Master's thesis
Theoretical and Computational Methods

Bayesian Structure and Parameter Learning of Sum-Product Networks

Noora Mäkelä

August 11, 2022

Supervisor: Mikko Koivisto

Examiners: Antti Hyttinen
Mikko Koivisto

University of Helsinki
Faculty of Science

PL 64 (Gustaf Hällströmin katu 2a)
00014 Helsingin yliopisto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Theoretical and Computational Methods

Tekijä — Författare — Author

Noora Mäkelä

Työn nimi — Arbetets titel — Title

Bayesian Structure and Parameter Learning of Sum-Product Networks

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	August 11, 2022	65

Tiivistelmä — Referat — Abstract

Sum-product networks (SPN) are graphical models capable of handling large amount of multi-dimensional data. Unlike many other graphical models, SPNs are tractable if certain structural requirements are fulfilled; a model is called tractable if probabilistic inference can be performed in a polynomial time with respect to the size of the model. The learning of SPNs can be separated into two modes, parameter and structure learning. Many earlier approaches to SPN learning have treated the two modes as separate, but it has been found that by alternating between these two modes, good results can be achieved. One example of this kind of algorithm was presented by Trapp et al. in an article Bayesian Learning of Sum-Product Networks (NeurIPS, 2019).

This thesis discusses SPNs and a Bayesian learning algorithm developed based on the earlier mentioned algorithm, differing in some of the used methods. The algorithm by Trapp et al. uses Gibbs sampling in the parameter learning phase, whereas here Metropolis-Hasting MCMC is used. The algorithm developed for this thesis was used in two experiments, with a small and simple SPN and with a larger and more complex SPN. Also, the effect of the data set size and the complexity of the data was explored. The results were compared to the results got from running the original algorithm developed by Trapp et al.

The results show that having more data in the learning phase makes the results more accurate as it is easier for the model to spot patterns from a larger set of data. It was also shown that the model was able to learn the parameters in the experiments if the data were simple enough, in other words, if the dimensions of the data contained only one distribution per dimension. In the case of more complex data, where there were multiple distributions per dimension, the struggle of the computation was seen from the results.

Avainsanat — Nyckelord — Keywords

Sum-product network, Bayesian statistics, Markov chain Monte Carlo

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

Acronyms

ABDA	Automatic Bayesian density analysis	28
AC	Arithmetic circuit	19
DAG	Directed acyclic graph	18
EM	Expectation maximization	27
MCMC	Markov chain Monte Carlo	14
NN	Neural network	8
PGM	Probabilistic graphical model	17
SPN	Sum-product network	8
SVD	Singular value decomposition	28

List of Symbols

$\boldsymbol{\theta}$	Set of all leaf node parameters in a sum-product network
$\boldsymbol{\theta}_L$	Set of leaf node parameters for a leaf node L
$\boldsymbol{A}$	Set of all partition nodes
$\boldsymbol{N}$	Set of all nodes in a sum-product network
$\boldsymbol{P}$	Set of all product nodes
$\boldsymbol{Q}$	Set of all nodes in a region graph
$\boldsymbol{R}$	Set of all region nodes
$\boldsymbol{S}$	Set of all sum nodes
$\boldsymbol{w}$	Set of sum node weights
$\mathcal{L}$	Set of likelihood models for each dimension
$\mathcal{R}$	A region graph DAG
$\mathcal{S}$	A sum-product network
$\mathcal{T}$	An induced tree
Ω	Likelihood weights of a leaf node
ψ	Scope function of a sum-product network
ξ	A scope function of a region graph
A	A partition node
$ch(N)$	Children of a node N
G	A computational graph
N	A node in a sum-product network
P	A product node
Q	A node in a region graph
R	A region node
S	A sum node

s	A leaf node latent variable
X	Set of random variables
y	A partition node latent variable
z	A sum node latent variable

1. Introduction

Using machine learning to analyze data has become widely popular in recent years because of its ability to solve various problems in many domains. Typically, a machine learning algorithm learns a model, i.e., a generalization, from a given training data set, after which it can apply the learned model to make inference regarding the data generating process or new, unforeseen data [1]. In the age of big data, there is a need to understand and analyze data sets which are multidimensional and large. This is a challenge for machine learning methods as large data sets with multidimensional data may cause both statistical and computational inefficiency.

Machine learning algorithms are often based on models which contain multiple *layers*. A layer is a group of nodes, all of which operate at a specific depth in a network. One example of this kind of a model is a neural network. Neural network (NN) is a graph composed of nodes which are connected by edges. The nodes are laid out in layers and the output from the previous layer is fed as an input to the next layer [2]. These models can have a high number of layers improving the representational power of the model but they can make inference slower and *intractable*. We call a model tractable if one can perform probabilistic inference with the model in polynomial time with respect to the model size [3]. Probabilistic inference is a type of statistical inference, where a probability function is marginalized. This means for example computing marginal probabilities. There is a need for a model with good representational power which is simultaneously tractable.

Sum-product network (SPN), introduced by Poon and Domingos [4], is a probabilistic graphical model which can handle large data sets with multidimensional data. Graphical models in general can represent many complex distributions, but usually they are not tractable. SPNs offer us tractability under some structural restrictions. They can model larger class of complex distributions than mixture or independence models by combining many simpler, tractable distributions, similar to mixture models, while still being tractable [5]. The probability distributions an SPN can model are closer to real-life situations, as they are more complex and have more dependencies.

Learning SPNs can be separated into two parts, parameter learning and structure learning. In this thesis an SPN learning algorithm is presented which alternates be-

tween these two modes. In many earlier approaches, parameter and structure learning has been seen as separate things, and many learning approaches have focused on only structure learning. Some approaches have attempted to perform both separately which have led to achieving local improvements in the SPN. Among the algorithm presented here, there are also other approaches for simultaneously learning the parameters and structure by alternating between these modes which has led to obtaining improvements globally.

The algorithm implemented for this thesis is an SPN learning algorithm based on the algorithm created by Trapp et al. in 2019 [6]. The difference between these two algorithms lies in the used methods. In the algorithm created for this thesis, Metropolis-Hastings Markov chain Monte Carlo (MCMC) is used in parameter learning whereas the original algorithm by Trapp et al. uses Gibbs sampling. In Metropolis-Hastings MCMC samples are either accepted or rejected by calculating an acceptance ratio, whereas in Gibbs sampling samples are always accepted [7]. In Gibbs sampling, sample for one random variable is sampled from a posterior conditional distribution with all of the other random variables fixed to their current value [8]. Both of these learning algorithms use collapsed Gibbs sampling in structure learning and the same Bayesian SPN model. This thesis aims to gather knowledge on how well the presented algorithm and methods used in it perform in SPN learning with SPNs of different sizes. The size of the SPN affects the learning process as the larger the SPN is the more nodes it has. This brings more possibilities for learning the structure and parameters as there are more nodes in the network for spreading the data and computing the node operations. Another aspect of having a larger SPN is that the larger the SPN is the heavier the learning process will be. This is a straight consequence of having more nodes which in turn mean more computations. In the experiments, we will also explore how the used data affects the learning process. We will look into using different sizes of data sets and examine the effect of the size on the learning results. Besides these, we will also explore having different number of dimensions in the data and having more complex data within a dimension.

The rest of the thesis is structured as follows. Chapter 2 supports the themes in the thesis by giving an introduction to some background information including topics related to machine learning, Bayesian statistics, Markov chain Monte Carlo and graphical models. These are important topics which will set grounds for the further topics in the thesis. Chapter 3 gives a detailed view on SPNs and an overview on data modelling. Chapter 4 discusses SPN learning, going through earlier approaches for it, including the algorithm created by Trapp et al. [6], and also presenting the learning algorithm created for this thesis. Chapter 5 describes the experiments carried out in this thesis and also the results from these experiments. Finally, Chapter 6 concludes the thesis,

discussing the limitations on the experiments done and the results in relation to the research questions.

2. Background

This chapter helps to support the main themes in this thesis. Here we will lay the grounds for sum-product networks, starting from important prerequisites, machine learning and graphical models. In addition to these, Bayesian statistics, the Bayes' rule and conjugate priors, are discussed. After these, Markov chain Monte Carlo methods are covered, including Metropolis-Hastings algorithm, Gibbs sampling and collapsed Gibbs sampling.

2.1 Machine Learning

Machine learning is a widely researched and used study area in computer science, especially now in the time of big data. Machine learning is about continuous automated data-analysis. In essence, it is about being able to automatically detect patterns from data, and based on those patterns predict future data or make decisions [9]. Machine learning algorithms can be used, for example, for classification, data clustering and regression. Developing these algorithms is an essential part of machine learning research field, as the algorithms need to be efficient. There are a few types of machine learning, of which we will cover supervised and unsupervised learning. Besides these, there are semi-supervised learning and reinforcement learning [10].

When using a machine learning model, there is a learning phase and a prediction phase. In the learning phase, some data is fed to the model from which the model tries to identify patterns. In prediction phase, the model aims to detect the patterns it learned during the learning phase. This can be, for example, a task for classifying data.

2.1.1 Supervised learning

Supervised learning is the most widely used field of machine learning [9]. In supervised learning we are given input x and output y, and the model aims to learn the mappings between the input and the output. The model is trained with a data set containing data points with labels assigned to them. Here, the input is the collection of data

points and the output is the corresponding labels. In the learning phase, the goal of the model is to learn how to classify these data points with the corresponding labels [11]. After the learning phase in a simple classification task, there are data points without the labels and the goal is to assign a label for each of the data points based on the learning. For example, we could have pictures of dogs and cats, and in learning phase we would feed the model with the pictures and the corresponding label 'dog' or 'cat'. In the prediction phase we would then have a group of pictures without these labels, which we would feed to the model and the goal would be to get the label for the data inputted to the model.

2.1.2 Unsupervised learning

In unsupervised learning, we are only given the input x instead of both input and output. The goal of the learning task is to learn the overall composition and the patterns of the data [12]. For this reason, for example, clustering of data is a common task in unsupervised learning [9].

Unsupervised learning is more applicable in many cases in real life, because the labels are not needed in the learning process. In addition to this, with unsupervised learning we can obtain more complex models as the labels in supervised learning do not contain very much information. One important aspect to note is that obtaining data with labels is expensive and more time consuming compared to obtaining data without any labels.

2.2 Bayesian Statistics

Bayesian definition of probability heavily lies in the prior knowledge on an event, in other words in a degree of belief in it. Probability of an event is updated when new information is retrieved. On the other hand, the Frequentist approach views the probability of an event as the limit of its frequency in many trials [13]. In Bayesian statistics the Bayes' rule is used in calculating probabilities after some new data is obtained. The Bayes' rule is discussed in the subsection 2.2.2. In Bayesian inference, the posterior probability can be updated using the prior probability and the likelihood function [14].

Definition 2.2.1. *Likelihood function* measures how well a model fits a set of observations.

In Bayesian statistics, the posterior probability is proportional to the product of the prior probability and the likelihood

$$posterior \propto prior \times likelihood. \tag{2.1}$$

2.2.1 Probability notations

Here we will go through some notations for probability as those will be used often in the thesis. Events and random variables are labeled with capital letters and the probability of something is labeled with p. For example, here X and Y are random variables, and the probability of these are $p(X)$ and $p(Y)$. The conditional probability of X given Y is $p(X|Y)$. The values a random variable takes are labeled with small letters. For example, random variable X takes value x. The probability of X taking the value x is $p(X = x)$, but it can as well be written as $p(x)$.

2.2.2 Bayes' rule

Bayes' rule is

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}. \tag{2.2}$$

Here A, B and C are events. In the rule $p(B|A)$ is a conditional probability of A given B. $p(A)$ and $p(B)$ are prior probabilities of events A and B. The posterior probability of A given B, which is $p(A|B)$, is obtained from using the Bayes' rule [15].

2.2.3 Conjugate priors

When we multiply a likelihood function with a density function and the results is a posterior density function which is in the same family of distributions as the prior, we can call the prior density function a conjugate prior [16]. This means that by choosing some likelihood function and by selecting a suitable prior, the posterior ends up in the same distribution family as the prior [13]. This is very useful in some cases, as we do not have to find out what is the posterior using the equation 2.1. In Bayesian inference, the posterior is updated each time we get more data, which means that less computation is required when using a conjugate prior.

For example, *beta* distribution $Beta(\alpha, \beta)$, with shape parameters α and β, is often used as a prior, because it is a conjugate prior for *Bernoulli*, *binomial*, *negative binomial* and *geometric* likelihoods, which are all discrete distributions. The resulting posterior in this case is *beta* distribution. *Gamma* distribution $\Gamma(\alpha, \beta)$ is a conjugate prior for *Poisson* and *Exponential* likelihoods. Here *Poisson* is a discrete distribution while *gamma* and *exponential* are continuous. Another example is choosing continuous

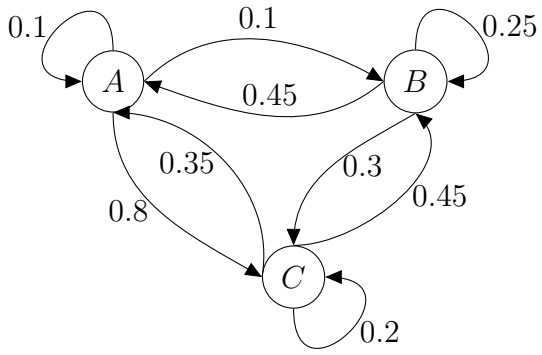


Figure 2.1: A three state Markov process with the probability of each state change as a label for the edges.

Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, with mean μ and variance σ^2 parameters, as a prior for *Gaussian* likelihood.

2.3 Markov chain Monte Carlo

In many cases the interest is to sample from a complex multidimensional probability distribution, although sampling from such distribution directly can be a hard task [17]. Markov chain Monte Carlo (MCMC) is a family of algorithms used for sampling from such distributions. They work by constructing a Markov chain. *Markov chain* is a stochastic model consisting of a sequence of consecutive steps, in other words, events. The probability of each of these events only depends on the previous event [18]. The more steps are included in the chain, the more the samples in the chain begin to resemble the desired distribution. Figure 2.1 describes a three state Markov process with states A, B and C, where the transition to any state depends only on the previous state.

2.3.1 Metropolis-Hastings algorithm

Metropolis-Hastings algorithm is a part of the MCMC algorithm family. It was first introduced by Metropolis et al. [19] and later extended by W. K. Hastings [20]. In Metropolis-Hastings algorithm, samples are created by utilizing a target density function proportional to the desired distribution [21]. Choosing the target density function is flexible since the target distribution must be only proportional to the desired distribution instead of having to be equal. Since the algorithm belongs to the MCMC family of algorithms, it works by building a Markov chain. The basic idea of the algorithm

is that based on the current value in the chain, a sample is generated from a proposal distribution which is a candidate for being a new value in the chain. The proposal distribution is an arbitrary distribution not dependent on the distribution used as the target [22]. When a new sample is generated, the sample is evaluated by calculating the acceptance ratio which tells whether the sample is accepted to the Markov chain or not. The acceptance ratio is calculated by comparing the value of the target density distribution with the sample and with the current value. The acceptance is decided by generating a uniform random number between zero and one. If this number is smaller than or equal to the acceptance ratio, the sample is accepted to the Markov chain. After this, a new interval is started with this value [23].

The Metropolis-Hastings algorithm attempts to move in a sample space by forming a Markov chain. The algorithm is more likely to move to points in high-density areas, which means the algorithm focuses on those areas while sometimes visiting the areas with lower densities if the move there is accepted. Because the algorithm starts from a random point in the sample space, it might take a while for the algorithm to find its way to the high-density areas. This is why a burn-in period is used and the values gathered during that period are thrown away. The values obtained during the burn-in period often describe completely other distributions. The more steps are taken in the Metropolis-Hasting algorithm, the better the generated values describe the desired distribution. It is worth noticing that the samples generated are correlated to each other. The correlation can be minimized by increasing the step size so that the new samples are farther away from each other. Although, the step size cannot be increased endlessly since it increases the probability for the model to jump over high-density areas.

Algorithm 1 Metropolis-Hastings

1: Initialization: choose value x_0 for the first sample, the proposal probability density function $q(x)$ and the target density function $f(x)$ that is proportional to the target probability density function $P(x)$.
2: **for** $i = 1$ to I **do**:
3: Generate a new candidate sample x' from the proposal distribution $q(x'|x_i)$
4: Calculate the acceptance ratio $\alpha = min\{1, \frac{f(x')q(x_i|x')}{f(x_i)q(x'|x_i)}\}$
5: Generate an uniform random number $u \in [0, 1]$
6: **if** $u \leq \alpha$ **then**
7: Accept the candidate sample and set it $x_{i+1} = x'$
8: **if** $u > \alpha$ **then**
9: Reject the candidate and set $x_{i+1} = x_i$

Because the target density function is proportional to the density function of $p(x)$ in Algorithm 1, we have

$$\alpha = \frac{f(x')}{f(x_i)} = \frac{p(x')}{p(x_i)}. \tag{2.3}$$

2.3.2 Gibbs sampling

Gibbs sampling was proposed by Stuart and Donald Geman [24] in 1984. In Gibbs sampling, the samples are generated from a multivariate probability distribution. The idea is to generate samples from the conditional distributions of the random variables. The posterior samples are generated by going through all of the random variables and sampling from their posterior conditional distribution with all of the other random variables fixed to their current value [23]. This is repeated until convergence.

Definition 2.3.1. *Convergence* happens when the sampled values have the same distribution as the true posterior joint distribution.

In the beginning of the algorithm, the initial values are sampled usually from the prior distribution q.

Algorithm 2 Gibbs sampling

1: Initialization: sample $x^{(0)} \sim q(x)$
2: **for** $i = 1$ to I **do**:
3: Sample from the posterior conditional distributions.
$x_1^{(i)} \sim p(X_1 = x_1 | X_2 = x_2^{(i-1)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
$x_2^{(i)} \sim p(X_2 = x_2 | X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
$\vdots$
$x_D^{(i)} \sim p(X_D = x_D | X_1 = x_1^{(i)}, X_2 = x_2^{(i)}, \ldots, X_D = x_{D-1}^{(i)})$

In Gibbs sampling, all of the proposed moves are accepted, which means the acceptance probability is always one. The proposal distribution is

$$Q(x_i', x_{-i} | x_i, x_{-i}) = p(x_i' | x_{-i}). \tag{2.4}$$

When this is applied to the Metropolis-Hastings algorithm, we get acceptance ratio of 1:

$$\begin{aligned} A(x_i', x_{-i}|x_i, x_{-i}) &= \min\left\{1, \frac{p(x_i'|x_{-i})Q(x_i, x_{-i}|x_i', x_{-i})}{p(x_i|x_{-i})Q(x_i', x_{-i}|x_i, x_{-i})}\right\} \\ &= \min\left\{1, \frac{p(x_i'|x_{-i})p(x_i|x_{-i})}{p(x_i|x_{-i})p(x_i'|x_{-i})}\right\} \\ &= \min\left\{1, \frac{p(x_i'|x_{-i})p(x_i)p(x_i|x_{-i})}{p(x_i|x_{-i}p(x_i)p(x_i'|x_{-i})}\right\} \\ &= \min\{1, 1\} \\ &= 1. \end{aligned} \tag{2.5}$$

Here x_i is the ith variable and x_{-i} a set of all variables except x_i.

2.3.3 Collapsed Gibbs sampling

Collapsed Gibbs sampling is a variant of Gibbs sampling and it was introduced by Jun Liu [25] in 1994. In collapsed Gibbs sampling, one or more random variables are marginalized out to make the sampling process simpler when sampling from the posterior conditional distribution of a random variable. In the case of three events, A, B and C, in Gibbs sampling, the sampling would happen from $p(A|B, C)$, then $p(B|A, C)$ and lastly from $p(C|A, B)$, whereas in collapsed Gibbs sampling, sampling for A could mean marginalizing over B and the samples would be then generated from $p(A|C)$ and $p(C|A)$.

2.4 Graphical Models

Graphical models represent probability distributions in an undirected or directed graph structures. They are able to represent joint distributions compactly [26]. The graph structure is a compact representation which visualizes dependence relations among random variables. The use of a graph allows the use of many algorithms which have been developed for graphs, for example, algorithms for calculating marginal or conditional probabilities of any variable. In this section, we will cover probabilistic graphical models and arithmetic circuits.

2.4.1 Probabilistic graphical models

Probabilistic graphical model (PGM) is a model representing a joint probability distribution by only including relevant independence relations to the graph structured model which reduces the complexity of the model. They are efficient in managing

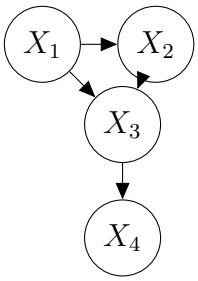


Figure 2.2: Bayesian network of random variables X_1, X_2, X_3 and X_4.

uncertainty-based probability theory [14]. PGMs are based on mathematical foundations of probability and graph theory. The goal of a PGM is to be able to compute the modes and the marginals of a distribution efficiently [4]. PGMs are used, for example, for performing inference.

Definition 2.4.1. *Inference* is about answering some probabilistic queries based on the model and some evidence. These probabilistic queries are for obtaining marginal or conditional probabilities. Marginal probability means a probability of one of the variables taking a certain value, and conditional probability means a probability of a event occurring given that another event has already occurred.

PGMs can be either directed or undirected graphs. One of the most used PGM is Bayesian network, which is a directed graphical model, and Markov random field, which in turn is an undirected graphical model.

Bayesian network is a Directed acyclic graph (DAG), where the direction of the edges between nodes represent the effect of one random variable on another random variable [27]. DAG is, like the name suggests, a graph where there are no cycles [28]. Figure 2.2 visualizes a Bayesian network of four random variables. The edges represent conditional distributions, and the joint probability distribution is formulated of these conditional distributions. One of the most prominent limitations of a Bayesian network is intractable inference [29]. The joint probability of a Bayesian network can be computed by taking the product of the conditional distributions for each node given its parents

$$p(X) = \prod_{i=1}^{N} p(X_i|Parents(X_i)). \tag{2.6}$$

Markov random field is an undirected graph, which can contain cycles. The edges in the graph represent joint probabilities of random variable cliques [30]. Figure 2.3 visualizes a Markov random field with three cliques, $\{X_1, X_2, X_3\}$, $\{X_3, X_4\}$ and $\{X_4, X_5, X_6\}$.

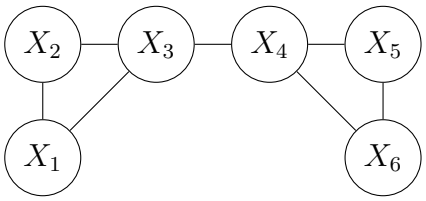


Figure 2.3: Markov random field of random variables X_1, X_2, X_3, X_4, X_5 and X_6.

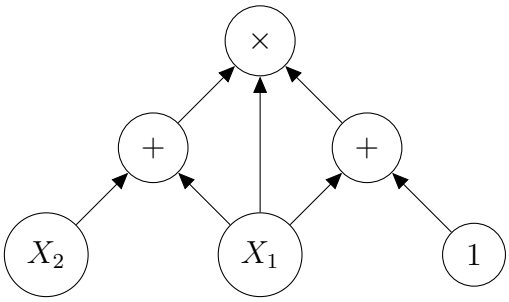


Figure 2.4: Arithmetic circuit of random variables X_1 and X_2.

The joint probability of the Markov random field in figure 2.3 is

$$p(X) = p(X_1, X_2, X_3)p(X_3, X_4)p(X_4, X_5, X_6). \tag{2.7}$$

2.4.2 Arithmetic circuits

Arithmetic circuit (AC) is a rooted, directed acyclic graph consisting of sum, product and leaf nodes [31]. It is a probabilistic graphical model which represents a group of random variables. ACs are tractable and flexible in their representation [32]. They were one of the first tractable models for probabilistic modeling presented, and there has been lot of new development based on ACs [33]. In an AC, sum nodes are for performing additions and product nodes for multiplications. Leaf nodes contain numerical values, for example, some model parameters or indicator variables which are computed from data passed through the AC. The leaf nodes are called input gates. For this there are some existing learning algorithms [34].

The complexity of an AC can be measured by the size and the depth of the graph. The depth is measured as the longest directed path in the graph. The size is measured as the number of nodes. For the complexity of the AC in figure 2.4, we can see that the size is six and the depth is two. The input nodes compute the value for the variable it is labeled by, in this figure X_1, X_2 and 1. The second layer of sum nodes compute the sum of their children, here $X_2 + X_1$ and $X_1 + 1$. The product node in the top calculates the product of its children, here $(X_2 + X_1)X_1(X_1 + 1)$.

3. Sum-Product Networks

This chapter introduces sum-product networks, which are probabilistic graphical models that, in essence, combine the structure of arithmetic circuits with mixture models. We begin by introducing relevant concepts such as independence and mixture models, and then move to sum-product networks. We will finish this chapter by discussing the applications of sum-product networks.

3.1 Independence and mixture models

There are two basic ways to compose more complex models from simple models: assuming independencies among the variables and taking mixtures of probability distributions.

3.1.1 Independence model

Independence model is used for modeling random variables independent from each other. It represents dependency and independency relations between random variables, similarly to earlier presented probabilistic graphical models. In the simplest case, the model assumes two random variables to be independent. This means their joint distribution can be written as $P(X, Y) = P(X)P(Y)$ [15].

The model can also consist of independent groups of random variables or vectors. Here we have a vector X that contains multiple random variables $X = \{X_1, X_2, \ldots, X_D\}$ and a vector Y which contains multiple random variables $Y = \{Y_1, Y_2, \ldots, Y_D\}$ as well. When combining these into a model, the independence relation is between the two vectors, $P(X, Y) = P(X)P(Y)$.

3.1.2 Mixture model

A mixture model is used for modeling complex distributions such as multimodal distributions which have more than one local maxima in their probability density function. A mixture model models these distributions using multiple distributions called mixture

components, which for complex distributions can describe the situation better than using just a single distribution. These components take a simple parametric form, for example, *Gaussian* or *Poisson*. A mixture model assumes that every data point comes from one of the components. The probability density function f of a mixture model is a weighted sum of K probability density functions f_k of the mixture components.

