSE (left)	842 ± 51	558 ± 27
	MSE (bottom)	877 ± 85	561 ± 29
Shuffled	Train LLH	793 ± 3	442 ± 14
	Test LLH	1193 ± 3	703 ± 14
	MSE (left)	811 ± 11	402 ± 16
	MSE (bottom)	817 ± 17	403 ± 17

pixel patches are statistically independent. The cluster architecture outperforms the Poon architecture across all measures on this dataset (see Table 2.1); this is due to its ability to focus resources on non-rectangular regions.

To demonstrate that the cluster architecture does not rely on the presence of spatially-local, strong interactions between the variables, we repeated the Olivetti experiment with the pixels in the images having been shuffled. In this experiment (see Table 2.1) the cluster architecture was, as expected, relatively unaffected by the pixel shuffling. The LLH measures remained basically unchanged from the Olivetti to the Olivetti-shuffled datasets. (The MSE results did not stay the same because the image completions happened over different subsets of the pixels.) On the other hand, the performance of the Poon architecture dropped

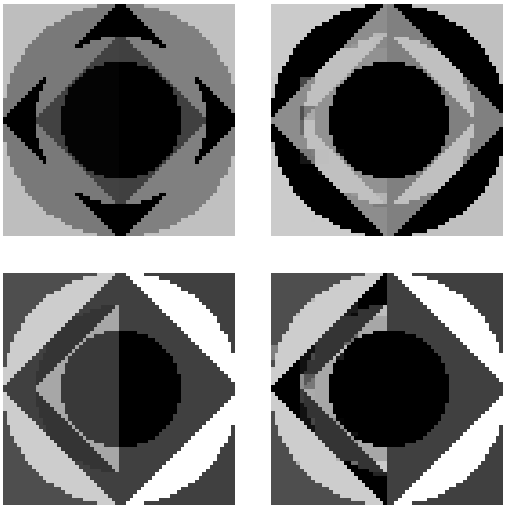


Figure 2.6: A cluster-architecture SPN completed the images in the left column and a Poon-architecture SPN completed the images in the right column. All images shown are left-half completions. The top row is the best results as measured by MSE and the bottom row is the worst results. Note the smooth edges in the cluster completions and the jagged edges in the Poon completions.

considerably due to the fact that it was no longer able to take advantage of strong correlations between neighboring pixels.

Figure 2.7 visually demonstrates the difference between the rectangular-regions Poon architecture and the arbitrarily-shaped-regions cluster architecture. Artifacts of the different region shapes can be seen in subfigure (a), where some regions are shaded lighter or darker, revealing region boundaries. Subfigure (b) compares the best of both architectures, showing image completion results on which both architectures did well, qualitatively speaking. Note how the Poon architecture produces results that look "blocky", whereas the cluster architecture produces results that are smoother-looking.

Algorithm 1 expands a region graph k times (lines 12 and 13). The value of k can significantly affect test set LLH, as shown in Table 2.2. A value that is too low leads to an insufficiently powerful model and a value that is too high leads to a model that overfits the training data and generalizes poorly.

(a) (b)

Figure 2.7: The completion results in subfigure (a) highlight the difference between the rectangular-shaped regions of the Poon architecture (top image) and the blob-like regions of the cluster architecture (bottom image), artifacts of which can be seen in the completions. Subfigure (b) shows ground truth images, cluster-architecture SPN completions, and Poon-architecture SPN completions in the left, middle, and right columns respectively. Left-half completions are in the top row and bottom-half completions are in the bottom row.

A singly-expanded model ($k = 1$) is optimal for the Olivetti dataset. This may be due in part to the Olivetti dataset having only one distinct class of images (faces in a particular pose). Datasets with more image classes may benefit from additional expansions. To experiment with this hypothesis we create two new datasets: Olivetti45 and Olivetti4590. Olivetti45 is created by augmenting the Olivetti dataset with Olivetti images that are rotated by -45 degrees. Olivetti4590 is built similarly but with rotations by -45 degrees and by -90 degrees. The Olivetti45 dataset, then, has two distinct classes of images: rotated and non-rotated. Similarly, Olivetti4590 has three distinct image classes. Table 2.2 shows that, as expected, the optimal value of k for the Olivetti45 and Olivetti4590 datasets is two and three, respectively.

Note that the Olivetti test set LLH with $k = 1$ in Table 2.2 is better than the test set LLH reported in Table 2.1. This shows that the algorithm for automatically selecting k in

Table 2.2: Test set LLH values for the Olivetti, Olivetti45, and Olivetti4590 datasets for different values of k. For each dataset the best LLH value is marked in bold.

Dataset	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
Olivetti	**650**	653	671	685	711	716	717	741
Olivetti45	523	**495**	508	529	541	528	544	532
Olivetti4590	579	576	**550**	554	577	595	608	592

Algorithm 1 is not optimal. Another option is to use a hold-out set to select k, although this method may not not be appropriate for small datasets.

2.5 Conclusion

The algorithm for learning a cluster architecture is simple, fast, and effective. It allows the SPN to focus its resources on explaining the interactions between arbitrary subsets of input variables. And, being driven by data, the algorithm guides the allocation of SPN resources such that it is able to model the data more efficiently. Future work includes experimenting with alternative clustering algorithms, experimenting with methods for selecting the value of k, and experimenting with variations of Algorithm 2 such as generalizing it to handle partitions of size greater than two.

Chapter 3

Greedy Structure Search for Sum-Product Networks

Published in the proceedings of IJCAI 2015

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.

3.1 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 [38]. SPN parameters can be learned using expectation-maximization or gradient descent in both the generative and discriminative settings [18, 38].

Results [18] on twenty real-world datasets compare SPNs to Bayesian networks learned using the WinMine toolkit [9] and to Markov networks learned using two other methods [13, 41]. 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 [44] were found when comparing an augmented SPN to mixtures of trees [31] and latent tree models [10].

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 [14, 19, 35, 44]. 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 [42].

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.

3.2 Previous Work

Delalleau & Bengio [12] 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 [11] 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 [38].

Lowd & Domingos [29] 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 [14], GD [19], RL [44], and PGP [35]. 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 [30]; 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 [28]; 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.

3.3 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 3.1. 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.

We do not use the similar definition from Gens & Domingos [19], even though it handles continuous variables more naturally, because Definition 3.1 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 3.2 (based on Chan & Darwiche [6])**.** 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 $\mathrm{ch}(n)$ be its children. Let w_{st} be the weight on the edge between sum node s and its child t; we assume that $\sum_{t \in \mathrm{ch}(s)} w_{st} = 1$. 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 $f_r = \prod_{t \in \mathrm{ch}(r)} f_t$; if r is a sum node then $f_r = \sum_{t \in \mathrm{ch}(r)} w_{rt} f_t$. 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 Appendix 3.A

Theorem 3.1. 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 3.1 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 3.1 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 [38]. Inferring $\text{argmax}_{X,Z} P(X, Z)$, 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 [6, 38]. We let c^*_x be an MPE circuit for variable setting $X = x$.

Poon & Domingos [38] 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

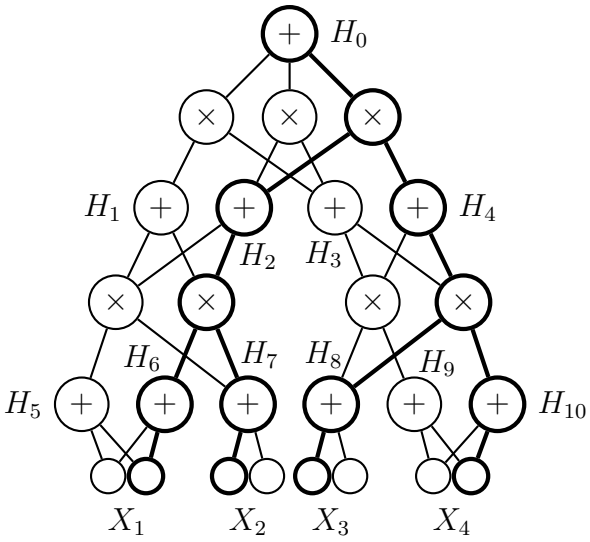


Figure 3.1: The bold lines show a complete sub-circuit. The circuit corresponds to variable settings $H_0{=}2,\ H_2{=}1,\ H_4{=}1,\ H_6{=}1,\ H_7{=}0,\ H_8{=}0,\ H_{10}{=}1,\ X_1{=}1,\ X_2{=}0,\ X_3{=}0$, and $X_4{=}1$.

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 $t \in \mathrm{ch}(s)$ 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 3.1.

3.4 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 $H_s, s \in \mathrm{ch}(p)$. Thus MPE inference gives us a sample of the hidden variables associated with children of p; this sample is added to T_p.

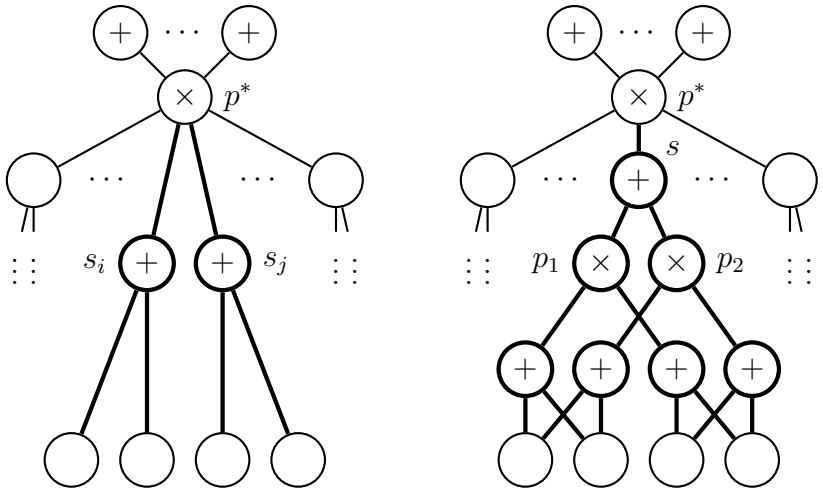


Figure 3.2: An example of the MixClones algorithm applied to the product node p^* and a subset of its children $S_1 = \{s_i, s_j\}$ in the network fragment on the left ($k = 2$). The algorithm replaces the bold nodes in the left with the bold nodes in the network fragment on the right.

In brief, a step in the search is as follows. We use the datasets T_p to select a product node p^* (Section 3.4.2). We pass T_{p^*} to modified versions of the variable- and instance-partitioning procedures (Section 3.4.2) found in GD [19]. The structure modification algorithm (Section 3.4.1) then uses the partitioning of T_{p^*} to change the 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.

3.4.1 Search Operator

The structure modification algorithm, or structure operator, we use is called MixClones and is outlined in Algorithm 4. An example of applying this operator to an SPN is shown in Figure 3.2. 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

Algorithm 4 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

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, $S_2, \ldots, S_k$, 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 $S_1, \ldots, S_k$ are the sub-distribution parameters. Outside of MixClones these parameters are chosen to (indirectly) increase the training set likelihood.

Algorithm 5 SEARCHSPN(N, T)

Input: SPN N, training instances T
while stop criteria not met **do**
  $p^* \leftarrow \text{argmin}\, p \prod_{w \in W_p} w$
  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

3.4.2 Search Algorithm

This section describes and justifies Algorithm 5 (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 3.1, 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

$$\tilde{\mathcal{L}}(f_r|T) = \prod_{x \in T} c_x^*(x), \tag{3.1}$$

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 3.1 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 $p^* = \operatorname{argmin} p \prod_{w \in W_p} w$ is the product node that contributes least to a high value of $\tilde{\mathcal{L}}$.

Selecting S_1 and k.

With p^* selected, we would like to change the structure of the SPN at p^* such that the distribution $\Phi_{p^*} = \prod_{s \in \mathrm{ch}(p^*)} \Phi_s$ 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 $\Psi_{p^*} = \prod_{s \in \mathrm{ch}(p^*)} \Psi_s$ better fits the training set T_{p^*}, where Ψ_s, for each $s \in \mathrm{ch}(p^*)$, is a categorical distribution over the variable H_s and its parameters are the weights of s.

If the variables in $V_{p^*} = \{H_s | s \in \mathrm{ch}(p^*)\}$ 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 i^{th} 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 [44]. 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.

3.5 Experiments

We compare SearchSPN and LearnSPN (from GD) on twenty datasets that were recently used in Rooshenas and Lowd [44] and Gens and Domingos [19]. 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 3.1 and Table 3.2 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 3.1: 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

3.5.1 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 [43]. We also assume it is a difficult problem for MACs since computing the permanent is #P-complete [52].

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

Table 3.2: 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

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 3.2 were generated by sampling from the constructed Markov networks using Gibbs sampling.

3.5.2 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 [19], although statistical significance cannot be determined in this comparison.

Comparing to the results reported in [44]—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 [45] 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 3.2 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.

3.6 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.

3.A Proof of Theorem 3.1

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 3.1. 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 $f_r = \sum_{t \in \mathrm{ch}(r)} w_{rt} f_t$ and by the inductive hypothesis $f_r = \sum_{t \in \mathrm{ch}(r)} w_{rt} \sum_{c' \in C_t} g_{c'}$. The double summation ranges over $C^{\cup}$ so $f_r = \sum_{c' \in C^{\cup}} w_{rt} g_{c'}$. Applying the mapping yields $f_r = \sum_{c \in C} g_c$. ■

Chapter 4

Online Structure Search for Sum-Product Networks

Abstract

A variety of algorithms exist for learning both the structure and parameters of sum-product networks (SPNs), a class of probabilistic model in which exact inference can be done quickly. The vast majority of them are batch learners, including a recently proposed algorithm, SearchSPN. However, SearchSPN has properties that make it particularly suited for adaptation to the online setting. In this paper we introduce the OnlineSearchSPN algorithm which does just that. We compare it to two general methods that build online learners from batch learners; one learns poor models quickly and the other learns good models slowly. Our experiments show that OnlineSearchSPN achieves the best of both methods. The test likelihood values of the models it learns are as good as the slow learner, while the training times needed to learn the models are much closer to the fast learner.

4.1 Introduction

Sum-product networks (SPNs) are probabilistic models designed to ensure that computing marginal probabilities takes time linear in the size of the network. The first paper describing SPNs focused on their theoretical foundations, the computations needed for performing inference, and algorithms for learning parameter values given a fixed network structure [38]. Many algorithms have been developed since for learning both the structure and the parameters. The first such algorithm, BuildSPN, combines several tree-strucured SPNs into a single SPN; it then uses the hard-EM algorithm from the original SPN paper to learn the parameters in a second step [14]. The most widely-adopted structure learning algorithm, LearnSPN, builds a single tree-structured SPN; it improves on BuildSPN by using a more principled tree construction algorithm and by learning the structure and parameters simultaneously [19]. Further work explores several modifications to LearnSPN that aim to improve and simplify it [55]. Another structure-learning algorithm constructs an SPN from the bottom up [35] and yet another incorporates Markov networks, encoded as arithmetic circuits, into the SPN structure [44].

The SearchSPN [15] algorithm works iteratively, adding nodes to a given input SPN so that the SPN better models the training data. The expanded SPN is then fed back into this process, which is repeated until a stopping criteria is met. It is not restricted to learning tree SPNS, which gives it certain advantages over LearnSPN. The AugmentSPN [27] algorithm also uses an iterative approach, but does so in a way that lets it work in the online setting; it only learns tree SPNs. The OnlineSearchSPN algorithm, which we introduce in this paper, adapts SearchSPN to the online setting, giving an online algorithm that is not restricted to learning tree SPNs.

OnlineSearchSPN is applicable to situations in which a model must adapt to a changing input distribution. As new and differently-distributed input examples arrive, it will adjust the SPN to fit the new data. It is also applicable in the offline setting when the training dataset is large [5]. Since it only ever operates on a small amount of data at any

given time, it is possible to train an SPN on a dataset that does not fit in memory. Doing so with most other SPN structure learning algorithms would be prohibitively slow.

4.2 Sum-Product Networks

A sum-product network (SPN) is represented using a rooted, directed acyclic graph (DAG). Its leaf nodes are univariate distributions and its internal nodes consist of sum nodes and product nodes. Without loss of generality, we assume that a child of a sum node is a product node and that a child of a product node is either a sum node or a leaf node. A sum node computes a convex combination [1] of its inputs and a product node computes the product of its inputs. Each node n in an SPN is the root of a sub-DAG whose leaf nodes are univariate distributions over a set of variables. The *scope* of node n is defined to be that set of variables. The following recursive definition places conditions on the scopes of nodes; these ensure that the SPN represents a valid joint probability distribution over its scope [38].

Definition 4.1 (based on [18])**.** A *sum-product network (SPN)* is:

1. a tractable [2] univariate distribution,
2. a product of SPNs with disjoint scopes, or
3. a conical combination of SPNs with identical scopes.

The remainder of this section presents results and notation that will help in describing the SearchSPN and OnlineSearchSPN algorithms. It was recently proven [15, 59] that an SPN N computes the function

$$f(x) = \sum_{c \in C} g_c(x),$$

where C is the set of all complete sub-circuits in N, g_c is the function computed by sub-circuit $c \in C$, and x is an instantiation of the random vector $X = [X_1, \ldots, X_m]$. A complete

[1] Conical combinations are allowed, but weights are often assumed to be normalized. Moreover, [37] showed that normalizing the weights in an SPN does not change the distribution it represents.

[2] Here, tractable means the partition function and mode can be computed in constant time.

