Parameter and Structure Learning Techniques for Sum Product Networks

by

Abdullah Rashwan

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor(s): Pascal Poupart
Professor, Dept. of Computer Science, University of Waterloo

Internal Member: Jesse Hoey
Professor, Dept. of Computer Science, University of Waterloo
Yaoliang Yu
Professor, Dept. of Computer Science, University of Waterloo

Internal-External Member: Matthias Schonlau
Professor, Dept. of Statistics & Actuarial Science, University of Waterloo

External Examiner: Pedro Domingos
Professor, Dept. of Computer Science & Engineering, University of Washington

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Probabilistic graphical models (PGMs) provide a general and flexible framework for reasoning about complex dependencies in noisy domains with many variables. Among the various types of PGMs, sum-product networks (SPNs) have recently generated some interest because exact inference can always be done in linear time with respect to the size of the network. This is particularly attractive since it means that learning an SPN from data always yields a tractable model for inference. Learning the parameters and the structure for SPNs is being explored by various researchers, having algorithms that scale are essential in the era of big data. In this thesis, I present tractable parameter and structure learning techniques for SPNs. First, I propose a new Bayesian moment matching (BMM) algorithm to learn the parameters for SPNs generatively. BMM operates naturally in an online fashion and that can be easily distributed. I demonstrate the effectiveness and scalability of BMM in comparison to other online algorithms in the literature.

Second, I present a discriminative learning algorithm for SPNs based on the Extended Baum-Welch (EBW) algorithm [6]. The experiments show that this algorithm performs better than both generative Expectation-Maximization, and discriminative gradient descent on a wide variety of applications. I also demonstrate the robustness of the algorithm in the case of missing features by comparing its performance to Support Vector Machines and Neural Networks.

Finally, I present the first online structure learning algorithm for recurrent SPNs. Recurrent SPNs were proposed by Mazen et. al [31] to model sequential data. They also proposed a structure learning algorithm which is slow, and it only operates in batch mode. I present the first online algorithm to learn the structure of recurrent SPNs. I also show how the parameters

can be learned simultaneously using a modified version of hard-EM algorithm. I compare the performance of the algorithm against different models on sequential data problems.

Acknowledgements

I would like to thank all who made this thesis possible. My deepest appreciation goes to my supervisor, Prof. Pascal Poupart, for his endless support throughout my PhD. I am very grateful for being one of his students. I would also like to thank my committee members Prof. Jesse Hoey, Prof. Yaoliang Yu, Prof. Matthias Schonlau, and Prof. Pedro Domingos for their time.

I would like to thank all my friends that I made during my PhD, they made the journey much more enjoyable. Special thanks to Mazen Melibari, Agastya Kalra, Priyank Jaini, Han Zhao, and Nabiha Asghar.

I would also like to show my appreciation to my parents for their support, and encouragement during the difficult times. And finally, I would like to thank my wife and kids who were there for me during my ups and downs, I couldn't have completed the journey without you.

Dedication

To my parents for their endless love ..

To my wife for her love and support ..

To my two kids, Yahia and Omar, for the joy they bring to my life ..

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Probabilistic graphical models provide a general and flexible framework for reasoning about complex dependencies in noisy domains with many variables. Among the various types of probabilistic graphical models, sum-product networks (SPNs) have recently generated interest because exact inference can always be done in linear time with respect to the size of the network. This is particularly attractive since it means that learning an SPN from data always yields a tractable model for inference. However, learning the parameters and the structure of an SPN can be intractable. In this thesis, I present tractable algorithms to learn the parameters for SPNs both generatively and discriminatively. I also present an online algorithm to learn the structure of regular and recurrent SPNs.

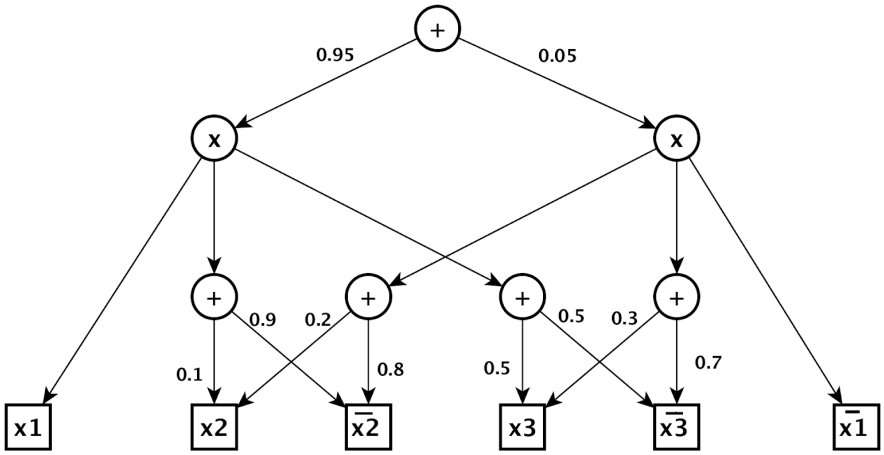


Figure 1.1: An example of a valid SPN over three binary variables; X_1, X_2, and X_3. This model is equivalent to naive Bayes classifier of the target variable (X_1), and two observable variables (X_2, and X_3).

1.1 Sum Product Networks

Sum-Product networks (SPNs) were first proposed by [45] as a new type of probabilistic graphical models. An SPN, as shown in Fig. 1.1, consists of an acyclic directed graph of sums and products that computes a non-linear function of its inputs. The edges linking each sum node to its children are labeled with non-negative weights. SPNs can be viewed as deep neural networks where non-linearity is achieved by products instead of sigmoid, softmax, hyperbolic tangent or rectified linear operations. They also have clear semantics in the sense that they encode a joint distribution over a set of leaf random variables in the form of a hierarchical mixture model. To better understand this distribution, SPNs can be converted into equivalent traditional probabilistic graphical models such as Bayesian networks (BNs) and Markov networks (MNs) by treating sum nodes as hidden variables [57]. An important advantage of SPNs over BNs and MNs is that

inference can be done without any approximation in linear time with respect to the size of the network. Hence, SPNs are gaining in popularity as a *tractable* class of probabilistic graphical models. SPNs have been used in image completion tasks [45], activity recognition [3, 2], scene understanding [14], speech modeling [41] and language modeling [9].

1.2 Parameter Learning for Sum Product Networks

The weights of an SPN are its parameters. They can be estimated by maximizing the likelihood of a dataset (generative training) [45] or the conditional likelihood of some output features given some input features (discriminative training) by Stochastic Gradient Descent (SGD) [15]. Since SPNs are generative probabilistic models where the sum nodes can be interpreted as hidden variables that induce a mixture, the parameters can also be estimated by Expectation Maximization (EM) [45, 37]. [58] provides a unifying framework that explains how likelihood maximization in SPNs corresponds to a signomial optimization problem where SGD is a first order procedure, sequential monomial approximations are also possible and EM corresponds to a concave-convex procedure that converges faster than the other techniques. Since SPNs are deep architectures, SGD and EM suffer from vanishing updates and therefore ”hard” variants have been proposed to remedy to this problem [45, 15]. By replacing all sum nodes by max nodes in an SPN, we obtain a max-product network where the gradient is constant (hard SGD) and latent variables become deterministic (hard EM).

1.3 Structure Learning for Sum Product Learning

Not all SPNs encode valid distributions. Some conditions on the architecture are sufficient to ensure that SPNs encode valid distributions. In the next chapter two sufficient conditions, decomposability and completeness, will be explained. Since it is difficult to specify network structures for SPNs that satisfy the decomposability and completeness properties, several automated structure learning techniques have been proposed [13, 16, 38, 26, 50, 1, 54, 46, 31]. The first two structure learning techniques [13, 16] are top down approaches that alternate between instance clustering to construct sum nodes and variable partitioning to construct product nodes. We can also combine instance clustering and variable partitioning in one step with a rank-one submatrix extraction by performing a singular value decomposition [1]. Alternatively, we can learn the structure of SPNs in a bottom-up fashion by incrementally clustering correlated variables [38]. These algorithms all learn SPNs with a tree structure and univariate leaves. It is possible to learn SPNs with multivariate leaves by using a hybrid technique that learns an SPN in a top down fashion, but stops early and constructs multivariate leaves by fitting a tractable probabilistic graphical model over the variables in each leaf [50, 54]. It is also possible to merge similar subtrees into directed acyclic graphs in a post-processing step to reduce the size of the resulting SPN [46]. Furthermore, [31] proposed dynamic SPNs (a.k.a. recurrent SPNs) for variable length data and described a search-and-score structure learning technique that does a local search over the space of network structures.

So far, all these structure learning algorithms are batch techniques that assume that the full dataset is available and can be scanned multiple times. [26] describes an online structure learning

technique that gradually grows a network structure based on mini-batches. The algorithm is a variant of LearnSPN [16] where the clustering step is modified to use online clustering. As a result, sum nodes can be extended with more children when the algorithm encounters a mini-batch that is better clustered with additional clusters. Product nodes are never modified after their creation. This technique requires large mini-batches to detect the emergence of new clusters and it assumes fixed length data (i.e., it does not generate structures for recurrent SPNs).

1.4 Contributions

- Online Parameter Learning for Discrete Sum Product Networks Using Bayesian Moment Matching [48]: I explored online algorithms that can process a dataset in one sweep and can continuously refine the parameters of an SPN with a fixed structure based on streaming data. I also present an online learning algorithm for SPNs with discrete leaf distributions using Bayesian moment matching. I show empirically that the algorithm outperforms other online algorithms on a wide range of real world datasets. I also show how to distribute the training over multiple machines.

- Online Parameter Learning for Continuous Sum Product Networks Using Bayesian Moment Matching [20]: I extend the Bayesian moment matching algorithm to learn the parameters for SPNs with Gaussian leaves. I also compare the results to other online algorithms. This part was done in collaboration with Priyank Jaini [20], he developed the Bayesian moment matching algorithm for mixture of Gaussians, my part was to extend his algorithm in the context of SPNs with continuous leaf distributions.

- Discriminative Training for Sum Product Networks using Extended Baum-Welch [47]: Discriminative parameter learning techniques haven't been investigated thoroughly for SPNs. In this thesis, I present a discriminative parameter learning technique for SPNs using Extended Baum-Welch. The Extended Baum-Welch algorithm was introduced to learn the parameters of Hidden Markov Models (HMMs) discriminatively. I derived the algorithm for the SPN case, and showed how to learn the parameters discriminatively for SPNs with discrete and Gaussian leaf distributions. I show how Extended Baum-Welch outperforms both generative-EM, and gradient descent on different datasets.

- Online Structure Learning for Recurrent Sum Product Networks [23]: Learning the network structure for RSPNs was done using a Search-and-Score algorithm that has a slow convergence rate, and it can only operate in batch mode. I present the first online algorithm to learn the structure of recurrent SPNs. I also show how the parameters can be learned simultaneously using a modified version of the hard-EM algorithm. This work was done in collaboration with Wei Shou Hsu and Agastya Kalra [23], they developed the algorithm for regular SPNs, and my part was to develop the algorithm for recurrent SPNs.

1.5 Thesis Structure

The rest of the thesis is structured as follows:

1.5.1 Chapter 2

Background materials are presented in Chapter 2. First, I talk about probabilistic graphical models (PGMs), and Bayesian networks as a common type of PGMs. I explain how inference is done in Bayesian networks using the variable elimination algorithm. After, I introduce sum product networks (SPNs), and how marginal and conditional inference can be done in linear time with respect to the size of the network. I review the research that has been done to learn the parameters and the structure for SPNs. I also talk about the recurrent version of SPNs that is used to model sequence of data. I end the chapter with different SPN applications.

1.5.2 Chapter 3

In Chapter 3, I present an online algorithm for learning the parameters of an SPN generatively. I start with explaining the Bayesian learning framework for SPNs, and show how the posterior can often be intractable. I then show how the exploding posterior can be approximated using moment matching. I show how to derive the algorithm for SPNs with discrete and Gaussian leaf distributions. At the end of the chapter, I present a set of experiments to show the performance of the online algorithm compared to other online algorithms. I also show how the training can be distributed over multiple machines.

1.5.3 Chapter 4

After presenting an algorithm to learn the parameters for SPNs generatively in Chapter 3, I present how the parameters of SPNs can be learned discriminatively using Extended Baum-Welch. I first present the Baum-Welch and Extended Baum-Welch algorithms. Then, I show how to derive the Extended Baum-Welch algorithm in the case of SPNs. Finally, I present a set of experiments to compare Extended Baum-Welch for SPNs to generative EM and discriminative gradient descent. I also show the effect of using discriminative training on the learned parameters of the SPNs.

1.5.4 Chapter 5

This chapter presents an online parameter and structure learning algorithm for recurrent SPNs. I first present the online parameter algorithm, and then the online structure learning algorithm for regular SPNs. I show how to extend the algorithm to learn the structure for recurrent SPNs. Then, the experiments section shows that the proposed algorithm outperforms the existing structure learning algorithm for recurrent SPNs.

1.5.5 Chapter 6

I conclude the thesis in Chapter 6 with a discussion of possible future directions.

Chapter 2

Background

This chapter introduces probabilistic graphical models (PGMs), and Bayesian networks (BNs) in particular. It explains how the joint distribution can be encoded in a graph, and how we can use the graph to answer any marginal or conditional queries. Section 2.2 introduces network polynomials, and how we can use them to encode a joint distribution. Section 2.3 introduces sum product networks (SPNs), which utilize network polynomials to build computational graphs that encode valid distributions. Sections 2.4 and 2.5 provide a literature review about the parameter and structure learning for SPNs. Section 2.6 introduces recurrent sum product networks (RSPNs), and I end the chapter with examples of the recent applications of SPNs in fields like speech recognition, language modeling, and scene understanding.

2.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) provide a general and flexible framework for reasoning about complex dependencies in noisy domains with many variables. PGMs utilize probability theory, graph theory, and machine learning to provide such flexible and elegant representation.

Modeling a joint distribution of N binary random variables using probability table would require 2^N parameters. Such a representation is inefficient and unscalable. For instance, modeling 30 binary random variables would require 2^{30} parameters. Learning the parameters for such model would be very challenging due to the exponential increase in the model size with respect to the number of variables. PGMs solve this problem by defining the interactions between random variables using graphs. As a result, the joint distribution can be model without paying the exponential cost.

A probabilistic graphical model G over a set of random variables ($\mathbf{X} = \{X_1, X_2, ..., X_N\}$) is defined using a graph $G = \langle V, E \rangle$, where each node $v \in V$ represents a random variable, and each edge $e \in E$ represents a probabilistic relationship between two nodes. Each node i in the graph is parameterized by a factor f_i. The graph is parametrized by the set of all factors in the graph; $\mathbf{F} = \{f_1, f_2, ..., f_N\}$. The joint distribution is proportional to the product of all factors in the graph $P(\mathbf{X}) \propto \prod_{\forall n} f_n$.

Probabilistic graphical models are classified into directed and undirected graphical models depending on the type of edges in the graph. Directed graphical models, also called Bayesian Networks (BNs), have directed edges that capture the causal relationship between the variables. Undirected graphical models, or Markov Random Fields (MRFs) [7], have undirected edges that

capture the correlation between the nodes in the graph.

For the purpose of this thesis, I will talk about Bayesian networks and dynamic Bayesian networks for the rest of this section since they will be mentioned very frequently throughout the thesis.

2.1.1 Bayesian Networks

Bayesian networks are the type of probabilistic graphical models where the links or edges are directed. The graph in that case is a directed acyclic graph (DAG). A directed edge in the graph represents a probabilistic dependency between two nodes. The node where the edge is coming from is called the parent node, while the node at the other end of the edge is called child node. The factors in that case are called conditional probability tables (CPTs). Each CPT is the conditional probability of the variable $\mathbf{X}_n$ given its parents, $f_n = P(X_n|parents(X_n))$. The joint distribution of the set of variables $\mathbf{X}$ is the product of all CPTs in the graph:

$$P(\mathbf{X}) = P(X_0, ..., X_N) = \prod_n P(X_n|parents(X_n) \tag{2.1}$$

Fig. 2.1 shows two examples of simple Bayesian networks. The first example captures the effect of season on the slippery condition of the grass through a series of random variables. The season has a direct effect on the weather condition being rainy or not. It also affects the usage of sprinklers. A rainy weather or the usage of a sprinkler can make the grass wet which results in a slippery condition. The causal relationships shown in the first example is not the only way to represent the effect of the weather on the slippery condition of the grass, we can

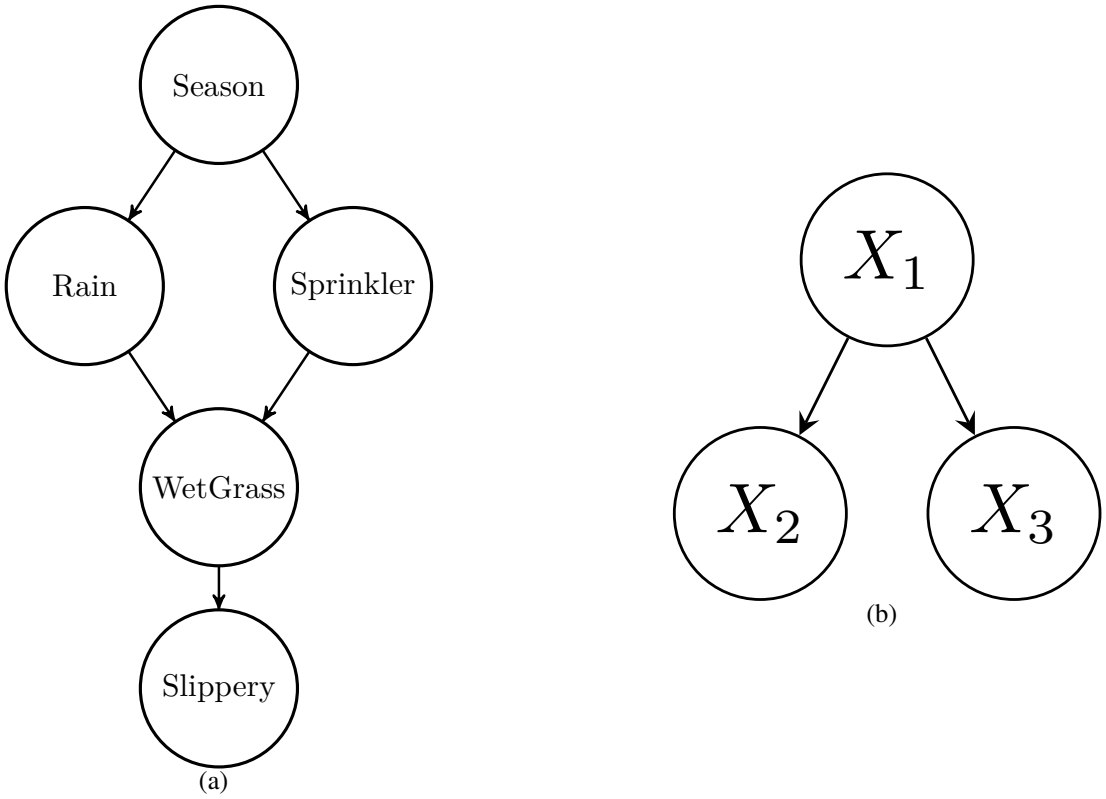


Figure 2.1: Two examples of Bayesian networks, (a) Example by Koller and Friedman [25] (b) A naive Bayes classifier where X_1 is the class label, and X_2, and X_3 are the two conditionally independent observable variables.

add or remove edges to the graph to better capture the relationships between any two nodes in the graph. By adding edges, we pay the price of increasing the parameters of the conditional probability tables. The second graph represents a naive Bayes classifier of the class variable X_1, and the two conditionally independent observable variables, X_2, and X_3.