We can assign a latent variable for each of the observables, a label Z describing from which mixture component the observable is from. When using a mixture model, the data is assumed to be generated first by sampling the latent variable Z and after that sampling the observable which depends on the earlier sampled Z:

$$p(Z, X) = p(Z = z)p(X|Z = z). \tag{3.1}$$

Here $p(X|Z = z)$ represents the mixture components. We can also compute the probability density function over mixture components by summing out z:

$$p(x) = \sum_z p(z)p(x|z).$$

3.2 Sum-Product Network

This chapter combines independence and mixture models by presenting sum-product networks in more detail. The earlier mentioned arithmetic circuits are similar to SPNs; however they differ in how they handle random variables. In arithmetic circuits, sum and product operations are performed between random variables, whereas in SPNs, those operations are performed between the distributions of the random variables. Similarly to mixture models, we can structure the model with latent variables [35].

3.2.1 Definition

For a mathematical definition of an SPN, we need two more primitive concepts. First, we call a directed acyclic graph a *computational graph* (G) if it contains only three types of nodes: sum nodes (S), product nodes (P) and leaf nodes (L). G is required to have only one node without a parent node, in other words a root node. We denote a generic node having any of these three types N, and $\boldsymbol{N}$ denotes all N in G. Equivalently $\boldsymbol{S}$, $\boldsymbol{P}$, $\boldsymbol{L}$ denote all S, all P and all L in G. The child nodes of a node N are marked by $ch(N)$. The nodes themself are used as indices. From the computational graph we can see the arrangement of the nodes in an SPN. Figure 3.1 shows the computational graph of an SPN.

Secondly, we need to define a scope for a node. It essentially consists of all of the random variables held by the leaves in a sub-SPN rooted at a node. The scope function ψ, which has some structural constraints, assigns a scope for a node.

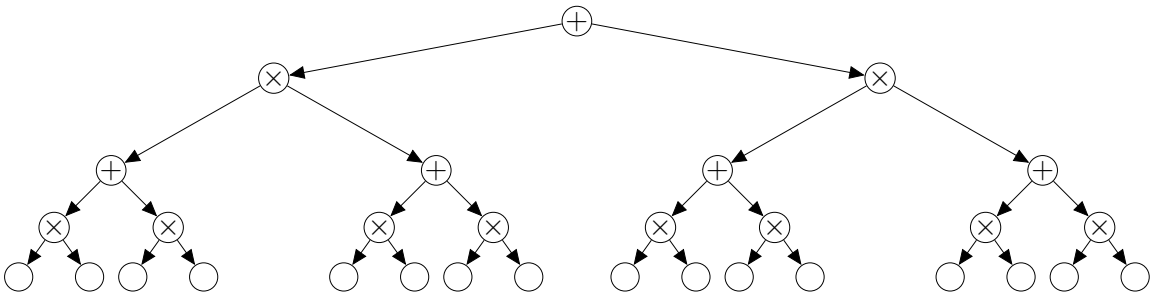


Figure 3.1: The computation graph G of an SPN.

Definition 3.2.1. A function $\psi : X \to Y$ is a *scope function* if it satisfies the following conditions:

1. If N is the root node, then $\psi(N) = X$.
2. If N is a sum or product, then $\psi(N) = \cup_{N' \in ch(N)} \psi(N')$.
3. For each $S \in \boldsymbol{S}$ we have $\forall N, N' \in ch(S)$: $\psi(N) = \psi(N')$ (completeness).
4. For each $P \in \boldsymbol{P}$ we have $\forall N, N' \in ch(P)$: $\psi(N) \cap \psi(N') = \emptyset$ (decomposability).

The completeness and decomposability conditions restrict the scope of an SPN to ensure every node is a probability distribution over their scope. Therefore, if an SPN is complete and decomposable, the SPN is also tractable. This is because marginalization and conditioning can be performed only to the leaf nodes, which contain only subsets of the random variables. By having the completeness condition, we ensure every child of a specific sum node has the same scope. In addition, the decomposability condition ensures the scope of every child of a product node must be pairwise disjoint [36]. Furthermore, if the scope function of an SPN respects these conditions, the SPN is valid. This thesis considers only valid SPNs.

The scope function defines the effective structure of an SPN as it combines the computational graph with the scopes of the nodes. Figure 3.2 shows an effective SPN structure with five random variables $\{X_1, X_2, X_3, X_4, X_5\}$. The sub-trees having empty scopes are evaluated to a constant 1.

Next, we will introduce the weights associated with a sum node, denoted by $\boldsymbol{w}$. These weights sum up to one and they assign data points to the children of s sum node [6]. The leaves in an SPN define a set of likelihood models, which are one-dimensional distributions for each random variable and the parameters of those are denoted by $\boldsymbol{\theta}$. In other words $\boldsymbol{\theta} = \{\boldsymbol{\theta}_L\}$, so $\boldsymbol{\theta}$ denotes a collection of distribution parameters for all leaf nodes. For example, if one of the likelihood models would be *Gaussian*, $\boldsymbol{\theta}$ would then include the *Gaussian* distribution parameters, mean μ and variance σ^2.

We are now ready to give a formal definition of an SPN.

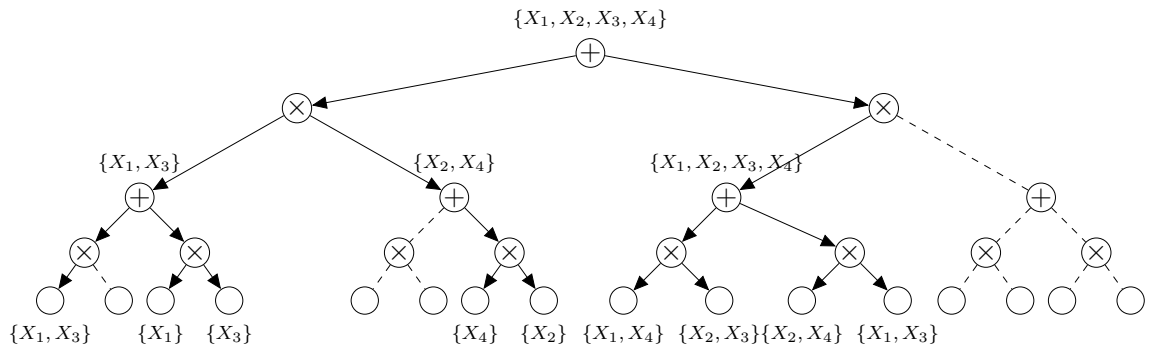


Figure 3.2: The effective structure of an SPN. The sum and leaf nodes have their scopes marked next to them. The sub-trees drawn with dotted lines have empty scopes.

Definition 3.2.2. An SPN $\mathcal{S}$ is a tuple $(G, \psi, \boldsymbol{w}, \boldsymbol{\theta})$, where G is a computational graph, ψ is a scope function over G, $\boldsymbol{w}$ is a set of weights associated with the sum nodes and $\boldsymbol{\theta}$ is a set of leaf node parameters.

Next, we will discuss how an SPN defines the joint distribution of the variables.

3.2.2 Node types and associated distributions

As earlier mentioned, an SPN has three node types, sums, products and leaves. Each of the nodes regardless of the node type define a local probability over their scope. Figure 3.3 visualizes the local probabilities each node in an SPN defines over their scope.

An SPN is built alternating between sum and product nodes, and only the bottom level consists of leaf nodes. In the Figures 3.1, 3.2 and 3.3, the sum nodes are denoted by + label, product nodes by $\times$ label and the leaf nodes at the bottom of the graph do not have any label assigned to them.

The leaf nodes in an SPN define a set of D likelihood models which are one-dimensional distributions for each random variable in their scope. These likelihood models can be picked freely and the leaf nodes contain their parameters. These parameters are determined by the input data. In some SPNs each leaf node defines values for one of the random variables. In the case of binary data, each binary variable would have two associated leaves, for each of the outcomes of the variable [3]. The leaves would then define a distribution over a fixed scope. However, in this thesis we will focus on SPNs associating arbitrary number of leaves for each variable.

Here, each leaf L is parametrized by $\boldsymbol{\theta}_L$ that consists of D parameters $\theta_{L,1}, ..., \theta_{L,D}$ of one-dimensional distributions, for example, *Gaussian* or *Poisson*. In addition to

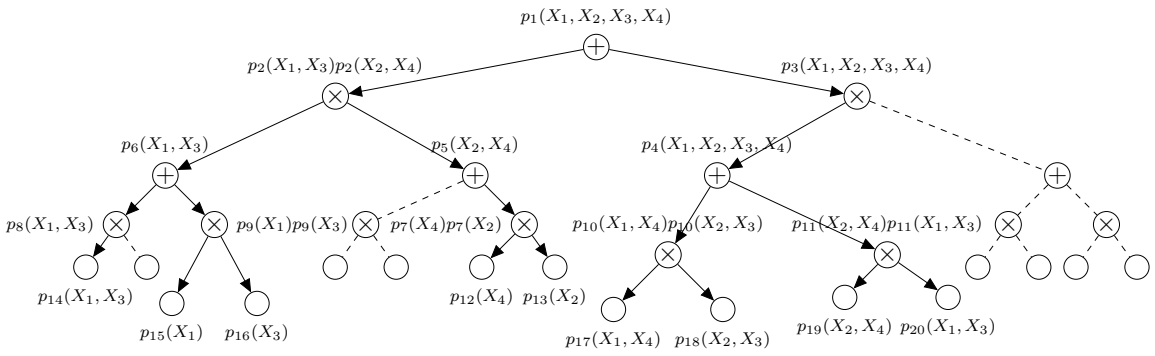


Figure 3.3: The local probability which each node defines over its scope.

this, for a given scope $\psi(L)$, the distribution in leaf L factorizes as

$$L = \prod_{X_i \in \psi(L)} p(X_i|\theta_{L,i}). \tag{3.2}$$

In this case the leaf nodes define distributions over all 2^D possible scope at all times [6]. This is an essential part of an SPN.

In addition to leaves, there are sum nodes, which calculate a weighted sum over their children. There are weights associated with each sum node, one for every child node they have. These weights are nonnegative, they sum up to one and they are used when assigning data points along the network. The third rule concerning a valid SPN, mentioned in subsection 3.2.1, describes that all children of a sum node must have the same scope. In essence, a sum node encodes a mixture model over all sub-SPNs which are rooted at its children. Each sum node has a set of latent variables for its children. Sum nodes, with the help of these latent variables, help to explain the interactions and dependencies between the random variables in their scopes. When a sum node is evaluated, the latent variables are summed out.

The final node type, product nodes, calculate a product over their children. They essentially calculate the product of distributions described by their child nodes over some locally independent features [37].

Equation 3.3 presents how probability p_v, associated with node v of any type, is calculated.

$$p_v = \begin{cases} p(X_i) & \text{if node is a leaf node over } X_i. \\ \prod_{u \in ch(v)} p_u & \text{if } v \text{ is a product node.} \\ \sum_{u \in ch(v)} \boldsymbol{w}_v p_u & \text{if } v \text{ is a sum node.} \end{cases} \tag{3.3}$$

Inference operations, like computing marginal probabilities, can be performed by applying the node operations recursively, starting from the root node.

In essence, sum and product nodes are like the hidden layers in a neural network,

as they are intermittent part of the network [38]. The structure and parameters, like the weights associated with sum nodes, must be learned using some learning method.

3.2.3 Applications of SPNs

As SPNs represent well complex distributions, they are useful for many purposes when there is a need for modeling complex distributions. This is the case in many real-life situations, as there are often data with high number of dimensions and many dependencies. SPNs have been shown to be comparable to other graphical models in likelihood but they offer better accuracy and speed [39]. Because of these reasons SPNs are used for many purposes.

Among other use-cases, SPNs are used, for example, in sequence labeling [40], where discriminative SPNs are used successfully to perform efficient sequence labeling. Image completion can also be performed with SPNs successfully [41]. SPNs are as well used in robotics for making probabilistic semantic maps [42] and spatial knowledge representations [43]. Here the fact that SPNs handle high dimensional data well is a key factor. The goal in both of these is to model the surroundings of a moving robot, which means there is a large amount of data that needs to be represented compactly and efficiently.

One further use case of SPNs is activity recognition [44]. There is need for modeling complex distributions and being able to perform fast and exact inference, but there is also a need of highly expressive models in this field. Here activities can be modeled with activity structure that is an arrangement of sub-activities. SPNs consist of multiple layers of nodes, which can efficiently represent these kinds of activity relations in a way that is easy to read and understand.

4. Learning Sum-Product Networks

Learning of a sum-product network consists of two phases, parameter and structure learning. There are many well-developed techniques for parameter learning, but the situation is different for learning of the structure. The first attempt at SPN structure learning was made by Dennis and Ventura in 2012 [3], where structure learning was done by first performing instance clustering to form sum nodes after which variable partitioning was performed to form product nodes, then alternating between these in top-down fashion. Although, this only optimizes some local criterion making the structural improvements only local. The ultimate goal of the learning process is to achieve global structural improvements, which lead to globally good SPN structure. SPN structure is commonly handpicked as identifying a good SPN structure is hard [3]. SPNs introduce many layers of latent variables over observed variables, and the number of variables, the connections between them or the size of the latent variable layers can affect the performance of the SPN. This is why it is often hard to identify a good structure for an SPN and why it is so important to achieve a globally good structure.

4.1 Learning algorithms for SPNs

There are many approaches for learning sum-product networks. Structure learning is well presented in related work and there are many algorithms which do not combine structure and parameter learning. LearnSPN [39] is a structure learning algorithm based on clustering variables using Expectation maximization (EM) algorithm. EM is used for learning sum nodes and graphical model structure learning is performed for product nodes. When learning the sum nodes, EM is performed over naive Bayes mixture model with exponential prior preventing overfitting. *Overfitting* happens when a model learns the test data set too deeply and performs poorly with other data. The algorithm alternates between partitioning features into sets creating a product node and clustering data points creating a sum node. This algorithm performs local structure improvements and does not optimize the global structure of the SPN.