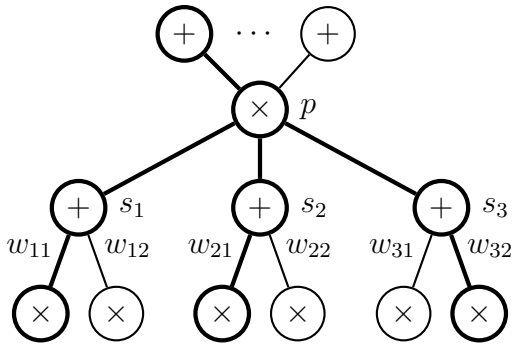


Figure 4.1: Product node p is shown, along with its parents, children, and grandchildren (the rest of the SPN is not shown). Bold lines indicate part of a complete sub-circuit that assigns $Y_p = [1, 1, 2]$; $w_p([1, 1, 2]) = [w_{11}, w_{21}, w_{32}]$.

sub-circuit c in an SPN is a tree whose root node is the root of the SPN; also, the children of each product node in c are also in the tree and exactly one child of each sum node in c is also in the tree [6]. To give an expression for $g_c(x)$ we let N define a distribution over X, let W be the set of edge-weight parameters in N, $W_c \subseteq W$ be the set of edge-weights in c, and u_c^i be the leaf node in c whose univariate distribution is over X_i. Then

$$g_c(x) = \prod_{w \in W_c} w \prod_{i \in [m]} u_c^i(x_i),$$

where $[m] = \{1, \ldots, m\}$.

Let P be the set of product nodes in SPN N and let $\mathrm{ch}(n) = \{h_1, \ldots, h_r\}$ denote the set of children of node n. For each product node $p \in P$ we define a random vector $Y_p = [Y_1, \ldots, Y_r]$, where $r = |\mathrm{ch}(p)|$ and Y_i corresponds to $h_i \in \mathrm{ch}(p)$. If p is in complete sub-circuit $c \in C$, then c assigns values to each variable in Y_p as follows. If $h_i \in \mathrm{ch}(p)$ is a sum node and $h_j \in \mathrm{ch}(h_i)$ is in c, then $Y_i = j$; if $h_i \in \mathrm{ch}(p)$ is a leaf node whose univariate distribution is over X_j, then $Y_i = x_j$. We define a closely-related function w_p that maps each value of Y_p to a vector $[a_1, \ldots, a_r]$. If $h_i \in \mathrm{ch}(p)$ is a sum node and $h_j \in \mathrm{ch}(h_i)$ is in c, then a_i takes the value of the weight on the edge between h_i and h_j; if h_i is a leaf node whose univariate distribution is over X_j, then $a_i = u_c^j(x_j)$. See Figure 4.1.

Given a dataset D of i.i.d. samples from a joint distribution over X, we generate mini-datasets D_p (SEARCHSPN makes heavy use of these) for every $p \in P$; each D_p contains instances of the random vector Y_p. This is done by identifying, for each $x \in D$, the (or a) complete sub-circuit c such that $g_c(x)$ is maximal; finding such a sub-circuit can be done efficiently using a Viterbi-style algorithm on N [6, 38], where sum nodes are replaced by max nodes[3]. This circuit c assigns a value to Y_p for each product node p in c; that value is added to D_p. We think of D_p as a matrix whose rows are filled with values of Y_p and whose i^{th} column corresponds to $h_i \in \text{ch}(p)$. Let $D_P = \{D_p | p \in P\}$.

With f the function computed by N, we follow [15] in writing the likelihood $\mathcal{L}$ of dataset D as

$$\begin{aligned}\mathcal{L}(f|D) &= \prod_{x \in D} f(x) = \prod_{x \in D} \sum_{c \in C} g_c(x) \\ &= \prod_{x \in D} \sum_{c \in C} \prod_{w \in W_c} w \prod_{i \in [m]} u_c^i(x_i)\end{aligned}$$

and in defining an *approximate likelihood* $\tilde{\mathcal{L}}$, sometimes called the Viterbi likelihood, as

$$\tilde{\mathcal{L}}(f|D) = \prod_{x \in D} \max_{c \in C} \prod_{w \in W_c} w \prod_{i \in [m]} u_c^i(x_i).$$

Notice that the function $\tilde{\mathcal{L}}$ is simply a list of values multiplied together. Each instance $x \in D$ is used in finding a maximal complete sub-circuit $c \in C$, whose edge-weights and leaf-node values are multiplied into the result. Now remember that these maximal complete sub-circuits are related to the product nodes in N through the mini-datasets D_p. This fact, along with the functions w_p, can be used to rewrite $\tilde{\mathcal{L}}$ in terms of product nodes instead of complete sub-circuits:

$$\tilde{\mathcal{L}}(f|D) = \alpha \prod_{p \in P} \prod_{y_p \in D_p} \prod_{w \in w_p(y_p)} w.$$

[3]Note we are not doing MAP/MPE inference and so do not face the issues raised in [36]

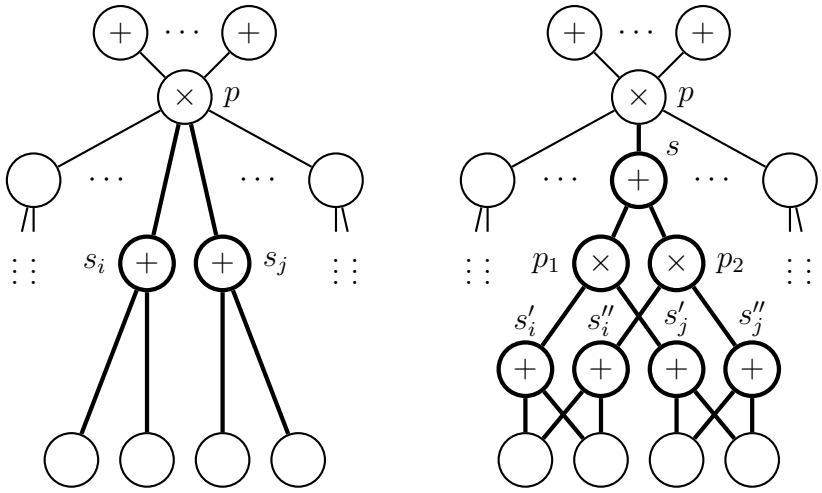


Figure 4.2: MixClones is applied to the network on the left. Given product node p and a subset of its children $A = \{s_i, s_j\}$, the bold nodes and edges at the left are replaced with the bold nodes and edges at the right.

Here α takes care of a technical detail: if the root node of N is a sum node, then its weights appear in $\tilde{\mathcal{L}}$, but are never part of the vector returned by any w_p; α accounts for these missing values and is one if the root node is not a sum node.

With this new formulation, $\tilde{\mathcal{L}}$ is still the product of a long list of values, but now the values are naturally grouped by product node. The expression $\prod_{y_p \in D_p} \prod_{w \in w_p(y_p)} w$ is taken as the contribution of p to $\tilde{\mathcal{L}}$, and a low value indicates that modifying the structure of the SPN at p is a promising way to increase $\tilde{\mathcal{L}}$. It is then hoped that increasing $\tilde{\mathcal{L}}$ will also increase $\mathcal{L}$. SearchSPN uses these ideas to guide its modifications of the input SPN. It selects the product node

$$p = \operatorname*{argmin}_q \prod_{y_q \in D_q} \prod_{w \in w_q(y_q)} w$$

and modifies the structure near this node; this process is repeated until a stopping criteria is met.

4.3 SPN Structure Search

This section describes the SearchSPN and OnlineSearchSPN algorithms. To make it suitable as a sub-routine in OnlineSearchSPN, our SearchSPN is modified from the

Algorithm 6 MixClones(p, A)

Input: product node p, $A \subseteq \text{ch}(p)$
remove the edge between p and each node in A
$s \leftarrow$ new sum node
set p as the parent of s
for all $i \in \{1, 2\}$ **do**
    $p_i \leftarrow$ new product node
    set s as the parent of p_i
    $A' \leftarrow$ clone of A
    set p_i as the parent of each node in A'
end for
return s

original; our description will be brief with a more in-depth discussion left to the original paper. We also simplify MixClones, an algorithm used by SearchSPN.

4.3.1 Offline Structure Search

The mechanics of the MixClones algorithm are outlined in Algorithm 6 and demonstrated in Figure 4.2. Given a product node p and a set $A \subseteq \text{ch}(p)$ as input, it increases the modeling power of the SPN by locally adding nodes around p. The algorithm makes exactly two clones of the nodes in A instead of an arbitrary number, as is done in the original algorithm which uses several runs of K-means in an attempt to find the best number. Additional arguments can be made to justify this choice; we refer the reader to several made by Vergari et al. [55], who justify a similar change to LearnSPN.

Algorithm 8 (SearchSPN) is given as input an SPN N and datasets D_P. The original algorithm takes D as input; changing to D_P makes it easier to use SearchSPN as a sub-routine in OnlineSearchSPN. Nodes are added to N by repeated calls to MixClones. The main tasks of SearchSPN are: intelligently select p, the next product node to expand; partition the children of p; expand N at p using the partition of $\text{ch}(p)$; update the parameters of newly-added sum and leaf nodes; and update the datasets in D_P.

[3] If h_l is a sum node, the counts of the values in D_l are computed and normalized to produce a categorical distribution. If h_l is a univariate distribution, its sufficient statistics are computed (e.g., the mean and variance if h_l were a Gaussian node).

Algorithm 7 UPDATE(s, A, D_p)

$D_A \leftarrow$ the columns in D_p that correspond to A
$\{p_1, p_2\} \leftarrow \text{ch}(s)$
$D_{p_1}, D_{p_2} \leftarrow$ divide the rows of D_A into two clusters
$a \leftarrow$ the cluster assignment vector
replace the D_A columns in D_p with a
edge weights of $s \leftarrow (|D_{p_1}|/|D_A|, |D_{p_2}|/|D_A|)$
for all $h_l \in \text{ch}(p_i), i \in \{1, 2\}$ **do**
    $D_l \leftarrow$ the l^{th} column of D_{p_i}
    parameters of $h_l \leftarrow \text{STATS}(D_l)$[3]
end for
return D_{p_1} and D_{p_2}

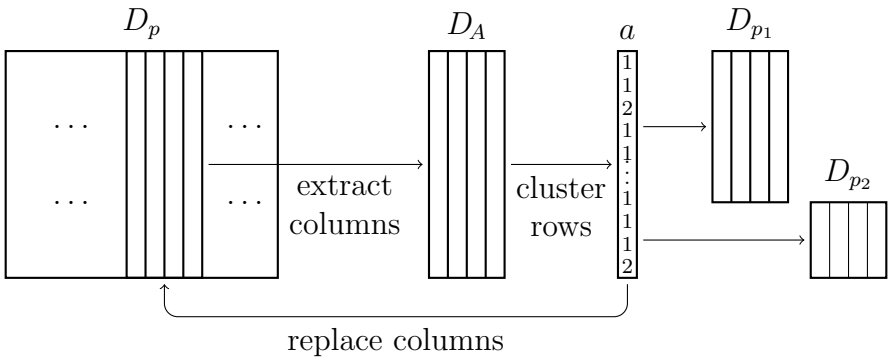


Figure 4.3: Algorithm 7 uses D_p to update the parameters of nodes affected when MIXCLONES is called with p as an argument. It also changes D_p and creates datasets D_{p_1} and D_{p_2} for p_1 and p_2, the newly-created grandchildren of p.

The preceding tasks are repeated until all product nodes in N have been added to blacklist L. The stopping criteria in the original algorithm uses the performance of N on a validation set; using a blacklist makes the algorithm simpler and more efficient. The blacklist holds product nodes that should not be expanded by MIXCLONES. After SEARCHSPN selects a product node p, it adds p to the blacklist if a) there are too few instances in D_p, or b) all variables in Y_p are (approximately) independent. In the former case we have too little data to justify making a structural change to N. In the latter case we do not expect that running MIXCLONES will significantly improve the model.

Selecting the next product node p, and partitioning its children, are both done in the same way as the original SEARCHSPN. It selects $p = \text{argmin}_q \prod_{y_q \in D_q} \prod_{w \in w_q(y_q)} w$, which is

Algorithm 8 SEARCHSPN(N, D_P)

Input: SPN N, set of product-node training datasets D_P
$L \leftarrow \emptyset$
while $L \neq P$ **do**
  $p \leftarrow \text{argmin}_{q \in P \setminus L} \prod_{y_q \in D_q} \prod_{w \in w_q(y_q)} w$
  partition ch(p) into independent subsets $\{A_i | i \in [k]\}$
  if $k = |\text{ch}(p)|$ or $|D_p|$ is too small **then**
    $L \leftarrow L \cup \{p\}$
    continue
  end if
  for all $i \in [k]$ s.t. $|A_i| > 1$ **do**
    $s \leftarrow$ MIXCLONES(p, A_i)
    $D_{p_1}, D_{p_2} \leftarrow$ UPDATE(s, A_i, D_p)
    $D_P \leftarrow D_P \cup \{D_{p_1}, D_{p_2}\}$
  end for
end while

the product node most blamed for a low approximate likelihood score $\tilde{\mathcal{L}}$. Then children of p are partitioned by looking for subsets of variables in Y_p that are statistically independent; subsets are found by looking for connected components in a graph whose edges indicate pairwise independence between variables [15, 19].

The parameters of N are updated using D_p. Various slices of D_p are assigned to the newly-created children and grandchildren of p. The data in the slices are then used to compute the parameters. Algorithm 7 and Figure 4.3 describe and illustrate this process. D_p itself is updated to reflect the changes made by MIXCLONES and a new dataset is created for each new grandchild of p.

4.3.2 Online Structure Search

Algorithm 9 (ONLINESEARCHSPN) takes in a seed SPN N and a stream of training batches. In our experiments we use a simple seed model consisting of a single product node with m children, each a univariate distribution over one of the variables in X. After each batch arrives, it is used to update the product node datasets D_p. This is done by finding the (or a)

Algorithm 9 OnlineSearchSPN(N, B)

Input: SPN N, training-instance batch-stream B
create an empty D_p for each $p \in P$
while more batches are available **do**
  $D \leftarrow$ next batch from B
  $E \leftarrow \left\{ \text{argmax}_{c \in C}\, g_c(x) | x \in D \right\}$
  for all (c, p) s.t. $c \in E$ and $p \in c$ **do**
    add[4] the value of Y_p to D_p
  end for
  for all (p, h_l) s.t. $p \in P$ and $h_l \in \text{ch}(p)$ **do**
    $D'_l \leftarrow$ most recent[5] r values in l^{th} column of D_p
    parameters of $h_l \leftarrow$ stats(D'_l)
  end for
  $D_P \leftarrow \{D_p | p \in P\}$
  SearchSPN(N, D_P)
end while

maximal complete subcircuit for each instance in the current batch. The complete subcircuits are used to assign values to the random vectors Y_p, and each instance of Y_p is added to D_p.

The most recent values added to the product node datasets are then used to update the parameters of N. This step allows N to adapt to non-stationary distributions. If, for example, the stream of batches consists of website traffic, then updating the parameters of N allows the SPN to model current trends in this traffic and helps it forget some of the older patterns it learned.

Finally, the structure of N is modified with a call to SearchSPN, the main workhorse of OnlineSearchSPN. Modifying the structure of N increases the modeling power of N, letting it grow more complex and better model complex input distributions. The whole process is repeated as long as more batches are available.

As mentioned, OnlineSearchSPN updates the parameters of the SPN as new examples are fed in, letting the SPN model the current input distribution and forget some of what it learned previously. There is less flexibility when it comes to the structure of the SPN. Once a particular set of nodes has been added to the SPN, there is no mechanism for removing them. This problem can partly be solved by adjusting the SPN parameters. For

instance, we can virtually remove nodes by setting to zero the weight between a sum node and one of its children; this removes the effects on the sum node of the sub-DAG rooted at the child, but still leaves the SPN with nodes that serve no beneficial function. A better solution would implement some kind of pruning step that actually removes nodes; we leave that to future work.

4.4 Experiments

We experiment with the online learning scenario in which data instances arrive in a stream of mini-batches. After each mini-batch is seen, the learning algorithm produces a model. To evaluate the algorithm we measure the time taken to produce the model and compute a likelihood score for the model. The next mini-batch is used in computing the likelihood score so that we measure generalization and not fitness to the training instances.

We study the behavior of three online structure-learning algorithms. The first two, which we identify as RECENT and ALL, simply use an offline algorithm in an online fashion. RECENT takes the current mini-batch and returns the result of running the offline algorithm on that mini-batch. ALL combines all mini-batches seen so far into a single batch and runs the offline algorithm on it. The third algorithm is ONLINESEARCHSPN; as noted, it adapts SEARCHSPN to the online setting. Unlike the first two algorithms, the approach used in ONLINESEARCHSPN would not work for adapting most other offline structure-learning algorithms. SEARCHSPN is particularly suited to adaptation because it augments an input SPN; this allows its output to be used as the input to a subsequent call, and can thus be used to produce a sequence of SPNs. It is less obvious how BUILDSPN or LEARNSPN, for example, could be adapted except using an approach similar to RECENT or ALL.

We use SEARCHSPN as the offline algorithm in RECENT and ALL. As mentioned, any offline algorithm would do, but using SEARCHSPN makes RECENT and ALL more similar to

[4]To deal with very large datasets (e.g., datasets that do not fit in memory), use reservoir sampling to add instances of Y_p to D_p.

[5]We set r to 1,000.

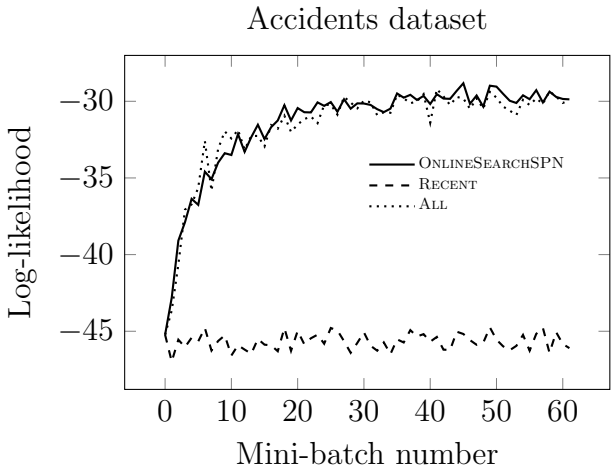


Figure 4.4: The test-likelihood plots during training on the Accidents dataset for the Recent, All, and OnlineSearchSPN algorithms.

OnlineSearchSPN, which must use it, and makes timing comparisons more meaningful since the three algorithms share code. Our experiments show that Recent and All have drawbacks compared to OnlineSearchSPN; we note that a different offline algorithm would not solve these problems.