2.1.2 Inference in Bayesian Networks

Inference in Bayesian Networks is the task of answering probabilistic queries. The query can be either a marginal probability of some variable (for example. $P(X = x)$), or a conditional probability, $P(X|E)$, where E is called the evidence. Exact inference for Bayesian networks, and PGMs in general, is #P-hard [10]. Approximate inference can be used to do inference in polynomial time, but without any guarantee for the accuracy.

Exact inference for Bayesian networks can be done with the variable elimination algorithm. As the name suggests, variable elimination answers a query by marginalizing (eliminating) one variable at a time.

Let's consider the simple example in Fig. 2.1 (b). The Bayesian network has three binary variables X_1, X_2, and X_3. Assume we want to get the marginal probability of P($X_2 = 1$). As stated in the previous section, we can exploit the graph structure to get the joint probability $P(X_1, X_2, X_3)$ which is the multiplication of all factors in the graph.

$$\begin{aligned} P(X_1, X_2 = 1, X_3) &= \prod_n P(X_n|parents(X_n) \\ &= P(X_1)P(X_2 = 1|X_1)P(X_3|X_1) \end{aligned} \tag{2.2}$$

Now, we only need to marginalize X_1, and X_3 which can be done by summing out both variables as follows,

$$P(X_2 = 1) = \sum_{X_1, X_3} P(X_1)P(X_2 = 1|X_1)P(X_3|X_1) \tag{2.3}$$

Marginalizing X_1, and X_3 according to the above equation is equivalent to enumerating all possible values of the joint probability for which $X_2 = 1$ and then summing out the result. This is computationally expensive since there are exponentially many entries. Variable elimination eliminates one variable at a time by reformulating the above equation as follows,

$$\begin{aligned} P(X_2 = 1) &= \sum_{X_1, X_3} P(X_1)P(X_2 = 1|X_1)P(X_3|X_1) \\ &= \sum_{X_1} \sum_{X_3} P(X_1)P(X_2 = 1|X_1)P(X_3|X_1) \end{aligned} \tag{2.4}$$

Now, we can compute $P(X_2 = 1)$ more efficiently using dynamic programming technique by pushing the sums inside the product to sum only the relevant terms.

$$P(X_2 = 1) = \sum_{X_1} P(X_1)P(X_2 = 1|X_1) \sum_{X_3} P(X_3|X_1) \tag{2.5}$$

Each summation results in an intermediate factor. The more variables involved in the summation, the bigger the size of the factor. Hence, the order by which we compute the marginal probability is important, and it has a great impact on the efficiency of the variable elimination algorithm. Ideally, we want to minimize the size of all intermediate factors, finding such optimal order is NP-hard.

Finally, answering conditional probability queries can be done by evaluating two marginal queries since,

$$P(X = x|E = e) = \frac{P(X = x, E = e)}{P(E = e)} \tag{2.6}$$

2.2 Network Polynomials

As we saw in variable elimination, answering a marginal query can be done through a series of sum and product operations. Hence, Darwiche [12] showed that polynomial functions can be used to encode joint distributions. A network polynomial that encodes a joint distribution is $(\sum \phi(x) \prod(x))$, where $\prod(x)$ is a product of indicators and $\phi(x)$ is a probability. For instance, a Bernoulli distribution over variable X parameterized by p can be encoded by the network polynomial $P(X) = px + (1-p)\bar{x}$. Many probabilistic queries can be answered by evaluating the network polynomial or its derivative with respect to the network indicators [12].

2.3 Sum Product Networks

A sum-product network (SPN) [45], as shown in Fig. 2.2, is a rooted acyclic directed graph whose internal nodes are sums and products, and leaves are tractable distributions over some random variables. The edges linking each sum node to its children are labeled with non-negative weights. For SPNs that encode discrete distributions, the leaf nodes can be the variable indicators. SPNs are equivalent to arithmetic circuits [12], which also consist of rooted acyclic directed graphs of sums and products, with the only difference that edges do not have weights, while leaves store numerical values. This is only a syntactic difference since edge weights in SPNs can always be transformed into leaves with corresponding numerical values in ACs (and vice-versa) [50].

The value of an SPN is the value computed by its root in a bottom up pass. More specifically,

$V_{node}(\mathbf{x})$ denotes the value computed by a node based on evidence $\mathbf{x}$.

$$V_{node}(\mathbf{x}) = \begin{cases} p_{node}(\mathbf{x}) & node \text{ is a leaf} \\ \sum_{c \in children(node)} w_c V_c(\mathbf{x}) & node \text{ is a sum} \\ \prod_{c \in children(node)} V_c(\mathbf{x}) & node \text{ is a product} \end{cases} \tag{2.7}$$

Here, $p_{node}(\mathbf{x})$ denotes the probability density or mass of the evidence $\mathbf{x}$ in the distribution at a leaf node, depending on whether $\mathbf{x}$ is continuous or discrete. When the evidence is empty or does not instantiate any of the variables in the distribution at a leaf node then $p_{node}(\mathbf{x}) = 1$, which is equivalent to marginalizing out all the variables for which we do not have any evidence.

Under suitable conditions, an SPN (or AC) encodes a joint distribution over a set of variables. Let $scope(node)$ be the set of variables in the leaves of the sub-SPN rooted at a node.

Definition 1 (Completeness/smoothness [11, 45])**.** *A sum node is complete/smooth when the scopes of its children are the same (i.e.,* $\forall c, c' \in children(sum)$, $scope(c) = scope(c')$*).*

Definition 2 (Decomposability [11, 45])**.** *A product node is decomposable when the scopes of its children are disjoint (i.e.,* $\forall c, c' \in children(sum)$ *and* $c \neq c', scope(c) \wedge scope(c') = \emptyset$*).*

When all sum nodes are complete/smooth and all product nodes are decomposable, an SPN is called valid, which ensures that the value computed by the SPN for some evidence is proportional to the probability of that evidence (i.e., $\Pr(\mathbf{x}) \propto V_{root}(\mathbf{x})$). This means we can answer marginal queries by evaluating the network twice as follows:

$$\Pr(\mathbf{x}_1) = \frac{V_{root}(\mathbf{x}_1)}{V_{root}(\mathbf{1})} \tag{2.8}$$

$V_{root}(\mathbf{1})$ computes the normalization constant, and it is done by setting all indicators or leaf distributions in the network to ones. Conditional queries $\Pr(\mathbf{x}_1|\mathbf{x}_2)$ can be done by two network evaluations $V_{root}(\mathbf{x}_1, \mathbf{x}_2)$ and $V_{root}(\mathbf{x}_2)$:

$$\Pr(\mathbf{x}_1|\mathbf{x}_2) = \frac{V_{root}(\mathbf{x}_1, \mathbf{x}_2)}{V_{root}(\mathbf{x}_2)} \tag{2.9}$$

Since each SPN evaluation consists of a bottom up pass that is linear in the size of the network, the complexity of inference in SPNs is linear in the size of the SPN (same holds for ACs). In contrast, inference may be exponential in the size of the network for BNs and MNs.

We can interpret SPNs as hierarchical mixture models since each sum node can be thought as taking a mixture of the distributions encoded by its children, where the probability of each child component is proportional to the weight labeling the edge of that child. Completeness/smoothness ensures that each child is a distribution over the same set of variables. Similarly, we can think of each product node as factoring a distribution into a product of marginal distributions. Decomposability ensures that the marginal distributions are independent.

2.3.1 Example for a Sum Product Network

Fig. 2.2 shows an example of a valid SPN over three binary variables X_1, X_2, and X_3. The network is a directed acyclic graph. The variable indicators (i.e. x_1, $\bar{x_1}$, x_2, $\bar{x_2}$, x_3, $\bar{x_3}$) are the leaf nodes of the network. All internal nodes are either sum or product nodes. A weight is assigned to each edge coming out of a sum node.

The model shown in Fig. 2.2 is equivalent to a naive Bayes classifier where the target variable

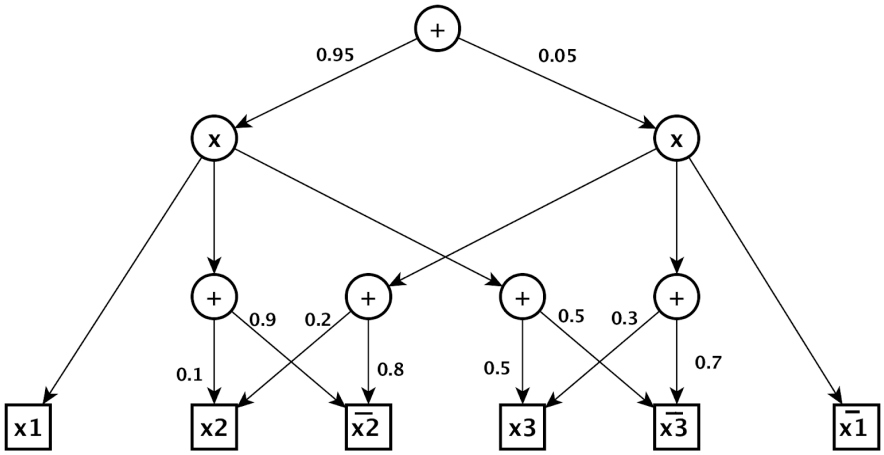


Figure 2.2: An example of a valid SPN over three binary variables; X_1, X_2, and X_3. This model is equivalent to naive Bayes classifier of the target variable (X_1), and two observable variables (X_2, and X_3).

is X_1, and the observable variables are X_2, and X_3. This network is called normalized SPN since the weights for each sum node are normalized (i.e. they sum up to 1). In that case, the normalization constant of Eq. 2.8 is equal 1, we can do inference using one pass through the network.

Let's try to compute $P(X_1 = 1, X_2 = 0)$. First, we need to marginalize X_3 which can easily be done by setting both indicators to 1 (i.e. $\bar{x_3} = 1$, and $x_3 = 1$). The rest of indicators are set according to the evidence, that is $x_1 = 1$, $\bar{x_1} = 0$, $x_2 = 0$, and $\bar{x_2} = 1$. Finally, we compute the value at the root by propagating the values of leaves upwards following the computations defined

by the graph.

$$\begin{aligned} P(X_1 = 1, X_2 = 0) &= V_{root}(X_1 = 1, X_2 = 0) \\ &= 0.95 \times (1 \times (0.9) \times 1) + 0.05 \times 0 \\ &= 0.855 \end{aligned} \tag{2.10}$$

Since this network is equivalent to a naive Bayes classifier, we can use it to estimate the posterior $P(X_1 = 1|X_2 = 0, X_3 = 1)$. Following Eq. 2.9 to compute the posterior, we need to evaluate the network twice to compute $V_{root}(X_1, X_2, X_3)$, and $V_{root}(X_2, X_3)$.

$$\begin{aligned} P(X_1 = 1, X_2 = 0, X_3 = 1) &= V_{root}(X_1 = 1, X_2 = 0, X_3 = 1) \\ &= 0.95 \times (1 \times (0.9) \times 1) \times 0.5 + 0 \\ &= 0.4275 \end{aligned} \tag{2.11}$$

To compute $V_{root}(X_2 = 0, X_3 = 1)$, X_1 is marginalized by setting its indicators to 1.

$$\begin{aligned} P(X_2 = 0, X_3 = 1) &= V_{root}(X_2 = 0, X_3 = 1) \\ &= 0.95 \times (0.9 \times 0.5) \times 0.5 + 0.05 \times (0.8 \times 0.3) \\ &= 0.4395 \end{aligned} \tag{2.12}$$

Finally, the posterior is computed by dividing $V_{root}(X_1, X_2, X_3)$ by $V_{root}(X_2, X_3)$, we get $P(X_1 = 1|X_2 = 0, X_3 = 1) = 0.973$.

2.4 Parameter Learning for SPNs

The weights of an SPN are its parameters. They can be estimated by maximizing the likelihood of a dataset (generative training) [45] or the conditional likelihood of some output features given some input features (discriminative training) by Stochastic Gradient Descent (SGD) [15]. Since SPNs are generative probabilistic models where the sum nodes can be interpreted as hidden variables that induce a mixture, the parameters can also be estimated by Expectation Maximization (EM) [45, 37]. [58] provides a unifying framework that explains how likelihood maximization in SPNs corresponds to a signomial optimization problem where SGD is a first order procedure, sequential monomial approximations are also possible and EM corresponds to a concave-convex procedure that converges faster than the other techniques. Since SPNs are deep architectures, SGD and EM suffer from vanishing updates and therefore "hard" variants have been proposed to remedy to this problem [45, 15]. By replacing all sum nodes by max nodes in an SPN, we obtain a max-product network where the gradient is constant (hard SGD) and latent variables become deterministic (hard EM). It is also possible to train SPNs in an online fashion based on streaming data [26, 48, 56, 20].

2.5 Structure Learning for SPNs

Since it is difficult to specify network structures for SPNs that satisfy the decomposability and completeness properties, several automated structure learning techniques have been proposed. The structure learning algorithms can be classified into two categories depending on the way they construct the network by either starting from the root and learning the structure in a top

down fashion, or starting from the leaves and building the structure in a bottom up fashion.

2.5.1 Top Down Structure Learning Algorithms

This is the common framework for learning the structure of SPNs, the learning algorithm starts from the root, and at each step, the algorithm recursively expands the leaves into product or sum nodes.

The first attempt to learn the structure of SPNs was done by Dennis et al. [13] in 2012. The idea was to use clustering of data instances to introduce sum nodes to the SPN, and clustering of variables to introduce product nodes to the network. When clustering data instances into n clusters, we can introduce n children nodes under a sum node. The weights of an edge between the sum node i and a child j is the number of instances in the cluster of the child j. Clustering of variables is used to split the variables into subsets. Variables in each subset have strong interactions with one another. The entire SPN is constructed by recursively expanding the SPN using instance and variable clustering until the leaf nodes are univariate leaf distributions.

The problem with the algorithm in [13] is the variable clustering based on the distance. Two variables might be highly correlated, but they get assigned to two different clusters because one is the negation of the other. Gens et al. [16] identified such problem, and they proposed a refined algorithm named LearnSPN that mitigates such problem. The algorithm decomposes the scope of a product node by measuring the pairwise mutual information between all variables in the scope. A threshold is used to identify whether two variables belong to the same scope subset. Similar to [13], the algorithm recursively expands the SPN until the leaf nodes are univariate

distributions.

Rooshenas et al. [50] improved LearnSPN by early stopping. Before reaching univariate scopes, arithmetic circuits are learned to create leaves that model the multivariate leaf distributions [30]. The algorithm used to learn the leaf arithmetic circuits can be restricted to produce valid SPNs. The entire algorithm is called ID-SPN.

Vergari et al. [54] further improved LearnSPNs by learning multivariate leaf distributions, and also improved the instance clustering using bagging. In LearnSPN, once an instance is clustered, that instance would only affect the future distributions that result from its cluster. To relax such limitation, Vergari et al. [54] proposed to bootstrap k samples with replacement to produce multiple copies of the available instances, then performing clustering on each copy. For each cluster, a new node is added as a child to the sum node being processed. As a result, an instance can be part of multiple clusters which makes the algorithm less sensitive to clustering errors.

Adel et al. [1] combine instance clustering and variable partitioning in one step with a rank-one submatrix extraction by performing a singular value decomposition. They also showed how to learn the structure both generatively and discriminatively. Computing SVD is a limitation to scaling this algorithm to large number of instances and/or variables.

2.5.2 Bottom Up Structure Learning Algorithms

Pehraz et al. [38] introduced the first bottom-up algorithm for learning the structure of SPNs, the algorithm is called greedy part-wise structure learning. They learn sub-SPNs over small

subsets of the variable scope. Then, product nodes are added to merge two independent disjoint distributions guided by independence tests. Sum nodes are added to introduce a mixture of distributions of sub-SPNs that share the same variable scopes. The algorithm recursively adds product, and sum nodes until the scope at the root includes all the variables.

2.6 Recurrent Sum Product Networks

Recurrent Sum-Product Networks (RSPNs) were introduced by Melibari et al. [32] to model sequential data. RSPNs are a generalization of regular SPNs for modeling temporal sequence of data. RSPNs consist of a template network that can be repeated as many times as needed to model a variable length data sequence. The template network is a recursive function of the input data at time t, and the previous output of the template network at time $t-1$. The repeated network encodes a valid distribution over a data sequence if the template SPN satisfies certain conditions [32] to ensure that the unrolled network is a valid SPN.