Another structure learning algorithm was proposed by Adel et al. [45] which is

based on clustering variables using Singular value decomposition (SVD) algorithm. In the algorithm, SVD is used in extracting rank 1 matrices. *A matrix rank* is the dimension of its column's vector space. The algorithm strives for global structure improvements, checking the data matrices across instance and variable dimensions. This makes the algorithm efficient, as it only has to perform one step, searching the submatrices, whereas other clustering algorithms perform two steps, clustering and identifying for independence. These submatrices estimate correlations and are used globally in the SPN to split data.

Other clustering approaches include an algorithm using the notation of region graphs [3]. This algorithm uses k-means algorithm for clustering variables into sets under latent variables describing dependencies between the variables. The goal of the algorithm is to identify variables strongly interacting with each other. The initial SPN architecture is learned from data.

Another structure learning algorithm with a greedy structure search presented is SearchSPN [5]. This is the first SPN structure learning algorithm which performs search operations over the SPN. The benefit here is that the SPN is not restricted in being tree structured. There is also another structure learning algorithm called Prometheus [37] which uses clustering. It is used in graph partitioning, where the partitioning constructs a maximum spanning tree.

Automatic Bayesian density analysis (ABDA) [46] incorporates Bayesian statistics to the learning algorithm. The algorithm performs Bayesian inference through Gibbs sampling and uses SPN learning technique called MSPN [47] for partitioning data over mixed discrete and continuous features by using randomized approximation of the Hirschfeld-Gebelein-Renyi maximum correlation coefficient. In this approach, the SPN leaf nodes hold likelihood model dictionaries for features. This approach is good for discovering data types, and handling complex data and likelihood models. Although, the learning does not achieve global improvements on the structure, as it performs posterior inference over parameters in the leaves which lead to local structure improvements. The overall structure of an SPN is kept fixed while only the leaf node parameters are being adapted in the learning process. The algorithm relies on predefined structure for an SPN.

4.2 Bayesian learning of Sum-Product Networks

The Bayesian SPN model discussed in this section was presented by Trapp et al. in 2019 [6]. In the article, they present the sum-product network learning algorithm based on Bayesian statistics. In this algorithm, SPN learning alternates between first updating sum weights and leaf parameters, and then updating the SPN structure. This is done

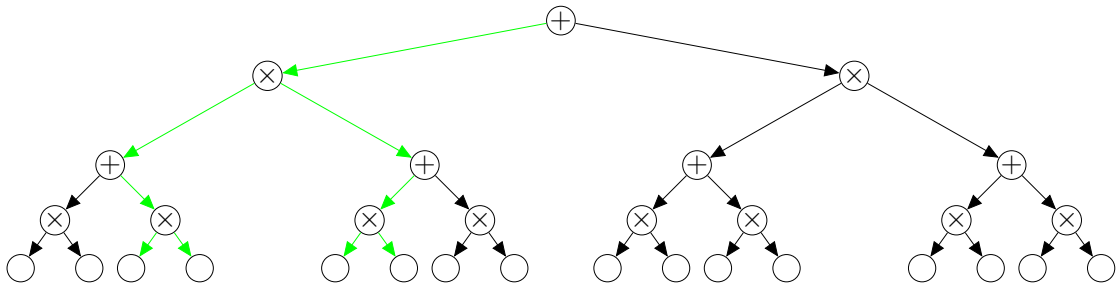


Figure 4.1: An induced tree in a computational graph G.

based on data fed to the network. The goal being that the SPN learns the patterns and distributions from the data, and can later use the information on other sets of data.

4.2.1 Induced trees in parameter learning

As said earlier, the sum nodes in an SPN cluster data over their child nodes. For each data point x_n and sum node S, there is a latent variable $z_{S,n}$ assigned indicating a component [46]. The latent variable has as many states as the sum node has children and a categorical distribution given by the sum node weights. It essentially represents the chosen child for each of the data point fed to the SPN. An important aspect when the data is clustered by the sum nodes is an induced tree which is formed when an assignment of $z_{S,n}$ is made [6]. Induced tree $\mathcal{T}$ is a path starting from the root node of an SPN and it is a sub-graph of computational graph G. An induced tree is created such that at each sum node one edge is selected based on the latent variable $z_{S,n}$ and at each product node, every edge is selected. In figure 4.1, the induced tree is presented with the green tree path.

The SPN distribution factorizes when conditioned with an induced tree $\mathcal{T}$. Because of this, the whole distribution of the SPN can be thought of as a mixture of all these factorized distributions over all of the induced trees:

$$\mathcal{S}(x) = \sum_{\mathcal{T} \sim \mathcal{S}} \prod_{S \in \mathcal{T}} w_S \prod_{L \in \mathcal{T}} L(x). \tag{4.1}$$

Here $L(x)$ is the leaf node L evaluated at its scope $\psi(L)$ with the restriction of x.

The conditional probability distribution of x given a latent variable z is

$$p(x|y) = \prod_{L \in T(z)} L(x). \tag{4.2}$$

And the prior in this case is

$$p(z) = \prod_{S \in G} w_{S,z_S}. \tag{4.3}$$

Here w_{S,z_S} is the sum weight indicated by a latent variable z_S. The SPN distribution represented as a latent variable model can be obtained by marginalizing out the latent

variables $\boldsymbol{z}$ from the joint distribution $p(x, z) = p(x|z)p(z)$. By placing the equations 4.2 and 4.3 into this, the following is obtained

$$\begin{aligned}\sum_{\boldsymbol{z}} \prod_{S\in G} w_{S,z_S} \prod_{L\in T(\boldsymbol{z})} L(x) &= \sum_{\mathcal{T}} \sum_{\boldsymbol{z}\in T^{-1}(\mathcal{T})} \prod_{S\in G} w_{S,z_S} \prod_{L\in T(\boldsymbol{z})} L(x) \\ &= \sum_{\mathcal{T}} \prod_{(S,N)\in\mathcal{T}} w_{S,N} \prod_{L\in\mathcal{T}} L(x) \underbrace{\left(\sum_{\bar{\boldsymbol{z}}} \prod_{S\in\bar{S}_{\mathcal{T}}} w_{S,\bar{z}_S}\right)}_{=1} \\ &= \mathcal{S}(x).\end{aligned} \tag{4.4}$$

This is the SPN distribution as a latent variable model, where z is marginalized out.

4.2.2 Region graph

Region graph is a vectorized representation of an SPN, used to help presenting the scope function ψ through the computational graph. Computational graph in this thesis follows a tree-shaped region graph. Region graph is a tuple $(\mathcal{R}, \xi)$, where ξ indicates the scope function and the $\mathcal{R}$ indicates a directed acyclic graph consisting of partition nodes A and region nodes R. $\boldsymbol{A}$ and $\boldsymbol{R}$ represent groups of all partition nodes and all region nodes in the graph. The graph $\mathcal{R}$ has only one root node.

Definition 4.2.1. The scope function of a region graph is a function $\xi : \boldsymbol{Q} \mapsto 2^X$ and it has the following properties:

1. If Q is the root node, then $\xi(Q) = X$.
2. If Q is a region with children or a partition, then $\xi(Q) = \cup_{Q'\in ch(Q)}\xi(Q')$.
3. For each $A \in \boldsymbol{A}$ we have $\forall R, R' \in ch(A)$: $\xi(R) = \xi(R') = \emptyset$.
4. For each $R \in \boldsymbol{R}$ we have $\forall A \in ch(R)$: $\xi(R) = \xi(A)$.

These properties correspond with the properties of a scope function of an SPN.

An SPN can be formed based on a region graph. For the root region node there would be a sum node introduced and for leaf regions there would be I SPN leaves introduced. For any other region that is not root or leaf, there would be J sum nodes introduced. Here I and J are hyperparameters of the model. For every partition in $\mathcal{R}$, all cross products of nodes from A's child regions $ch(A) = \{R_1, ..., R_K\}$ would be introduced. Here $\boldsymbol{N}_k$ is a set of nodes in each region R_k, forming all the possible cross-products is then $N_1 \times \cdots \times N_K$ for $1 \leq k \leq K$. Here $N_k \in \boldsymbol{N}_k$. The scope function ψ of an SPN is inherited from ξ.

4.2.3 Region graphs in structure learning

When performing structure learning, the dimensions of a data set are passed through an SPN. There is a latent variable $y_{A,d}$ assigning a dimension d to a particular child $R_1, ..., R_{|ch(A)|}$ of a partition A. This latent variable has as many states as the partition has children. When passing the data dimensions through the network, an induced scope function is used. Induced scope function defines that when introducing a latent variable to assign a dimension to a child, all partitions before this in the induced tree have decided the assign the dimension to this path. The induced scope function defines a unique path in the region graph since it is tree shaped.

4.2.4 Bayesian SPN model

In the Bayesian SPN model, the sum weights, leaf node parameters and latent variables $\boldsymbol{y}$ are equipped with priors. Sum weights and latent variables $\boldsymbol{y}$ use *Dirichlet* priors. From this we get the following Bayesian SPN model:

$$\begin{aligned} \boldsymbol{w}_S \mid \alpha &\sim Dir(\boldsymbol{w}_S \mid \alpha)\, \forall S, & z_{S,n} \mid \boldsymbol{w}_S &\sim Cat(z_{S,n} \mid \boldsymbol{w}_S)\, \forall S\, \forall n, \\ \mathbf{v}_P \mid \beta &\sim Dir(\mathbf{v}_P \mid \beta)\, \forall P, & y_{S,n} \mid \mathbf{v}_P &\sim Cat(v_{P,d} \mid \mathbf{v}_P)\, \forall P\, \forall d, \\ \theta_L \mid \gamma &\sim p(\theta_L \mid \gamma)\, \forall L, & \boldsymbol{x}_n \mid \boldsymbol{z}_n, \boldsymbol{y}, \theta &\sim \prod_{L \in T(\boldsymbol{z}_n)} L(\boldsymbol{x}_{\boldsymbol{y},n} \mid \theta_L)\, \forall n. \end{aligned} \tag{4.5}$$

The plate notation of the model is presented in the figure 4.2 where the directed edges present dependencies between variables.

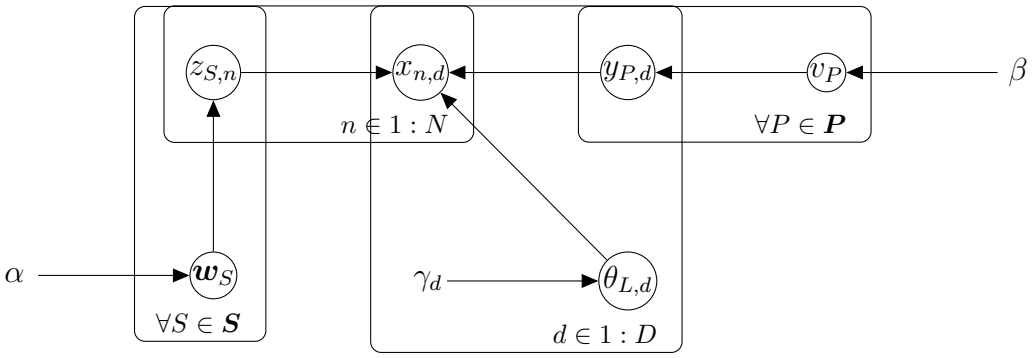


Figure 4.2: Plate notation of the Bayesian SPN model.

4.3 Algorithm for Bayesian learning of Sum-Product Networks

In this section we will discuss the algorithm implemented for Bayesian learning of an SPN, where we simultaneously perform parameter and structure learning by alternating

between them and aim for globally good SPN structure. The algorithm* implemented for this thesis is a variation of the algorithm from the article Bayesian Learning of Sum-Product Networks by Trapp et al. [6].

4.3.1 Parameter Learning

In parameter learning, the latent variables z, the sum weights w and the leaf parameters θ are sampled. We begin by sampling the latent variables for each sum node. They are sampled by using SPN ancestral sampling which means the sampling happens by forming induced trees [46]. Induced trees were discussed in subsection 4.2.1. This is done for one data point at the time, the data point in question is passed through the network along the induced tree the latent variable assignments are forming. The latent variable assignments are sampled using Metropolis-Hasting MCMC. The used proposal distribution is a discrete uniform distribution. In this way, the leaf nodes in the induced trees end up having some sets of data points, based on which the leaf node parameters are updated later. After the data point has ended up to a certain leaf, it is assigned to some likelihood model in the leaf. The set of all likelihood models for each dimension is denoted by $\mathcal{L}$. The sample is assigned to a likelihood model inside the leaf using a latent variable s which is sampled in the same way as the latent variable z. The latent variable s has also likelihood weights Ω associated with it and it has as many stated as there are likelihood models. Each leaf node has a likelihood weight assigned to each likelihood model.