The first set of experiments we run draws dataset mini-batches from a stationary distribution. This is not a particularly compelling use-case for an online algorithm since there is no need to adapt to changes in the input distribution over time. However, it does let us focus on the quality of the produced models and the time required to produce them, without needing to wonder how much these metrics are being affected by a changing input distribution. So we separate this concern and leave it to the second set of experiments, which looks explicitly at how the algorithms adapt to a change in input distribution. We do this in a controlled manner by sampling from one stationary distribution and then switching to a different stationary distribution. The third set of experiments evaluates the algorithms as they train on a sequential dataset whose distribution is non-stationary.

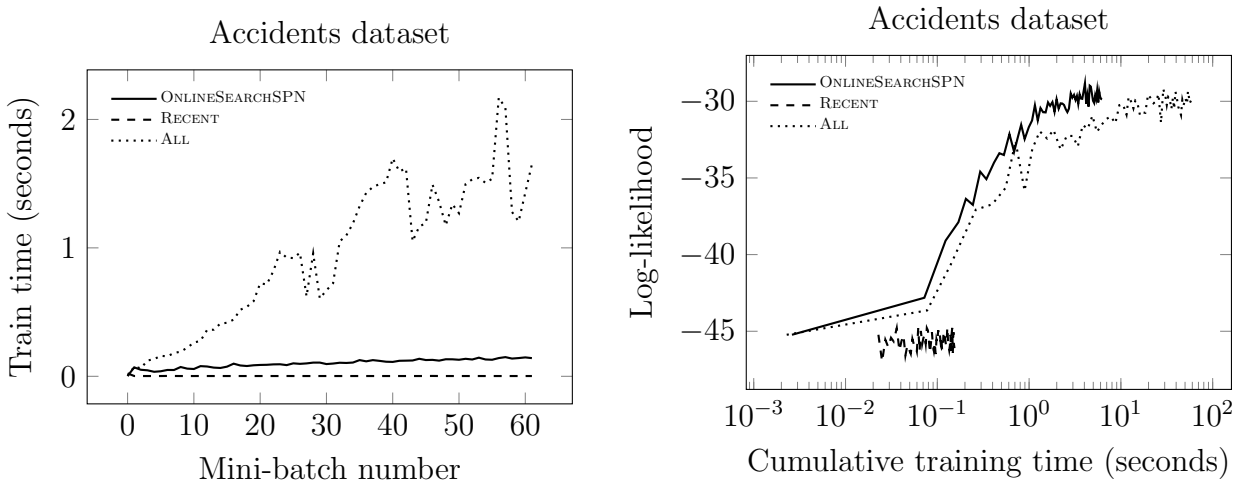


Figure 4.5: Both plots show data collected while training on the Accidents dataset. The plot on the left shows the time taken for training on each mini-batch. The plot at the right shows the likelihoods from Figure 4.4 as a function of cumulative training time, on a log scale. All requires roughly 60 seconds to train, OnlineSearchSPN about 6 seconds, and Recent less than 1 second. OnlineSearchSPN achieves the likelihood of All using a training time closer to that taken by Recent.

4.4.1 Stationary-Distribution Datasets

In our stationary-distribution experiments we use the twenty datasets from Van Haaren [54], which have been used in many other SPN structure-learning papers, starting with Gens & Domingos [19]. Table 4.1 in Appendix 4.A lists basic statistics for these datasets. All have between 16 and $1,556$ binary variables, with between about 2k and 300k training instances. We divide these into batches of size 200 and feed them one at a time into Recent, All, and OnlineSearchSPN. In Figure 4.4 we show results from the Accidents dataset. Figure 4.8 in Appendix 4.A contains plots for all 20 datasets, most of which are qualitatively similar: in terms of likelihood, Recent under-performs and the other two algorithms perform comparably.

The speed of learning for the three algorithms varies quite a bit, as illustrated in the plots in Figure 4.5. The upper plot shows the time taken for training on each mini-batch. All requires more time than the other algorithms since it trains on all previously-seen mini-batches, not just the current mini-batch. Also, its trend line has a larger slope, indicating

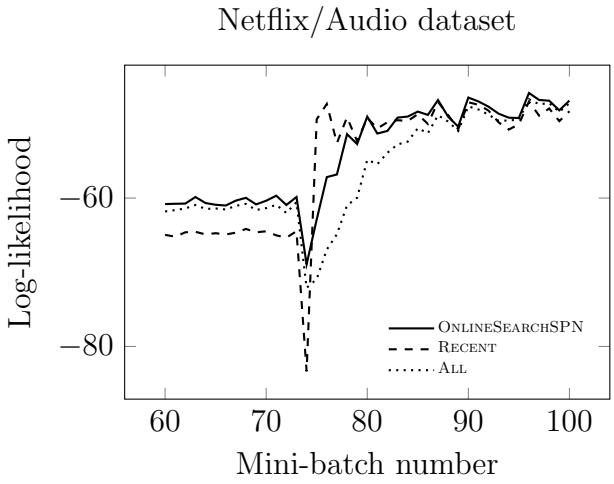


Figure 4.6: Test-likelihood plots during training on the Netflix/Audio dataset. To highlight how well the algorithms adapt to a changing input distribution, the plot is focused on the transition from Netflix mini-batches to Audio mini-batches.

that it does scale as well. The lower plot shows the same data from Figure 4.4 but plotted as a function of cumulative training time instead of number of mini-batches seen. Note the log scale on the time axis. Here we see that Recent quickly learns poor models and All learns good models, slowly. OnlineSearchSPN provides a nice alternative that combines the advantages of both, learning models that are as good as those produced by All, but taking much less time.

4.4.2 Abrupt-Change Dataset

The next experiment combines the Netflix and Audio datasets (two of the twenty), which is easy to do since both datasets have 100 binary variables. The Netflix instances are ordered before the Audio instances so that the mini-batches given to the learning algorithms abruptly shift their distribution midway through training. Figure 4.6 shows the test likelihood of the algorithms during the shift. The algorithms adjust to the change, as is evidenced by the obvious move in likelihood values from around -60 to about -50.

All three algorithms scramble to adjust to the new distribution (Audio), but Recent clearly adjusts more quickly than the others. In addition to short training times, then,

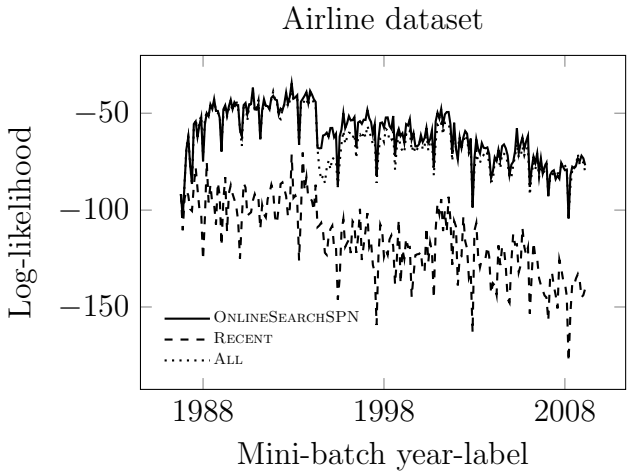


Figure 4.7: Test-likelihood plots during training on the Airline dataset for Recent, All, and OnlineSearchSPN. The algorithms adapt to seasonal- and longer-term-changes in the training data.

Recent is able to adapt quickly to changes in the input distribution. This is because it only ever looks at a single mini-batch; after seeing one mini-batch of new data, it produces a model that is about as good as any it will produce thereafter. These models have lower likelihood, of course, which remains the disadvantage of Recent.

Both All and OnlineSearchSPN adapt less quickly, but once they do, both eventually produce models with higher likelihood than Recent. This is not seen in Figure 4.6 because, for clarity, it zooms in on the period of training in which the algorithms are adjusting; but if the plot were extended to the right, a gap would appear between the likelihood of Recent and the other two algorithms, similar to the gap seen at the left of the plot during the period in which the algorithms had only ever seen Netflix training instances.

4.4.3 Non-Stationary Airline Dataset

In the last experiment we took a subset of the data from the 2009 ASA Data Expo [56]. This dataset is 120 million rows by 29 columns and contains information about the arrival and departure of commercial flights in the US from 1987 to 2008. We pared this data down, looking at 16 columns and subsets of size 40 of the flights departing each Monday

at 10am. The columns measured in minutes or miles were removed, resulting in a dataset of only discrete variables; keeping only the Monday-at-10am rows was an effort to remove high-frequency variation in the data while keeping the seasonal and year-to-year changes in the input distribution. The results are shown in Figure 4.7. The likelihood plots take a noticeable dip about once each year as the models adjust, it appears, to seasonal changes in air travel patterns. The likelihoods also adjust from decade to decade as the models appear to adapt to longer-term trends in the data. Instead of converging on a particular value as in the stationary experiment with the Accidents dataset, the likelihoods change as the learning algorithms encounter shifts in the input distribution and adapt accordingly.

The ALL and ONLINESEARCHSPN algorithms, as in the other experiments, clearly model the data with higher likelihood than RECENT. Unlike in the Netflix/Audio experiment, RECENT does not clearly have the advantage in terms of adapting more quickly to changes in the data. Apparently the distribution shifts in the Airline dataset are not as large or as abrupt.

4.5 Conclusion

Various batch-learning algorithms have been proposed to learn the structure and parameters of SPNs. We take one of these, SEARCHSPN, and use it to develop an online SPN structure learner called ONLINESEARCHSPN. The advantage of this algorithm is its ability to adapt to a changing input distribution and to handle larger datasets than the batch learners.

Our experiments compare ONLINESEARCHSPN with two methods, RECENT and ALL, that adapt SPN batch-learners to the online setting. Results show that ONLINESEARCHSPN learns models that are as good as those learned by ALL, and does so almost as quickly as RECENT.

4.A Stationary-Distribution Experiments and Statistics

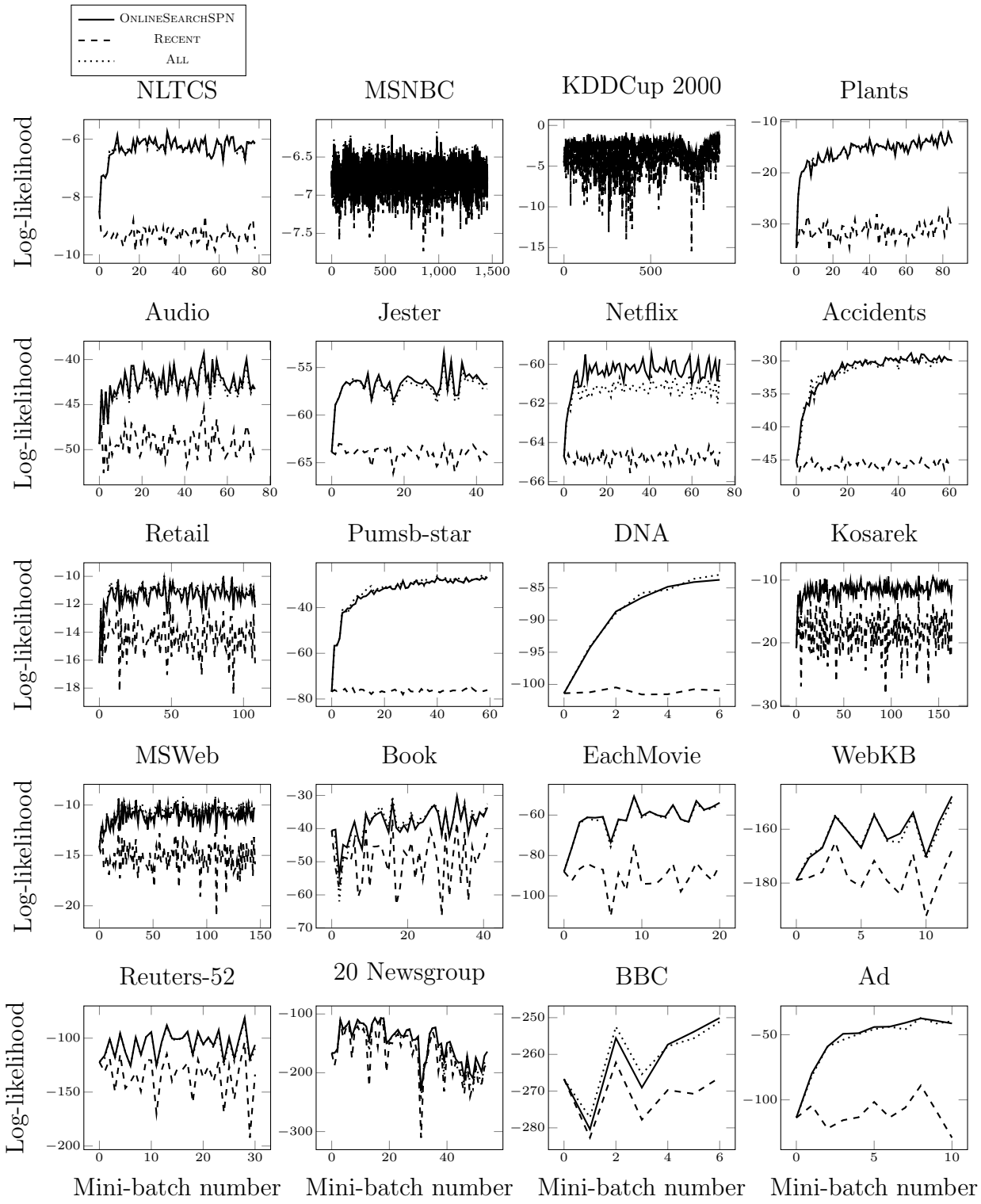


Figure 4.8: The test-likelihood plots during training on the 20 datasets for the Recent, All, and OnlineSearchSPN algorithms.

Dataset	# Vars	Train	Valid	Test
NLTCS	16	16181	2157	3236
MSNBC	17	291326	38843	58265
KDDCup 2000	64	180092	19907	34955
Plants	69	17412	2321	3482
Audio	100	15000	2000	3000
Jester	100	9000	1000	4116
Netflix	100	15000	2000	3000
Accidents	111	12758	1700	2551
Retail	135	22041	2938	4408
Pumsb-star	163	12262	1635	2452
DNA	180	1600	400	1186
Kosarek	190	33375	4450	6675
MSWeb	294	29441	32750	5000
Book	500	8700	1159	1739
EachMovie	500	4524	1002	591
WebKB	839	2803	558	838
Reuters-52	889	6532	1028	1540
20 Newsgroup	910	11293	3764	3764
BBC	1058	1670	225	330
Ad	1556	2461	327	491

Table 4.1: Basic statistics for the Van Haaren 20 datasets.

Chapter 5

Autoencoder-Enhanced Sum-Product Networks

Abstract

Sum-product networks (SPNs) are probabilistic models with many nice properties, chief among them their guarantee that exact inference can be done in time linear in the size of the network. We use autoencoders in concert with SPNs to model high-dimensional, high-arity random vectors (e.g., image data). Experiments show that our proposed model, the autoencoder-SPN (AESPN), which combines two SPNs and an autoencoder, produces better samples than an SPN alone. This is true whether we sample all variables, or whether a set of unknown query variables is sampled, given a set of known evidence variables.

5.1 Introduction

Sum-product networks (SPNs) are probabilistic models that guarantee exact inference can be done in time linear in the size of the network [38]. Since their introduction many algorithms have been introduced to automatically learn the structure and parameters of an SPN from data, and these algorithms have fared well on many datasets compared to other models that allow fast, exact inference [14, 15, 19, 35, 44, 55].

The paper introducing SPNs [38] as well as two follow-on structure-learning papers [14, 35] all experiment with the Olivetti face-image dataset. They show impressive results, especially on the task of image reconstruction where they fill in half of the image given the values of the pixels in the other half of the image. Poon and Domingos [38] also experimented with the Caltech-101 dataset, doing the same kinds of image reconstructions. Running the Caltech-101 experiments using the code they make available[1] results in much less impressive image reconstructions than those from the Olivetti dataset. This is not surprising as the Olivetti dataset contains centered faces looking directly at the camera; it contains a great deal of structure and symmetry that the Caltech-101 dataset lacks.

Another interesting aspect of these papers [14, 35, 38] is that they all model pixels using a mixture of four Gaussians instead of using a categorical variable taking the integer values in the range $[0, 255]$. It is not unusual that the pixels are modeled as continuous variables, but using a mixture with exactly four components in some sense reduces (fuzzily) the arity of the pixel variables from 256 to 4. We speculate that SPNs may struggle modeling high-dimensional, high-arity random vectors. In this paper we train an SPN on MNIST, using categoricals to model the pixels, and the results seem consistent with this speculation in that the samples from the SPN are fuzzy-looking (see Figure 5.2). We leave the investigation of our speculation for future work, but are inspired by it to propose the use of autoencoders to aid SPNs in modeling image data. We use one SPN to model the input variables and a

[1] http://spn.cs.washington.edu/spn/downloadspn.php

second SPN to model the hidden representation of the autoencoder. Our experiments show that both additions help the SPN produce better-looking samples.

More than a decade ago Bengio and Bengio [4] proposed adapting an autoencoder to become a density estimator by computing an "autoregressive" function. Essentially they adapt the architecture of the autoencoder so that the i^{th} output node computes a function of the first $i-1$ input nodes. Using a sigmoid activation function on the output nodes lets them treat the output values as probabilities $P(x_i|x_1,\ldots,x_{i-1})$; thus multiplying the output nodes gives a density estimate for the input.

In recent years this thread of research has been taken up again in several studies [20, 21, 26, 50, 51]. All of these newly-proposed models compute joint probabilities $P(x)$, but vary with respect to other model properties. Some can draw independent samples from $P(X)$ and some only draw correlated samples. Some can compute marginal probabilities, some can sample from the marginal probability distributions in order to, say, do image reconstruction. The time complexity for these tasks also vary. Our approach is less a descendant of this work on autoregressive neural networks, and more a cousin. We incorporate an autoencoder so that it is an important part of a probabilistic model. However, instead of directly adapting autoencoders to become density estimators, we use SPNs for density estimates and use an autoencoder to aid the SPNs in producing samples from the joint distribution and samples from various marginal distributions.