Let **X** be a data sequence generated by n variables over T time steps:

$$< x_1, ..., x_n >^1, < x_1, ..., x_n >^2, ... < x_1, ..., x_n >^T \tag{2.13}$$

The template network can be formally defined as follows:

Definition 3 (Template Network [32])**.** *A template network over n binary variables at time t, $< x_1, ..., x_n >^t$, is a directed acyclic graph with k roots, and $k + 2n$ leaf nodes. The $2n$ leaf nodes are the binary variable indicators. The k nodes at the inputs and outputs are the interface*

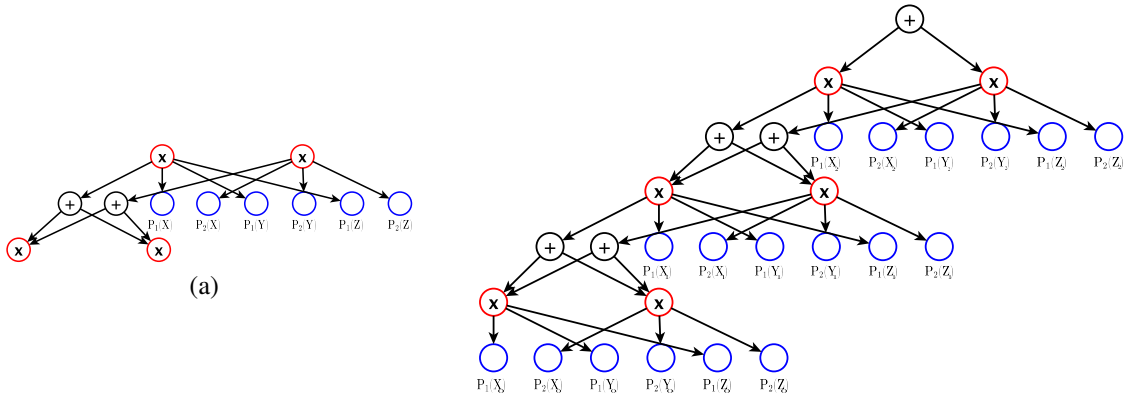
nodes between consecutive time steps. The network nodes are either sum or product nodes. Each edge (i, j) emanating from a sum node i has a non-negative weight w_{ij}

The template network, as shown in Figure 2.3(a), is a directed acyclic graph with k output interface nodes. The inner nodes are sums or products like regular SPNs. At the leaves, we have leaf distributions over the observable variables at time step t, and latent leaf distributions over the input interface nodes. The output interface nodes at time step t are the input interface nodes at time step $t + 1$.

To repeat a template network over a sequence of data, we have to deal with two cases. First, at time $t = 0$, we don't have input interface nodes from the previous time step, hence we need to alter the template network to deal with such case. Second, since inference for SPNs is computed by evaluating the network at the root, we need to add a top network after the last time step that can combine the last k interface nodes into a single rooted graph. [32] defined two additional networks named bottom and top networks that deal with these two cases.

Definition 4 (Bottom Network [32])**.** *A bottom network for the first time step of n binary variables is a directed acyclic graph with k roots and $2n$ leaf nodes. The $2n$ leaf nodes are the indicator variables. The k roots are interface nodes to the template network for the next time step. The interface and interior nodes are either sum or product nodes. Each edge (i, j) emanating from a sum node i has a non- negative weight w_{ij} as in regular SPNs.*

Although the bottom network can have a different structure than template network, we can obtain the bottom network directly from the template network by marginalizing the interface nodes. Marginalizing the interface nodes can be obtained by setting all interface nodes to 1's.



(a) (b)

Figure 2.3: (a) An example of a generic template network. Interface nodes are drawn in red, while leaf distributions are drawn in blue. Leaf distributions can be either binary or Gaussian. (b) An RSPN unrolled over 3 time steps. Three template networks are stacked and capped by the top network.

Definition 5 (Top Network [32])**.** *A top network is a rooted directed acyclic graph composed of sum and product nodes with k leaves. The leaves of this network are interface nodes of the last time step. Each edge (i, j) emanating from a sum node i has a non-negative weight w_{ij} as in regular SPNs.*

Once the structure and the weights of the three networks are defined, unrolling the network can be done as shown in Figure 2.3(b), (1) The bottom network is used at the first time step. (2) The template networks are stacked on top of the bottom network as many time steps as needed. (3) The top network is stacked on top of the last template network.

To ensure that the unrolled network encodes a valid distribution, the unrolled network needs to be consistent and complete. Since verifying these conditions each time the network is unrolled is impractical, [32] define an invariant property for the template network that ensures that the

unrolled network is consistent and complete without the need for unrolling the network.

The invariant conditions [32] restrict the relations among the input and output interface nodes. The scope of each pair of nodes among the input interface nodes needs to be either the same or disjoint, the same relation also needs to hold between the corresponding output interface nodes. The interior nodes in the template network must satisfy the completeness and consistency conditions.

2.7 Sum Product Networks Applications

SPNs do very well in applications where a subset of the leaf variables are unobserved (missing features). SPNs also do well for the task of structure prediction since Most Probable Explanation (MPE) inference can be done naturally by a forwardpass followed by a backward pass on the network. In the first SPN paper, Poon et al. [45] showed the superiority of SPNs to other types of deep networks (ex. deep belief networks) on the task of image completion. In the task, the network was given part of a human face as an input, and then it was asked to infer the rest of the face. SPNs outperformed several other models by wide margins according to the mean squared error metric.

Amer et al. [3] used SPNs for activity recognition (ex. loading of a car). Some primitive actions (e.g. walking, or jumping) were extracted from the videos along with their locations. SPNs are used to model the presence of such actions in an activity. A separate SPN is trained per activity. For testing, primitive actions are extracted from the video, then the activity for which the SPN yields the highest MPE is selected.

SPNs have been used for speech recognition. Pehraz et al. [41] used SPNs for speech recognition of telephone signals. The work was motivated by the success of SPNs in the task of image completion. Similarly, high frequency signals are being lost by the telephone band-pass filters, hence SPNs can be used to recover those high frequency parts. An HMM was used to model the temporal information, and they used SPNs to model the emission distributions. At inference time, the high frequency parts are reconstructed using the most probable explanation inference. The reconstructed signal results in a better speech quality due to the added high frequency component.

Ratajczak et al. [49] applied SPNs in speech recognition by combining SPNs with conditional random fields (CRFs). The new model outperformed state-of-the-art models in the task of optical character recognition, and speech recognition.

SPNs were also used for language modeling. Cheng et al. [9] used SPNs to model relations between words. A special SPN architecture was used to predict the next word using the N previous words. The parameters of the SPN were trained discriminatively. In their experiments, they show that the SPN-based language model outperformed other language models in terms of the perplexity scores.

Chapter 3

Online Parameter Learning for SPNs Using Bayesian Moment Matching

Sum-product networks (SPNs) have recently emerged as an attractive representation due to their dual interpretation as a special type of deep neural network with clear semantics and a tractable probabilistic graphical model. A variety of algorithms have been proposed to learn the structure [50, 16] and the parameters [45] of SPNs from data. Since existing algorithms operate in a batch way, they do not scale to large datasets. In this chapter, we explore online algorithms that can process a dataset in one sweep and can continuously refine the parameters of an SPN with a fixed structure based on streaming data. It is relatively easy to extend existing algorithms like gradient descent (GD), exponentiated gradient and expectation maximization (EM) to the online setting by restricting the algorithms to a single iteration (i.e., single pass through the data) in which incremental updates to the parameters are performed after processing each data point.

Since those algorithms maximize the likelihood of the data, they are subject to overfitting and local optima (the optimization problem is non-convex). Furthermore, restricting the algorithms to a single iteration means that some information is lost. We propose a new Bayesian learning algorithm that does not frame the problem as optimization. Bayesian learning is robust to overfitting and naturally operates in an online, asynchronous and distributed fashion since Bayess theorem allows us to incrementally update the posterior after each data point and the updates can be computed in any order and distributed on different machines. The main issue with Bayesian learning is that the posterior is often intractable, which is the case for parameter learning in SPNs. To that effect, we propose to approximate the posterior after each update with a tractable distribution that matches some moments of the exact, but intractable posterior. We call this algorithm *online Bayesian Moment Matching* (oBMM). The approximation introduced by moment matching yields a loss of information, however we show that oBMM performs better than online expectation maximization (oEM), online Exponentiated Gradient (oEG), and Stochastic gradient descent (SGD) on a suite of binary benchmark datasets and some large language modeling problems. We also demonstrate the scalability of oBMM by distributing the computation on many machines.

Most of the algorithms for SPNs assume that the variables of interest are categorical. However, many real-world applications are best modeled by continuous variables. Hence, there is a need for SPN algorithms that work with continuous data. Several papers mention that it is possible to generalize SPNs to continuous variables by considering leaf distributions over continuous variables [45, 15, 16, 57], but only a recent paper by [40] proposes an EM algorithm for continuous SPNs and reports experiments with continuous data. So, we show how to extend oBMM to

SPNs with continuous variables. More specifically, we consider SPNs with Gaussian leaf distributions and show how to derive an online Bayesian moment matching algorithm to learn from streaming data. We compare the resulting generative models to stacked restricted Boltzmann machines and generative moment matching networks on real-world datasets.

3.1 Online and Distributed Bayesian Moment Matching

We propose to use Bayesian learning to obtain a new online algorithm. As pointed out by [8] Bayesian learning naturally lends itself to online learning and distributed computation. In the case of an SPN, the weights are the parameters to be learned. Bayesian learning starts by expressing a prior $\Pr(\mathbf{w})$ over the weights. Learning corresponds to computing the posterior distribution $\Pr(\mathbf{w}|data)$ based on the data observed according to Bayes' theorem:

$$\Pr(\mathbf{w}|data) \propto \Pr(\mathbf{w}) \Pr(data|\mathbf{w}) \tag{3.1}$$

Since the data consists of a set of instances $\mathbf{x}^{1:N} = \{\mathbf{x}^1, ..., \mathbf{x}^N\}$ that is assumed to be sampled identically and independently from some underlying distribution, we can rewrite Bayes' theorem in a recursive way that facilitates incremental online learning:

$$\Pr(\mathbf{w}|\mathbf{x}^{1:n+1}) \propto \Pr(\mathbf{w}|\mathbf{x}^{1:n}) \Pr(\mathbf{x}^{n+1}|\mathbf{w}) \tag{3.2}$$

The computation of the posterior in Bayes' theorem can also be distributed over several machines that each process a subset of the data. For example, suppose that we have K machines

and a dataset of KN instances, then each machine (indexed by k) can compute a posterior $\Pr(\mathbf{w}|\mathbf{x}^{(k-1)N+1:kN})$ over N instances. Those posteriors can be combined to obtain the posterior of the entire dataset as follows:

$$\Pr(\mathbf{w}|\mathbf{x}^{1:KN}) = \Pr(\mathbf{w}) \prod_{k=1}^{K} \frac{\Pr(\mathbf{w}|\mathbf{x}^{(k-1)N+1:kN})}{\Pr(\mathbf{w})} \tag{3.3}$$

Hence, exact Bayesian learning can be performed naturally in an online and distributed fashion. Unfortunately, the computation of the posterior is often intractable. This is the case for parameter learning in SPNs. Thus, we propose to approximate the posterior obtained after each data instance with a tractable distribution by matching a set of moments. We first describe how to estimate the parameters of an SPN with a fixed structure by exact Bayesian learning. Then we discuss the exponential complexity of this approach and how to circumvent this intractability by Bayesian moment matching.

3.2 Bayesian Moment Matching for Discrete SPNs

3.2.1 Exact Bayesian Learning

The parameters of an SPN consist of the weights associated with the edges emanating from each sum node. The first step is to define a prior over the weights. While the weights can be any non-negative number, any SPN can be transformed into an equivalent *normal* SPN with normalized weights (i.e., $w_{ij} \geq 0$ and $\sum_j w_{ij} = 1 \ \forall i \in$ sumNodes) that correspond to local distributions [42, 57]. Without loss of generality, we will restrict ourselves to normal SPNs since

the likelihood of a data instance $\Pr(\mathbf{x})$ is obtained by a single network evaluation $V_{root}(\mathbf{e}(\mathbf{x}))$ (i.e., the normalization constant $V_{root}(\mathbf{1})$ is always 1) and this allows us to use a Dirichlet for the prior over each local distribution associated with the weights of each sum node. We start with a prior that consists of a product of Dirichlets with respect to the weights $\mathbf{w}_{i\cdot} = \{w_{ij} | j \in children(i)\}$ of each sum node i:

$$\Pr(\mathbf{w}) = \prod_{i \in \text{sumNodes}} Dir(\mathbf{w}_{i\cdot} | \alpha_{i\cdot}) \tag{3.4}$$

The posterior is obtained by multiplying the prior by the likelihood $V_{root}(\mathbf{x})$ of each data instance:

$$\Pr(\mathbf{w}|\mathbf{x}^{1:n+1}) \propto \Pr(\mathbf{w}|\mathbf{x}^{1:n}) V_{root}(\mathbf{x}^{n+1}) \tag{3.5}$$

Since a network evaluation consists of an alternation of sums and products, we can rewrite $V_{root}(\mathbf{x})$ as a polynomial with respect to the weights. Furthermore, this polynomial can always be re-written as a sum of monomials that each consists of a product of weights:

$$V_{root}(\mathbf{x}) = \sum_c monomial_c^{\mathbf{x}}(\mathbf{w}) = \sum_c \prod_{ij \in monomial_c^{\mathbf{x}}} w_{ij} \tag{3.6}$$

Intuitively, if we distribute the sums over the products in the network evaluation, we obtain a large sum of small products of weights where each product of weights is a monomial. The number of monomials is exponential in the number of sum nodes. Note that products of Dirichlets are conjugate priors with respect to monomial likelihood functions. This means that multiplying a monomial by a product of Dirichlets gives a distribution that is again a product of Dirichlets.

However, since our likelihood function is a polynomial that consists of a sum of monomials, the posterior becomes a mixture of products of Dirichlets:

$$\Pr(\mathbf{w}|\mathbf{x}) = \prod_{i \in \text{sumNodes}} Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot}) \sum_c monomial_c^{\mathbf{x}}(\mathbf{w}) \tag{3.7}$$

$$= \sum_c k_c \prod_{i \in \text{sumNodes}} Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot} + \delta_{i\cdot}^{\mathbf{x},c}) \tag{3.8}$$

$$\text{where } \delta_{ij}^{\mathbf{x},c} = \begin{cases} 1 & ij \in monomial_c^{\mathbf{x}} \\ 0 & ij \notin monomial_c^{\mathbf{x}} \end{cases} \tag{3.9}$$

The nice thing about the above derivation is that the posterior has a closed form (mixture of products of Dirichlets), however it is computationally intractable. The number of mixture components is exponential in the number of sum nodes after the first data instance (in the worst case). If we repeatedly update the posterior after each data instance according to Eq. 3.2, the number of mixture components will grow exponentially with the amount of data and doubly exponentially with the number of sum nodes.

3.2.2 Moment Matching

Moment matching is a popular frequentist technique to estimate the parameters of a distribution based on the empirical moments of a dataset. For instance, it has been used to estimate the parameters of mixture models, latent Dirichlet allocation and hidden Markov models while ensuring consistency [4]. Moment matching can also be used in a Bayesian setting to approximate an intractable posterior distribution. More precisely, by computing a subset of the moments of an

intractable distribution, another distribution from a tractable family that matches those moments can be selected as a good approximation. Expectation propagation [33] is a good example of such an approach. We describe how to use Bayesian Moment Matching to approximate mixtures of products of Dirichlets obtained after processing each data instance by a single product of Dirichlets.

A Dirichlet $Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot}) \propto \prod_j (w_{ij})^{\alpha_{ij}-1}$ is defined by its hyper-parameters α. However it could also be defined by a set of moments. Consider the following first and second order moments which are expectations of w_{ij} and w_{ij}^2:

1. $M_{Dir}(w_{ij}) = \int_{w_{ij}} w_{ij} Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot}) dw_{ij}$
2. $M_{Dir}(w_{ij}^2) = \int_{w_{ij}} w_{ij}^2 Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot}) dw_{ij}$

We can express the hyperparameters α_{ij} in terms of the above moments as follows:

$$\alpha_{ij} = M_{Dir}(w_{ij}) \frac{M_{Dir}(w_{ij}) - M_{Dir}(w_{ij}^2)}{M_{Dir}(w_{ij}^2) - (M_{Dir}(w_{ij}))^2} \quad \forall ij \tag{3.10}$$

When approximating a distribution P by a Dirichlet, we can compute the moments $M_P(w_{ij})$ and $M_P(w_{ij}^2)$ of P and then use Eq. 3.10 to set the hyperparameters of the Dirichlet so that it has the same first and second order moments. More generally, since we are interested in approximating a joint distribution $P(\mathbf{w})$ by a *product* of Dirichlets, we can compute the moments $M_{P(\mathbf{w}_{i\cdot})}(w_{ij})$ and $M_{P(\mathbf{w}_{i\cdot})}(w_{ij}^2)$ of each marginal $P(\mathbf{w}_{i\cdot})$ in order to set the hyperparameters $\alpha_{i\cdot}$ of each Dirichlet in the product of Dirichlets. Hence, instead of explicitly computing the intractable posterior after each data instance, we compute the first and second order moments of the posterior

which are sufficient to approximate the posterior by Bayesian moment matching with a product of Dirichlets. It turns out that we can exploit the structure of SPNs to compute efficiently the first and second order moments of the posterior. Algorithm 1 describes how to compute a moment of the posterior in one bottom-up pass. Since the number of moments is linear in the size of the network and computing each moment is also linear, the overall time to approximate the posterior by a product of Dirichlets is quadratic in the size of the network.