After the latent variables are sampled, it is time to sample the sum weights. These are sampled from a *Dirichlet* distribution which is $Dir(\alpha + c_{S,1}, ..., \alpha + c_{S,K_S})$. Here $c_{S,k}$ is the amount of data points or latent variable assignments for sum node S, i.e., $c_{S,k} = \sum_{n=1}^{N} \mathbb{1}[z_{S,n} = k]$. In other words, it is a counter of all of the data points passed through the sum node. The α is a predefined constant, for example, one. After this, the likelihood weights Ω need to be sampled similarly from a *Dirichlet* distribution $Dir(\alpha + b_{d,1}, ..., \alpha + b_{d,l_d})$, where $b_{d,l} = \sum_{n=1}^{N} \mathbb{1}[s_n^d = l]$. In other words, this is a counter of how many data points has been assigned to a specific likelihood model. Similarly, here α is a constant.

After sum weight sampling, the leaf parameters are updated by using conjugate distributions. After previous steps, the likelihood models in leaf nodes have some set of data points, but it might also be that the leaf node did not get included into any induced tree and was left without data. It might as well be that some specific likelihood model in a leaf was left without data. As mentioned earlier, each of the leaf nodes has a predefined set of likelihood models for all of the random variable. These likelihood models

*https://github.com/mnoora/Bayesian-learning-of-SPNs

are equipped with suitable priors, making it viable to use the features of conjugate distributions. The data points in each likelihood model in each leaf are used to update model parameters using the aspects of conjugate distributions. The hyperparameters of a distribution can be updated with the data in a leaf, and the hyperparameters can in turn be used to update the posterior parameters. These posterior parameters can be used as the prior distribution parameters in the next iteration, enabling the model to pick up new information on every iteration.

4.3.2 Structure Learning

In structure learning, the dimensions of the data are spread across the SPN. Collapsed Gibbs sampler is used to achieve this, details about the method were discussed deeper in Section 2.3.3. The method is used to sample all $y_{A,d}$ assignments from a conditional distribution where the dependency structure $\mathbf{v}$ is marginalized out. $y_{A,d}$ is a latent variable assigning dimension d to a particular child of partition node A. The probability of assigning dimension d under child k under partition node A is the following:

$$p(y_{A,d} = k|\boldsymbol{y}_{A,\not d}, \boldsymbol{y}_{\boldsymbol{A}\setminus A,d}, X, \boldsymbol{z}, \theta, \beta) \propto p(y_{A,d} = k|\boldsymbol{y}_{A,\not d}, \beta)p(X|y_{A,d} = k, \boldsymbol{y}_{\boldsymbol{A}\setminus A,d}, \boldsymbol{z}, \theta). \tag{4.6}$$

Here $\mathbf{y}_A$ indicates the set of all dimension assignments to A, $\mathbf{y}_{A,\not d}$ indicates $\mathbf{y}_A$ without dimension d and $\mathbf{y}_{\mathbf{A}\setminus A,d}$ indicates all assignments of dimension d on all partition nodes except node A. The same equation was presented in the article by Trapp et al. [6] where the left and right sides of the equation were marked as equal although in reality, they are proportional to each other.

The first term on the right side of the equation can be calculated with the following

$$p(y_{A,d} = k|\boldsymbol{y}_{A,\not d}, \beta) = \frac{\beta + m_{A,k}}{\sum_{j=1}^{|ch(A)|} \beta + m_{A,k}}. \tag{4.7}$$

Here m is a component count, $m_{A,k} = \sum_{d\in\psi(A)\setminus d} \mathbb{1}[y_{A,d} = k]$. In other words, it is the number of dimension assignments the node A has received.

The second term on the right side of the equation is a product of the marginal likelihood terms of all product nodes P in A. It is known that assignments to $y_{A,d}$ are more probable if other dimensions have been assigned to the same child node, making the component count larger.

With this formula, we can learn a structure for an SPN.

5. Experimental Results

This chapter presents and discusses the experiments done in this thesis. The main goal of the experiments is to observe the outcomes of using the algorithm developed in different scenarios which include using data sets of different sizes and data with varying complexity. Lastly, conclusions are discussed.

5.1 A small and simple SPN

The first experiment focuses on exploring the abilities of a small and simple SPN in learning patterns from a simple data set.

5.1.1 Details of the experiment

In this experiment, we will test the ability of an SPN in learning a data set. We will use data sets of three different sizes, of 1000, 5000 and 20000 data points, to see how the data set size affects the learning results. We expect to obtain more accurate results when using larger data sets.

The experiment is kept simple by using data sets with only a few dimensions and distributions. In addition to this, the SPN is kept small and simple. The data sets used here were generated from an SPN constructed with SPFlow framework [48], hence the used data is synthetic. The approach of using synthetic data to evaluate machine learning models is widely used [49]. The three data sets have the same dimensions and distributions, the only difference in them being the size of the data set.

SPFlow* is an open-source Python library for creating SPNs and operating with them. It offers utilities for performing probabilistic operations with SPNs such as inference and computing marginals and conditionals. In addition to this, it helps in visualization as it has functionality for plotting an SPN based on the configuration code [48]. The data set used in the experiment was generated by using the functionality of the framework.

The SPN presented in Figure 5.1 is used for generating a simple data set with two

*https://github.com/SPFlow/SPFlow

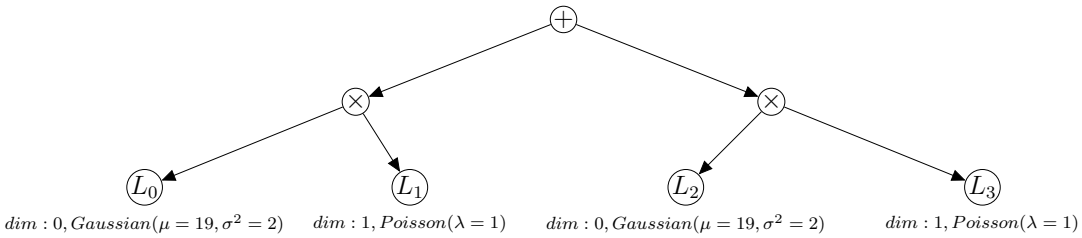
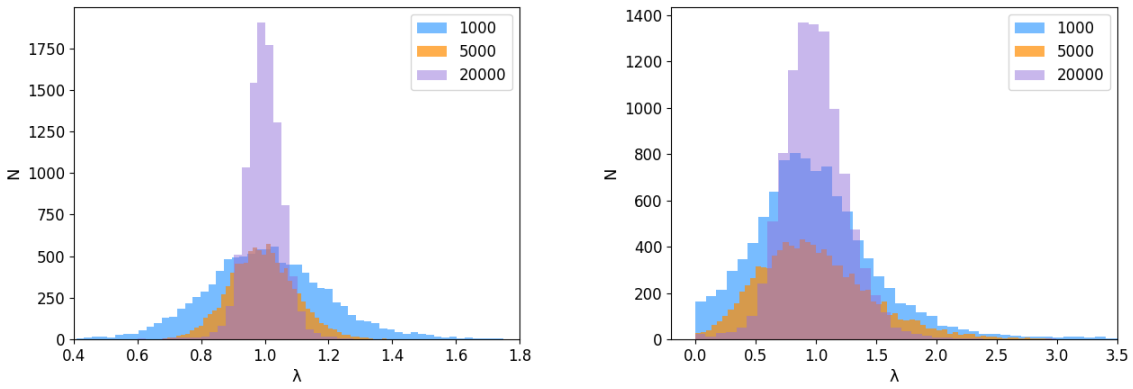


Figure 5.1: The SPN used for generating data.



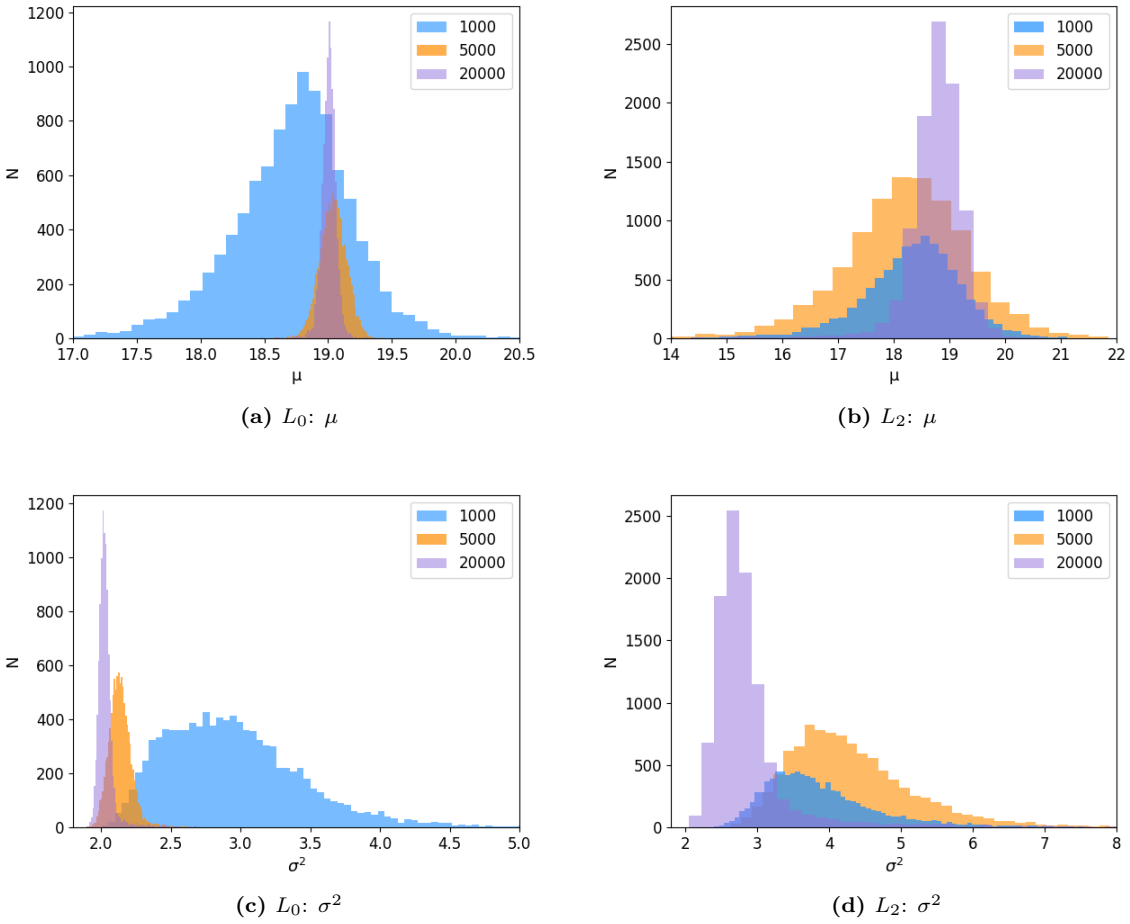
(a) L_1 **(b)** L_3

Figure 5.2: Histograms of the distribution of *Poisson* λ for leaves L_1 and L_3 for experiments with data sets of 1000, 5000 and 20000 data points.

dimensions, $Gaussian(\mu = 19, \sigma^2 = 2)$ as the first dimension and $Poisson(\lambda = 1)$ as the second dimension. In addition to this SPN, another SPN with the same structure is constructed without any weights or parameters. The data set produced by the data generating SPN is fed through the second SPN with empty weights and parameters, with the goal of it learning from the data the same weights and parameters as the data generating SPN had.

5.1.2 Results

In this first experiment, the goal is to experiment how a small SPN with a simple structure, without any weights or parameters, can learn these from data. The experiment was run for 10000 iterations with data sets consisting of 1000, 5000 and 20000 data points. After the learning process, the weights and parameters between the data generating SPN and the test SPN were compared to see how close they would be to each other. It is worth noticing that the parameter values will not be exact matches, because of the nature of the learning algorithm. If the SPN has been able to learn successfully, the sampled parameters should center around the corresponding original parameter values from the generating SPN.



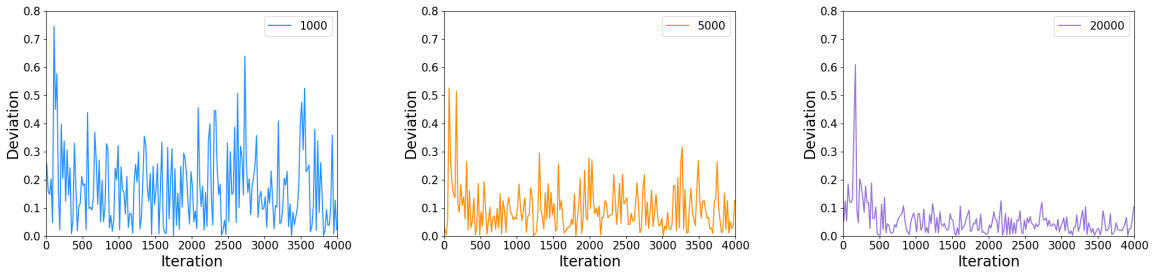
(a) L_0: μ **(b)** L_2: μ

(c) L_0: σ^2 **(d)** L_2: σ^2

Figure 5.3: Histograms of the distribution of *Gaussian* μ and σ^2 for leaves L_0 and L_2 for experiments with data sets of 1000, 5000 and 20000 data points.