5.2 Sum-Product Networks and Autoencoders

A sum-product network (SPN) is a network of sum and product nodes that represent a joint probability distribution P over some set of variables X. Mixture models and product-of-marginals are used to recursively combine distributions over subsets of X in the following manner. A mixture model i expresses a distribution over $A \subseteq X$ as

$$P_i(A) = \sum_j w_j P_j(A),$$

where the w_j are mixing coefficients and the P_j are distributions over A. A product-of-marginals i expresses a distribution over $A \subseteq X$ as

$$P_i(A) = \prod_{B \in S} P_j(B),$$

where S is a partition [2] of A and the P_j are distributions over the blocks in the partition. The base case in the recursion occurs when $|A| = 1$ or $|B| = 1$; in this case the distributions P_j are univariate distributions over the single variable in A or B. Sum nodes in an SPN compute mixtures, product nodes compute product-of-marginals, and leaf nodes are univariate distributions. Several algorithms construct SPNs using a top-down strategy that starts with $A = X$ and recursively constructs mixture models and product-of-marginal models; some [14, 19] continue until $|A| = 1$, at which point a univarate leaf node is inserted, while others stop short before $|A| = 1$ and insert a multivariate distribution such as a Markov Network [44] or Chow-Liu tree [55] instead. See [19, 38] for more precise definitions of SPNs.

SPNs factor distributions differently than Bayesian networks. Instead of multiplying conditional distributions to form the joint, SPNs multiply marginal distributions to form the joint. This change gives an SPN the advantage of having an exact inference algorithm that is efficient. Any marginal probability can be computed in time linear in the size of the network. However, this also means that dependencies between variables cannot be modeled using conditional distributions. The marginal factorizations in an SPN implicitly assume statistically independent variables. Therefore, all dependency modeling in an SPN is done using its mixture models.

For many problems SPNs model the distribution quite well. For others, variable dependencies may be easier to learn or more compact to represent using a method other than mixture models. This motivates our use of autoencoders to augment SPNs. While

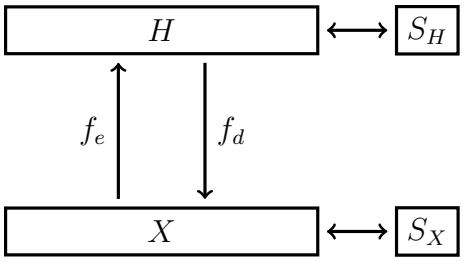
[2] $\bigcup_{B \in S} B = A$, $\emptyset \notin S$, and $B \cap C = \emptyset$ for any two distinct sets $B, C \in S$

autoencoders do not explicitly model variable dependencies, they are able to capture important characteristics of a target distribution and encode these as hidden variables.

An autoencoder can be thought of as a pair of functions (f_e, f_d) such that f_e encodes inputs into an intermediate representation and f_d decodes these representations back into the original input space. The learning objective is to fix these functions such the $f_d(f_e(x))$ is very similar to x for all instances x in the training set. To avoid simply computing the identity function, regularization terms and other constraints are usually placed on the functions. We take the common approach of using feed-forward neural networks to compute the encoder and decoder functions.

Let P^* be the distribution from which our training set is drawn and let P be the distribution represented by an SPN. We use autoencoders to improve the quality of a sample, x, drawn from an SPN. With x in hand we compute $y = f_d(f_e(x))$. Because the autoencoder is trained on samples from P^* and because it cannot compute the identity function, we argue that an autoencoder tends to map less-likely inputs (with respect to P^*) to more-likely inputs ; in other words, we expect $P^*(y) > P^*(x)$.

This gets to the heart of our paper. The situation in which we imagine ourselves is that the distribution P represented by an SPN does not quite capture important aspects of P^*; there are regions of the input space where P assigns higher probability than P^* and vice versa. We use an autoencoder in hopes of compensating for these poorly-modeled regions. For example, given a sample x from the SPN such that $P(x) > P^*(x)$, we modify the sample by computing $y = f_d(f_e(x))$. The idea is to shift probability mass away from x, which has low probability in P^*, to y, which has higher probability in P^*. Note that if the autoencoder is always used, then *all* the probability mass is shifted from x to y. So, in effect, we are really sampling from a distribution specified by the autoencoder. But we cannot simply use the autoencoder alone, since without the SPN there is no way to sample from it.



(a)

P_X	$S_X \to X$
P_2	$S_X \to f_e \to f_d \to X$
P_3	$S_X \to f_e \to S_H \to f_d \to X$

(b)

Figure 5.1: In the AESPN model, autoencoder functions encode visible variables X into hidden variables H and decode H into X. Two SPNs, S_X and S_H, model the variables X and H, respectively. This is shown in 5.1a. To sample from an AESPN we can use three different strategies, as indicated in 5.1b. We label the distributions associated with these strategies as P_X, P_2, and P_3. Sampling P_X is done using S_X, sampling P_2 involves encoding then decoding a sample from S_X, and sampling P_3 involves using S_H to help infer a value for H before it is decoded.

5.3 The AESPN Model

We call our model the autoencoder sum-product network (AESPN). An AESPN A is a tuple $A = (S_X, f_e, f_d, S_H)$ consisting of two SPNs S_X and S_H, an encoder function f_e, and a decoder function f_d. Figure 5.1 shows how the autoencoder functions and SPNs interact. S_X models the visible variables X and S_H models the hidden representation H computed by the autoencoder. We call the distributions represented by S_X and S_H, respectively, P_X and P_H. The function f_e maps instances of X to H and f_d maps instances of H to X. We use a neural network to implement the autoencoder functions and designate one of its layers as representing H. This layer uses sigmoid activations functions and thus the variables in H are binary variables taking the value zero or one. For instance $X = x$ we compute $H = h$ by rounding the vector $f_e(x)$.

5.3.1 Sampling in AESPNs

We evaluate an AESPN model by generating full images from it and by performing inpainting. The first task is accomplished by drawing samples from a joint distribution over all the variables X. The second task is accomplished by drawing samples from a conditional

Algorithm 10 SAMPLE1(S, Y_q, y_e)

Input: SPN S, query variables Y_q, and evidence y_e
Set variables in Y_e according to y_e
Set variables in Y_q to unknown
Evalute S in a forward pass, caching node values v_i
$Q \leftarrow \{\text{root of } S\}$
while $|Q| > 0$ **do**
  $n \leftarrow$ pop node from Q
  if n is a sum node **then**
    $w_{nj} \leftarrow$ weight between n and j^{th} child
    select j^{th} child randomly with $p(j) \propto w_{nj} v_j$
    $Q \leftarrow Q \cup \{\text{child } j\}$
  end if
  if n is a product node **then**
    $C_n \leftarrow$ children of n
    $Q \leftarrow Q \cup C_n$
  end if
  if n is a univariate distribution node over $Y_i \in Y_q$ **then**
    $Y_i \leftarrow$ sample the univariate distribution
  end if
end while

distribution over query variables X_q given evidence variables X_e, where $X = X_q \cup X_e$ and $X_q \cap X_e = \emptyset$. The observed pixels in the in-painting task are associated with the variables in X_e, and the areas in which pixel values should be guessed are associated with the variables in X_q. Also note that sampling from the joint distribution is the same as sampling from the conditional distribution in which $X_q = \emptyset$. The experiments in this paper compare three methods for using an AESPN to draw samples of X_q given X_e.

The first sampling method draws samples from $P_X(X_q|x_e)$ in the usual manner using S_X. This is shown in Algorithm 10. In short, a backward pass through S, starting at the root node, is performed. At each sum node a child is randomly selected and the backward pass proceeds to that child. At each product node every child is selected in turn and the backward pass proceeds to that child. Thus a tree is traced out in the SPN S until it ends at univariate distribution leaf nodes. These distributions are sampled to fill in values for the variables in X_q.

Algorithm 11 SAMPLE2(S_X, f_e, f_d, S_H, X_q, x_e)

Input: SPNs S_X, S_H, autoencoder functions f_e, f_d, query variables X_q, and evidence x_e
$x_q \leftarrow$ sample from $P_X(X_q|x_e)$
$x \leftarrow x_q \cup x_e$
$h \leftarrow f_e(x)$
$y \leftarrow f_d(h)$
$y_q \leftarrow$ values in y for the variables in X_q
return y_q

The second method, shown in Algorithm 11, draws a sample x_q from $P_X(X_q|x_e)$ using Algorithm 10. It then sets H to the value $f_e(x)$, where $x = x_q \cup x_e$. Next, it decodes $H = h$ to produce the sample x_q. Given $H = h$ it computes $y = f_d(h)$ and then assigns the variables in X_q to have values consistent with y. In this method the autoencoder can be thought of as "cleaning" the sample drawn from S_X.

The third method similary decodes a value $H = h$ to produce a sample of the variables in X_q, but it generates the value of H differently. As shown in Algorithm 12, it computes D_H, a set of m assignments[3] to H, by doing the same thing as Algorithm 11: sampling X_q and then computing $f_e(x)$. These values are then used to partition H into evidence variables H_e and query variables H_q. If $H_i \in H$ always takes value h_i in D_H then $H_i = h_i$ is placed in H_e; otherwise H_i is placed in H_q. Then, using S_H and Algorithm 10, a sample h_q is drawn from $P_H(H_q|h_e)$ and the hidden variable is set to $H = h_q \cup h_e$.

To distinguish between these strategies we label the distributions from which the second and third methods draw their samples $P_2(X_q|X_e)$ and $P_3(X_q|X_e)$, respectively. Also, in the special case that $X_e = \emptyset$ we do not compute D_H when sampling from P_3. Instead, we directly decode a sample drawn from P_H since, if no variables in X are known, we assume that no variables in H will be known either.

[3]We use $m = 100$.

Algorithm 12 SAMPLE3(S_X, f_e, f_d, S_H, X_q, x_e)

Input: SPNs S_X, S_H, autoencoder functions f_e, f_d, query variables X_q, and evidence x_e
$D_H \leftarrow \emptyset$
for all $i \in \{1, \ldots, m\}$ **do**
  $x_q \leftarrow$ sample from $P_X(X_q|x_e)$
  $x \leftarrow x_q \cup x_e$
  $D_H \leftarrow D_H \cup \{f_e(x)\}$
end for
$H_e = \{H_i | \forall H_i^{(j)}, H_i^{(k)} \in D_H, H_i^{(j)} = H_i^{(k)}\}$
$H_q = \{H_i | \exists H_i^{(j)}, H_i^{(k)} \in D_H, H_i^{(j)} \neq H_i^{(k)}\}$
$h_q \leftarrow$ sample from $P_H(H_q|h_e)$
$h \leftarrow h_q \cup h_e$
$y \leftarrow f_d(h)$
$y_q \leftarrow$ values in y for the variables in X_q
return y_q

5.3.2 Distributions

We can write $P_2(x_q|x_e)$ in terms of P_X. The first step in Algorithm 11 is to draw a sample x'_q from $P_X(X_q|x_e)$, which is then used to produce another value y_q for X_q. Therefore, letting S be the set of samples x'_q that lead to value $y_q = x_q$, we have $P_2(x_q|x_e) = \sum_{x'_q \in S} P_X(x'_q|x_e)$. Note that there may be values x_q for which S is the empty set, and thus $P_2(x_q|x_e) = 0$, even when $P_X(x_q|x_e) \neq 0$; in fact, in practice this will often be the case, especially when the number of values that H can take is much less than the number of values X can take.

Analyzing $P_3(x_q|x_e)$ is more difficult, except in the special case where $X_e = \emptyset$. We can write $P_3(x)$ by considering all possible values h such that $x = f_d(h)$. If T is this set of values then $P_3(x) = \sum_{h \in T} P_H(h)$. As with P_2, $P_3(x)$ may be zero for many values of x.

5.3.3 Learning

In our experiments we train the AESPN autoencoder on a dataset T using the Adam [24] optimization algorithm, a fixed number of epochs, and a fixed mini-batch size. The output layer and the middle layer (the one representing the hidden variables H) use sigmoid activation functions. The other layers use ReLU activations. For our experiments the exact choice of

Figure 5.2: The samples at the left are drawn from P_X. Those in the middle are drawn from P_2, and those at the right are drawn from P_3. "Cleaning" the samples from P_X using an autoencoder improves the samples, but decoding samples drawn from an SPN modeling the hidden representation of the autoencoder produces even better results.

autoencoder architecture and how it is trained is not so important as long as the resulting autoencoder does its job reasonably well. We train the SPN S_X on a dataset T and train SPN S_H on dataset T', where $T' = \{f_e(x)|x \in T\}$ is the dataset T encoded by the autoencoder. To train the SPNs S_X and S_H we use the LearnSPN algorithm [19]. LearnSPN is a template algorithm: parts of the algorithm can be swapped out using various methods. In comparison to the original implementation of LearnSPN, we use K-means with $K = 2$ instead of incremental expectation maximization and we test variable dependence using a threshold on the correlation coefficient instead of using a G-test score. These changes speed up the learning process; and although we do not take advantage of this, swapping the G-test score for the correlation coefficient also makes it possible to use LearnSPN on real-valued variables.

5.4 Experiments

We compare the three methods of sampling from an AESPN. First we compare the samples generated when $X_e = \emptyset$. Then we compare the sampling methods in the task of image reconstruction, or in-painting, where $X_e \neq \emptyset$. Our experiments show that sampling from P_3 is better than sampling from P_2, and both are better than sampling from P_X. The autoencoder functions and S_H both seem to help produce a better model than the SPN alone.

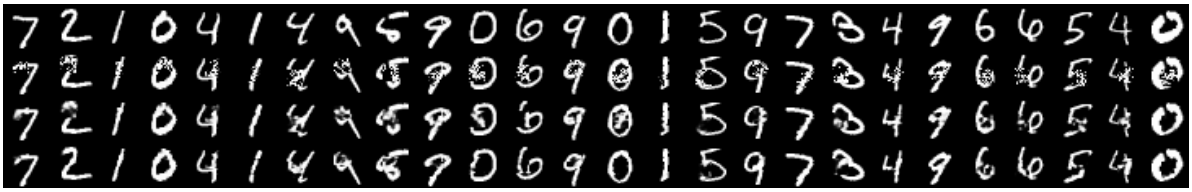


Figure 5.3: The top row is the first 26 examples in the test set. A 10×10 patch of pixels was covered and samples were drawn from conditional distributions to fill in the patch. The patches in the second row were filled using $P_X(X_q|X_e)$. In-painting in the third row used $P_2(X_q|X_e)$, and in-painting in the fourth row used $P_3(X_q|X_e)$. Using the autoencoder improves on using S_X only, and using both the autoencoder as well as S_H produces even better results.

The quality of a probabilistic model is typically measured using its likelihood given a test set. However, measuring the likelihood of P_2 or P_3 is problematic since we do not have an efficient way of computing $P_2(x)$ or $P_3(x)$. Also, the likelihoods could very well be zero since P_2 and P_3 typically assign zero probability to many instances of X. We argue that measuring likelihood is not altogether appropriate here anyway, since the autoencoder and SPN S_H in an AESPN are not meant to be a probabilistic models in the traditional sense. Instead they serve mainly as methods to improve the samples of the SPN S_X.

Therefore we adopt a different measure for analyzing the samples drawn from P_X, P_2, and P_3. In the case that $X_e = \emptyset$ we attempt to gauge how similar the samples are to the sampes in the test set. To do this we simply find the nearest test set neighbor for each sample, compute the Euclidean distance between the two, and average across all samples. While this measure is not ideal, it does give us a quantitative comparison of P_X, P_2, and P_3. We also compare these sampling strategies qualitatively by visual inspection.

Figure 5.2 shows samples generated from an AESPN trained on MNIST. Given 1000 samples and using the test set, the Euclidean measures for the samples from S_X, P_2, and P_3 are, respectively, 1702, 1230, and 1212. The P_3 samples at the right are both qualitatively and quantitatively better than those from P_2, which are qualitatively and quantitatively better than those from S_X.

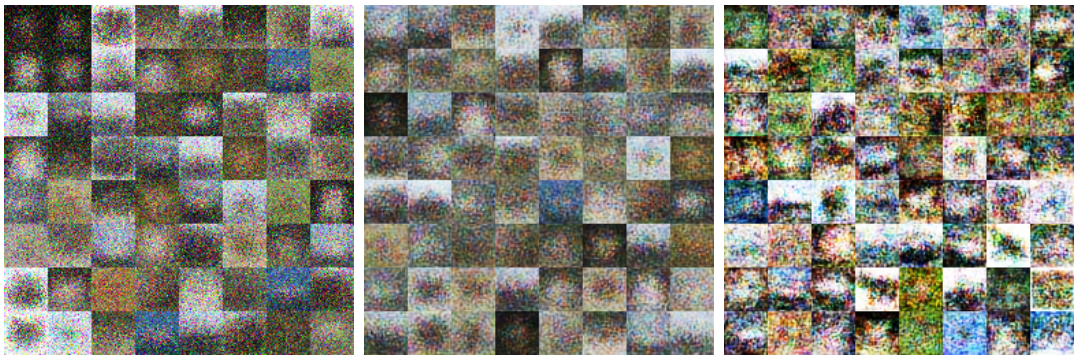


Figure 5.4: The samples at the left are drawn from P_X. Those in the middle are drawn from P_2, and those at the right are drawn from P_3.

Our MNIST in-painting experiments remove random 10×10 patches of pixels from the center 20×20 patch of pixels. Samples drawn from the conditionals $P_X(X_q|X_e)$, $P_2(X_q|X_e)$, and $P_3(X_q|X_e)$ are used to fill in the 10×10 patches. For the in-painting experiments, the Euclidean distance between the filled-in image and the original image is used instead of the distance to the nearest-neighbor in the test set.

Figure 5.3 shows results from the in-painting experiments. The Euclidean measures for the samples drawn from the conditionals $P_X(X_q|X_e)$, $P_2(X_q|X_e)$, and $P_3(X_q|X_e)$ are, respectively, 1235, 1011, and 950. Again, the P_3 samples at the right are both qualitatively and quantitatively better than those from P_2, which are qualitatively and quantitatively better than those from S_X.