Algorithm 1 Compute marginal moment for w_{ij}^k (k is an exponent indicating the order of the moment) in the posterior obtained after observing $\mathbf{x}$

if $isLeaf(node)$ **then**
  return $V_{node}(\mathbf{x})$
else if $isProduct(node)$ **then**
  return $\prod_{child} computeMoment(child)$
else if $isSum(node)$ and $node == i$ **then**
  return $\sum_{child} M_{Dir}(w_{ij}^k w_{i,child}) \times$
    $computeMoment(child)$
else
  return $\sum_{child} w_{node,child}\ computeMoment(child)$

In practice, many structure learning algorithms produce SPNs that are trees (i.e., each node has a single parent) [13, 16, 38, 50, 39, 1]. When an SPN is a tree, it is possible to compute all the moments simultaneously in time that is linear in the size of the network. The key is to compute two coefficients $coef_i^0, coef_i^1$ at each node i in a top-down pass of the network. Algorithm 2 shows how those coefficients are computed. Once we have the coefficients, we can compute each moment as follows:

$$M_{posterior}(w_{ij}^k) = \int_{w_{i\cdot}} w_{ij}^k Dir(\mathbf{w}_{i\cdot}|\alpha_{i\cdot})(coef_i^0 + coef_i^1 \sum_{j'} w_{ij'} V_{j'}(\mathbf{x}))dw_{i\cdot}$$

Algorithm 2 $computeCoefficients(node)$ based on $\mathbf{x}$ and prior $\prod_i Dir(w_{i\cdot}|\alpha_{i\cdot})$

if $isRoot(node)$ **then**
  $coef^0_{node} \leftarrow 0$
  $coef^1_{node} \leftarrow 1$
else if $isProduct(parent(node))$ **then**
  $coef^0_{node} \leftarrow coef^0_{parent}$
  $coef^1_{node} \leftarrow coef^1_{parent} \prod_{sibling} v_{sibling}(e(\mathbf{x}))$
else if $isSum(parent(node))$ **then**
  $coef^0_{node} \leftarrow coef^0_{parent} + coef^1_{parent} \sum_{sibling} \frac{\alpha_{parent,sibling}}{\sum_j \alpha_{parent,j}} v_{sibling}(e(\mathbf{x}))$
  $coef^1_{node} \leftarrow coef^1_{parent} \frac{\alpha_{parent,node}}{\sum_j \alpha_{parent,j}}$
if $isNotLeaf(node)$ **then**
  $computeCoefficients(child)\ \forall child$

At a high level, Alg. 2 maintains a posterior distribution over all the sum nodes in the subtree of a given node. This is achieved by a bottom-up pass of the network to compute all the $v_i(\mathbf{x})$ at each node followed by a top-down recursive pass to compute the required moments at all the sum nodes. In Alg. 2, $v_i(\mathbf{x})$ is defined as:

$$v_i(\mathbf{x}) = \int_{\mathbf{w}} \prod_j Dir(\mathbf{w}_j|\boldsymbol{\alpha}_j) V_i(\mathbf{x})\, d\mathbf{w} \tag{3.11}$$

where the integration is over all the model parameters $\mathbf{w}$ that appear in the sub-tree rooted at node i, and upper case $V_i(\mathbf{x})$ is the value computed at node i when the input instance is $\mathbf{x}$. Note that by utilizing the same network structure, all the $v_i(\mathbf{x})$ can be computed in a single bottom-up pass.

3.3 Bayesian Moment Matching for Continuous SPNs

For continuous SPNs, the leaf nodes are multivariate Gaussian distributions. The Normal-Wishart distribution is a multivariate distribution with four parameters. It is the conjugate prior of a multivariate Gaussian distribution with unknown mean and covariance matrix. This makes a Normal-Wishart distribution a natural choice for the parameters at the leaf nodes. The mixture weights for sum nodes can still be modeled using dirichlet distributions as shown in the previous section.

As we showed in the previous section, the likelihood at any node under the condition of independent sum nodes can be written as a function of two coefficients; $coef^0_{node}$ and $coef^1_{node}$, hence at the leaf nodes, the likelihood is, $P_{node}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}) = (coef^0_{node} + coef^1_{node}\mathcal{N}_d\big(\mathbf{x}; \boldsymbol{\mu}, (\kappa\boldsymbol{\Lambda})^{-1}\big)$.

In Bayesian learning, we start from a family of distributions for the prior, in our case the prior is $P_0(\boldsymbol{\mu}, \boldsymbol{\Lambda}) = \mathcal{NW}(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\alpha}, \kappa, \mathbf{W}, \nu)$. The set of sufficient moments for the posterior in this case would be given by $S(P(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{x})) = \{\boldsymbol{\mu}, \boldsymbol{\mu}\boldsymbol{\mu}^T, \boldsymbol{\Lambda}, \Lambda^2_{kl}\}$ where Λ_{kl} is the $(k, l)^{th}$ element of the matrix $\boldsymbol{\Lambda}$. Notice that, since $\boldsymbol{\Lambda}$ is a symmetric matrix, we only need to consider the moments of the elements on and above the diagonal of $\boldsymbol{\Lambda}$.

To compute the posterior after observing $\mathbf{x_1}$, we get:

$$\begin{aligned} P_0(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &\propto \mathcal{NW}(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\alpha}, \kappa, \mathbf{W}, \nu)\big(coef^1_{node}\mathcal{N}_d\big(\mathbf{x_1}; \boldsymbol{\mu}, (\kappa\boldsymbol{\Lambda})^{-1}\big) + coef^0_{node}\big) \\ &= coef^1_{node}\ \ c\,\mathcal{NW}(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\mu}^*_0, \kappa^*, \mathbf{W}^*, \nu^*) + coef^0_{node}\ \ \mathcal{NW}(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\mu}_0, \kappa, \mathbf{W}, \nu) \quad (3.12) \end{aligned}$$

Where,

$$
\begin{aligned}
\boldsymbol{\mu}_0^* &= \frac{\kappa\boldsymbol{\mu}_0 + \mathbf{x_1}}{\kappa + 1} \\
\kappa^* &= \kappa + 1 \\
\mathbf{W}^* &= \mathbf{W} + \frac{\kappa}{\kappa + 1}(\mathbf{x_1} - \mu_0)^T(\mathbf{x_1} - \mu_0) \\
\nu^* &= \nu + 1 \\
c &= \big(\frac{\kappa}{\kappa^*}\big)^{\frac{d}{2}} \frac{|\boldsymbol{\Lambda}|^\nu}{|\boldsymbol{\Lambda}^*|^{\nu^*}} \frac{\frac{1}{2}\nu\Gamma(\frac{1}{2}\nu^*)}{\Gamma(\frac{1}{2}(\nu^* - d))}
\end{aligned}
\tag{3.13}
$$

The posterior is a mixture of Normal-Wishart with exactly two components. Such distribution can be approximated using a single Normal-Wishart by matching the sufficient moments as shown in the following subsection.

3.3.1 Moment Matching for Normal-Wishart Distribution

Let $\boldsymbol{\mu}$ be a d-dimensional vector and $\boldsymbol{\Lambda}$ be a symmetric positive definite $d \times d$ matrix of random variables respectively. Then, a Normal-Wishart distribution over $(\boldsymbol{\mu}, \boldsymbol{\Lambda})$ given parameters $(\boldsymbol{\mu}_0, \kappa, \mathbf{W}, \nu)$ is such that $\boldsymbol{\mu} \sim \mathcal{N}_d\big(\boldsymbol{\mu}; \boldsymbol{\mu}_0, (\kappa\boldsymbol{\Lambda})^{-1}\big)$ where $\kappa > 0$ is real, $\boldsymbol{\mu}_0 \in \mathbb{R}^d$ and $\boldsymbol{\Lambda}$ has a Wishart distribution given as $\boldsymbol{\Lambda} \sim \mathcal{W}(\boldsymbol{\Lambda}; \mathbf{W}, \nu)$ where $\mathbf{W} \in \mathbb{R}^{d\times d}$ is a positive definite matrix and $\nu > d - 1$ is real. The marginal distribution of $\boldsymbol{\mu}$ is a multivariate t-distribution i.e $\boldsymbol{\mu} \sim t_{\nu-d+1}\big(\boldsymbol{\mu}; \boldsymbol{\mu}_0, \frac{\mathbf{W}}{\kappa(\nu-d+1)}\big)$.

In the case of the Normal-Wishart distribution and to do Bayesian moment matching, we would require at least four different moments to estimate the four parameters that characterize it. A set of sufficient moments in this case is $S = \{\boldsymbol{\mu}, \boldsymbol{\mu}\boldsymbol{\mu}^T, \boldsymbol{\Lambda}, \Lambda_{ij}^2\}$ where Λ_{ij}^2 is the $(i, j)^{th}$ element

of the matrix $\mathbf{\Lambda}$. The expressions for sufficient moments are given by

$$\begin{aligned}
\mathbb{E}[\boldsymbol{\mu}] &= \boldsymbol{\mu}_0 \\
\mathbb{E}[(\boldsymbol{\mu} - \boldsymbol{\mu}_0)(\boldsymbol{\mu} - \boldsymbol{\mu}_0)^T] &= \frac{\kappa + 1}{\kappa(\nu - d - 1)}\mathbf{W}^{-1} \\
\mathbb{E}[\mathbf{\Lambda}] &= \nu\mathbf{W} \\
Var(\Lambda_{ij}) &= \nu(W_{ij}^2 + W_{ii}W_{jj})
\end{aligned} \tag{3.14}$$

3.3.2 Online Bayesian Moment Matching for Continuous SPNs

Similar to the discrete case, Algorithm 2 shows how $coef_i^0, coef_i^1$ are computed at each node i in a top-down pass of the network; assuming that a bottom-up pass was done to compute the likelihood at each node in the network. Once we have the coefficients, we can compute each moment as follows. For leaf nodes that model multivariate Gaussian distributions, the set of sufficient moments is $S_i = \{\boldsymbol{\mu}_i, \boldsymbol{\mu}_i\boldsymbol{\mu}_i^T, \mathbf{\Lambda}_i, \Lambda_{i_{kl}}^2\}\ \forall i \in leafNodes$ and we get:

$$M_{P(\mathbf{\Theta}|\mathbf{x})}(s) = \int_s s\mathcal{NW}_i(\mu_i, \Lambda_i|\alpha_i, \kappa_i, W_i, \nu_i)\Big(coef_i^0 + coef_i^1\mathcal{N}(x|\mu_i, \Lambda_i^{-1})\Big)ds \ \ \forall s \in S_i$$

For the rest of the nodes:

$$M_{P(\mathbf{\Theta}|\mathbf{x})}(w_{ij}^k) = \int_{w_{i\cdot}} w_{ij}^k Dir(\mathbf{w}_{i\cdot}|\boldsymbol{\alpha}_{i\cdot})\Big(coef_i^0 + coef_i^1\sum_{j'} w_{ij'}V_{j'}(\mathbf{x})\Big)dw_{i\cdot}\ \forall i \in sumNodes$$

After computing the moments, any mixture can be approximated by a simple distribution by matching sufficient number of moments.

3.4 Experimental Results

3.4.1 Discrete SPNs

We evaluated oBMM on 2 sets of datasets; 20 small datasets and 4 large datasets. The 20 datasets span diverse sets of domains, and the number of variables range from 16 to 1556 binary variables. These datasets were used before for comparisons in [16, 50]. The 4 large datasets are bag of words datasets and they are publicly available online from the UCI machine learning repository. Each dataset contains a collection of documents and the corresponding word counts. The number of variables (dictionary size) ranges from 6906 to 102660 variables. The variables are binarized by ignoring the word count and only used 0 or 1 for the absence or presence of a word in a document.

In order to evaluate the performance of oBMM, we evaluated it and compared it to three online algorithms: stochastic gradient descent (SGD), online exponentiated gradient (oEG) and online expectation maximization (oEM). We measure both the quality of the three algorithms in terms of average log-likelihood scores on the held-out test data sets and their scalability in terms of running time. To test the statistical significance of the results, we apply the Wilcoxon signed rank test [55] to compute the p-value and report statistical significance with p-value less than 0.05. The log-likelihood scores for the structural learning algorithm from LearnSPN [16] are reported as a reference for the comparison of the three online parameter learning algorithms. The online algorithms are not expected to beat a structural learning algorithm, but we will show that online algorithms can outperform LearnSPN for the large datasets in terms of running time and accuracy.

Since we don't do structural learning, a fixed structure is used to perform the experiments. We used a simple structure generator which can take 2 hyperparameters (number of variables, and the depth of the SPN) and generate the SPN accordingly. The generator keeps track of the scope at each node and it starts from a sum node as the root node that has all variables in the scope. Recursively, for each sum node, a number of children product nodes are generated with the same scope as the sum node, and for each product node, a number of children sum nodes are generated while randomly factoring the scope among the children. When the level at a product node is 2 or less, a number of children sum nodes are generated such that each child has only a single variable in its scope. Finally, a sum node that contains a single variable in its scope, two leaf nodes are generated corresponding to the indicators of the variables. For the small datasets, SPNs of depth 6 are used, while for the large datasets SPNs of depth 4 are used.

Online Bayesian Moment matching (oBMM) doesn't have any hyper-parameters to tune, except for the prior which we set randomly by selecting hyperparameters α_{ij} uniformly at random in $[0, 1]$. For the online Distributed Bayesian Moment Matching (oDMM), the training set is partitioned into 10 smaller sets, and each set is sent to a different machine to be processed. Once the machines finish, the output sum-product networks are collected and combined into a single sum-product network. For SGD and oEG, we use backtracking line search to shrink the step size if the log-likelihood scores on the training set decreases. The initial step size is set to 1.0 for all data sets. oEM does not require a learning rate nor weight shrinking. To avoid possible numeric underflow we use a smoothing constant equal to 0.01 in all the experiments.

Table 4.1 shows the average log-likelihoods on the test sets for various algorithms. oBMM outperforms the rest of the online algorithms in 19 out of the 20 datasets. The results show

Table 3.1: Log-likelihood scores on 20 data sets. The best results among oBMM, SGD, oEM, and oEM are highlighted in bold font. $\uparrow$ ($\downarrow$) indicates that the method has significantly better (worse) log-likelihoods than oBMM under Wilcoxon signed rank test with p value < 0.05.

Dataset	Var#	LearnSPN	oBMM	SGD	oEM	oEG
NLTCS	16	-6.11	**-6.07**	$\downarrow$-8.76	$\downarrow$-6.31	$\downarrow$-6.85
MSNBC	17	-6.11	**-6.03**	$\downarrow$-6.81	$\downarrow$-6.64	$\downarrow$-6.74
KDD	64	-2.18	**-2.14**	$\downarrow$-44.53	$\downarrow$-2.20	$\downarrow$-2.34
PLANTS	69	-12.98	**-15.14**	$\downarrow$-21.50	$\downarrow$-17.68	$\downarrow$-33.47
AUDIO	100	-40.50	**-40.7**	$\downarrow$-49.35	$\downarrow$-42.55	$\downarrow$-46.31
JESTER	100	-53.48	**-53.86**	$\downarrow$-63.89	$\downarrow$-54.26	$\downarrow$-59.48
NETFLIX	100	-57.33	**-57.99**	$\downarrow$-64.27	$\downarrow$-59.35	$\downarrow$-64.48
ACCIDENTS	111	-30.04	**-42.66**	$\downarrow$-53.69	-43.54	$\downarrow$-45.59
RETAIL	135	-11.04	**-11.42**	$\downarrow$-97.11	$\downarrow$-11.42	$\downarrow$-14.94
PUMSB-STAR	163	-24.78	**-45.27**	$\downarrow$-128.48	$\downarrow$-46.54	$\downarrow$-51.84
DNA	180	-82.52	**-99.61**	$\downarrow$-100.70	$\downarrow$-100.10	$\downarrow$-105.25
KOSAREK	190	-10.99	**-11.22**	$\downarrow$-34.64	$\downarrow$-11.87	$\downarrow$-17.71
MSWEB	294	-10.25	**-11.33**	$\downarrow$-59.63	$\downarrow$-11.36	$\downarrow$-20.69
BOOK	500	-35.89	**-35.55**	$\downarrow$-249.28	$\downarrow$-36.13	$\downarrow$-42.95
MOVIE	500	-52.49	**-59.50**	$\downarrow$-227.05	$\downarrow$-64.76	$\downarrow$-84.82
WEBKB	839	-158.20	**-165.57**	$\downarrow$-338.01	$\downarrow$-169.64	$\downarrow$-179.34
REUTERS	889	-85.07	**-108.01**	$\downarrow$-407.96	-108.10	$\downarrow$-108.42
NEWSGROUP	910	-155.93	**-158.01**	$\downarrow$-312.12	$\downarrow$-160.41	$\downarrow$-167.89
BBC	1058	-250.69	-275.43	$\downarrow$-462.96	**-274.82**	$\downarrow$-276.97
AD	1556	-19.73	**-63.81**	$\downarrow$-638.43	$\downarrow$-63.83	$\downarrow$-64.11

Table 3.2: Log-lielihood scores on 4 large datasets. The best results are highlighted in bold font, and dashes "-" are used to indicate that an algorithm didn't finish or couldn't load the dataset into memory.

Dataset	Var#	LearnSPN	oBMM	oDMM	SGD	oEM	oEG
KOS	6906	-444.55	**-422.19**	-437.30	-3492.9	-538.21	-657.13
NIPS	12419	-	**-1691.87**	-1709.04	-7411.20	-1756.06	-3134.59
ENRON	28102	-	**-518.842**	-522.45	-13961.40	-554.97	-14193.90
NYTIMES	102660	-	-1503.65	-1559.39	-43153.60	**-1189.39**	-6318.71

Table 3.3: Running time in minutes on 4 large datasets. The best running time is highlighted in bold font, and dashes "-" are used to indicate that an algorithm didn't finish or couldn't load the dataset into memory.

Dataset	Var#	LearnSPN	oBMM	oDMM	SGD	oEM	oEG
KOS	6906	1439.11	89.40	**8.66**	162.98	59.49	155.34
NIPS	12419	-	139.50	**9.43**	180.25	64.62	178.35
ENRON	28102	-	2018.05	**580.63**	876.18	694.17	883.12
NYTIMES	102660	-	12091.7	**1643.60**	5626.33	5540.40	6895.00

that oBMM has significantly better log-likelihood on most of the datasets. Surprisingly, oBMM matches the performance of LearnSPN in 11 datasets despite the fact that the structure used for oBMM is generated randomly. The advantage of online algorithms over batch algorithms including LearnSPN appears when the dataset size increases to the limit since it becomes hard to fit it in memory. Tables 4.2 and 3.3 show the likelihood scores and the running times of different methods on large datasets. Dashes are used to indicate that the algorithm didn't finish in a week or the training data could not fit in memory. We were able to run LearnSPN only on the KOS dataset since the rest of the datasets didn't finish in a week or they were too big to fit in memory. Both oBMM and oDMM outperform LearnSPN, which runs 16 times slower than oBMM and 166 times slower than oDMM. oBMM and oDMM also outperform SGD by a large margin on the large data sets. In terms of the running time, oDMM is significantly faster than the rest of the algorithms since we are distributing the training data over multiple machines.