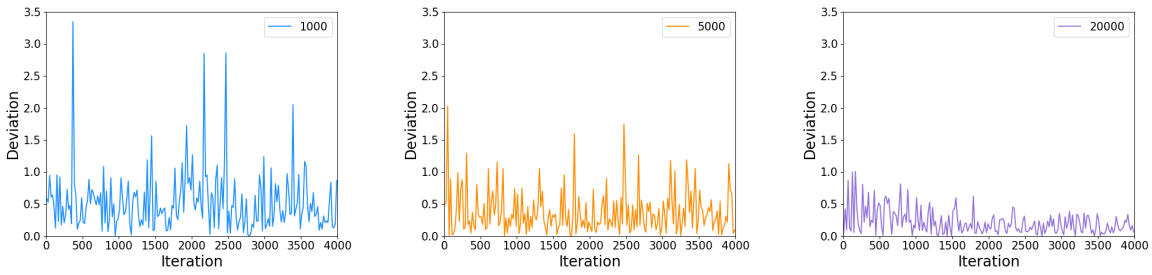
Figure 5.2 shows the histograms of *Poisson* λ parameter values for leaves L_1 and L_3 for the experiments with the three different size data sets. The histograms are centered around 1, which is the target value in this experiment for λ. We can see that the histograms are narrower and more focused when using larger data sets. This indicates that by having more data, the results are more precise. In figure 5.2a, the histograms are more closely centered around 1 and in figure 5.2b, the histograms are wider.

Figure 5.3 shows the corresponding histograms for leaves L_0 and L_2 for parameters μ and σ^2. Similarly, here we can see that the more data there is in the experiment, the more the histograms are centered around the target values, 19 for μ and 2 for σ^2. One interesting thing to notice is in figure 5.3c where we can see the histogram for the data set of size 1000 centering approximately around 3. We can see that by increasing the size of the data set, the histogram shifts towards the correct value, 2. From the histograms for L_3 we can see that the histogram for data set of 5000 is the widest one.



(a) Data set of 1000 data points. **(b)** Data set of 5000 data points. **(c)** Data set of 20000 data points.

Figure 5.4: Deviation of *Poisson* λ parameter samples compared to the parameter of the generating model as a function of iterations for leaf L_1 with three different size data sets.



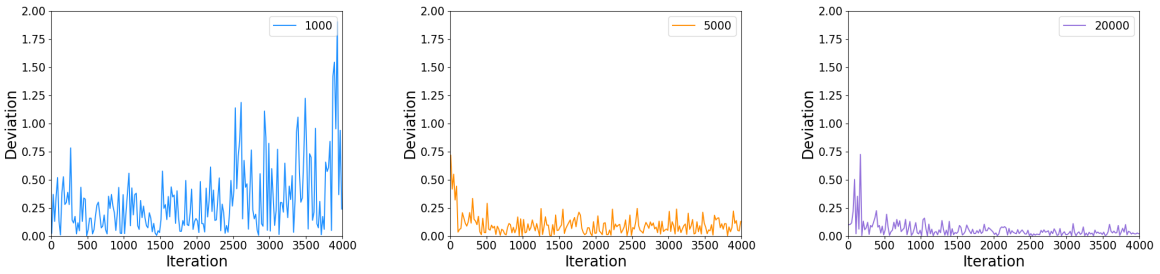
(a) Data set of 1000 data points. **(b)** Data set of 5000 data points. **(c)** Data set of 20000 data points.

Figure 5.5: Deviation of *Poisson* λ parameter samples compared to the parameter of the generating model as a function of iterations for leaf L_3 with three different size data sets.

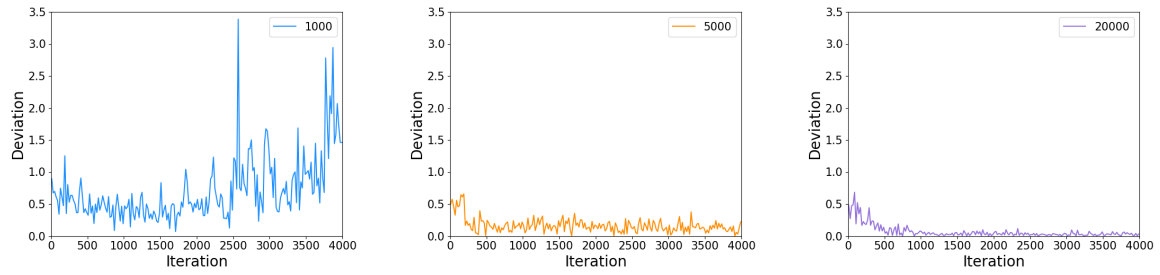
This result is surprising because the assumption was that the histogram with less data would be wider, like we have seen for the other leaves and parameters. Although, the histogram for the largest data set is the narrowest, as we expected.

Figure 5.4 and figure 5.5 show the deviation of the samples compared to the parameters of the generating model for *Poisson* λ as a function of the iterations for leaves L_1 and L_3. The deviation is expected to decrease as the number of iterations grow, as the SPN should learn the parameters better and therefore decrease the deviation of the parameter samples. Although, it is important to notice that some amount of deviation should always stay as the parameters have some variance in them. From the figures, it can be clearly seen that the deviation goes towards zero as the number of iterations grow when using the two largest data sets. For the largest data set, the deviation decreases the most. When using the smallest data set, the deviation does not decrease. This can be understood to happen because the size of the data set is small and causing the learning process to have difficulties in being able to spot patterns from the data.

We can notice that the deviation stabilizes after certain number of iterations have passed when using the two largest data sets. This happens roughly at 500 iterations for the second largest data set and at 1000 iterations for the largest data set. This



(a) Data set of 1000 data points: μ **(b)** Data set of 5000 data points: μ **(c)** Data set of 20000 data points: μ

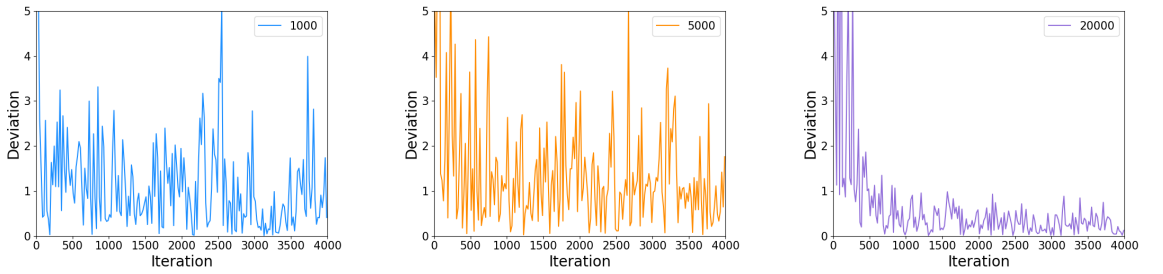


(d) Data set of 1000 data points: σ^2 **(e)** Data set of 5000 data points: σ^2 **(f)** Data set of 20000 data points: σ^2

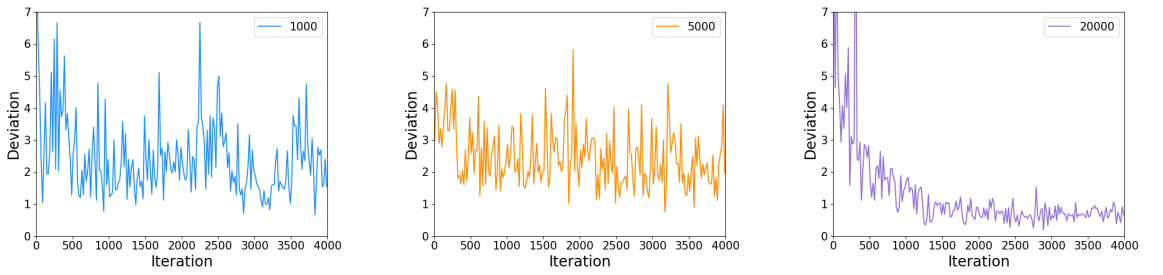
Figure 5.6: Deviations of *Gaussian* μ and σ^2 parameters compared to the parameters of the generating model as a function of iterations for leaf L_0 for experiments with three different size data sets.

can be understood to happen because the learning process will require more iterations when dealing with larger sets of data. The fact that the deviation rate stabilizes after a rather small number of iterations can be explained by the simplicity of the SPN and the used data. For this reason, the figures go only to 3000 iterations whereas the whole experiment was run for 10000 iterations. There were not any clear decreases of the deviation between iterations 3000 and 10000 which led to the decision to only show a subset of iterations in these figures, as only they show relevant results.

Figure 5.6 and figure 5.7 show the deviation of *Gaussian* parameters μ and σ^2 compared to the parameters of the generating model as a function of iterations for leaves L_0 and L_2. Like in the previous figures for *Poisson* parameters, we can see the deviation significantly decreasing as the number of iterations grow when using the two largest data sets. Between these two sizes, the deviation for the largest one decreases the most. Here as well, the deviations stabilize after the same number of iterations has passed, like described earlier. Afterwhich, there are no clear signs of the deviation decreasing. For the leaf L_2, there is significantly more deviation in the beginning of the learning process than for leaf L_0, but it also decreases efficiently with the two largest data sets. The deviation when using the smallest data set does not decrease when the iterations grow, which can be explained by the small amount of data.



(a) Data set of 1000 data points: μ **(b)** Data set of 5000 data points: μ **(c)** Data set of 20000 data points: μ



(d) Data set of 1000 data points: σ^2 **(e)** Data set of 5000 data points: σ^2 **(f)** Data set of 20000 data points: σ^2

Figure 5.7: Deviations of *Gaussian* μ and σ^2 parameters compared to the parameters of the generating model as a function of iterations for leaf L_2 for experiments with three different size data sets.

5.2 A larger and more complex SPN

The second experiment focuses on exploring the ability of a larger and more complicated SPN to learn patterns from data. In addition to this, the data used in this experiment is more complicated.

5.2.1 Details of the experiment

The goal of this experiment is to examine how the learning algorithm functions with a larger SPN and a data set with more dimensions. This data set is produced by

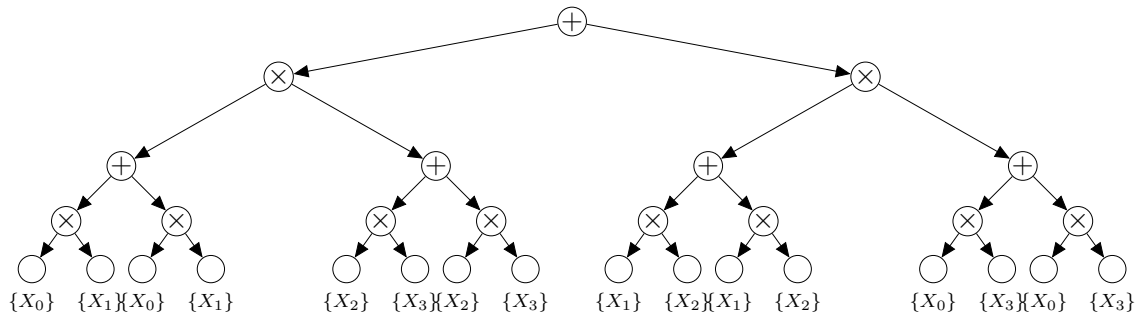
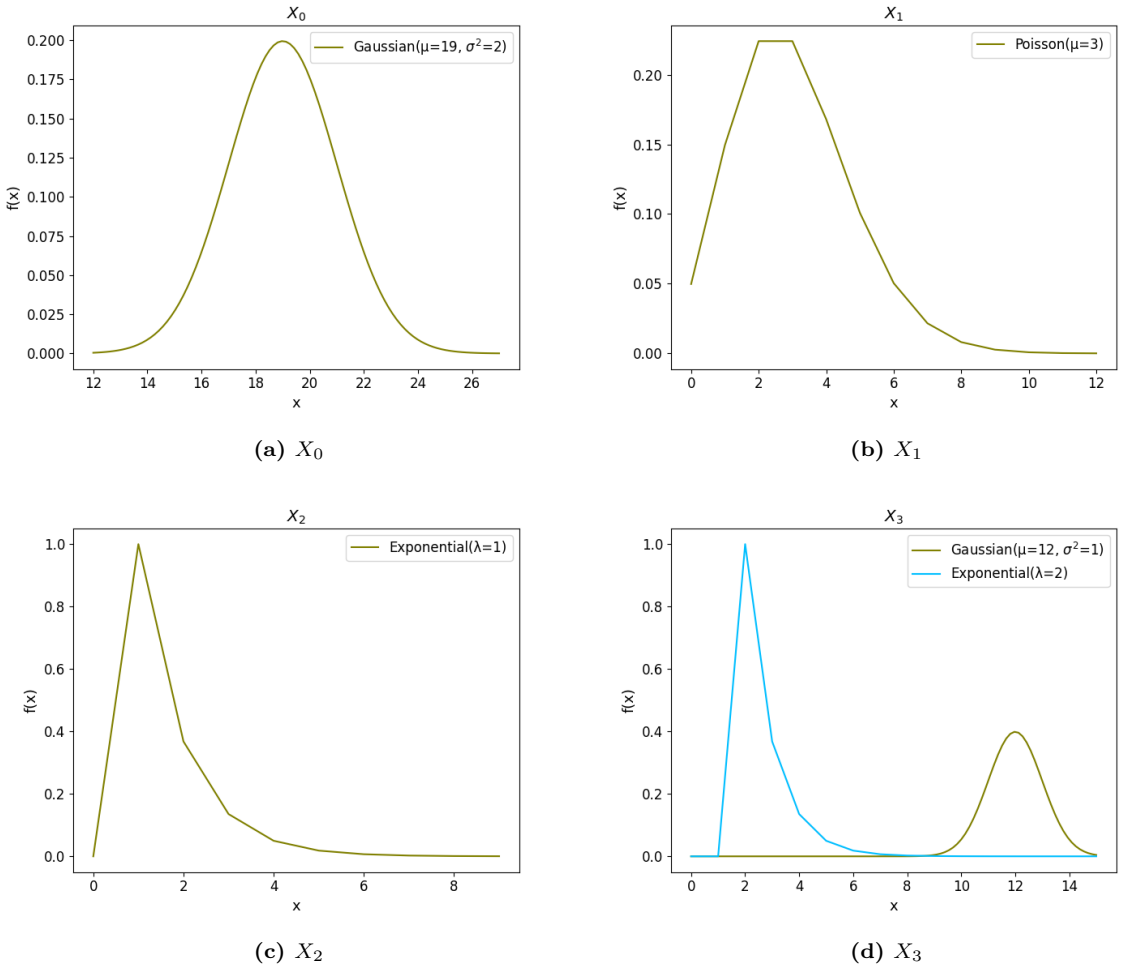


Figure 5.8: The SPN used for generating data.