We also ran the sampling experiments on the CIFAR-10 dataset. Example results are in Figure 5.4. While the image samples do not form intelligible pictures, there is a marked shift in coherence between the samples drawn from P_3 compared to the other sampling methods. The Euclidean measure against the test set for the samples from P_X, P_2, and P_3 are, respectively, 2896, 1778, and 2885. In this instance our intuitive sense for which results are better does not align with our chosen objective criteria, pointing perhaps to the need for a better way of measuring samples than Euclidean distance.

Table 5.1: Shown are the Euclidean distance measures for the sample sets generated in our experiments.

	MNIST		CIFAR-10
	Sampling	In-painting	Sampling
P_X	1702	1235	2896
P_2	1230	1101	1778
P_3	1212	950	2885

5.5 Conclusion

In this paper we propose the AESPN model, which uses autoencoders to improve the samples generated by SPNs. Two sampling methods are detailed and experimented with. Both methods allow full samples $X = x$ to be drawn or conditional samples of X_q to be drawn given values for X_e. The latter method facilitates the use of AESPNs for image reconstruction tasks.

Our experiments show that adding an SPN that models the hidden representation of the autoencoder helps improve the samples beyond what the autoencoder alone can do. This improvement occurs both when taking full samples and when taking conditional samples. We note that this technique itself may be interesting in autoencoder research. Adding a model over H that can be sampled from adds the ability to "sample" from an autoencoder and provides a method for visually inspecting the "distribution" modeled by the autoencoder. We leave the exploration of this idea to future work. Other possible future work includes experiments that further our understanding of the high-arity problem. With improved understanding would hopefully come ideas for improving SPNs in this regard.

Chapter 6

Nonmonotone Sum-Product Networks

Abstract

Sum-product networks (SPNs) are probabilistic models with many interesting properties, chief among them their guarantees regarding efficient, exact inference. In this paper we generalize the definition of an SPN to allow for negative weight-parameters, explore the consequences of this generalization, and verify its utility experimentally. We prove that efficient, exact inference is still possible even in the presence of negative weights. The move from theory to practice is made by showing that for a sub-class of negative-weight SPNs it is easy to check that they never assign negative values to the probability of an event. A novel algorithm for learning the parameters of these SPNs is also introduced. Experimental results exercise this algorithm and demonstrate the utility of negative weights in SPNs.

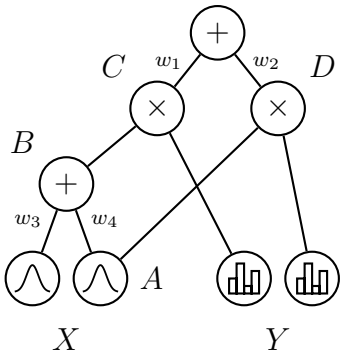


Figure 6.1: A small SPN with two mixture models (sum nodes), two independence models (product nodes), and four base distributions (two normals and two categoricals). Mixing coefficients are shown as edge-weights in the SPN.

6.1 Introduction

A mixture model p combines probability models q_i, each over some set of variables A, using a weighted average so that $p(A) = \sum_i w_i q_i(A)$, where the w_i are non-negative mixing coefficients. Now let the disjoint sets A_i be a partition of A; further, for each A_i let q_i be a probability model over A_i. Then we call a model p an independence model if it combines the models q_i using multiplication so that $p(A) = \prod_i q_i(A_i)$. A sum-product network (SPN) uses these two methods of combining probability models but does so in a recursive manner over various subsets of a set of variables A. An SPN is represented as a directed, acyclic graph of nodes; a sum node appears for each mixture model, a product node for each independence model, and a leaf node for each base distribution [19, 38]. Figure 6.1 shows a simple SPN with two mixture models, two independence models, and four base distributions. Leaf node A, a base normal distribution over continuous variable X, is part of the mixture model computed by sum node B. Product node C is an independence model combining a categorical distribution over discrete variable Y and the mixture model at B. Note that nodes can be re-used; for example, node A is a component of the mixture model at B and the independence model at D.

A remarkable property of SPNs is that, as long as the components in a mixture each cover the same set of variables and the components in an independence model each cover

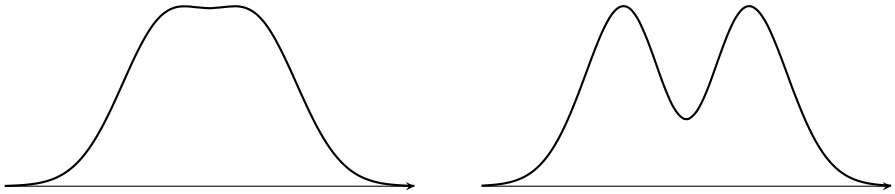


Figure 6.2: Two mixtures of the normal distributions with means $\mu_1 = \mu_2 = 0$ and standard deviations $\sigma_1 = 1, \sigma_2 = 0.9$. At left the mixing coefficients are $(1, -0.75)$ and at right the mixing coefficients are $(1, -0.87)$.

disjoint sets of variables, computing the value of any marginal probability can be done in time linear in the size of the network [38]. The ability to efficiently perform exact inference is one reason SPNs have been the subject of much recent research. Several methods have been developed for learning their parameters from data, as well as the structure of their networks [14–16, 18, 19, 27, 28, 32, 35, 38, 40, 44, 55, 59]. Other more theoretical work has also explored the modeling power of SPNs [12], their relationship to traditional graphical models [58], and other interesting properties [37].

This paper introduces for the first time a definition of SPNs that relaxes one of its key properties, the non-negativity of mixing coefficients in its mixture models. Our definition allows negative mixture-coefficients. While uncommon, negative weights in mixture models have been used before [39]. Figure 6.2 shows a simple example of two normal distributions mixed using negative mixing coefficients.

Theoretical results from circuit complexity theory [49] show that negative weights in arithmetic circuits can be very powerful. Jerrum and Snir [23] prove that any monotone circuit[1] computing the permanent has size $2^{\Omega(n)}$. Two other results show an exponential separation between monotone and nonmonotone circuits. Valiant [53] proves that a single subtraction node is enough to allow a polynomial-sized arithmetic circuit to compute a particular function, and proves that any monotone circuit computing the same function must be exponential in size. Sengupta [47] proves a similar result by showing that any monotone circuit computing the determinant is exponential in size and by noting that the determinant

[1]Roughly-speaking, a monotone circuit is one without negative weights, or coefficients.

can be computed by polynomial-sized nonmonotone circuits. These results indicate that there may be significant benefit from allowing negative weights in SPNs, just as there are significant benefits in arithmetic circuits.

6.2 Definitions

For simplicity we consider the Boolean random vector $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$, but our results can be extended to other discrete and continuous variables. A value of X_i is written x_i and a value of $\boldsymbol{X}$ as $\boldsymbol{x}$. Each X_i takes values in $\{0, 1\}$ and $\boldsymbol{X}$ takes values in $\mathcal{X} = \{0, 1\}^n$. To handle variable marginalization and/or incomplete knowledge of the values of the variables in $\boldsymbol{X}$, we introduce the *evidence set* $\mathcal{E} = \{0, 1, *\}^n$; for $\boldsymbol{e} \in \mathcal{E}$, $e_i = x_i$ indicates that $X_i = x_i$, while $e_i = *$ indicates that the value of X_i is unknown or that X_i is being marginalized. A vector $\boldsymbol{x}$ is *consistent* with $\boldsymbol{e} \in \mathcal{E}$, written $\boldsymbol{x} \sim \boldsymbol{e}$, iff $\forall i (e_i = x_i$ or $e_i = *)$; for example, $(0, 0)$ and $(0, 1)$ are both consistent with $(0, *)$ but not consistent with $(1, *)$. An indicator variable (IV) is defined for each (X_i, x_i) pair as follows:

$$\lambda_{X_i = x_i} = \begin{cases} 1 & \text{if } X_i = x_i \text{ or } X_i \text{ is unknown/marginalized} \\ 0 & \text{otherwise.} \end{cases}$$

The possible settings of the IVs are in a one-to-one correspondence with the vectors in $\mathcal{E}$. This is so because an IV $\lambda_{X_i = x_i}$ takes its value based on whether $X_i = 0$, $X_i = 1$ or X_i is unknown or being marginalized; these three conditions correspond to $e_i = 0$, $e_i = 1$, and $e_i = *$, respectively.

Definition 6.1. A *sum-product network (SPN)* is a rooted, directed acyclic graph whose root node is:

1. an indicator node associated with some $\lambda_{X_i = x_i}$
2. a product node whose children are SPNs with disjoint scopes [2], or

[2] The scope of an SPN is the set of variables whose IVs are associated with the leaf nodes of the SPN.

3. a sum node whose children are SPNs with identical scopes.

The value of an indicator node is the value of its indicator variable. The value of product node i is $\prod_{j \in \mathrm{ch}(i)} v_j$, where $\mathrm{ch}(i)$ is the set of children of node i and v_j is the value of node j. The value of sum node i is $\sum_{j \in \mathrm{ch}(i)} w_{ij} v_j$, where $w_{ij} \in \mathbb{R}$ is the edge-weight between nodes i, j.

Definitions of an SPN vary slightly in the literature [19, 38], but all require $w_{ij} \in \mathbb{R}_+$. The definition above allows negative edge-weights. We distinguish between SPNs that use this extra flexibility and those that do not as follows.

Definition 6.2. A *monotone sum-product network* is an SPN in which all edge-weights w_{ij} are greater than or equal to zero.

Definition 6.3. A *nonmonotone sum-product network* is an SPN in which one or more edge-weights w_{ij} are negative.

An SPN $\mathcal{S}$ computes a function $f_{\mathcal{S}}$ whose domain is $\mathcal{E}$: the value of $f_{\mathcal{S}}(\boldsymbol{e})$ is the value of the root node of $\mathcal{S}$ after setting the indicator nodes in accordance with $\boldsymbol{e}$ and evaluating the other nodes, from the leaves to the root, with children evaluated before parents. In a nonmonotone SPN it is possible that $f_{\mathcal{S}}(\boldsymbol{e}) < 0$ for some $\boldsymbol{e} \in \mathcal{E}$, which would be problematic since an SPN is supposed to represent a (possibly-unnormalized) probability distribution. We deal with this issue later but in preparation introduce the following two definitions.

Definition 6.4. A *positive sum-product network* is an SPN $\mathcal{S}$ such that $\forall \boldsymbol{e} \in \mathcal{E}, f_{\mathcal{S}}(\boldsymbol{e}) \geq 0$.

Definition 6.5. A *negative sum-product network* is an SPN $\mathcal{S}$ such that $\exists \boldsymbol{e} \in \mathcal{E}, f_{\mathcal{S}}(\boldsymbol{e}) < 0$.

With our distinction between monotone/nonmonotone and positive/negative SPNs, we have four classes of SPN; however, note that monotone, negative SPNs are not possible.

6.3 Positive SPNs are Valid

Every positive SPN $\mathcal{S}$ represents a possibly-unnormalized probability distribution $\Phi_{\mathcal{S}}$. To simplify our discussion we drop the phrase "possibly-unnormalized" when writing about

$\Phi_{\mathcal{S}}$; for example, we use marginal probability even though possibly-unnormalized marginal probability would be more accurate. [38] prove that every monotone SPN $\mathcal{S}$ can efficiently compute any marginal probability in $\Phi_{\mathcal{S}}$. We will extend this result by proving that the same holds for positive SPNs, whether monotone or nonmonotone.

For positive SPN $\mathcal{S}$ the distribution $\Phi_{\mathcal{S}}$ is defined in terms of $f_{\mathcal{S}}$ as $\Phi_{\mathcal{S}}(\boldsymbol{x}) \triangleq f_{\mathcal{S}}(\boldsymbol{x})$. We do not define $\Phi_{\mathcal{S}}$ for negative SPNs since $\Phi_{\mathcal{S}}$, being a distribution, is a mapping to non-negative values. The joint and marginal probabilities in $\Phi_{\mathcal{S}}$ are in one-to-one correspondence with the vectors $\boldsymbol{e} \in \mathcal{E}$. If $\boldsymbol{e} \in \mathcal{X}$ then it corresponds to a joint probability; otherwise it corresponds to a marginal probability. For example, $\boldsymbol{e} = (0, *, \ldots, *)$ corresponds to the marginal probability that $X_1 = 0$. For each $\boldsymbol{e} \in \mathcal{E}$, its associated probability is $\Phi_{\mathcal{S}}(\boldsymbol{e}) \triangleq \sum_{\boldsymbol{x} \sim \boldsymbol{e}} \Phi_{\mathcal{S}}(\boldsymbol{x}) = \sum_{\boldsymbol{x} \sim \boldsymbol{e}} f_{\mathcal{S}}(\boldsymbol{x})$; the summation sums out all variables marked by an asterisk in $\boldsymbol{e}$. The special case $\boldsymbol{e} = (*, \ldots, *)$, where all variables are summed out, is written as $\Phi_{\mathcal{S}}(*)$.

Poon and Domingos [38] define an SPN to be **valid** iff $f_{\mathcal{S}}(\boldsymbol{e}) = \Phi_{\mathcal{S}}(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$ and then prove that all monotone SPNs are valid. Equivalently, an SPN is valid iff $f_{\mathcal{S}}(\boldsymbol{e}) = \sum_{\boldsymbol{x} \sim \boldsymbol{e}} f_{\mathcal{S}}(\boldsymbol{x})$. Thus a valid SPN can compute any joint or marginal probability with a single forward pass through the network. Before proving that all positive SPNs are valid, we first establish Lemma 6.1; this Lemma and its proof are inspired by the proof of Lemma 5 in [53]. The proof is by construction. Given an SPN $\mathcal{S}$, monotone SPNs $\mathcal{S}^+$ and $\mathcal{S}^-$ are constructed such that $f_{\mathcal{S}} = f_{\mathcal{S}^+} - f_{\mathcal{S}^-}$. Figure 6.3 illustrates the operations used in building $\mathcal{S}^+$ and $\mathcal{S}^-$.

Lemma 6.1. For any SPN $\mathcal{S}$ there exists monotone SPNs $\mathcal{S}^+, \mathcal{S}^-$ such that $f_{\mathcal{S}} = f_{\mathcal{S}^+} - f_{\mathcal{S}^-}$.

Proof. Let $\mathcal{S}$ be an SPN. Remove all product nodes that only have a single child. Replace all remaining product nodes in $\mathcal{S}$ with a chain of product nodes such that each has exactly two children. If u is a product node with k children $v_1, \cdots, v_k$, replace it with $k-1$ product nodes $u_2, \ldots, u_k$, with the children of u_2 being $\{v_1, v_2\}$ and the children of $u_j, j > 2$, being $\{u_{j-1}, v_j\}$. The function computed by $\mathcal{S}$ has not changed.

We now build SPN $\mathcal{S}'$. For each node u in $\mathcal{S}$ introduce nodes u^+ and u^- in $\mathcal{S}'$. We construct $\mathcal{S}'$ such that $f_u = f_{u^+} - f_{u^-}$ for all nodes u in $\mathcal{S}$. If u is an indicator node in $\mathcal{S}$,

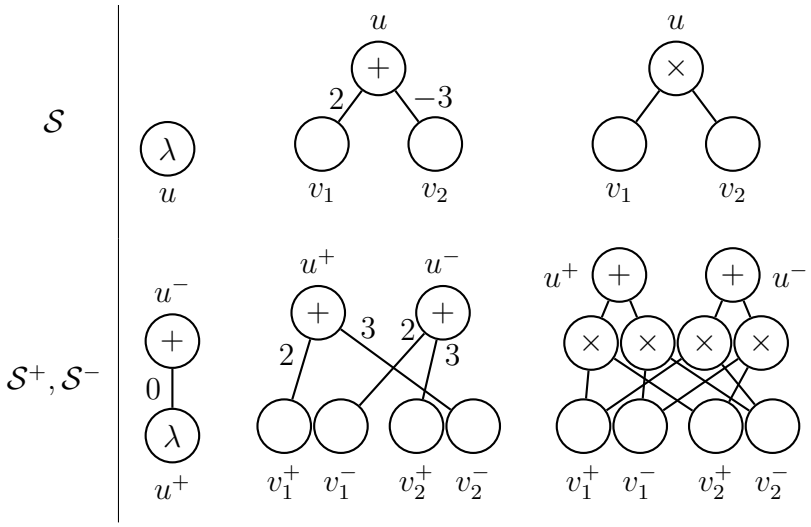


Figure 6.3: The transformations used in Lemma 6.1 converting an SPN $\mathcal{S}$ into monotone SPNs $\mathcal{S}^+$ and $\mathcal{S}^-$. The top row shows indicator, sum, and product nodes from $\mathcal{S}$ and the bottom row shows how they are transformed. Each node u in $\mathcal{S}$ has corresponding nodes u^+ and u^- in $\mathcal{S}^+, \mathcal{S}^-$. Letting f_u, f_{u^+}, and f_{u^-} denote the functions computed by the nodes u, u^+, and u^-, respectively, the transformation is done such that $f_u = f_{u^+} - f_{u^-}$.

let u^+ be an identical indicator node in $\mathcal{S}'$ and let u^- be a sum node whose single child is u^+ and whose edge weight is 0. If u is a sum node or product node then u^+ and u^- will be sum nodes. Create edges in $\mathcal{S}'$ as follows. Let (u, v) be an edge between sum node u and $v \in \mathrm{ch}(u)$, with w the edge weight. If $w \geq 0$ then add edges (u^+, v^+) and (u^-, v^-), each with edge weight $|w|$. If $w < 0$ then add edges (u^+, v^-) and (u^-, v^+), each with edge weight $|w|$. For each product node u in $\mathcal{S}$ we create the following nodes and edges in $\mathcal{S}'$. Let v_1 and v_2 be the children of u. Add two product nodes as children to u^+, with edge weights both 1, with the children of the first product node being $\{v_1^+, v_2^+\}$ and the children of the second product node being $\{v_1^-, v_2^-\}$. Add two product nodes as children to u^-, with edge weights both 1, with the children of the first product node being $\{v_1^+, v_2^-\}$ and the children of the second product node being $\{v_1^-, v_2^+\}$.