3.4.2 Continuous SPNs with Gaussian Leaves

We performed experiments on 7 real-world datasets from the UCI machine learning repository [29] and the function approximation repository [18] that span diverse domains with 3 to 48

attributes. We compare the performance of online algorithms for Sum-Product Networks with continuous variables as well as other generative deep networks like Stacked Restricted Boltzman Machines (SRBMs) and Generative Moment Matching Networks (GenMMNs). We evaluated the performance of SPNs using both online EM (oEM) and online Bayesian Moment Matching (oBMM). Stochastic gradient descent and exponentiated gradient descent could also be used, but since their convergence rate was shown to be much slower than that of oEM [48, 58], we do not report results for them. Since there are no structure learning algorithms for continuous SPNs, we used a random structure and a structure equivalent to a Gaussian Mixture Model for the experiments using both oEM and oBMM. In both cases, the leaves are multivariate Gaussian distributions. Before discussing the results, we provide additional details regarding the training and likelihood computation for oEM, SRBM and GenMMN.

Online EM. [40] describe how to estimate the leaf parameters in a batch mode by EM. To adapt their algorithm to the online setting, we follow the framework introduced by [34] where increments of the sufficient statistics are accumulated in a streaming fashion. More specifically, we sample one data instance at a time to compute the corresponding local sufficient statistics from that instance. Then we accumulate the local sufficient statistics as increments to the globally maintained sufficient statistics. The update formula is obtained by renormalization according to the global sufficient statistics. This online variant of EM is also known as incremental EM [28]. There are no hyperparameters to be tuned in the online EM (oEM) algorithm for SPNs. In the experiments, we randomly initialize the weights of SPNs and apply oEM to estimate the parameters. For each data set, oEM processes each data instance exactly once.

SRBM. We trained stacked restricted Boltzmann machine (SRBM) with 3 hidden layers

of binary nodes where the first and second hidden layers have the same number of nodes as the dimensionality of the data and the third hidden layer has four times as many nodes as the dimensionality of the data. This yielded a number of parameters that is close to the number of parameters of the random SPNs for a fair comparison. This $1 : 1 : 4$ structure is taken from [19]. Computing the exact likelihood of SRBMs is intractable. Hence, we used Gibbs sampling to approximate the log-likelihood. We averaged 1000 runs of 500 iterations of Gibbs sampling where in each iteration the value of each hidden node is re-sampled given the values of the adjacent nodes. At the end of each run, the log-likelihood of each data instance is computed given the last values sampled for the hidden nodes in the layer above the leaf nodes.

Gen MMN. We trained fully-connected networks with multiple layers. The number of layers and hidden units per layer depends on the training dataset and was selected to match the number of parameters in random SPNs for a fair comparison. Inputs to the networks are vectors of elements drawn independently and uniformly at random in $[-1, 1]$. The activation function for the hidden layers is ReLU and for the output layer is sigmoid. We used the Gaussian kernel in MMD and followed the algorithm described in [27] for training. To compute the average log-likelihood reported in Table 4.1, we generate $10,000$ samples using the trained model and fit a kernel density estimator to these samples. The average log-likelihood of the test set is then evaluated using this estimated density. The kernel for our density estimator is Gaussian and its parameter is set to maximize the log-likelihood of the validation set. This technique was used since there is no explicit probability density function for this model. However, as shown in [53] this method only provides an approximate estimate of the log-likelihood and therefore the log-likelihood reported for GenMMNs in Table 4.1 may not be directly comparable to the likelihood

of other models.

Table 4.1 reports the average log-likelihoods (with standard error) of each model on the 7 real-world datasets using 10-fold cross-validation. No results are reported for random SPNs trained by oBMM and oEM when the number of attributes is too small (less than 5) to consider a structure that is more complex than GMMs. No results are reported for GMM SPNs trained by oBMM and oEM when the number of attributes is large (48) in comparison to the amount of data since the number of parameters to be trained in the covariance matrix of each Gaussian component is square in the number of attributes. We can think of GMMs as a baseline structure for SPNs when the dimensionality of the data is low and random SPNs as more data efficient models for high-dimensional datasets. Note also that since the data is continuous, the log-likelihood can be positive when a model yields a good fit as observed for the Flow Size dataset. The best results for each dataset are highlighted in bold. oBMM consistently outperforms oEM on each dataset. In comparison to other deep models, the results are mixed and simply suggest that continuous SPNs trained by oBMMs are competitive with SRBMs and GenMMNs. Note that it is remarkable that SPNs with a random or basic GMM structure can compete with other deep models in the absence of any structure learning. Note also that SPNs constitute a tractable probabilistic model in which inference can be done exactly in linear time. This is not the case for SBRMs and GenMMNs where inference must be approximated.

Table 3.4: Log-likelihood scores on real-world data sets. The best results are highlighted in bold font.

Dataset # of vars	Flow Size 3	Quake 4	Banknote 4	Abalone 8	Kinematics 8	CA 22	Sensorless Drive 48
oBMM (random)	-	-	-	-1.82 ± 0.19	-11.19 ± 0.03	-2.47 ± 0.56	**1.58** ± 1.28
oEM (random)	-	-	-	-11.36 ± 0.19	-11.35 ± 0.03	-31.34 ± 1.07	-3.40 ± 6.063
oBMM (GMM)	**4.80** ± 0.67	-3.84 ± 0.16	-4.81 ± 0.13	**-1.21** ± 0.36	-11.24 ± 0.04	**-1.78** ± 0.59	-
oEM (GMM)	-0.49 ± 3.29	-5.50 ± 0.41	-4.81 ± 0.13	-3.53 ± 1.68	-11.35 ± 0.03	-21.39 ± 1.58	-
SRBM	-0.79 ± 0.004	**-2.38** ± 0.01	-2.76 ± 0.001	-2.28 ± 0.001	**-5.55** ± 0.02	-4.95 ± 0.003	-26.91 ± 0.03
GenMMN	0.40 ± 0.007	-3.83 ± 0.21	**-1.70** ± 0.03	-3.29 ± 0.10	-11.36 ± 0.02	-5.41 ± 0.14	-29.41 ± 1.185

3.5 Conclusion

SPNs gained popularity for being able to provide exact marginal and conditional inference in linear time, however learning the parameters in a tractable way is still challenging. In this chapter, I explored online algorithms for learning the parameters of an SPN tractably. I also proposed online Bayesian Moment Matching as an online framework for parameter learning, and I showed how to distribute the algorithm over many machines. I showed how oBMM outperforms the rest of the algorithms. I also showed that distributing the algorithm over multiple machines is very effective when running time is the bottleneck.

For SPNs with Gaussian leaves, I presented the first online algorithm for parameter learning. I also reported on the first comparison between continuous SPNs and other generative deep models such as SRBMs and GenMMNs.

Chapter 4

Discriminative Training of Sum-Product Networks by Extended Baum-Welch

Various generative learning algorithms have been designed to estimate the parameters of SPNs, including Gradient Descent and hard EM [45], soft EM [37], Bayesian moment matching [48], collapsed variational Bayes [56], sequential monomial approximations and the concave-convex procedure [58]. Discriminative training of SPNs by gradient descent was introduced by [15]. Although gradient descent is tractable, convergence can be quite slow since it uses first order approximations. [1] introduced a novel discriminative algorithm for SPNs that learns the structure of the SPN while extracting features that are maximally correlated with the labels. The algorithm has been shown to perform well compared to generative structure learning algorithms. However, it is a batch algorithm that recursively performs singular value decompositions that are very expensive.

In this chapter, we present a novel algorithm to train SPNs discriminatively based on the Extended Baum-Welch technique. While Expectation Maximization and Baum-Welch are equivalent in generative training, they cannot be used directly in discriminative training and their extensions for maximizing conditional likelihoods are not the same. Extended Baum-Welch (for discriminative training) [17] is simpler both conceptually and computationally than conditional EM (for discriminative training) [21, 22, 51] and therefore has become the most popular approach to train HMMs discriminatively. Extended Baum-Welch provides a general approach to optimize rational functions such as conditional likelihoods. It also offers faster convergence than gradient descent while guaranteeing monotonic improvement at each iteration [35]. In order to apply Extended Baum-Welch to discriminative SPNs, we will formulate the conditional distribution as a rational function. We will develop the algorithm for SPNs with both multinomial and univariate Gaussian distributions at the leaves.

The chapter is structured as follows, we first review Baum-Welch and Extended Baum-Welch algorithms. Section 4.2 introduces discriminative SPNs and explains how to compute the conditional likelihood for a classification task. Then, update formulas for discriminative gradient descent and Extended Baum-Welch are derived for SPNs with discrete and continuous variables. Section 4.3 presents three sets of experiments that we carried out to evaluate the performance of our algorithm.

4.1 Extended Baum-Welch Algorithm

The Baum-Welch algorithm was introduced in 1970 to estimate the parameters for HMMs [6]. The algorithm was based on the Baum-Eagon inequality [5], which monotonically maximizes polynomials that satisfy certain conditions. The algorithm was then extended to maximizing rational functions (i.e., ratio of two polynomial functions) [17], which made it very useful in discriminative training settings. Extended Baum-Welch (EBW) was used to discriminatively train HMMs, GMMs, as well as discrete distributions [44, 24, 36] In this Section, we will review the original Baum-Welch algorithm, then we will explain how it was extended to work for rational polynomials. Finally, we will show how to use EBW to train SPNs discriminatively in Section 4.2.

Theorem 1. *[5] Let $S(\theta)$ be a homogeneous degree d polynomial with non-negative coefficients. Let $\bar{\theta} = \{\bar{\theta}_{ij}\}$ be any point in the domain $\mathcal{D} : \sum_j \theta_{ij} = 1, \forall i$. Let $\hat{\theta} = T(\bar{\theta}) = T(\{\bar{\theta}_{ij}\})$ be a transformation function such that*

$$\hat{\theta}_{ij} = \frac{\bar{\theta}_{ij} \frac{\partial S}{\partial \theta_{ij}}(\bar{\theta})}{\sum_j \bar{\theta}_{ij} \frac{\partial S}{\partial \theta_{ij}}(\bar{\theta})}, \tag{4.1}$$

where $\sum_j \bar{\theta}_{ij} \frac{\partial S}{\partial \theta_{ij}}(\bar{\theta}) \neq 0$, $\frac{\partial S}{\partial \theta_{ij}}(\bar{\theta})$ is the value of $\frac{\partial S}{\partial \theta_{ij}}$ at $\bar{\theta}$. Then, $S(\hat{\theta}) > S(\bar{\theta})$ unless $T(\bar{\theta}) = \bar{\theta}$.

Theorem 1 is applied iteratively to optimize a polynomial $S(\theta)$. The transformation $T(\bar{\theta})$ is called a growth transform since it increases $S(\theta)$ monotonically. In discrete HMMs and other discrete mixture models, the likelihood function is a polynomial in the parameters θ and therefore Eq. 4.1 can be used to iteratively improve the parameters in a way that the likelihood

monotonically improves. Interestingly, we obtain the same update formula as for Expectation-Maximization.

The growth transform can be extended to rational functions, $R_d(\theta) = \frac{Num(\theta)}{Den(\theta)}$, where both the numerator, $Num(\theta)$, and the denominator, $Den(\theta)$, are polynomials.

Theorem 2. *[17] Let $R_d(\theta) = \frac{Num(\theta)}{Den(\theta)}$ be a rational function. Let $\bar{\theta} = \{\bar{\theta}_{ij}\}$ be any point in the domain $\mathcal{D} : \sum_j \theta_{ij} = 1, \forall i$. Let's construct polynomials $Q(\theta)$ and $S(\theta)$ as follows*

$$\begin{aligned} Q(\theta) &= Num(\theta) - R_d(\bar{\theta}) Den(\theta) \\ S(\theta) &= Q(\theta) + C(\theta) \end{aligned} \tag{4.2}$$

where $C(\theta) = c\left[\sum_{i,j} \theta_{ij} + 1\right]^d$, c is chosen such that it cancels all negative coefficients in $Q(\theta)$, and d is the degree of $Q(\theta)$. Based on the above construction, Theorem 1 can be applied to $S(\theta)$ such that $R_d(\hat{\theta}) > R_d(\bar{\theta})$, unless $T(\bar{\theta}) = \bar{\theta}$.

In discriminative learning, the conditional likelihood can typically be expressed as a rational function $R(\theta)$ (i.e., the data likelihood divided by the marginal of the inputs). In the next section, we will show how to construct a polynomial $S(\theta)$ from the rational function corresponding to the conditional likelihood of SPNs and then apply the growth function to improve the conditional likelihood monotonically.

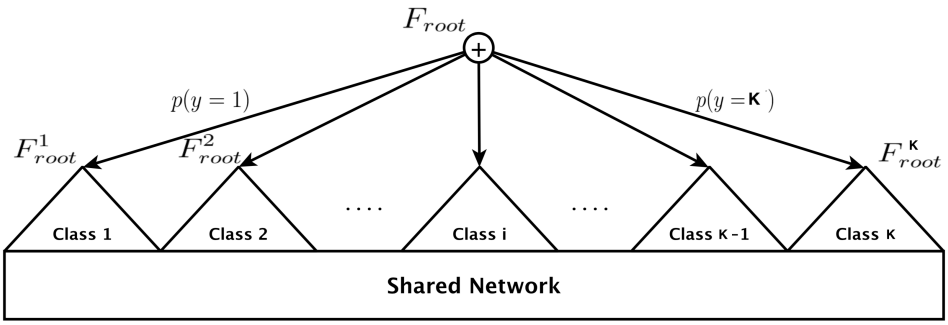


Figure 4.1: Discriminative SPN architecture.

4.2 Discriminative Sum-Product Networks

Let the training set be $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_N\}$ and the corresponding labels $\mathbf{Y} = \{y_1, ..., y_N\}$, where $\mathbf{x}_i \in \{0, 1\}^M$, $y_i \in \{0, ..., K\}$, N is the number of training examples, M is the feature size, and K is the number of class labels. In this section, we first provide some preliminary derivations for discriminative SPNs, then extend the discriminative gradient descent technique proposed by [15] to continuous SPNs with Gaussian leaves, and finally we describe our new discriminative learning technique based on Extended Baum-Welch.

In discriminative training, we maximize the conditional probability distribution $P(y|\mathbf{x})$. To do that, we use the discriminative SPN architecture shown in Figure 4.1 where there is a sub-SPN, SPN_y, for each class y , and sub-SPNs can share part of the network. Each sub-SPN models the likelihood of an observation given the class label, $F^y_{root}(\mathbf{x}) = p(\mathbf{x}|y)$. Evaluating the whole network gives us the probability of an observation, $p(\mathbf{x}) = \sum_y p(y) F^y_{root}(\mathbf{x})$. According to Bayes rule, the conditional probability can be computed as follows:

$$
\begin{aligned}
P(y|\mathbf{x}) &= \frac{p(y)p(\mathbf{x}|y)}{p(\mathbf{x})} \\
&= \frac{p(y)F^{y}_{root}(\mathbf{x})}{F_{root}(\mathbf{x})}
\end{aligned}
\tag{4.3}
$$

The label associated with the sub-SPN that maximizes the conditional probability distribution is selected.

$$
\arg max_{y=1,\ldots,Y} p(y)F^{y}_{root}(\mathbf{x}) \tag{4.4}
$$

In the training phase, the posterior $P(\mathbf{Y}|\mathbf{X})$ is maximized. The posterior is computed in terms of the prior and the likelihoods as follows

$$
\begin{aligned}
P(\mathbf{Y}|\mathbf{X}) &= \prod_{n=1}^{N} P(y_n|\mathbf{x}_n) \\
&= \prod_{n=1}^{N} \frac{P(y_n)F^{y_n}_{root}(\mathbf{x}_n)}{F_{root}(\mathbf{x}_n)}
\end{aligned}
\tag{4.5}
$$

Similarly $log(P(\mathbf{Y}|\mathbf{X}))$ is computed as follows.

$$
log(P(\mathbf{Y}|\mathbf{X})) = \sum_{n=1}^{N} log(p(y_n)F^{y_n}_{root}(\mathbf{x}_n)) - log\big(F_{root}(\mathbf{x}_n)\big) \tag{4.6}
$$

The above equations show that discriminative training aims at maximizing the likelihood of the correct class, similar to generative training, while minimizing the likelihoods of the remaining classes. Estimating $P(y)$ can be done easily and robustly by normalizing the class frequencies in the training data. Estimating the parameters of $P(\mathbf{x}|y)$, which are the weights of $F^{y}_{root}(\mathbf{x})$, is

harder and usually doesn't have a closed form solution. Iterative methods are used in this case.

We will use $F_j^y(\mathbf{x})$ to refer to the value of the sub-SPN associated with class y at node j. Throughout the derivations, we assume that the SPN is always normalized to ensure that $P(\mathbf{x}|y) = F_{root}^y(\mathbf{x})$. This can be done by normalizing the weights after each iteration.