(a) X_0 **(b)** X_1 **(c)** X_2 **(d)** X_3

Figure 5.9: Distributions for each dimension in the data set.

an SPN seen in the figure 5.8. Similarly to the last experiment, the data will be fed to another SPN and we aim to see how well the SPN can learn the structure and parameters from the data by comparing those to the structure and the parameters of the generating model. The SPN used in this experiment is notably larger than in the first experiment, making the learning task more challenging. With a larger SPN, there are more possibilities for the scope functions, for example. It is not guaranteed that the layout of the scope is optimal in the data generating SPN and because of this it is possible for the SPN to arrange differently and obtain a possibly more suitable setup for the data set in question.

From figure 5.9 we can see the distributions each of the dimensions contain in the data set. Dimensions X_0, X_1 and X_2 each contain only one distribution to restrict the amount of complexity in the data. Dimension X_3 has two distributions, *Gaussian* and *Exponential*, bringing more challenges for the learning algorithm as it would need to identify two different patterns from the data for this dimension.

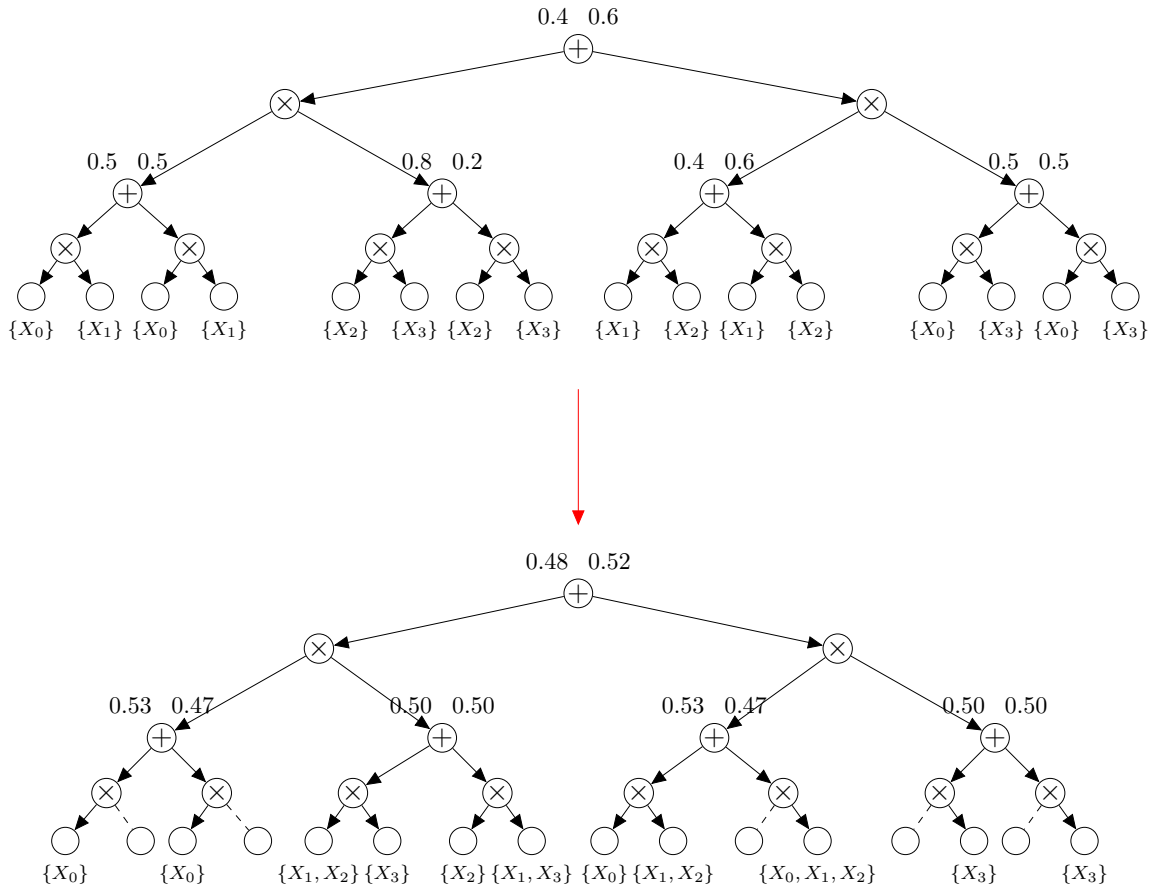
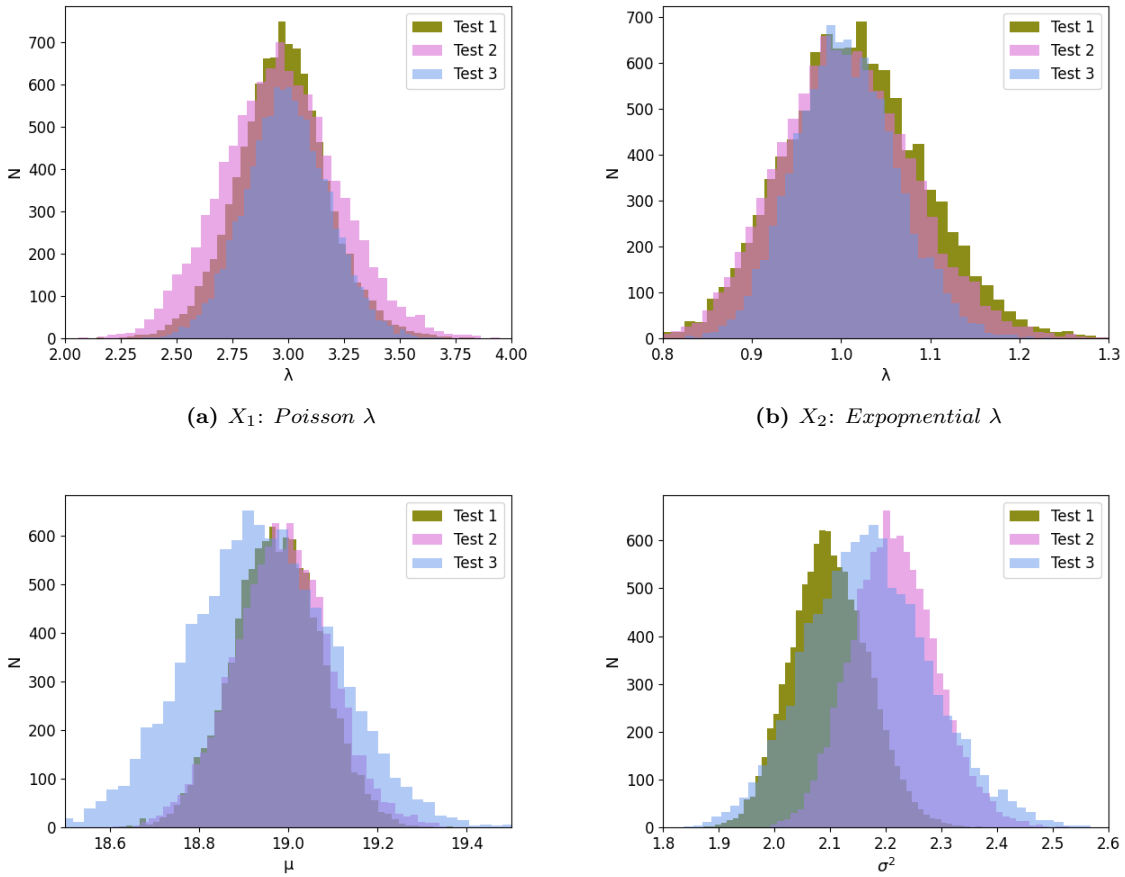


Figure 5.10: The learned effective structure at iteration 10000 with sum node weights. The sum node weights are shown with two decimals.

5.2.2 Results

In this experiment the goal was to see how a larger SPN can learn a data set. The experiment was run for 10000 iterations with a data set of 7000 data points. This experiment was repeated three times using different starting values and random seed. The expectation is that all of the outcomes from the repeated experiments should be similar.

From the figure 5.10 we can see the effective structure learned during the last iteration of the experiment, at iteration 10000. We can notice some differences in the structure between the SPNs, which was expected as there are more possibilities for the learning outcome with a larger SPN. Apart from the differences, there are also similarities. If we focus on the right side of the SPN, we can see that the arrangement of the dimensions is similar compared to the generating model. We can see that the main difference there is that the left sub-tree has an extra dimension, X_0, which originally was on the right sub-tree. When focusing on the left side of the SPN, we can see that here the differences and similarities are similar compared to the other side of the SPN.



(a) X_1: *Poisson* λ

(b) X_2: *Expopnential* λ

(c) X_0: *Gaussian* μ

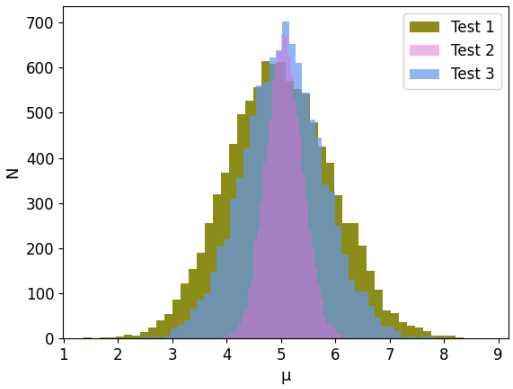
(d) X_0: *Gaussian* σ^2

Figure 5.11: Histograms of the leaf distributions for dimensions X_0, X_1 and X_2 from three repeated runs of the experiment.

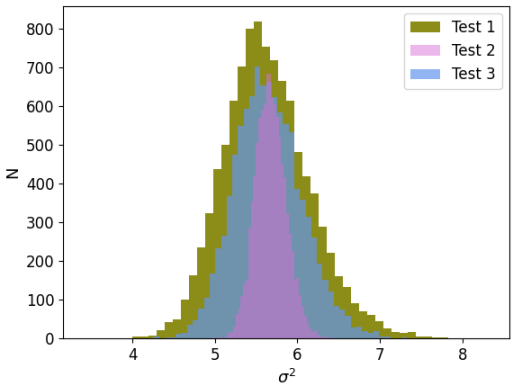
Here X_1 is in a different sub-tree, leaving X_0 to be the only dimension in the most left sub-tree in the SPN. As the learned effective structure viewed here is from the last iteration, Appendix A has additional effective structures from iterations 5000, 6000, 7000, 8000 and 9000 with sum node weights attached to them to shed light on the structure during the learning process.

Another thing we can notice from figure 5.10 are the learned sum node weights. From the figure we can see that all of the weights are close to 0.5 which is different compared to the generating model. Here it is important to notice that the weights are related to the overall structure of the SPN which is also different between the two SPNs.

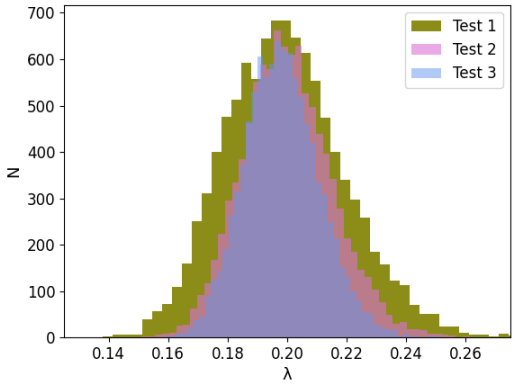
Figure 5.11 shows histograms of distribution parameters for dimensions X_0, X_1 and X_2. From the figures we can see that the parameters center around or near the correct values for $Gaussian(\mu = 19, \sigma^2 = 2)$ in dimension X_0, $Poisson(\lambda = 3)$ in dimension X_1 and $Exponential(\lambda = 1)$ in dimension X_2. For dimensions X_1 and X_2,



(a) X_3: *Gaussian* μ



(b) X_3: *Gaussian* σ^2

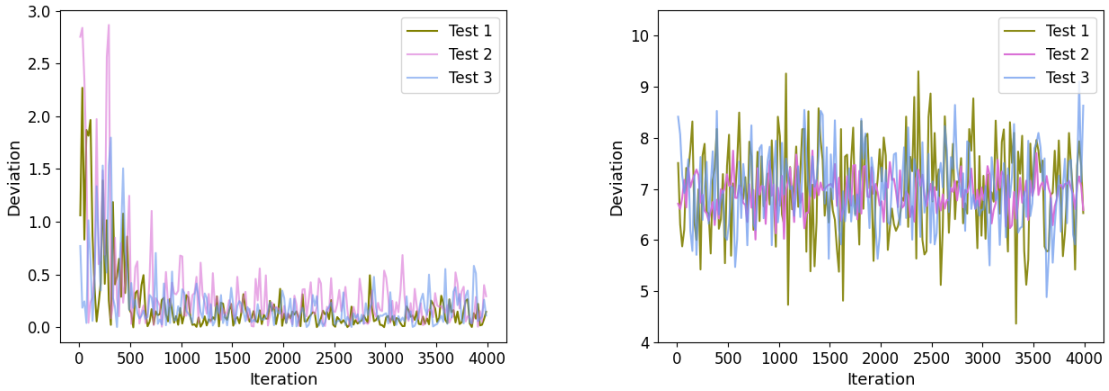


(c) X_3: *Exponential* λ

Figure 5.12: Histograms of *Gaussian* and *Exponential* parameters for dimension X_3 from three repeated runs of the experiment.

the histograms center around the correct value, whereas for dimension X_0 the situation is different. The histograms for the μ parameter are centered around the correct value but we can see one of the histograms being wider than the others. Also, the histograms for σ^2 are shifted from the correct value 2. Here, we can see that the histograms from the repeated experiments center around different values which means that the learning outcome varied between repetitions which is not desirable. Based on this, the SPN has been able to learn the patterns from data concerning these dimensions.