Now let u be the root node of $\mathcal{S}$. Then $\mathcal{S}^+$ is the SPN rooted at u^+ and $\mathcal{S}^-$ is the SPN rooted at u^-. Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are monotone since all edge weights in them are 0, 1, or the absolute value of an edge weight in $\mathcal{S}$. And by construction $f_{\mathcal{S}} = f_{\mathcal{S}^+} - f_{\mathcal{S}^-}$. ■

The number of nodes v' and edges e' of the SPN $\mathcal{S}'$ constructed in Lemma 6.1 are both bounded by a polynomial in the number of nodes v and edges e in the original SPN $\mathcal{S}$. After ensuring that product nodes have exactly two children, the resulting SPN has fewer than ve nodes and fewer than $2e$ edges. The indicator transformations add n nodes and n edges. The sum-node transformations at most double the number of nodes and edges. The product-node transformations add at most six nodes for every product node and ten edges for every two edges. See Figure 6.3. In total the SPN $\mathcal{S}'$ has $v' = O(ve)$ nodes and $e' = O(e)$ edges, a polynomial increase in size.

Theorem 6.1. Every positive SPN is valid.

Proof. Let $\mathcal{S}$ be a positive SPN. In the case that $\mathcal{S}$ is also monotone we appeal to Theorem 1 from [38] to show that it is valid. Now assume $\mathcal{S}$ is nonmonotone. Using Lemma 6.1 we can construct monotone SPNs $f_{\mathcal{S}^+}$ and $f_{\mathcal{S}^-}$ such that $f_{\mathcal{S}} = f_{\mathcal{S}^+} - f_{\mathcal{S}^-}$. Both $\mathcal{S}^+$ and $\mathcal{S}^-$ are valid since each is monotone; thus $f_{\mathcal{S}^+}(\boldsymbol{e}) = \Phi_{\mathcal{S}^+}(\boldsymbol{e})$ and $f_{\mathcal{S}^-}(\boldsymbol{e}) = \Phi_{\mathcal{S}^-}(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$. Now let $\boldsymbol{e}$ be an arbitrary vector from $\mathcal{E}$; then

$$\begin{aligned} f_{\mathcal{S}}(\boldsymbol{e}) &= f_{\mathcal{S}^+}(\boldsymbol{e}) - f_{\mathcal{S}^-}(\boldsymbol{e}) = \Phi_{\mathcal{S}^+}(\boldsymbol{e}) - \Phi_{\mathcal{S}^-}(\boldsymbol{e}) \\ &= \sum_{\boldsymbol{x} \sim \boldsymbol{e}} \Phi_{\mathcal{S}^+}(\boldsymbol{x}) - \sum_{\boldsymbol{x} \sim \boldsymbol{e}} \Phi_{\mathcal{S}^-}(\boldsymbol{x}) = \sum_{\boldsymbol{x} \sim \boldsymbol{e}} \left(f_{\mathcal{S}^+}(\boldsymbol{x}) - f_{\mathcal{S}^-}(\boldsymbol{x}) \right) \\ &= \sum_{\boldsymbol{x} \sim \boldsymbol{e}} f_{\mathcal{S}}(\boldsymbol{x}) = \sum_{\boldsymbol{x} \sim \boldsymbol{e}} \Phi_{\mathcal{S}}(\boldsymbol{x}) = \Phi_{\mathcal{S}}(\boldsymbol{e}). \end{aligned}$$

■

6.4 Twin SPNs

Theorem 6.1 shows that positive SPNs are valid. The question remains whether or not a particular SPN is positive. If inspection of the edge-weights in the SPN finds they are all non-negative then, by definition, the SPN is monotone and thus positive. The difficult case is to determine if a nonmonotone SPN is positive. Is it positive even with the presence of

negative weights? Obviously, a brute force algorithm can answer this question. Simply check that $f_S(\boldsymbol{x}) \geq 0$ for all inputs $\boldsymbol{x} \in \mathcal{X}$. However, this quickly becomes intractable since the size of $\mathcal{X}$ is exponential in the number of variables. While we do not know a fast algorithm for determining positivity in the general case, it can be guaranteed quickly in the special case of twin SPNs. Before proving this result we give the definition of a twin SPN and make some observations about it.

Definition 6.6. A *twin SPN* is an SPN $\mathcal{S}$ that meets the following three criteria:

1. $f_{\mathcal{S}} = f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$, where $\mathcal{S}^+, \mathcal{S}^-$ are both monotone and $0 \leq \alpha \leq 1$.
2. $\mathcal{S}^+$ and $\mathcal{S}^-$ have identical architectures.
3. $w_{ij}^+ \geq w_{ij}^-$ holds for each pair (w_{ij}^+, w_{ij}^-) in $\mathcal{S}$, where w_{ij}^+ and w_{ij}^- are corresponding edge-weights in $\mathcal{S}^+$ and $\mathcal{S}^-$, respectively.

The first and second criteria are not restrictive. All SPNs can be converted to meet the first criterion, as shown in Lemma 6.1, with only a polynomial increase in size. The second criterion can also be met as follows. Let $f_{\mathcal{S}} = f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$, where $\mathcal{S}^+$ and $\mathcal{S}^-$ are both monotone, but not identical. Now $f_{\mathcal{S}} = (f_{\mathcal{S}^+} + 0 f_{\mathcal{S}^-}) - \alpha(0 f_{\mathcal{S}^+} + f_{\mathcal{S}^-}) = f_A - \alpha f_B$, where A and B are monotone SPNs with identical architecture. The root of A is a sum node whose children are the root of $\mathcal{S}^+$ and the root of a duplicate of $\mathcal{S}^-$, with the edge weight to $\mathcal{S}^+$ being 1 and the other edge weight 0. Similarly, the root of B is a sum node whose children are the root of $\mathcal{S}^-$ and the root of a duplicate of $\mathcal{S}^+$, with the edge weight to $\mathcal{S}^-$ being 1 and the other edge weight 0. Thus this criteria can be met at the cost of doubling the size of the SPN.

The name twin SPN is in reference to the identical architectures of $\mathcal{S}^+$ and $\mathcal{S}^-$. Unlike general positive SPNs, it is straightforward to check whether an SPN is twin or not and can be done in time linear in the size of $\mathcal{S}$. We next prove that twin SPNs are positive; applying Theorem 6.1 then leads to the corollary that twin SPNs are valid.

Theorem 6.2. Every twin SPN is positive.

Proof. Let $\mathcal{S}$ be a twin SPN; therefore it computes $f_{\mathcal{S}} = f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$, where $\mathcal{S}^+$ and $\mathcal{S}^-$ are monotone SPNs with identical architectures. Consider each node q^+ in $\mathcal{S}^+$, its corresponding node q^- in $\mathcal{S}^-$, and the twin SPN $f_q = f_{q^+} - \alpha' f_{q^-}$, where f_{q^+} and f_{q^-} are the functions computed by the SPNs rooted at q^+ and q^-, respectively, and $0 \leq \alpha' \leq 1$. Proceeding by induction on the nodes of $\mathcal{S}^+$, we show that $f_{q^+}(\boldsymbol{e}) \geq f_{q^-}(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$; reaching the step in which q^+ is the root of $\mathcal{S}^+$ completes the proof by showing that $f_{\mathcal{S}^+}(\boldsymbol{e}) \geq f_{\mathcal{S}^-}(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$.

If q^+ is an indicator node, then $f_{q^+} = f_{q^-}$ and the claim is obviously true. Now we can assume the claim is true for all children of q^+. Because $\mathcal{S}^+$ and $\mathcal{S}^-$ are monotone we have that 1) for any node q in $\mathcal{S}^+$ or $\mathcal{S}^-$, $f_q(\boldsymbol{e}) \geq 0$ and 2) for any weight w in $\mathcal{S}^+$ or $\mathcal{S}^-$, $w \geq 0$. If q^+ is a product node then $f_{q^+}(\boldsymbol{e}) = \prod_{c^+ \in \mathrm{ch}(q^+)} f_{c^+}(\boldsymbol{e}) \geq \prod_{c^- \in \mathrm{ch}(q^-)} f_{c^-}(\boldsymbol{e}) = f_{q^-}(\boldsymbol{e})$. The inequality holds since, for any child $c^+ \in \mathrm{ch}(q^+)$ and its corresponding node c^- in $\mathcal{S}^-$, $f_{c^+}(\boldsymbol{e}), f_{c^-}(\boldsymbol{e}) \geq 0$ and by the inductive hypothesis $f_{c^+}(\boldsymbol{e}) \geq f_{c^-}(\boldsymbol{e})$. If q^+ is a sum node then $f_{q^+}(\boldsymbol{e}) = \sum_{c^+ \in \mathrm{ch}(q^+)} w_{c^+} f_{c^+}(\boldsymbol{e}) \geq \sum_{c^- \in \mathrm{ch}(q^-)} w_{c^-} f_{c^-}(\boldsymbol{e}) = f_{q^-}(\boldsymbol{e})$. The inequality holds since, for any child $c^+ \in \mathrm{ch}(q^+)$ and its corresponding node c^- in $\mathcal{S}^-$, $f_{c^+}(\boldsymbol{e}), f_{c^-}(\boldsymbol{e}), w_{c^+}, w_{c^-} \geq 0$, by the inductive hypothesis $f_{c^+}(\boldsymbol{e}) \geq f_{c^-}(\boldsymbol{e})$, and $w_{c^+} \geq w_{c^-}$ by the definition of a twin SPN. ■

Usually we also want to guarantee that $f_{\mathcal{S}} \neq 0$. This can be done by requiring $f_{\mathcal{S}^+}(*) > 0$ and $\alpha < 1$.

The definition of an SPN can be generalized by letting leaf nodes be any univariate probability distribution, not just indicator nodes. In this case twin SPNs are still positive as long as, for each pair of corresponding leaf nodes u^+ and u^- in $\mathcal{S}^+$ and $\mathcal{S}^-$, respectively, we have $u^+(x) \geq u^-(x)$ for all possible inputs x. Of course, if u^+ and u^- are normalized, then setting $u^+ = u^-$ is the only way to satisfy this condition. But if they are not normalized then other possibilities arise. For example, the condition is satisfied if u^+ is the continuous uniform distribution over $[0, 5]$ and u^- is uniform over $[1, 2]$, with the constraint that $\int_{-\infty}^{\infty} u^- dx \leq 1/5$.

6.4.1 Sampling Twin SPNs

Let monotone SPN $\mathcal{S}$ represent a distribution over the variables in $\boldsymbol{X}$. Let X denote these variables, Y and Z be disjoint sets forming a partition of X, and $\boldsymbol{Y}, \boldsymbol{Z}$ be vectors containing the variables in Y, Z, respectively. Using $\mathcal{S}$ we can draw samples from any conditional distribution $p(\boldsymbol{Y}|\boldsymbol{Z}{=}\boldsymbol{z})$. In the case that $Z = \emptyset$ the samples comes from the joint distribution over X. Sampling is done as follows. First, $\boldsymbol{e} \in \mathcal{E}$ is chosen such that $e_i = *$ if $X_i \in Y$ and $e_i = x_i$ if $X_i \in Z$. Then $f_{\mathcal{S}}(\boldsymbol{e})$ is computed using a forward pass through the SPN and the value of each node i in $\mathcal{S}$ is cached as v_i. Next, a backward pass through the network is done, starting at the root node and tracing out a tree embedded in $\mathcal{S}$; not every node in $\mathcal{S}$ is necessarily visited, as will be seen. If the current node i—which, again, is initially set to be the root node—is a sum node then one child j is selected, with the probability of selecting it proportional to $w_{ij}v_j$; the sampling procedure is then called recursively with this child set as the current node. If the current node is a product node then the sampling procedure is called recursively, once for each of its children. If the current node is a univariate distribution over X_i then a sample is drawn from it; if the node is an indicator then the sample is set to the value indicated by the indicator. After the recursive calls finish, one and only one sample will have been drawn for each variable $X_i \in X$. The samples drawn for the variables in Y make up a sample from $p(\boldsymbol{Y}|\boldsymbol{Z}{=}\boldsymbol{z})$.

Unfortunately this sampling procedure does not work for nonmonotone SPNs since choosing a sum-node child j with probability proportional to $w_{ij}v_j$ does not make sense when $w_{ij}v_j < 0$. However, we can still sample from twin SPNs using rejection sampling, with the distribution represented by $\mathcal{S}^+$ used as the proposal distribution. Here we describe sampling from the joint, but sampling from $P(Y|Z{=}z)$ is also possible. First, a sample $\boldsymbol{x}$ is drawn from $\mathcal{S}^+$ using the sampling procedure described in the preceding paragraph. Then we compute $q^+ = f_{\mathcal{S}^+}(\boldsymbol{x})$ and draw a sample y from the uniform distribution over the interval $[0, q^+]$. In the last step we compute $q = f_{\mathcal{S}}(\boldsymbol{x}) \leq q^+$ and keep the sample $\boldsymbol{x}$ if $y \leq q$; otherwise the rejection sampling procedure is repeated until a sample is found. Actually, care must be

Algorithm 13 TwinGD($\mathcal{S}$, $\mathcal{D}$)

Input: twin SPN $\mathcal{S}$, data $\mathcal{D}$, learning rate η
Initialize Θ, the parameters of $\mathcal{S}$
while $\mathcal{L}(\mathcal{S}|\mathcal{D})$ improves **do**
  $\Theta \leftarrow \Theta + \eta \frac{\partial}{\partial \Theta} \log \mathcal{L}(\mathcal{S}|\mathcal{D})$
  Adjust Θ so that $\mathcal{S}$ is still twin ($w_{ij}^{+} \geq w_{ij}^{-}$)
end while

taken to avoid accepting a sample in the corner case in which $q = 0$ and $y = 0$, since $q = 0$ implies that the probability of $\boldsymbol{x}$ under the distribution represented by $\mathcal{S}$ is zero.

6.4.2 Learning Twin SPNs

Given a dataset $\mathcal{D}$ we would like to learn a twin SPN $\mathcal{S}$ that maximizes the likelihood

$$\mathcal{L}(\mathcal{S}|\mathcal{D}) = \prod_{\boldsymbol{x} \in \mathcal{D}} p(\boldsymbol{x}) = \frac{1}{f_{\mathcal{S}}(*)} \prod_{\boldsymbol{x} \in \mathcal{D}} f_{\mathcal{S}}(\boldsymbol{x}).$$

Given the structure of a twin SPN we can learn its parameters using a modified gradient descent on the negative of the log-likelihood function. Algorithm 13 outlines this process. The main difference from plain gradient descent is that the parameters of $\mathcal{S}$ may need to be adjusted after each gradient step to ensure that $w_{ij}^{+} \geq w_{ij}^{-}$ for all edge-weight pairs in $\mathcal{S}^{+}, \mathcal{S}^{-}$. For example, we may set $w_{ij}^{-} = w_{ij}^{+}$ if $w_{ij}^{+} < w_{ij}^{-}$. While this works in principle, our experience suggests that TwinGD is prone to finding poor local optima, at least for some datasets. We found that an intelligent parameter-initialization strategy can help overcome this problem. The basic idea is to train $\mathcal{S}^{+}$, use it to train $\mathcal{S}^{-}$, and then fine-tune $\mathcal{S}$. This algorithm, LearnTwin, is outlined in Algorithm 14.

LearnTwin receives as input monotone SPN $\mathcal{S}^{+}$ which has already been trained on $\mathcal{D}$. In our experiments with synthetic data we use a pre-determined architecture for $\mathcal{S}^{+}$ and then learn its parameters with TwinGD, but the output of any of the many structure-learning algorithms in the literature can be used to train $\mathcal{S}^{+}$ instead. We assume $\mathcal{D}$ is drawn i.i.d. from some distribution p^{*} and the distribution represented by $\mathcal{S}^{+}$, which we call p^{+}, is an

Algorithm 14 LearnTwin($\mathcal{D}$, $\mathcal{S}^+$)

Input: dataset $\mathcal{D}$, monotone SPN $\mathcal{S}^+$ trained on $\mathcal{D}$
Let function $m(\boldsymbol{x}, A) = \min\{\|\boldsymbol{x} - \boldsymbol{y}\|_2 | \boldsymbol{y} \in A\}$
$t \leftarrow \max\{m(\boldsymbol{x}, \mathcal{D} \setminus \{\boldsymbol{x}\}) | \boldsymbol{x} \in \mathcal{D}\}$
$\mathcal{D}^+ \leftarrow |\mathcal{D}|$ samples from $\mathcal{S}^+$
$\mathcal{D}^- \leftarrow \{\boldsymbol{x} \in \mathcal{D}^+ | m(\boldsymbol{x}, \mathcal{D}) > t\}$
$\mathcal{S}^- \leftarrow$ copy of $\mathcal{S}^+$
Learn parameters of $\mathcal{S}^-$, aiming to maximize $\mathcal{L}(\mathcal{S}^-, \mathcal{D}^-)$
Add a sum node as the root of $\mathcal{S}$ such that $f_{\mathcal{S}} = f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$.
Adjust parameters of $\mathcal{S}$ so that it is twin
TwinGD($\mathcal{S}, \mathcal{D}$)
return $\mathcal{S}$

approximation to it. The first goal of LearnTwin is to learn the parameters of $\mathcal{S}^-$ in such a way that $p = p^*$, where p is the distribution of the final twin SPN $\mathcal{S}$. Loosely speaking then, we want $\mathcal{S}^-$ to decrease the value of p^+ whenever $p^+(\boldsymbol{x}) > p^*(\boldsymbol{x})$, and do nothing otherwise. In other words, if $p^+(\boldsymbol{x}) > p^*(\boldsymbol{x})$ then we want $f_{\mathcal{S}^-}(\boldsymbol{x}) = p^+(\boldsymbol{x}) - p^*(\boldsymbol{x})$, and otherwise we want $f_{\mathcal{S}^-}(\boldsymbol{x}) = 0$. Of course this is rarely possible to do perfectly in practice[3], for at least two reasons. The first is that we almost never have access to p^*. The second is that, even if we did have access to p^*, the SPN $\mathcal{S}^-$ must meet the constraints in the definition of a twin SPN since that is what we are trying to construct. So instead, LearnTwin uses heuristics to approximate the ideal approach.