Finally, we will need to compute the partial derivative of the log likelihood with respect to an arbitrary parameter θ^y in the SPN, where superscript y indicates that θ^y is a parameter in the sub-SPN F_{root}^y. The derivative of the log likelihood can be obtained as follows:

$$\begin{aligned}\frac{\partial log(P(\mathbf{Y}|\mathbf{X}))}{\partial \theta^y} &= \sum_{n=1}^{N} \frac{p(y_n)\frac{\partial F_{root}^{y_n}}{\partial \theta^y}(\mathbf{x}_n)}{p(y_n)F_{root}^{y_n}(\mathbf{x}_n)} - \frac{\frac{\partial F_{root}}{\partial \theta^y}(\mathbf{x}_n)}{F_{root}(\mathbf{x}_n)} \\ &= \sum_{n=1}^{N} \frac{1}{F_{root}^{y_n}(\mathbf{x}_n)}\frac{\partial F_{root}^{y_n}}{\partial \theta^y}(\mathbf{x}_n) - \frac{1}{F_{root}(\mathbf{x}_n)}\frac{\partial F_{root}}{\partial \theta^y}(\mathbf{x}_n)\end{aligned} \tag{4.7}$$

Since we know that $\frac{\partial F_{root}^{y_n}}{\partial \theta^y} = 0$ when $y \neq y_n$, we can rewrite $\frac{\partial F_{root}^{y_n}}{\partial \theta^y}$ as follows:

$$\frac{\partial F_{root}^{y_n}}{\partial \theta^y} = \mathbb{1}_{[y \in SPN_{y_n}]}\frac{\partial F_{root}^{y}}{\partial \theta^y} \tag{4.8}$$

Also, since $F_{root}(\mathbf{x}_n) = \sum_y p(y)F_{root}^y(\mathbf{x}_n)$, we can rewrite $\frac{\partial F_{root}}{\partial \theta^y}(\mathbf{x}_n)$ as follows:

$$\frac{\partial F_{root}}{\partial \theta^y}(\mathbf{x}_n) = p(y)\frac{\partial F_{root}^{y}}{\partial \theta^y}(\mathbf{x}_n) \tag{4.9}$$

Finally, based on Eq. 4.8 and 4.9 we can rewrite Eq. 4.7 as follows:

$$\begin{aligned}\frac{\partial log(P(\mathbf{Y}|\mathbf{X}))}{\partial \theta^y} &= \sum_{n=1}^{N} \frac{\partial F^y_{root}}{\partial \theta^y}(\mathbf{x}_n)\left[\frac{\mathbb{1}_{[y=y_n]}}{F^{y_n}_{root}(\mathbf{x}_n)} - \frac{p(y)}{F_{root}(\mathbf{x}_n)}\right] \\ &= \sum_{n=1}^{N} \frac{\partial F^y_{root}}{\partial \theta^y}(\mathbf{x}_n)\gamma_n\end{aligned} \tag{4.10}$$

where $\gamma_n \geq 0$. Throughout the rest of this section, we will use the above equation whenever we need the gradient of the log of the conditional likelihood.

4.2.1 Discriminative Learning for SPNs Using Gradient Descent

[15] showed how to train edge weights for SPNs discriminatively by taking the gradient of the conditional log likelihood $log(P(\mathbf{Y}|\mathbf{X}))$. We briefly state how to compute discriminative gradients in continuous SPNs with Gaussian leaves.

Using Eq. 4.10 and knowing that $\frac{\partial F^y_{root}}{\partial w^y_{ij}}(\mathbf{x}) = F^y_j(\mathbf{x}_n)\frac{\partial F^y_{root}}{\partial F^y_i}(\mathbf{x}_n)$, the following formula can be used to update each edge weight w^y_{ij} by taking a small step η in the direction of the gradient.

$$\begin{aligned}w^y_{ij} &\leftarrow w^y_{ij} + \eta\frac{\partial log(P(\mathbf{Y}|\mathbf{X}))}{\partial w^y_{ij}} \\ &= w^y_{ij} + \eta\sum_n F^y_j(\mathbf{x}_n)\frac{\partial F^y_{root}}{\partial F^y_i}(\mathbf{x}_n)\gamma^y_n\end{aligned} \tag{4.11}$$

Similarly, for leaf univariate Gaussian distributions, μ^y_{ij} and $(\sigma^2)^y_{ij}$ can be updated by taking a small step η in the direction of the gradient.

$$\mu^y_{ij} \leftarrow \mu^y_{ij} + \eta\sum_n \mathcal{N}^y_{ij}(x_{nj})\frac{x_{nj} - \mu^y_{ij}}{(\sigma^2)^y_{ij}}\frac{\partial F^y_{root}}{\partial F^y_i}(\mathbf{x}_n)\gamma^y_n \tag{4.12}$$

$$(\sigma^2)^y_{ij} \leftarrow (\sigma^2)^y_{ij} + \eta \sum_n \frac{\mathcal{N}^y_{ij}(x_{nj})}{2(\sigma^2)^y_{ij}} \left[\frac{(x_{nj} - \mu^y_{ij})^2}{(\sigma^2)^y_{ij}} - 1 \right] \frac{\partial F^y_{root}}{\partial F^y_i}(\mathbf{x}_n)\gamma^y_n \tag{4.13}$$

4.2.2 Discriminative Learning for SPNs using Extended Baum-Welch

We first derive the Baum-Welch algorithm to maximize the log likelihood for SPNs in a generative way. Theorem 1 can be applied to SPNs assuming that the network is normalized. In that case, the polynomial $S(\theta)$ is the likelihood of the data, which corresponds to a product of network polynomials, i.e., $P(\mathbf{X}|\mathbf{Y}) = \prod_n F_{root}(\mathbf{x}_n)$. The parameters $\theta = \{w_{ij}\}$ of the polynomial satisfy the condition $\sum_j w_{ij} = 1$ since the sum of the weights for each sum node is one. To apply Theorem 1, we need to deal with the sum of the log-likelihoods, $log(P(\mathbf{X}|\mathbf{Y}))$, instead of the likelihood of the data, $P(\mathbf{X}|\mathbf{Y})$, since $log(P(\mathbf{X}|\mathbf{Y}))$ is easier to differentiate. We have

$$\frac{\partial log(P(\mathbf{X}|\mathbf{Y}))}{\partial w_{ij}} = \frac{1}{P(\mathbf{X}|\mathbf{Y})} \frac{\partial P(\mathbf{X}|\mathbf{Y})}{\partial w_{ij}} \tag{4.14}$$

Hence, we can rewrite Eq. 4.1 as follows.

$$\begin{aligned} \hat{w}_{ij} &= \frac{w_{ij} P(\mathbf{X}|\mathbf{Y}) \frac{\partial log(P(\mathbf{X}|\mathbf{Y}))}{\partial w_{ij}}}{\sum_j w_{ij} P(\mathbf{X}|\mathbf{Y}) \frac{\partial log(P(\mathbf{X}|\mathbf{Y}))}{\partial w_{ij}}} \\ &= \frac{w_{ij} \frac{\partial log(P(\mathbf{X}|\mathbf{Y}))}{\partial w_{ij}}}{\sum_j w_{ij} \frac{\partial log(P(\mathbf{X}|\mathbf{Y}))}{\partial w_{ij}}} \end{aligned} \tag{4.15}$$

The above formula is the same update formula obtained by Expectation-Maximization and the Convex-Concave Procedure (CCCP) [58].

In discriminative training for SPNs, applying Expectation-Maximization or CCCP doesn't lead to a closed form update formula [15]. [21, 22] derived a conditional version of EM that turned out to be complicated and computationally demanding. [51] derived a simpler and faster update formula, but it requires second order derivatives, which is not tractable in large models with many parameters such as SPNs. In contrast, EBW can be applied to maximize the conditional likelihood with a closed form formula. Before applying Theorem 2 to SPNs, we will do the same as we did above to work with $\frac{\partial log R_d(\theta)}{\partial \theta_{ij}}$. We start by taking the derivative of $S(\theta)$ in Eq. 4.2 with respect to the parameters θ_{ij}

$$\frac{\partial S(\theta)}{\partial \theta_{ij}} = \frac{\partial Num(\theta)}{\partial \theta_{ij}} - R_d(\bar{\theta})\frac{\partial Den(\theta)}{\partial \theta_{ij}} + \frac{\partial C(\theta)}{\partial \theta_{ij}} \tag{4.16}$$

where $\frac{\partial C(\theta)}{\partial \theta_{ij}} = cd\left[\sum_{i,j}\theta_{ij} + 1\right]^{d-1}$ is also a constant.

Since,

$$\begin{aligned}\frac{\partial log(R_d(\theta))}{\partial \theta_{ij}} &= \frac{1}{Num(\theta)}\frac{\partial Num(\theta)}{\partial \theta_{ij}} - \frac{1}{Den(\theta)}\frac{\partial Den(\theta)}{\partial \theta_{ij}} \\ &= \frac{1}{Num(\theta)}\left[\frac{\partial Num(\theta)}{\partial \theta_{ij}} - R_d(\theta)\frac{\partial Den(\theta)}{\partial \theta_{ij}}\right]\end{aligned} \tag{4.17}$$

Then we obtain the following equation.

$$\begin{aligned}\frac{\partial S(\bar{\theta})}{\partial \theta_{ij}} &= Num(\bar{\theta})\left[\frac{\partial log R_d}{\partial \theta_{ij}}(\bar{\theta}) + \frac{1}{Num(\bar{\theta})}\frac{\partial C(\bar{\theta})}{\partial \theta_{ij}}\right] \\ &= Num(\bar{\theta})\left[\frac{\partial log R_d}{\partial \theta_{ij}}(\bar{\theta}) + D\right]\end{aligned} \tag{4.18}$$

Substituting Eq. 4.18 in Equation 4.1, we get the following update formula

$$\hat{\theta}_{ij} = \frac{\bar{\theta}_{ij}\left[\frac{\partial log R_d}{\partial \theta_{ij}}(\bar{\theta}) + D\right]}{\sum_j \left[\bar{\theta}_{ij}\frac{\partial log R_d}{\partial \theta_{ij}}(\bar{\theta})\right] + D} \tag{4.19}$$

where $D = \frac{1}{Num(\bar{\theta})}\frac{\partial C(\bar{\theta})}{\partial \theta_{ij}}$. D controls how different $\hat{\theta}_{ij}$ is from $\bar{\theta}_{ij}$. Small values for D allow $\hat{\theta}_{ij}$ to change freely, while large values restrict the magnitude of changes. In practice, D is chosen such that $D < \frac{1}{Num(\theta)}\frac{\partial C(\bar{\theta})}{\partial \theta_{ij}}$ for faster convergence.

To apply EBW to SPNs, we have to define the rational function R_d, which in this case is the posterior $P(\mathbf{Y}|\mathbf{X})$. The parameters θ of the posterior are the weights w_{ij}^y, the mean μ_{ij}^y, and the variance $(\sigma^2)_{ij}^y$.

For sum nodes, the weights will be updated as follows.

$$\hat{w}_{ij}^y = \frac{w_{ij}^y\left[\sum_n F_j^y(\mathbf{x}_n)\frac{\partial F_{root}^y}{\partial F_i^y}(\mathbf{x}_n)\gamma_n^y\right] + Dw_{ij}^y}{\left[\sum_{j\in Children(i)}\sum_n F_j^y(\mathbf{x}_n)\frac{\partial F_{root}^y}{\partial F_i^y}(\mathbf{x}_n)\gamma_n^y\right] + D} \tag{4.20}$$

[36] proposed a discrete approximation for univariate Gaussian distributions that allows us to update μ_{ij}^y and $(\sigma^2)_{ij}^y$ in the Gaussian leaves of SPNs as follows.

$$\hat{\mu}_{ij}^y = \frac{\left[\sum_n \left(\mathcal{N}_{ij}^y(\mathbf{x}_n)\frac{\partial F_{root}^y}{\partial F_i^y}(\mathbf{x}_n)\gamma_n^y\right)x_{nj}\right] + D\mu_{ij}^y}{\left[\sum_n \mathcal{N}_{ij}^y(\mathbf{x}_n)\frac{\partial F_{root}^y}{\partial F_i^y}(\mathbf{x}_n)\gamma_n^y\right] + D} \tag{4.21}$$

$$(\hat{\sigma^2})_{ij}^{y} = \frac{\left[\sum_n \left(\mathcal{N}_{ij}^{y}(\mathbf{x}_n)\frac{\partial F_{root}^{y}}{\partial F_i^{y}}(\mathbf{x}_n)\gamma_n^{y}\right)x_{nj}^2\right] + D\left[(\sigma^2)_{ij}^{y} + (\mu_{ij}^{y})^2\right]}{\left[\sum_n \mathcal{N}_{ij}^{y}(\mathbf{x}_n)\frac{\partial F_{root}^{y}}{\partial F_i^{y}}(\mathbf{x}_n)\gamma_n^{y}\right] + D} - (\hat{\mu}_{ij}^{y})^2 \tag{4.22}$$

Constant D plays an important role in the discriminative training part. D tries to preserve the previous parameters. The larger D is, the stronger will be the influence of the previous parameters on the current ones. Choosing constant D is not trivial since small values can prevent convergence, while large values induce slow convergence. A rule of thumb is to start from a small value and to increase it after each epoch.

4.3 Experiments

To evaluate Discriminative SPNs using EBW, we carried out three sets of experiments. In the first experiment, we compared EBW to generative EM (discriminative EM requires second order derivatives [51], which is not tractable for SPNs of 1 thousand to 1 million parameters) and discriminative gradient descent. The second experiment aims to illustrate the advantage of SPNs over Support Vector Machines (SVMs) and Neural Networks in the case of missing features. In a third experiment, we trained an SPN on MNIST images using generative EM and discriminative EBW. We sample images from the resulting SPNs, and we show the effect of using discriminative training on the model parameters.

For all experiments, we generated dense SPNs by using a variant of the algorithm proposed in [45]. We recursively construct the SPN structure in a top down fashion as follows. We treat the

Table 4.1: The accuracies of EBW, generative EM, and discriminative GD on the test data of eight different datasets.

Dataset	Var#	Dataset Size	Classes#	Var Type	EBW	genEM	discGD
Banknote	4	1371	2	Binary	**86.13%**	83.94%	**86.13%**
Voice	20	3167	2	Cont	**97.15%**	96.20%	96.20%
Credit Card	29	284806	2	Cont	**99.92%**	99.38%	**99.92%**
Breast Cancer	30	865	2	Cont	**96.42%**	92.85%	91.07%
Sensorless Drive	48	85508	11	Cont	**99.44%**	99.36%	55.41%
Fault Detection	70	14354	41	Cont	**60.45%**	58.67%	58.12%
Activity Recognition	561	10299	6	Cont	**90.53%**	88.66%	76.45%
MNIST	784	70000	10	Binary	**95.07%**	93.35%	62.89%

variables of each problem as a 1D array (or 2D array in the case of MNIST) based on the order of the features in the data. For each sum node, we construct children product nodes corresponding to all splits of the scope in two sub-arrays of variables (all vertical and horizontal splits in two 2D arrays in the case of MNIST). We stop when the scope has a single variable, in which case, a univariate leaf distribution is generated. To control the size of the network, we randomly skip some partitions.

4.3.1 EBW versus Other Parameter Learning Algorithms

In this experiment, we used eight different datasets[1] that span a wide spectrum of domains. The number of classes ranges from 2 to 41, and the number of variables ranges from 4 to 784. For the datasets that don't have training and testing splits, we use 10% of the data for testing and the rest for training. While we applied the algorithm on datasets where the variables are binary and continuous, our implementation can also handle categorical variables.

[1]The datasets are publicly available at archive.ics.uci.edu/ml/ and kaggle.com except fault detection, which is a private dataset collected by Huawei.

EBW has one hyper-parameter D. Initializing D to 0.1 and increasing it after each epoch by 0.1 produces the best results. Generative EM doesn't have any hyper-parameters. For gradient descent, we found that initializing the learning rate to 1 and decreasing it after each epoch by multiplying by 0.9 produces the best results. During the experiments, we limited the number of epochs to 20.

Table 4.1 shows that EBW always outperforms genEM. We also observed that discGD converges quickly to good solutions for small and shallow SPNs, but not deep SPNs, which suggests that it suffers from the gradient vanishing problem.

We explored the convergence speeds for EBW and discGD on the training data. We ran every algorithm for 20 epochs. Figure 4.2 shows the convergence speed and performance for both algorithms. Both algorithms take the same time per epoch. The figure shows that EBW converges faster to a better solution than discGD. Furthermore, discGD struggles to achieve good results consistently as we can see in the Activity Recognition plot where it didn't converge to a solution in 20 epochs while EBW was able to converge after a few epochs.

4.3.2 EBW for Problems with Missing Features

Missing features is a problem that commonly happens in wearable devices where sensors can fail frequently. SPNs can naturally handle missing features by summing out the corresponding unobserved variables when doing inference. We show the robustness of SPNs trained using EBW in the absence of some features. We compare the performance of SPNs to SVMs and Neural Networks.

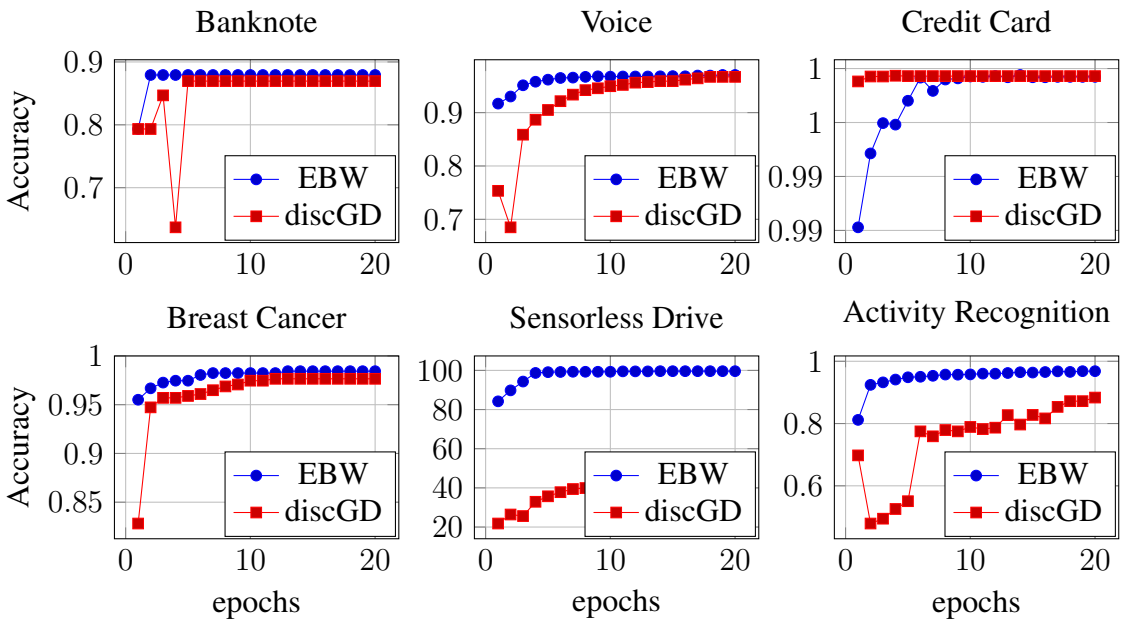


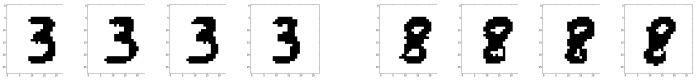
Figure 4.2: Accuracies on training data versus number of epochs for EBW and discriminative GD algorithms. The time per epoch is the same for both algorithms.