To train $\mathcal{S}^-$, LearnTwin constructs a dataset $\mathcal{D}^-$ which it uses in an SPN parameter-learning algorithm. In our experiments we use plain gradient descent, but hard or soft EM could be used instead [16, 18]. Creating $\mathcal{D}^-$ is done by filtering samples drawn from p^+. Given a sample $\boldsymbol{x}$ from p^+, we add the sample to $\mathcal{D}^-$ whenever we guess that $p^+(\boldsymbol{x}) > p^*(\boldsymbol{x})$. The guess is based on whether or not the distance from $\boldsymbol{x}$ to its nearest neighbor in $\mathcal{D}$ is greater than a computed threshold t. The threshold t is computed by looking at the nearest-neighbor distances for each instance in $\mathcal{D}$; t is set to the maximum such nearest-neighbor distance. The assumption is that if the sample $\boldsymbol{x}$ is far away from any instance in $\mathcal{D}$, and in fact is farther away than the distance between any two pairs of instances in $\mathcal{D}$, then it does not

[3] In fact, the only case in which it is possible may be when $p^+ = p^*$.

belong in $\mathcal{D}$, or at least $p^*(\boldsymbol{x})$ is low. We also assume that $p^+(\boldsymbol{x})$ is relatively high since sampling, by definition, tends to draw samples from the modes of p^+. Of course we can find situations, such as when p^+ is uniform, in which these assumptions do not necessarily hold, but for many problems they will be reasonable.

The SPN $\mathcal{S}^-$ is initially created as a copy of $\mathcal{S}^+$ to satisfy the second condition of twin SPNs and then its parameters are learned using $\mathcal{D}^-$. These SPNs are combined into twin SPN $\mathcal{S}$ such that $f_{\mathcal{S}} = f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$, by creating a sum node with edge-weights $(1, -\alpha)$ whose children are the root nodes of $\mathcal{S}^+, \mathcal{S}^-$. The last two steps in LearnTwin set the parameters of $\mathcal{S}$ to satisfy the third condition of twin SPNs and fine-tune the parameters using TwinGD.

6.5 Continuous Variables

The definition of an SPN can be generalized by allowing univariate-distribution nodes at the leaves instead of just indicator nodes. Discrete distributions such as the categorical or the Poisson and continuous distributions such as the normal or the uniform can also appear at leaf nodes. The definitions and results from Sections 6.2, 6.3, and 6.4 can also be extended to the case in which univariate distributions are allowed at leaf nodes.

When using continuous variables we add to the third criteria of a twin SPN as follows. For any univariate u_i^+ in $\mathcal{S}^+$ and its corresponding univariate u_i^- in $\mathcal{S}^-$ we require that $u_i^+(\boldsymbol{e}) \geq u_i^-(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$. If both u_i^+ and u_i^- are normalized, then the only way to meet this condition is to set $u_i^+ = u_i^-$. We can allow leaf nodes more flexibility by introducing a scalar factor $\alpha_i \in [0, 1]$ that functions similarly to α in the expression $f_{\mathcal{S}^+} - \alpha f_{\mathcal{S}^-}$. Say $u_i^+(\boldsymbol{e}) = g(\boldsymbol{e}; \theta_i^+)$, where g is some univariate distribution function. Then we set $u_i^-(\boldsymbol{e}) = \alpha_i g(\boldsymbol{e}; \theta_i^-)$, picking α_i such that $u_i^+(\boldsymbol{e}) \geq u_i^-(\boldsymbol{e})$ for all $\boldsymbol{e} \in \mathcal{E}$.

If $\alpha_i = 0$ then the condition will always hold but also means that u_i^- does not contribute to the model. We would like to know how large a value α_i can take before the condition no longer holds. The following subsections present such bounds for a few different

univariate distributions. The bounds will be functions of the distribution parameters θ_i^+ and θ_i^-.

6.5.1 Scalar Bound for the Normal Distribution

Let $\theta_i^+ = (\mu_1, \sigma_1)$ and $\theta_i^- = (\mu_2, \sigma_2)$ be the means and standard deviations of two normal probability density functions (PDFs) g_1 and g_2, respectively. With $h = g_1 - \alpha_i g_2$, and letting $r(x) = g_1(x)/g_2(x)$, it can be seen that if $\alpha_i \leq \min_x r(x)$, then $h(x) \geq 0$ for all $x \in \mathbb{R}$. Since

$$r(x) = \frac{\sigma_2}{\sigma_1} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{(x-\mu_2)^2}{2\sigma_2^2}},$$

observe that if g_1, g_2 share parameter values then $r(x) = 1$. Now assuming that the parameters are not identical we analyze the three cases $\sigma_1 < \sigma_2$, $\sigma_1 = \sigma_2$, and $\sigma_1 > \sigma_2$. If $\sigma_1 < \sigma_2$ then $\lim_{|x|\to\infty} r(x) = 0$. Assuming $\sigma_1 = \sigma_2$, we can see that 1) if $\mu_1 < \mu_2$ then $\lim_{x\to\infty} r(x) = 0$, and similarly 2) if $\mu_1 > \mu_2$ then $\lim_{x\to-\infty} r(x) = 0$. If $\sigma_1 > \sigma_2$ then $r(x)$ has a non-zero minimum and $\lim_{|x|\to\infty} r(x)$ diverges to infinity. The minimum in this case can be found by solving $\frac{\partial r(x)}{\partial x} = 0$ and plugging the solution into $r(x)$. Now

$$\frac{\partial r(x)}{\partial x} = \left(-\frac{x-\mu_1}{\sigma_1^2} + \frac{x-\mu_2}{\sigma_2^2} \right) \frac{\sigma_2}{\sigma_1} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{(x-\mu_2)^2}{2\sigma_2^2}},$$

and the solution to $\frac{\partial r(x)}{\partial x} = 0$ is $x^* = \frac{\sigma_1^2\mu_2 - \sigma_2^2\mu_1}{\sigma_1^2 - \sigma_2^2}$. Thus if $g_1 = g_2$ then $\min r(x) = 1$; if $\sigma_1 < \sigma_2$, or $\sigma_1 = \sigma_2$ but $\mu_1 \neq \mu_2$, then $\min r(x) = 0$; and if $\sigma_1 > \sigma_2$ then $\min r(x) = r(x^*)$. These results are summarized in Table 6.1.

6.5.2 Scalar Bounds for the Uniform and Π-Sigmoid Distributions

Let $\theta_i^+ = (a_1, b_1)$ and $\theta_i^- = (a_2, b_2)$ be the lower- and upper-bounds of two continuous uniform probability density functions g_1 and g_2, respectively. It is easy to see that $h(x) =$

Table 6.1: To ensure that $h(x) = g_1(x) - \alpha_i g_2(x) \geq 0$ for all $x \in \mathbb{R}$, the bounds on α_i listed below must be met.

Parameters	**Bound on α_i**
$\mu_1 = \mu_2, \sigma_1 = \sigma_2$	$0 \leq \alpha_i \leq 1$
$\mu_1 \neq \mu_2, \sigma_1 = \sigma_2$	$\alpha_i = 0$
$\sigma_1 < \sigma_2$	$\alpha_i = 0$
$\sigma_1 > \sigma_2$	$0 \leq \alpha_i \leq r(x^*), x^* = \frac{\sigma_1^2\mu_2 - \sigma_2^2\mu_1}{\sigma_1^2 - \sigma_2^2}$

$g_1(x) - \alpha_i g_2(x) \geq 0$ for all $x \in \mathbb{R}$ as long as

$$\alpha_i \leq \begin{cases} \frac{b_2 - a_2}{b_1 - a_1} & \text{if } a_1 \leq a_2 < b_2 \leq b_1 \\ 0 & \text{otherwise.} \end{cases}$$

The LearnTwin algorithm uses gradient descent to learn the parameters of a twin SPN. Since the uniform density function is not differentiable, LearnTwin would not be able to learn the parameters of any leaf nodes having uniform distributions. This motivates our use of the Π-sigmoid distribution [2] as a differentiable approximation to the uniform. Its density is computed by subtracting two sigmoid functions that have been shifted with respect to each other:

$$g(x; a, b, \lambda) = \frac{\sigma_\lambda(x - a) - \sigma_\lambda(x - b)}{b - a}, b > a,$$

where $\sigma_\lambda(x) = 1/(1 + e^{-\lambda x}), \lambda > 0$. Along with being differentiable, g also has support over all the reals.

Let $\theta_i^+ = (a_1, b_1, \lambda_1)$ and $\theta_i^- = (a_2, b_2, \lambda_2)$ be the parameters of two Π-sigmoid distributions g_1 and g_2, respectively. Again we want α_i such that $h(x) = g_1(x) - \alpha_i g_2(x) \geq 0$ for all $x \in \mathbb{R}$. If we restrict the parameters such that $\lambda_1 = \lambda_2$ and $a_1 \leq a_2 < b_2 \leq b_1$, then $h(x) \geq 0$ for all x if $\alpha_i \leq (b_2 - a_2)/(b_1 - a_1)$, which is the same bound used when the g_j functions are

uniforms. The bound can be justified by writing α_i as

$$\alpha_i \leq \frac{g_1(x)}{g_2(x)} = \left(\frac{b_2 - a_2}{b_1 - a_1}\right)\left(\frac{\sigma_{\lambda_1}(x - a_1) - \sigma_{\lambda_1}(x - b_1)}{\sigma_{\lambda_2}(x - a_2) - \sigma_{\lambda_2}(x - b_2)}\right) = \left(\frac{b_2 - a_2}{b_1 - a_1}\right)\frac{\Delta_1}{\Delta_2}$$

and then showing that $\frac{\Delta_1}{\Delta_2} \geq 1$. First, because $\lambda_1 = \lambda_2$ we know that $\sigma_{\lambda_1}(x) = \sigma_{\lambda_2}(x)$. Because $\sigma_\lambda(\cdot)$ is monotonically increasing, if $x < y$ then $\sigma_\lambda(x) < \sigma_\lambda(y)$. These facts, along with the condition that $a_1 \leq a_2 < b_2 \leq b_1$, lead to the result that $\sigma_{\lambda_1}(x - a_1) \geq \sigma_{\lambda_2}(x - a_2) > \sigma_{\lambda_2}(x - b_2) \geq \sigma_{\lambda_1}(x - b_1)$. This ordering ensures that $\Delta_1 \geq \Delta_2$ and so $\frac{\Delta_1}{\Delta_2} \geq 1$.

Note that unlike for the normal-distribution and uniform-distribution cases, our bound on α_i is not as loose as possible, and it also relies on λ_1 being equal to λ_2. In practice we simply enforce $\lambda_1 = \lambda_2$ and keep within the extra-tight bound on α_i, but finding a looser bound that takes into account differing values for λ_1 and λ_2 could be valuable future work.

6.6 Experiments

We now turn to an empirical investigation of twin SPNs and focus in particular on understanding the advantages and disadvantages that twin SPNs have compared with monotone SPNs. In other words, we ask what might be gained from allowing SPNs to have negative weights. Our experiments will use real-world datasets as well as datasets sampled from a class of synthetic distributions we call **box distributions**.

6.6.1 Box Distribution

A box distribution $\mathcal{B}(w)$ places probability density in $\mathbb{R}^n$ as follows:

$$f(\boldsymbol{x}; w) = \begin{cases} \frac{1}{1-w^n} & \exists i (0 \leq x_i \leq \frac{1-w}{2} \text{ or } \frac{1+w}{2} \leq x_i \leq 1) \\ 0 & \text{otherwise} \end{cases}$$

where its single parameter w lies in the interval $(0, 1)$. Density is spread out uniformly over the outside "faces" of the unit hypercube to a depth $(1 - w)/2$, where each face is

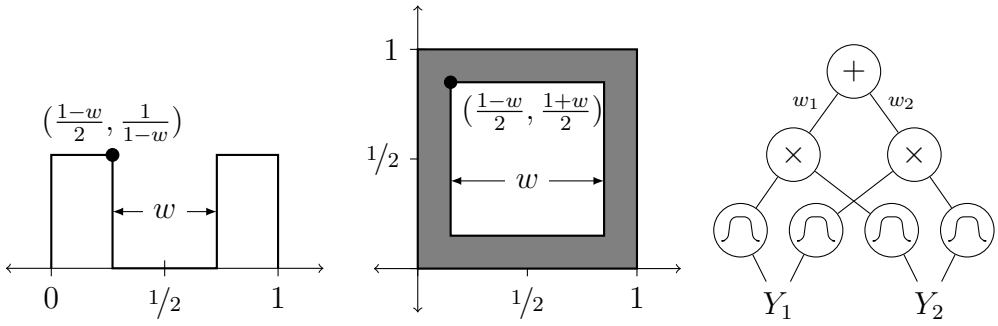


Figure 6.4: The box distributions on the left and middle are in $n = 1$ and $n = 2$ dimensions, respectively. The non-zero density on the left is $\frac{1}{1-w}$ and the non-zero density on the right, indicated by the shaded region, is $\frac{1}{1-w^2}$. At right is a simple $k = 2$ empty-mixture model SPN architecture over $n = 2$ variables Y_1 and Y_2. The term empty-mixture is used because the distribution represented by each product node is equivalent to one represented by an empty BN, or one with no edges, and the sum node is a mixture of them.

an $(n-1)$-dimensional hypercube. Another way to say this: within the unit hypercube there is a centered n-dimensional hypercube with edges of length w in which the probability density is 0; elsewhere in the unit hypercube the desnity is spread uniformly. In three dimensions we can imagine a cube-shaped cardboard box for which w is the perpendicular distance between the inside faces of two opposing sides; in the box distribution the probability density is spread evenly throughout the cardboard shell. Thus we can also write $f(\boldsymbol{x}; w) = u(\boldsymbol{x}; \mathbf{0}, \mathbf{1}) - u(\boldsymbol{x}; \frac{\mathbf{1}-w\mathbf{1}}{2}, \frac{\mathbf{1}+w\mathbf{1}}{2})$, where $u(\boldsymbol{x}; \boldsymbol{a}, \boldsymbol{b})$ is the n-dimensional uniform probability density function with lower and upper bounds in each dimension specified by the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. See Figure 6.4.

If we choose the parameter of $\mathcal{B}(w)$ to be $w = (1 - v)^{1/n}$, then for any number of dimensions n the volume of the shell of non-zero density is a constant v and the density of the shell is a constant v^{-1}, which means that the ground-truth average log-likelihood for any number of dimensions n is $-\log(v)$. In our experiments we choose values for v, n and then set w using the expression above. SPNs are trained on samples drawn from the resulting box distribution, and we compute the average log-likelihood of the samples under the SPNs, comparing the models against each other and against the ground-truth.

6.6.2 Box Distribution Experiments

The goal of the box distribution experiments is not to see how close we can get to the ground-truth likelihood. Rather we show that nonmonotone SPNs can model this data better than monotone SPNs with identical architecture. We then allow the monotone SPNs to increase in size and observe how much larger they get before reaching the performance of the nonmonotone SPNs.

The formulation of $\mathcal{B}(w)$ as one n-dimensional uniform distribution subtracted from another motivates the use of some simple, fixed SPN architectures. The root of each is a sum node with k children, each one a product node. Each product node has exactly n children, each a Π-sigmoid distribution over one of the input variables. See the right-hand side of Figure 6.4. We call this model an empty-mixture becuase it corresponds to a mixture model with k components, each component being an empty Bayesian network (a BN with zero edges). The value of k uniquely identifies the SPN architecture of an empty-mixture model. The nonmonotone twin SPNs that we train will have $k = 2$ components, but the weights on the edges from the sum node to the left and right children will be fixed to 1 and -1, respectively. We compare these twin SPNs with monotone SPNs that also have $k = 2$ components, but whose sum-node edge-weights are not fixed.

Before presenting the empirical results we analyze the problem at hand. If we were to hand-engineer the architecture of a nonmonotone SPN to model a box distribution, it is hard to imagine how we could find something smaller than the $k = 2$ architecture, which consists of $2n + 3$ nodes. These nodes are sufficient to exactly model $\mathcal{B}(w)$.

On the other hand, the $k = 2$ architecture does not give monotone SPNs enough modeling power to exactly represent $\mathcal{B}(w)$, at least not for $n > 1$. How many more nodes, or what different architecture might be required for a monotone SPN to do so? It turns out that the $k = 2n$ architecture, which uses $2n^2 + 2n + 1$ nodes, is enough. To see why this is so observe that there are $2n$ $(n-1)$-dimensional hypercubes formed by the edges in the n-dimensional hypercube [3]. These $(n-1)$-dimensional hypercubes correspond to the

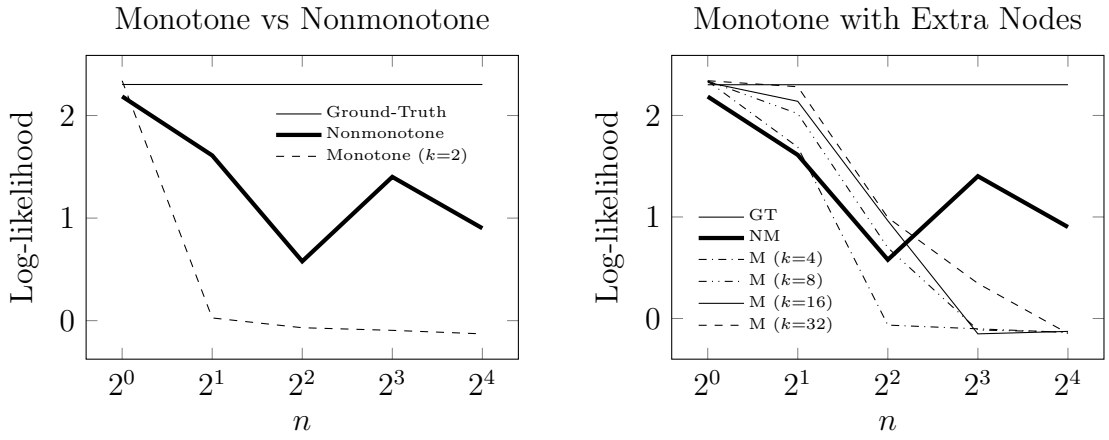


Figure 6.5: Both plots show the log-likelihood of the best SPN models found after doing a grid search. As seen at the left, the nonmonotone SPN performs better than the monotone SPN with identical architecture, for all except the $n = 1$ box-distribution dataset. At right we see the effect of allowing the monotone SPN more nodes.