Table 4.2: The test accuracies of EBW-trained SPNs, SVMs and NNs on seven datasets.

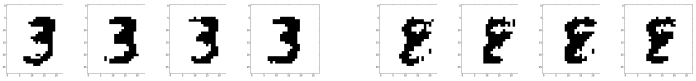
Algorithm	EBW		NN		SVM	
Dataset	0%	50%	0%	50%	0%	50%
Banknote	86.13%	64.90%	86.13%	62.04%	86.13%	54.74%
Voice	97.15%	88.60%	97.46%	88.60%	96.20%	77.53%
Credit Card	99.92%	99.80%	99.87%	99.80%	99.96%	99.73%
Breast Cancer	96.24%	89.28%	94.60%	83.92%	94.64%	87.50%
Sensorless Drive	99.44%	52.03%	96.03%	47.90%	75.50%	12.40%
Fault Detection	60.45%	48.40%	50.01%	46.80%	57.04%	47.09%
Activity Recognition	90.53%	88.59%	93.82%	81.57%	96.23%	61.14%

We randomly set 50% of the features for each instance to be missing, SPNs can handle such missing features by summing/integrating out the leaf distributions. Since SVMs and NNs need values for all features, we set the missing features to their average.[2] For SVMs, a polynomial kernel was used and the penalty constant was tuned for best performance. For NNs, we limited the number of parameters to be equal to the number of parameters in SPNs. We used neural networks with two hidden layers and rectified linear units (ReLU) in each layer. We set the width of each layer such that the number of parameters for the NNs is the same as the SPNs. We used TensorFlow to build and train NNs. We set the number of epochs to 20. Table 4.2 shows that EBW-trained SPNs are consistently more robust to missing features than both SVMs and NNs.

[2]It is possible to deal with missing features in kernel methods in a principled way by modifying the loss function to take into account the uncertainty induced by the missing features, however modeling assumptions are needed and the optimization problem is changed [43]. Alternatively, one can also deal with missing features by casting kernel methods as estimation problems in exponential families, but this yields non-convex optimization problems [52]



(a) Sampled images for digits ’3’ and ’8’ from generatively trained SPNs



(b) Sampled images for digits ’3’ and ’8’ from discriminatively trained SPNs

Figure 4.3: The above samples show the effect of training SPNs discriminatively. The sampled images for digit ’8’ in the discriminative training case illustrate that the SPN for digit ’8’ was tuned to focus on the parts that discriminate the digit ’8’ from the digit ’3’.

4.3.3 Visualizing the Parameters Learned by EBW

This experiment aims at visualizing the parameters learned by both EBW and generative EM by sampling different images from the learned SPNs. We chose two classes, ’3’ and ’8’, with visual similarities from the MNIST dataset. We trained an SPN using generative EM and a second SPN using discriminative EBW. We sampled images from the resulting SPNs to analyze the effect of discriminative EBW on the learned parameters.

As shown in Figure 4.3, the sampled images from the generatively trained SPN resemble the appearance of digit ’3’ and digit ’8’. On the other hand, the sampled images from the discriminatively trained SPN show the parts of the digits ’3’ and ’8’ that are discriminative. We know that the left part of digit ’8’ differentiates it from digit ’3’, which is what was learned by the SPN.

4.4 Conclusion

I proposed a framework to train SPNs discriminatively using Extended Baum-Welch. I did so by formulating the conditional likelihood as a rational function and applied Extended Baum-Welch to maximize that function. I derived the update formulas for cases where the leaf nodes are either multinomial or univariate normal distributions. The experiments show that EBW outperforms generative EM and discriminative gradient descent in a wide variety of applications. I also demonstrated the advantage of SPNs for classification tasks when some features are missing. We also illustrated the effect of learning the parameters of SPNs using EBW.

Chapter 5

Online Parameter and Structure Learning for Recurrent SPNs

In the previous two chapters, I discussed how to train the parameters of an SPN generatively, and discriminatively. In this chapter, I propose a new technique to learn the network structure of recurrent SPNs.

Recurrent Sum-Product Networks (RSPNs) were introduced by Melibari et al. [32] to model sequential data. RSPNs consist of a template network that can be unrolled as many times as needed to model a variable length data sequence. The unrolled network encodes a valid distribution over a data sequence if the template SPN satisfies certain conditions [32] to ensure that the unrolled network is a valid SPN.

Structure learning for RSPNs is important to model the conditional dependencies among the variables. Several automated structure learning techniques have been proposed for regular SPNs

[13, 16, 38, 26, 50, 1, 54, 46, 31]. These algorithms learn the structure for a fixed length input, but they fail to generalize to variable length data. Melibari et al. [32] proposed a structure learning algorithm called Search-and-Score. The algorithm starts from an initial template network, and gradually increases the size of the network through a series of search and score iterations. At each iteration, the algorithm modifies the network slightly (search step), and the new modification is accepted if it improves the log likelihood score on a development set (score step). In the search step, the algorithm samples a product node uniformly from the template network, and the scope of the product node is randomly partitioned into a set of disjoint scopes. The sub-SPN rooted at the product node is replaced by a product of naive Bayes models of the partitioned scope. Finally, the parameter of the new SPN is learned. In the score step, the log likelihood score is computed on a development dataset. If the log likelihood score improves, the new structure is accepted. Search-and-score based structure learning was demonstrated to perform well compared to HMMs and other models on a number of datasets. But, it is slow and doesn't scale well. It also locally optimizes the structure at each search and score iteration.

The algorithm proposed by Melibari et al. [32] doesn't scale well since it performs a slight modification to the network at each iteration. When the size of the network gets bigger, the effect of these modifications diminish, which results in very slow convergence. The scoring step is also computationally expensive since we need to evaluate the network after each structure update. To that effect, I propose a new *online* parameter and structure learning technique for recurrent SPNs. The approach starts with a network structure that assumes that all features are independent. This network structure is then updated as a stream of data points is processed. Whenever a non-negligible correlation is detected between some features, the network structure

is updated to capture this correlation.

5.1 Proposed Algorithm

In recurrent SPNs, there are four types of nodes: sum nodes, product nodes, interface and leaf nodes. Sum nodes and product nodes produce, respectively, the sums and products of the values of their children. Interface nodes are the latent inputs and outputs of each time step, they are either sum or product nodes. Each leaf node represents a probability distribution over a subset of the variables, and given an input, it produces the value of the probability density function at that input.

To simplify the exposition, we assume that the leaf nodes have Gaussian distributions. A leaf node may have more than one variable in its scope, in which case it follows a multivariate Gaussian distribution. For interface nodes, we consider the case where all interface nodes have the same scope, this makes it easier to grow the network without violating any of the completeness/consistency constraints. This assumption helps as well with interpreting the interface nodes as the predicates of a single multinomial latent variable, L.

Suppose we want to model a probability distribution over a d-dimensional input space, and k-dimensional latent space. The algorithm starts with a fully factorized joint probability distribution over all d variables plus the latent variable L, $p(\mathbf{x}, L) = p(x_1, x_2, \ldots, x_d, L)$
$= p_1(x_1)p_2(x_2)\cdots p_d(x_d)p(L)$. This distribution is represented by a product node with $d+1$ children. Initially, we assume that the variables are independent, and the algorithm will update this probability distribution as new data points are processed. Fig. 5.2 shows an example of an

initial structure for three input variables, and two interface nodes.

Given a mini-batch of data sequences, the algorithm passes the points through the network from the root to the leaf nodes and updates each node along the way. This update includes two parts: i) updating the parameters of the SPN, and ii) updating the structure of the template network.

5.1.1 Parameter update

There are two types of parameters in the model: weights on the branches under a sum node, and parameters for the Gaussian distribution in a leaf node. The parameters are updated in an online fashion by keeping track of running sufficient statistics.

Algorithm 3 describes the pseudocode of this procedure. Every node in the network has a count, n_c, initialized to 1. When a data point is received, the template network is unrolled (Fig. 5.2.b) as many times as needed and the likelihood of this data sequence is computed at each node. Then the parameters of the network are updated in a recursive top-down fashion by starting at the root node. When a sum node is traversed, its count is increased by 1 and the count of the child with the highest likelihood is increased by 1. This effectively increases the weight of the child with the highest likelihood while decreasing the weights of the remaining children. As a result, the overall likelihood at the sum node will increase. The weight $w_{s,c}$ of a branch between a sum node s and one of its children c is estimated as $w_{s,c} = \frac{n_c}{n_s}$ where n_s is the count of the sum node and n_c is the count of the child node. We recursively update the subtree of the child with the highest likelihood. In the case of ties, we simply choose one of the children with highest

likelihood at random to be updated. Since the template network is copied as many times as the length of the input sequence, the same sum node could be traversed multiple times. At each time, the parameters of the count of the sum node, and the child that is traversed are incremented.

Since there are no parameters associated with a product node, the only way to increase its likelihood is to increase the likelihood at each of its children. We increment the count at each child of a product node and recursively update the subtrees rooted at each child. For Gaussian leaf nodes, we keep track of the empirical mean vector μ and covariance matrix Σ for the variables in their scope. When a leaf node with a current count of n receives a batch of m data points $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$, the empirical mean μ and covariance Σ are updated according to the following equations:

$$\mu_i' = \frac{1}{n+m}\left(n\mu_i + \sum_{k=1}^{m} x_i^{(k)}\right) \tag{5.1}$$

$$\begin{aligned}\Sigma_{i,j}' =& \frac{1}{n+m}\left[n\Sigma_{i,j} + \sum_{k=1}^{m}\left(x_i^{(k)} - \mu_i\right)\left(x_j^{(k)} - \mu_j\right)\right] \\ & - (\mu_i' - \mu_i)(\mu_j' - \mu_j)\end{aligned} \tag{5.2}$$

where i and j index the variables in the leaf node's scope.

This parameter update technique is related to hard SGD and hard EM used in [45, 15, 26]. Hard SGD and hard EM also keep track of a count for the child of each sum node and increment those counts each time a data point reaches this child. However, to decide when a child is reached by a data point, they replace all descendant sum nodes by max nodes and evaluate the resulting max-product network. In contrast, we retain the descendant sum nodes and evaluate the original

sum-product network as it is. This evaluates more faithfully the probability that a data point is generated by a child.

Algorithm 3 does a single pass through the data. The complexity of updating the parameters after each data point is linear in the size of the network (i.e., # of edges) since it takes one bottom up pass to compute the likelihood of the data point at each node and one top-down pass to update the sufficient statistics and the weights. The update of the sufficient statistics can be seen as locally maximizing the likelihood of the data. The empirical mean and covariance of the Gaussian leaves locally increase the likelihood of the data that reach that leaf. Similarly, the count ratios used to set the weights under a sum node locally increase the likelihood of the data that reach each child. We prove this result below.

Algorithm 3 parameterUpdate(root(UnrolledRSPN),data)

Require: RSPN and m data points
Ensure: RSPN with updated parameters
  $n_{root} \leftarrow n_{root} + m$
  if $isProduct(root)$ **then**
    for each $child$ of $root$ **do**
      $parameterUpdate(child, data)$
  else if $isSum(root)$ **then**
    for each $child$ of $root$ **do**
      $subset \leftarrow \{x \in data \mid likelihood(child, x) \geq likelihood(child', x)\ \forall child'$ `of` $root\}$
      $parameterUpdate(child, subset)$
      $w_{root,child} \leftarrow \frac{n_{child}}{n_{root}}$
  else if $isLeaf(root)$ **then**
    update mean $\mu^{(root)}$ based on Eq. 5.1
    update covariance matrix $\Sigma^{(root)}$ based on Eq. 5.2

5.1.2 Structure update

The simple online parameter learning technique described above can be easily extended to enable online structure learning. Alg. 5 describes the pseudocode of the resulting procedure called oSLRAU (online Structure Learning with Running Average Update). Similar to leaf nodes, each product node also keeps track of the empirical mean vector and empirical covariance matrix of the variables in its scope. These are updated in the same way as the leaf nodes.

In RSPNs, the scope of the same product node at each copy of the template network is different when taking into account copies at previous time steps. In such case, keeping track of the empirical means and covariances is difficult and intractable. To solve this problem, the scope of product nodes can be computed with respect to the template network input and the latent multinomial variable L. This ensures that the variables in each copy of the template network are equivalent and therefore we can maintain a shared covariance matrix at each product node of the template network.

The latent multinomial distribution $P(L)$ for interface nodes, can be approximated with a k-dimensional normal distribution. Each interface node is treated as a separate variable when estimating the mean and the covariance. To learn the parameters, we construct the sample vector $L = l_1, l_2, ..., l_k$ by setting the indicator associated with the traversed interface node to 1 while setting the rest of indicators to 0's. This vector is used to update the means and covariances for any product node that has variable L in its scope.

Initially, when a product node is created, all variables in the scope are assumed independent (see Alg. 4). As new data sequences arrive at a product node, the covariance matrix is updated,

and if the absolute value of the Pearson correlation coefficient between two variables are above a certain threshold, the algorithm updates the structure so that the two variables become correlated in the model. We ignore inter correlations between interface node variables since we can't partition the scope of the latent variable L into disjoint scopes. We correlate the latent variable L with another variable if the absolute value between this variable and any of the interface node variables is above certain threshold.

We correlate two variables in the model by combining the child nodes whose scopes contain the two variables. The algorithm creates a mixture of two components over the variables (Alg. 4). As shown in Figure 5.1. On the left, a product node with scope $x_1, \ldots, x_5$ originally has three children. The product node keeps track of the empirical mean and covariance for these five variables. Suppose it receives a mini-batch of data and updates the statistics. As a result of this update, x_1 and x_3 now have a correlation above the threshold. To correlate x_1 and x_3, a mixture is created as shown in the right part of Figure 5.1. The mixture has two components. The first component contains the original children of the product node that contain x_1 and x_3. The second component is a new product node, which is again initialized to have a fully factorized distribution over its scope (see Alg. 4). The mini-batch of data points are then passed down the new mixture to update its parameters. Although the children are drawn like leaf nodes in the diagrams, they can in fact be entire subtrees. Since the process does not involve the parameters in a child, it works the same way if some of the children are trees instead of single nodes.

Note that this structure learning technique does a single pass through the data and therefore is entirely online. The time and space complexity of updating the structure after each data point is linear in the size of the network (i.e., # of edges) and quadratic in the number of features plus

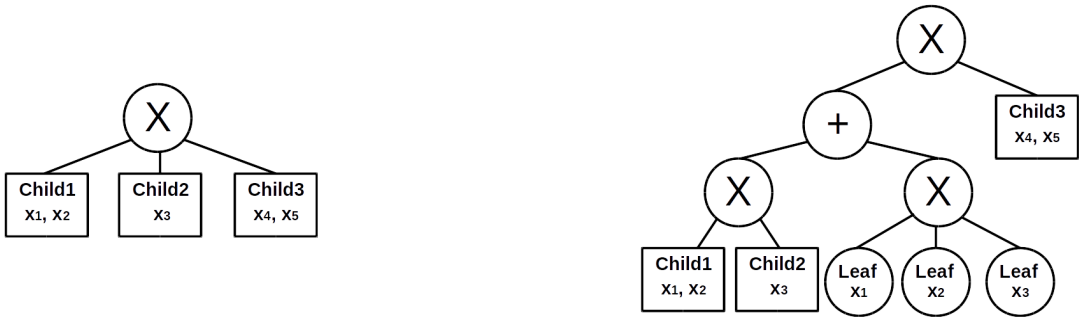


Figure 5.1: Depiction of how correlations between variables are introduced. Left: original product node with three children. Right: create a mixture to model the correlation (Alg. 4).

number of interface nodes (since product nodes store a covariance matrix that is quadratic in the size of their scope). The algorithm also ensures that the decomposability and completeness properties are preserved after each update.

Algorithm 4 $createFactoredModel(scope)$

Require: scope (set of variables)
Ensure: fully factored SPN

1: $factoredModel \leftarrow$ create product node
2: **for** each $i \in scope$ **do**
3: add $N_i(\mu{=}0, \sigma{=}\Sigma_{i,i}^{(root)})$ as child of $factoredModel$
4: $\Sigma^{(factoredModel)} \leftarrow \mathbf{0}$
5: $n_{factoredModel} \leftarrow 0$
6: return $factoredModel$

5.2 Experiments

Table 5.1 reports the average log likelihood based on 10-fold cross validation with 5 sequence datasets. The number of sequences, the average length of the sequences and the number of observed variables is reported under the name of each dataset. We compare oSLRAU to the previous search-and-score (S&S) technique [31] for recurrent SPNs (RSPNs) with Gaussian leaves

Algorithm 5 $oSLRAU(root(unrolledRSPN), data)$

Require: RSPN and m data sequences
Ensure: RSPN with updated parameters

$n_{root} \leftarrow n_{root} + m$
if $isProduct(root)$ **then**
  update covariance matrix $\Sigma^{(root)}$ based on Eq. 5.2
  $highestCorrelation \leftarrow 0$
  for each $c, c' \in children(root)$ where $c \neq c'$ **do**
    $correlation_{c,c'} \leftarrow \max_{i \in scope(c), j \in scope(c')} \frac{|\Sigma_{ij}^{(root)}|}{\sqrt{\Sigma_{ii}^{(root)} \Sigma_{jj}^{(root)}}}$
    if $correlation_{c,c'} > highestCorrelation$ **then**
      $highestCorrelation \leftarrow correlation_{c,c'}$
      $child_1 \leftarrow c$
      $child_2 \leftarrow c'$
  if $highest \geq threshold$ **then**
    $createMixture(root, child_1, child_2)$
  for each $child$ of $root$ **do**
    $oSLRAU(child, data)$
else if $isSum(root)$ **then**
  for each $child$ of $root$ **do**
    $subset \leftarrow \{x \in data \mid likelihood(child, x) \geq likelihood(child', x)\ \forall child' \texttt{ of } root\}$
    $oSLRAU(child, subset)$
    $w_{root,child} \leftarrow \frac{n_{child}+1}{n_{root}+\#children}$
else if $isLeaf(root)$ **then**
  update mean $\mu^{(root)}$ based on Eq. 5.1
  update covariance matrix $\Sigma^{(root)}$ based on Eq. 5.2

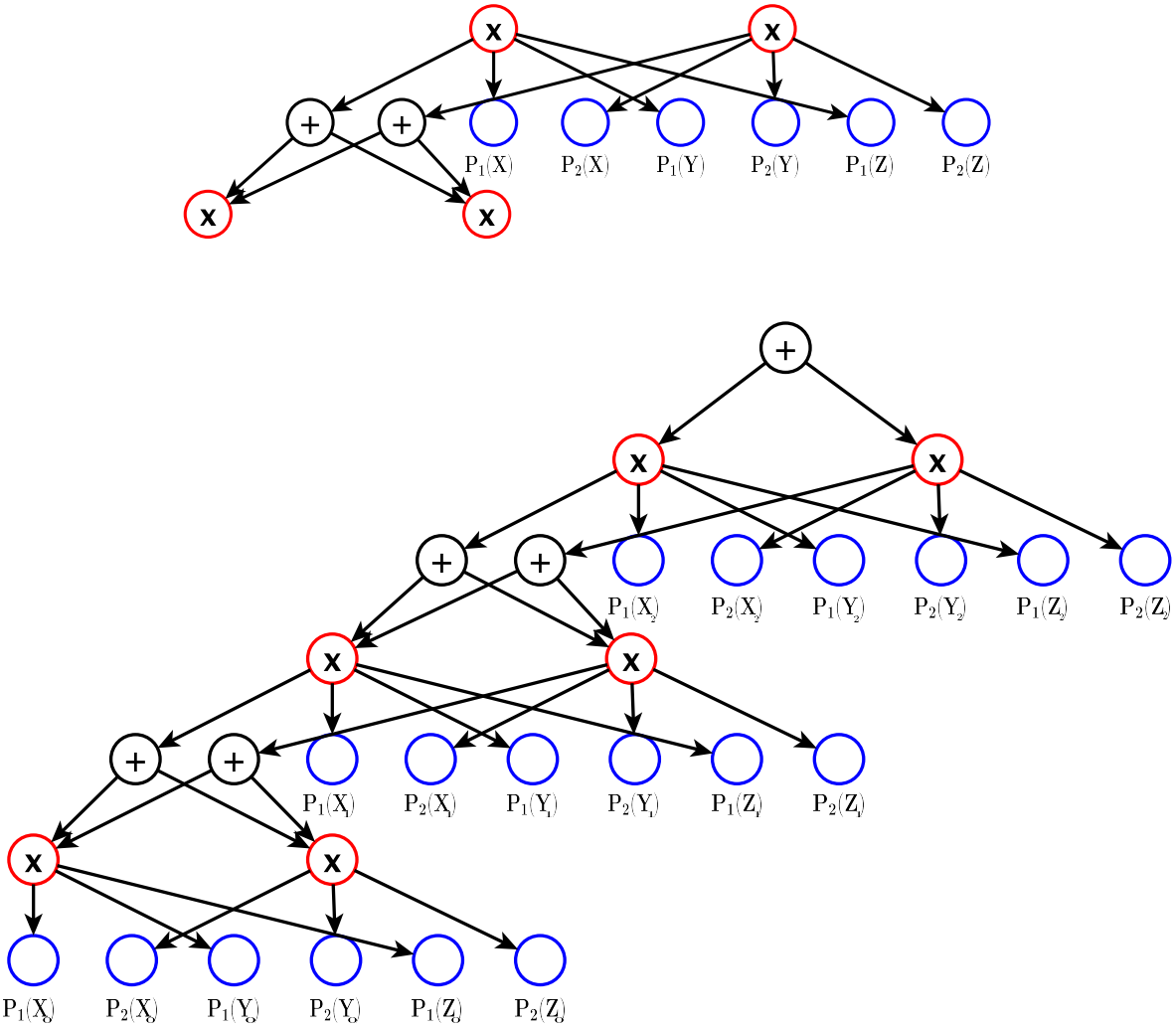


Figure 5.2: Top: Generic template network with interface nodes drawn in red and leaf distributions drawn in blue. Bottom: A recurrent SPN unrolled over 3 time steps.

as well as HMMs with mixture of Gaussians emission distributions and recurrent neural networks (RNNs) with LSTM units and output units that compute the mean of Gaussians. The number of interface nodes in RSPNs, hidden states in HMMs and LSTM units in RNNs was bounded to 15. Parameter and structure learning was performed for the RSPNs while only parameter learning

was performed for the HMMs and RNNs. The RNNs were trained by minimizing squared loss, which is mathematically equivalent to maximizing the data likelihood when we interpret each output as the mean of a univariate Gaussian. The variance of each Gaussian was optimized by a grid search in [0.01,0.1] in increments of 0.01 and in [0.1,2] in increments of 0.1. We did this solely for the purpose of reporting the log likelihood of the test data with RNNs, which would not be possible otherwise. oSLRAU outperformed the other techniques on 4 of the 5 datasets. It learned better structures than S&S in less time. oSLRAU took less than 11 minutes per dataset while S&S took 1 day per dataset.

Table 5.1: Average log-likelihood and standard error based on 10-fold cross validation. (#i,length,#oVars) indicates the number of data instances, average length of the sequences and number of observed variables per time step.

Dataset (#i,length,#oVars)	hillValley (600,100,1)	eegEye (14970,14,1)	libras (350,90,1)	JapanVowels (270,16,12)	ozLevel (2170,24,2)
HMM	286 $\pm$ 6.9	22.9 $\pm$ 1.8	-116.5 $\pm$ 2.2	-275 $\pm$ 13	-34.6 $\pm$ 0.3
RNN	205 $\pm$ 23	15.2 $\pm$ 3.9	-92.9 $\pm$ 12.9	-257 $\pm$ 35	**-15.3** $\pm$ 0.8
RSPN+S&S	296 $\pm$ 16.1	25.9 $\pm$ 2.1	-93.5 $\pm$ 7.2	-241 $\pm$ 12	-34.4 $\pm$ 0.4
RSPN+oSLRAU	**299.5** $\pm$ 18	**36.9** $\pm$ 1.4	**-83.5** $\pm$ 5.4	**-231** $\pm$ 12	-30.1 $\pm$ 0.4

5.3 Conclusion

This chapter describes a new *online* structure learning technique for recurrent SPNs. oSLRAU can learn the structure of SPNs in domains for which it is unclear what might be a good structure, including sequence datasets of varying length. This algorithm can also scale to large datasets efficiently.

Chapter 6

Conclusion

Sum-product networks (SPNs) generated some interest because exact marginal and conditional inference can always be done in linear time with respect to the size of the network. This is particularly attractive since it means that learning an SPN from data always yields a tractable model for inference. In this thesis, I proposed tractable parameter and structure learning techniques for SPNs. Having tractable learning algorithms for SPNs would add an additional attractive property to SPNs, and make them more practical to use in various applications.

First, I proposed online Bayesian Moment Matching as an online framework for generative parameter learning, and I showed how to distribute the algorithm over many machines. I showed how oBMM outperforms existing algorithms in online settings, I also showed that distributing the algorithm over multiple machines is very effective when running time is the bottleneck. I also showed how to apply the algorithm on SPNs with Gaussian leaves. I reported on the first comparison between continuous SPNs and other generative deep models such as SRBMs and

GenMMNs.

Second, I focused on discriminative parameter learning for SPNs, I proposed a framework to train SPNs discriminatively using Extended Baum-Welch. I did so by formulating the conditional likelihood as a rational function and applied Extended Baum-Welch to maximize that function. I derived the update formulas for cases where the leaf nodes are either multinomial or univariate normal distributions. The experiments show that EBW outperforms generative EM and discriminative gradient descent in a wide variety of applications. I also demonstrated the advantage of SPNs for classification tasks when some features are missing. We also illustrated the effect of learning the parameters of SPNs using EBW.

In the final piece, I tackled the problem of structure learning for recurrent SPNs. Recurrent SPNs is a recent model introduced by [31] to model sequential data. Learning the structure for recurrent SPNs was done using random search and score which is very inefficient and time consuming. I proposed a new *online* structure learning technique for recurrent SPNs. oSLRAU can learn the structure of SPNs in domains for which it is unclear what might be a good structure, including sequence datasets of varying length. This algorithm can also scale to large datasets efficiently.

6.1 Future Work

In the subsection below, I will talk about possible future directions to extend the work for each technique I presented in this thesis.

6.1.1 Online Parameter Learning for SPNs Using Bayesian Moment Matching

Discrete or Gaussian leaves might not be the best choice for modeling some random events, this work can be extended to cover more types of leaf distributions. I would also like to investigate the consistency of Bayesian moment matching.

6.1.2 Discriminative Training of SPNs by Extended Baum-Welch

This work can be extended by applying this framework to recurrent SPNs. Extended Baum-Welch was first introduced to learn the parameters for Hidden Markov Models discriminatively, extending this algorithm to cover recurrent SPNs would help improving the performance of recurrent SPNs on sequential classification tasks like speech recognition. I would also like to investigate extending this framework to different types of leaf distributions such as exponential family distributions.

6.1.3 Online Parameter and Structure Learning for Recurrent SPNs

The work can be extended by learning the structure of SPNs in an *online* and *discriminative* fashion. Discriminative learning is essential to attain good accuracy in classification. Another direction for extending this work is investigating the combination of our structure learning technique with other parameter learning methods. Currently, I simply learn the parameters by keeping running statistics for the weights, mean vectors, and covariance matrices. It might be possible to

improve the performance by using more sophisticated parameter learning algorithms.

It would also be interesting to investigate other ways to form mixture components. The current algorithm creates new mixture components when variables are found to be correlated according to the Pearson coefficient. But there are situations when clusters cannot be detected using this method.

I would also like to extend the structure learning algorithm to discrete variables. Finally, we would like to look into ways to automatically control the complexity of the networks. For example, it would be useful to add a regularization mechanism to avoid possible overfitting.

References

[1] Tameem Adel, David Balduzzi, and Ali Ghodsi. Learning the structure of sum-product networks via an svd-based algorithm. In *UAI*, 2015.

[2] Mohamed Amer and Sinisa Todorovic. Sum product networks for activity recognition. 2015.

[3] Mohamed R Amer and Sinisa Todorovic. Sum-product networks for modeling activities with stochastic structure. In *Computer Vision and Pattern Recognition*, 2012.

[4] Animashree Anandkumar, Daniel Hsu, and Sham M. Kakade. A method of moments for mixture models and hidden Markov models. *Journal of Machine Learning Research - Proceedings Track*, 23:33.1–33.34, 2012.

[5] Leonard E Baum, John Alonzo Eagon, et al. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bull. Amer. Math. Soc*, 73(3):360–363, 1967.

[6] Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The annals of mathematical statistics*, 41(1):164–171, 1970.

[7] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[8] Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C Wilson, and Michael I Jordan. Streaming variational bayes. In *NIPS*, pages 1727–1735, 2013.

[9] Wei-Chen Cheng, Stanley Kok, Hoai V Pham, Hai Leong Chieu, and Kian Ming A Chai. Language modeling with sum-product networks. *INTERSPEECH*, 2014.

[10] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Information Theory, IEEE Transactions on*, 42(3):393–405, 1990.

[11] Adnan Darwiche. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

[12] Adnan Darwiche. A differential approach to inference in Bayesian networks. *JACM*, 50(3):280–305, 2003.

[13] Aaron Dennis and Dan Ventura. Learning the architecture of sum-product networks using clustering on variables. In *NIPS*, 2012.

[14] Abram L Friesen and Pedro Domingos. Submodular sum-product networks for scene understanding. 2017.

[15] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. In *NIPS*, pages 3248–3256, 2012.

[16] Robert Gens and Pedro Domingos. Learning the structure of sum-product networks. In *ICML*, pages 873–880, 2013.

[17] Ponani S Gopalakrishnan, Dimitri Kanevsky, Arthur Nádas, and David Nahamoo. An inequality for rational functions with applications to some statistical estimation problems. *IEEE Transactions on Information Theory*, 37(1):107–113, 1991.

[18] H. Altay Guvenir and I. Uysal. Bilkent university function approximation repository. 2000.

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

[20] Priyank Jaini, Abdullah Rashwan, Han Zhao, Yue Liu, Ershad Banijamali, Zhitang Chen, and Pascal Poupart. Online algorithms for sum-product networks with continuous variables. In *International Conference on Probabilistic Graphical Models (PGM)*, 2016.

[21] Tony Jebara and Alex Pentland. Maximum conditional likelihood via bound maximization and the CEM algorithm. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, pages 494–500. MIT Press, 1998.

[22] Tony Jebara and Alex Pentland. On reversing Jensen's inequality. In *NIPS*, pages 231–237, 2000.

[23] Agastya Kalra, Abdullah Rashwan, Wei-Shou Hsu, Pascal Poupart, Prashant Doshi, and Georgios Trimponias. Online structure learning for feed-forward and recurrent sum-product networks. In *Advances in Neural Information Processing Systems*, pages 6944–6954, 2018.

[24] Aldebaro Klautau, Nikola Jevtic, and Alon Orlitsky. Discriminative Gaussian mixture models: A comparison with kernel classifiers. In *ICML*, pages 353–360, 2003.

[25] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[26] Sang-Woo Lee, Min-Oh Heo, and Byoung-Tak Zhang. Online incremental structure learning of sum–product networks. In *NIPS*, pages 220–227, 2013.

[27] Yujia Li, Kevin Swersky, and Rich Zemel. Generative moment matching networks. In *ICML*, pages 1718–1727, 2015.

[28] Percy Liang and Dan Klein. Online EM for unsupervised models. In *HLT-NAACL*, pages 611–619, 2009.

[29] M. Lichman. UCI machine learning repository, 2013.

[30] Daniel Lowd and Amirmohammad Rooshenas. Learning Markov networks with arithmetic circuits. In *AISTATS*, pages 406–414, 2013.

[31] Mazen Melibari, Pascal Poupart, Prashant Doshi, and George Trimponias. Dynamic sum-product networks for tractable inference on sequence data. In *JMLR Conference and Workshop Proceedings - International Conference on Probabilistic Graphical Models (PGM)*, 2016.

[32] Mazen Melibari, Pascal Poupart, Prashant Doshi, and George Trimponias. Dynamic sum product networks for tractable inference on sequence data. pages 345–355, 2016.

[33] Thomas Minka and John Lafferty. Expectation-propagation for the generative aspect model. In *UAI*, pages 352–359, 2002.

[34] Radford M Neal and Geoffrey E Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

[35] Yves Normandin. *Hidden Markov models, maximum mutual information estimation, and the speech recognition problem*. PhD thesis, McGill University, Montreal, 1991.

[36] Yves Normandin and Salvatore D Morgera. An improved MMIE training algorithm for speaker-independent, small vocabulary, continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 537–540. IEEE, 1991.

[37] Robert Peharz. *Foundations of Sum-Product Networks for Probabilistic Modeling*. PhD thesis, Medical University of Graz, 2015.

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

[39] Robert Peharz, Robert Gens, and Pedro Domingos. Learning selective sum-product networks. *ICML Workshop on Learning Tractable Probabilistic Models*, 2014.

[40] Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2016.

[41] Robert Peharz, Georg Kapeller, Pejman Mowlaee, and Franz Pernkopf. Modeling speech with sum-product networks: Application to bandwidth extension. In *ICASSP*, pages 3699–3703, 2014.

[42] Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On theoretical properties of sum-product networks. In *AISTATS*, pages 744–752, 2015.

[43] Kristiaan Pelckmans, Jos De Brabanter, Johan AK Suykens, and Bart De Moor. Handling missing values in support vector machine classifiers. *Neural Networks*, 18(5-6):684–692, 2005.

[44] Franz Pernkopf and Michael Wohlmayr. *Large Margin Learning of Bayesian Classifiers Based on Gaussian Mixture Models*, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[45] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *UAI*, pages 2551–2558, 2011.

[46] Tahrima Rahman and Vibhav Gogate. Merging strategies for sum-product networks: From trees to graphs. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence, UAI*, 2016.

[47] Abdullah Rashwan, Pascal Poupart, and Chen Zhitang. Discriminative training of sum-product networks by extended baum-welch. In *International Conference on Probabilistic Graphical Models*, pages 356–367, 2018.

[48] Abdullah Rashwan, Han Zhao, and Pascal Poupart. Online and Distributed Bayesian Moment Matching for Sum-Product Networks. In *AISTATS*, 2016.

[49] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. Sum-product networks for sequence labeling. *arXiv preprint arXiv:1807.02324*, 2018.

[50] Amirmohammad Rooshenas and Daniel Lowd. Learning sum-product networks with direct and indirect variable interactions. In *ICML*, pages 710–718, 2014.

[51] Jarkko Salojärvi, Kai Puolamäki, and Samuel Kaski. Expectation maximization algorithms for conditional likelihoods. In *Proceedings of the 22nd international conference on Machine learning*, pages 752–759. ACM, 2005.

[52] Alexander J Smola, SVN Vishwanathan, and Thomas Hofmann. Kernel methods for missing variables. In *AISTATS*, 2005.

[53] Lucas Theis, Aäron Oord, and Matthias Bethge. A note on the evaluation of generative models. *arXiv:1511.01844*, 2015.

[54] Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Simplifying, regularizing and strengthening sum-product network structure learning. In *ECML-PKDD*, pages 343–358. 2015.

[55] Frank Wilcoxon. Some rapid approximate statistical procedures. *Annals of the New York Academy of Sciences*, pages 808–814, 1950.

[56] Han Zhao, Tameem Adel, Geoff Gordon, and Brandon Amos. Collapsed variational inference for sum-product networks. In *ICML*, 2016.

[57] Han Zhao, Mazen Melibari, and Pascal Poupart. On the relationship between sum-product networks and Bayesian networks. In *ICML*, 2015.

[58] Han Zhao and Pascal Poupart. A unified approach for learning the parameters of sum-product networks. *arXiv:1601.00318*, 2016.