"sides" of the density shell in the box distribution. And since each side of the shell has depth, we can break the box distribution into parts, one part for each side, each of which is an n-dimensional hyperrectangle. Imagine cutting the three-dimensional cardboard box apart into its six side pieces, each of which is a three-dimensional hyperrectangle. Each part can be exactly represented using one component from an empty-mixture model, and thus a $k = 2n$ architecture is enough to exactly represent the full box distribution.

Note that we have ignored the details of how exactly to break apart the sides where, since each side has depth, they overlap. We also note here that if we continue to hand-engineer the monotone SPN architecture, clever re-use of univariate distributions may allow us to compress the $k = 2n$ architecture into an architecture with something like $5n + 1$ nodes.

Figure 6.5 shows results from the box-distribution experiments. Both plots show the log-likelihood of the best SPN models found while doing a grid search[4]. In the plot on the left we compare the performance of a monotone and a nonmonotone SPN with identical architectures. The nonmonotone SPN outperforms the monotone SPN for all datasets except

[4]The hyperparameters in the grid search: $\lambda \in \{100, 300, 1000\}$, $\eta \in \{.01, .003, .001, .0003, .0001, .00003\}$, $\delta \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$, $b \in \{1, 10, 100, 1000\}$, $e \in \{1, 10, 100, 200\}$, where we fix the slope parameter in Π-sigmoid distributions to λ, and in TwinGD η is the learning rate, δ the log-likelihood improvement threshold, b the mini-batch size, and e the maximum number of epochs.

that derived from the $n = 1$ box-distribution dataset. The plot on the right demonstrates the effect of allowing the monotone SPN more nodes. After $n = 2$, the $k = 4$ monotone SPN starts to underperform the nonmonotone SPN. After $n = 4$, the $k = 8$ monotone SPN begins to underperform. We would expect this pattern to continue but, interestingly, no monotone SPN is able to match the nonmonotone SPN for $n = 8$ and $n = 16$. Again, in principle the nonmonotone SPN is able to reach the ground-truth log-likelihood for every n, and the monotone SPN is able to reach it whenever $k \geq 2n$. While for $n > 1$ the learning algorithms fail to find the ideal solutions in both cases, it is interesting that the failure is greater in the monotone case for $n \geq 8$. This can partly be explained by the fact that the nonmonotone SPN has fewer nodes and fewer parameters—making the task of learning them easier—than the $k = 2n$ monotone SPN that could in theory match its performance.

6.7 Conclusion

In this paper we provide, for the first time, a definition of a sum-product network that allows for negative edge-weights. When using this generalized definition it is helpful to make the distinction between what we call monotone SPNs, which only have positive edge-weights, and nonmonotone SPNs, which have at least one negative edge-weight. A distinction is also made between positive SPNs, which only compute positive values, and negative SPNs, which compute a negative value for some input. We also define twin SPNs, a particular class of nonmonotone SPN.

We prove some important results for nonmonotone, positive, and twin SPNs. We prove that any SPN can be converted into a nonmonotone SPN with a single negative edge-weight. Then we generalize a result from Poon and Domingos [38] that showed monotone SPNs to be valid, or in other words, that monotone SPNs can compute any marginal probability quickly. We prove the same result for positive SPNs, a strict superset of monotone SPNs. This implies that some SPNs with negative edge-weights are positive and thus valid. Because negative SPNs cannot be used as probabilistic models, it is important to distinguish between

nonmonotone SPNs that are positive and those that are negative. We address this problem in the particular case of twin SPNs, proving that all twin SPNs are positive. This makes them useful in practice since it is easy to check whether an SPN is twin or not.

Another contribution we make is the proposed use of gradient descent for learning the parameters of a twin SPN as well as the introduction of the LearnTwin algorithm. LearnTwin can be thought of as a strategy for initializing the parameters of a twin SPN prior to using gradient descent. Our experience indicates this strategy may help gradient descent avoid shallow local optima.

The utility of negative edge-weights in general, and twin SPNs in particular, is shown in our experimental results with data drawn from box distributions of varying dimensionality. These results compare the log-likelihood of two SPNs that have the same architecture. The only difference is that one is a twin SPN that can use negative edge-weights while the other is restricted to using positive edge-weights. As the number of variables n increases we see the log-likelihood of these two models diverge, demonstrating the advantage of negative edge-weights. We also experiment with increasing the size of the architecture of the monotone SPN, giving it an advantage. For smaller values of n this helps the monotone SPN be competitive, but for larger values of n the smaller twin SPN achieves higher log-likelihood.

There are many avenues for future research on nonmonotone SPNs. Expectation-maximization and signomial programming [16, 59] have both been used to learn the parameters of monotone SPNs. Can they be adapted to the nonmonotone case? A structure learning algorithm for twin SPNs could also be very useful. Currently the structure of $\mathcal{S}^+$ is learned and then passed to LearnTwin, which constructs a twin SPN by combining $\mathcal{S}^+$ with a copy, $\mathcal{S}^-$. It may be better to learn the structure of $\mathcal{S}^+$ and $\mathcal{S}^-$ in a single step that is designed to take advantage of twin SPNs and negative weights. Or more incrementally, improvements to LearnTwin, such as exploring other strategies for generating the dataset $\mathcal{D}^-$, could be experimented with.

On the theoretical side, connections with results from circuit-complexity could be investigated. For instance, the size of some arithmetic circuits can be exponentially reduced when using negative weights. Can a similar benefit be shown in the more specific case of SPNs? The relationship between nonmonotone SPNs and twin SPNs could also be examined more closely. For example, perhaps it is always possible to convert a positive, nonmonotone SPN into a twin SPN with only a polynomial increase in size. In that case there would be no great disadvantage to restricting ourselves to twin SPNs. On the other hand, if performing such a conversion is not always possible, then what do we lose by only using twin SPNs?

Chapter 7

Conclusion

This dissertation builds on the work of Poon and Domingos [38], which introduced the sum-product network model. It in turn built on the work of Darwiche [11], which introduced the concept of the network polynomial and its representation as an arithmetic circuit. Our contribution to the field includes three algorithms for learning the structure of an SPN, one of which, BuildSPN, was the first SPN structure learning algorithm ever proposed. Another, SearchSPN, adapts ideas from the LearnSPN algorithm [19] and uses a search approach to learn the SPN structure. The third, OnlineSearchSPN, is an adaptation of SearchSPN to the online learning setting. We also contribute by showing how SPNs can work with an autoencoder to model image data. Our last contribution extends the definition of SPNs to allow negative edge-weights. We explore some of the implications of this definition, prove some theoretical results, and provide algorithms for learning. We detail more specifically these accomplishments as follows.

- We define a region graph in Definition 2.1 and propose a method for building one from data. In some important ways this method foreshadows the approach LearnSPN takes in constructing a tree-structured SPN. Both recursively partition the rows and columns of the training dataset to guide the construction of a tree-structured object, a region graph in the case of our method and an SPN in the case of LearnSPN. We also propose a method for converting a region graph into a DAG-structured SPN.

- The preceding bullet point outlines the main parts of our BuildSPN algorithm, the first published SPN structure learning algorithm.

- Our experiments with BuildSPN show the advantage of learning the structure of an SPN instead of using a pre-specified, fixed SPN architecture.

- We prove in Theorem 3.1 that the function, f, computed by an SPN is equal to the sum of all the functions computed by the complete sub-circuits (see Definition 3.2) embedded in the SPN.

- Theorem 3.1 lets us express the likelihood function, and an approximation of it, in such a way that we can justifiably calculate the amount each product node in an SPN contributes to a low likelihood value. This is crucial in the SearchSPN algorithm (see Algorithm 5) because it guides the selection of a product node at each step; and the structure of the SPN near the selected product node is what then gets changed in that step of the search procedure.

- The SearchSPN algorithm changes the structure of an SPN using the MixClones operator (see Figure 3.2 and Algorithm 4). This operator is also a unique contribution of this dissertation. The location at which it is applied is described in the previous bullet point. Other features of its application are guided by partitioning the rows and columns of subsets of the training data; the partitioning strategy we use is partly inspired by the approach taken in LearnSPN.

- We propose the permanent distribution and an approximation to it in Section 3.5.1, which we sample to produce the MNPerm datasets. Our experiments with these datasets show that SearchSPN finds SPNs that are about the same size as those found by LearnSPN but whose likelihood is much better. We hypothesize that the difference is due to the ability of SearchSPN to find DAG-structured SPNs and the fact that LearnSPN is restricted to learning tree-structured SPNs.

- In Algorithm 8 we propose OnlineSearchSPN, an adaptation of the SearchSPN algorithm to the online setting.

- We experimentally compare OnlineSearchSPN with two methods for adapting offline structure learning algorithms to work as online algorithms. One of these methods is slow but produces good models and the other is fast but produces poor models. In our results we see that OnlineSearchSPN is not much slower than the fast algorithm but produces models that are about as good as the slow algorithm. These results are presented in Section 4.4.

- We propose the auotencoder-SPN (AESPN) model. It pairs an autoencoder with two SPNs, one to model the input data and another to model the hidden representation, or the encoded data. Three methods for drawing samples from this model are described and experimentally compared on both an image generation task as well as an image in-painting task. The results show that an autoencoder can help SPNs produce better samples; the autoencoder also benefits by gaining the ability to generate images.

- In Definition 6.1 we generalize the definition of an SPN to allow negative edge-weights. We also define monotone, nonmonotone, positive, and negative SPNs in Definitions 6.2, 6.3, 6.4, and 6.5.

- We prove that all positive SPNs, whether monotone or nonmonotone, are valid. In other words, efficient, exact marginalization can be performed in any positive SPN. This generalizes the previously-known result that all monotone SPNs are valid.

- We define a class of nonmonotone SPNs called twin SPNs and prove that every twin SPN is positive. It is easy to check whether an SPN is twin or not, making this class of nonmonotone SPNs especially useful in practice.

- Gradient descent is proposed as a method for learning the parameters of a twin SPN. LearnTwin, outlined in Algorithm 14, is given as an improved method for learning twin SPNs.

- We experimentally compare nonmonotone SPNs with monotone SPNs and show them to perform better when learning to model data drawn from box distributions, a distribution we introduce for this purpose.

7.1 Future Work

An SPN combines probability distributions using mixture models and independence models, or sum nodes and product nodes. If the scopes of the distributions in the mixture models are the same, if the scopes of the distributions in the independence models are disjoint, and if any marginal can be computed quickly in the constituent distributions, then any marginal can be computed quickly in the SPN. Are mixture and independence models the only two ways of combining probability distributions such that efficient, exact marginalization is preserved? Proving this true would help us understand the limits of SPNs, and finding alternate methods of combining distributions could expand the modeling power of SPNs.

We have presented several SPN structure learning algorithms, but there is room to improve all of them. One possible improvement to both SearchSPN and OnlineSearchSPN would be to incorporate an SPN structure operator that removes nodes. Currently both algorithms use the MixClones operator which only ever expands the SPN architecture. A node-removal operator could help undo poorly-taken steps made during the structure search and perhaps increase the number of possible structures that are reachable by the search. Another idea is to combine a top-down and bottom-up structure learning algorithm. There has only been one structure learning algorithm that learns by building an SPN from the leaves to the root. Presumably this approach has a different bias than learning an SPN from the root to the leaves, and combining the two could be beneficial.

The LearnTwin algorithm provides a method for learning nonmonotone SPNs. However, the structure of $\mathcal{S}^+$ is learned without regard to the fact that it is eventually paired with $\mathcal{S}^-$. A structure learning algorithm that directly learns the structure of a twin SPN could be an improvement over LearnTwin. Twin SPNs are only a subset of the positive,

nonmonotone SPNs. Developing learning algorithms for this larger set of SPNs could make nonmonotone SPNs even more useful. On the other hand, if every positive, nonmonotone SPN can be converted to a twin SPN with only a small increase in size then proving it true would also be a useful result.

References

[1] Ryan P. Adams, Hanna M. Wallach, and Zoubin Ghahramani. Learning the structure of deep sparse graphical models. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 1–8, 2010.

[2] Anastasios Alivanoglou and Aristidis Likas. Probabilistic models based on the π-sigmoid distribution. In *Proceedings of the 3rd IAPR Workshop on Artificial Neural Networks in Pattern Recognition*, pages 36–43. Springer, 2008.

[3] Thomas F. Banchoff. *Beyond the Third Dimension: Geometry, Computer Graphics, and Higher Dimensions*. Scientific American Library series. Scientific American Library, 1996.

[4] Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems 12*, pages 400–406. MIT Press, 1999.

[5] Léon Bottou and Yann LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems 16*, pages 217–224. MIT Press, 2003.

[6] Hei Chan and Adnan Darwiche. On the robustness of most probable explanations. In *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence*, pages 63–71. AUAI Press, 2006.

[7] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1306–1312, 2005.

[8] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. In *Proceedings of the 2nd European Workshop on Probabilistic Graphical Models*, pages 49–56, 2004.

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

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

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

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

[13] 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, April 1997.

[14] 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. Curran Associates, 2012.

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

[16] Mattia Desana and Christoph Schnörr. Expectation maximization for sum-product networks as exponential family mixture models. *arXiv preprint arXiv:1604.07243*, 2016.

[17] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding*, 106:59–70, April 2007.

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

[19] 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.

[20] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked autoencoder for distribution estimation. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 881–889, 2015.

[21] Karol Gregor, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra. Deep autoregressive networks. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1242–1250, 2014.

[22] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, July 2006.

[23] Mark Jerrum and Marc Snir. Some exact complexity results for straight-line computations over semirings. *Journal of the ACM*, 29(3):874–897, July 1982.

[24] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[26] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 29–37, 2011.

[27] Sang-Woo Lee, Min-Oh Heo, and Byoung-Tak Zhang. Online incremental structure learning of sum-product networks. In *Proceedings of the 20th International Conference on Neural Information Processing*, pages 220–227. Springer, 2013.

[28] Sang-Woo Lee, Christopher Watkins, and Byoung-Tak Zhang. Non-parametric Bayesian sum-product networks. In *Workshop on Learning Tractable Probabilistic Models at the 31st International Conference on Machine Learning*, 2014.

[29] Daniel Lowd and Pedro Domingos. Learning arithmetic circuits. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 383–392. AUAI Press, 2008.

[30] Daniel Lowd and Amirmohammad Rooshenas. Learning Markov networks with arithmetic circuits. In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics*, pages 406–414, 2013.

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

[32] Mazen Melibari, Pascal Poupart, and Prashant Doshi. Dynamic sum-product networks for tractable inference on sequence data. *arXiv preprint arXiv:1511.04412*, 2015.

[33] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[34] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, October 2011.

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

[36] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *arXiv preprint arXiv:1601.06180*, 2016.

[37] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On theoretical properties of sum-product networks. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics*, pages 744–752, 2016.

[38] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the 27th Annual Conference on Uncertainty in Artificial Intelligence*, pages 337–346. AUAI Press, 2011.

[39] Guillaume Rabusseau and François Denis. Learning negative mixture models by tensor decompositions. *arXiv preprint arXiv:1403.4224*, 2014.

[40] Abdullah Rashwan, Han Zhao, and Pascal Poupart. Online and distributed Bayesian moment matching for parameter learning in sum-product networks. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 1469–1477, 2016.

[41] Pradeep Ravikumar, Martin J Wainwright, John D Lafferty, et al. High-dimensional Ising model selection using ℓ_1-regularized logistic regression. *The Annals of Statistics*, 38(3):1287–1319, June 2010.

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

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

[44] 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.

[45] Pedram Rooshenas. Personal communication, 2014.

[46] Ferdinando Samaria and Andy Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of the 2nd IEEE Workshop on Applications of Computer Vision*, pages 138 –142, 1994.

[47] Rimli Sengupta. Cancellation is exponentially powerful for computing the determinant. *Information Processing Letters*, 62(4):177–181, May 1997.

[48] Prakash Shenoy and Glenn Shafer. Propagating belief functions with local computations. *IEEE Expert*, 1(3):43–52, September 1986.

[49] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3–4): 207–388, December 2010.

[50] Benigno Uria, Iain Murray, and Hugo Larochelle. RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems 26*, pages 2175–2183. Curran Associates, 2013.

[51] Benigno Uria, Iain Murray, and Hugo Larochelle. A deep and tractable density estimator. In *Proceedings of The 31st International Conference on Machine Learning*, pages 467–475, 2014.

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

[53] Leslie G. Valiant. Negation can be exponentially powerful. *Theoretical Computer Science*, 12(3):303–314, November 1980.

[54] Jan Van Haaren and Jesse Davis. Markov network structure learning: A randomized feature generation approach. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, pages 1148–1154, 2012.

[55] Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Simplifying, regularizing and strengthening sum-product network structure learning. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 343–358. Springer, 2015.

[56] Hadley Wickham. ASA 2009 data expo. *Journal of Computational and Graphical Statistics*, 20(2):281–283, January 2011.

[57] Nevin L. Zhang. Hierarchical latent class models for cluster analysis. *Journal of Machine Learning Research*, 5:697–723, December 2004.

[58] Han Zhao, Mazen Melibari, and Pascal Poupart. On the relationship between sum-product networks and Bayesian networks. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 116–124, 2015.

[59] Han Zhao, Pascal Poupart, and Geoff Gordon. A unified approach for learning the parameters of sum-product networks. In *Advances in Neural Information Processing Systems 29*. Curran Associates, 2016. To appear.