Figure 5.12 shows the histograms for *Gaussian* and *Poisson* parameters for dimension X_3 in leaf L_{15}. For this dimension, the generating model had $Gaussian(\mu = 12, \sigma^2 = 1)$ and $Exponential(\lambda = 2)$ distributions. In these figures, the histograms center around completely different values compared to the values of the generating model, for example, in figure 5.12a, the histogram is centered around 5 whereas the value of the parameter in the generating model is 12. Similarly, the histogram for the *Exponential* λ is centered around 0.2 whereas the parameter value of the generating model was



(a) Dimension X_0.

(b) Dimension X_3.

Figure 5.13: Deviation of *Gaussian* μ parameter samples compared to the parameter of the generating model as a function of iterations for dimension X_0 in leaf L_4 and dimension X_3 in leaf L_{10} from three repeated runs of the experiment.

2. Based on the results, the model has not been able to identify these two patterns from the data and therefore been unable to learn these parameters. Although, here we can see no variation on the value around which the histograms between repeated experiments center around.

From figure 5.13 we can see the deviation of *Gaussian* μ parameter samples compared to the parameter of the generating model. Figure 5.13a shows it for dimension X_0 in leaf L_{14} and from it we can see the deviation decreasing and then stabilizing at around 500 iterations. In the first experiment we experienced a similar effect, where at some number of iterations the deviation stabilized to values close to zero. An important aspect here is to acknowledge that there will always be some variance in the sampled parameter values, thus the deviation will not disappear completely. The behavior in this figure shows that the SPN has been able to identify patterns from the data as the deviation has efficiently decreased.

Similarly, figure 5.13b shows the deviation of the *Gaussian* μ parameter samples for dimension X_3 in leaf L_{10}. Dimension X_3 contains both *Gaussian* and *Exponential* distributions, and from the earlier histograms we were able to see that the SPN was not able to identify these distributions from the data. Instead of that, the learned patterns were from somewhere in between the parameters from these two distributions. This figure also supports the same claim. The deviation does not decrease and instead stays at a high value compared to the figure on the left. From this we can see that the SPN has not been able to reduce the deviation by learning, thus it has been unable to learn the patterns from the data.

We calculated average log-likelihood of the SPN which resulted from one of the repeated experiments with a test data set of 2000 data points. Log-likelihood measures

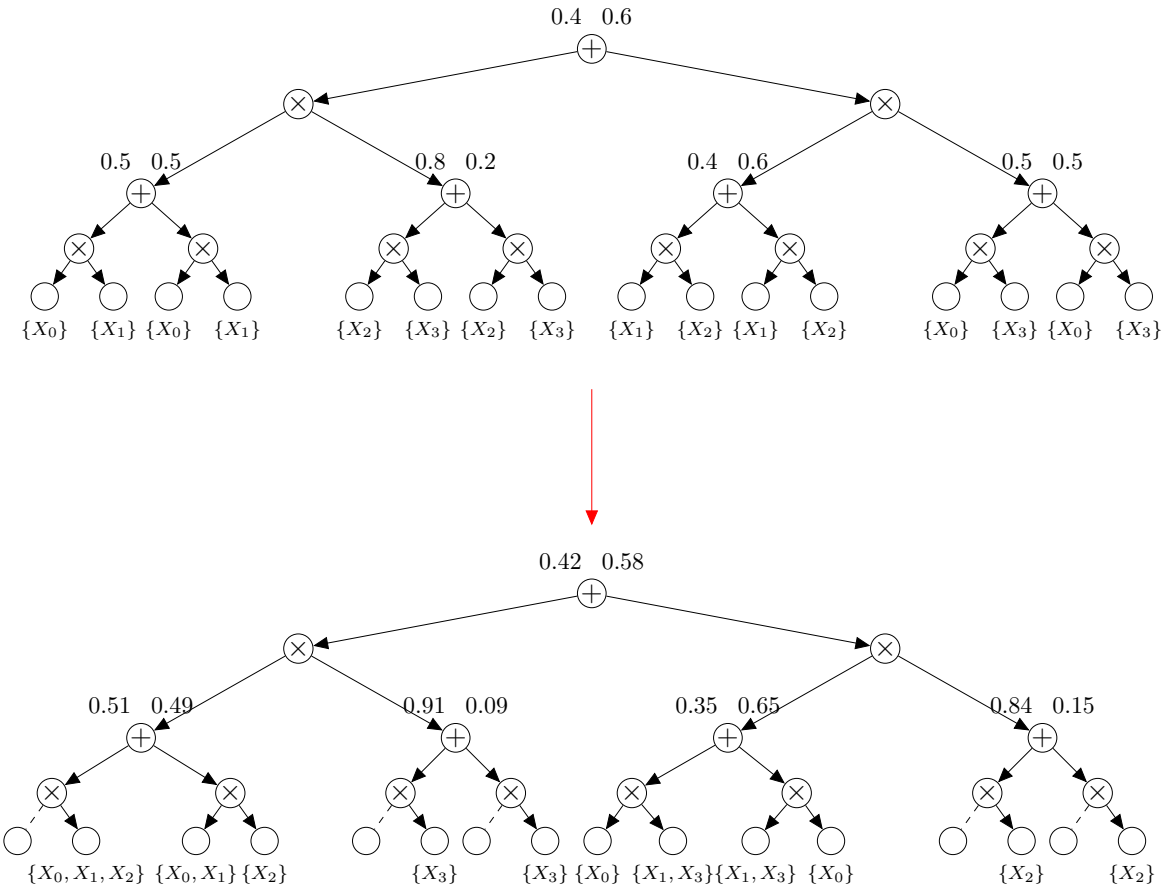


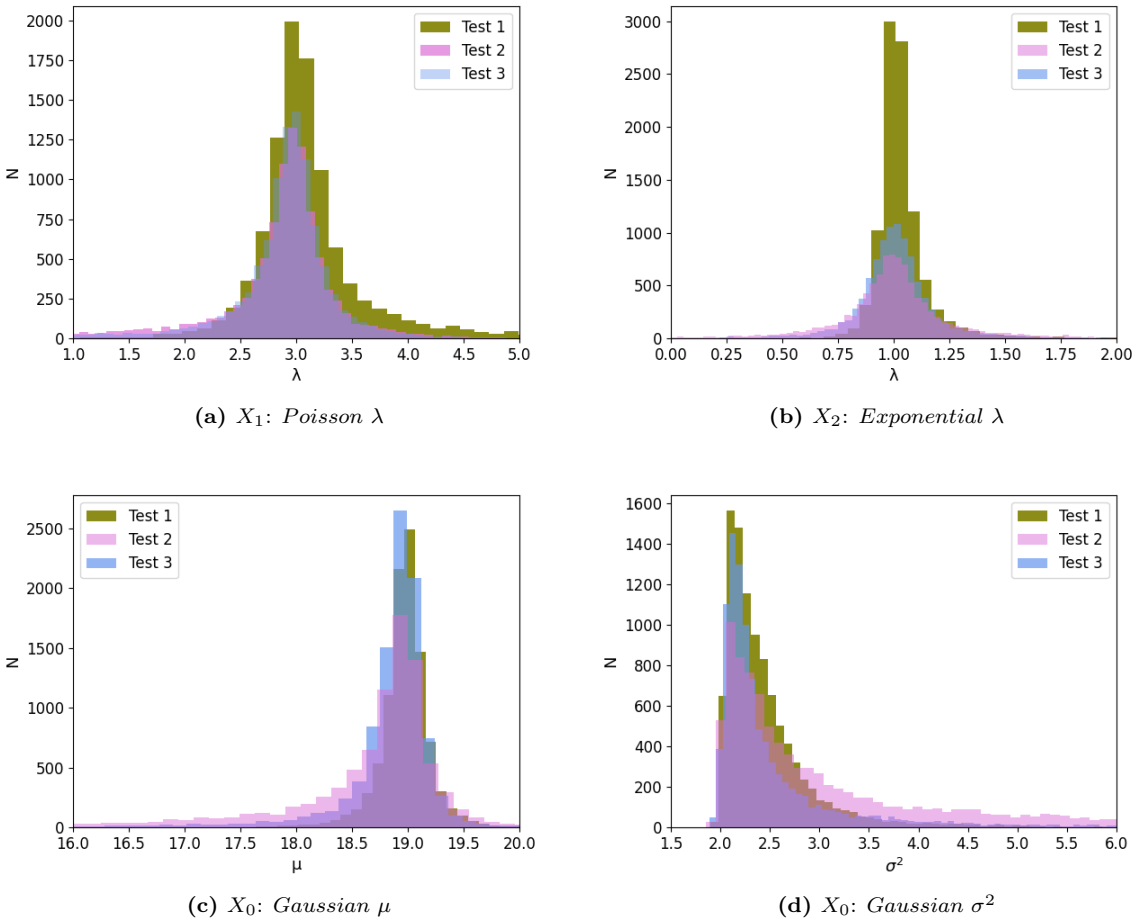
Figure 5.14: The learned effective structure at iteration 10000.

how well a model fits a data set, in other words, how well a model describes a data set. The larger value the log-likelihood has, the better the model represents the data. The result we got for the average log-likelihood for the test data set using the SPN was **-61.7** and the corresponding average log-likelihood for the generating model was **-6.5**. From these values we can see that the SPN resulted from the learning does worse in describing the data compared to the generating model. This is although something which we could assume as the model was not able to learn one of the dimensions, X_3.

Using Gibbs sampling

The algorithm was additionally implemented with Gibbs sampling, instead of using Metropolis-Hastings MCMC, to be able to compare between using these two methods. The same experiment was ran using Gibbs sampling, repeated three times using different starting values and random seed.

The learned effective structure in figure 5.14 shows the layout of the scope function and the learned sum node weights from the last iteration of the learning process. From the figure we can see the similarity to figure 5.10 from the previous experiment regarding



(a) X_1: *Poisson* λ

(b) X_2: *Exponential* λ

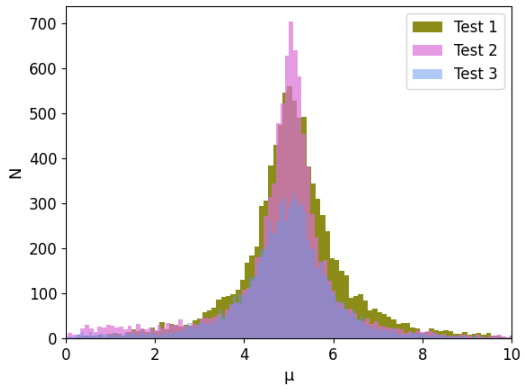
(c) X_0: *Gaussian* μ

(d) X_0: *Gaussian* σ^2

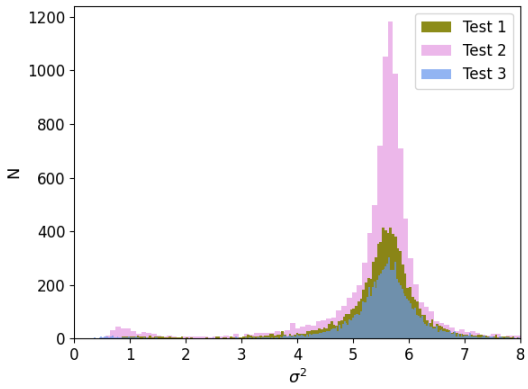
Figure 5.15: Histograms of the leaf distributions for dimensions X_0, X_1 and X_2 from three repeated runs of the experiment.

the scope function. The learned arrangement of the dimensions is different from the original arrangement, although some similarities can be found. For example, on the left side of the SPN the dimension X_2 has switched from the right sub-tree to the left one. Even though the result regarding the scope function is similar compared to the previous experiment, the situation is different with the sum node weights. The learned sum node weights in figure 5.10 were all pairs of approximately 0.5 and 0.5, whereas here the weights have different values. In many cases the learned weights are very close to the correct weight values as we can see from the figure. Although, it is important to keep in mind that the learned sum node weights are only relevant with the SPN at that moment of the last iteration and depend on the other aspects of the SPN. Similarly, here Appendix B has additional effective structures from iterations 5000, 6000, 7000, 8000 and 9000 with sum node weights attached to them to shed light on the structure during the learning process.

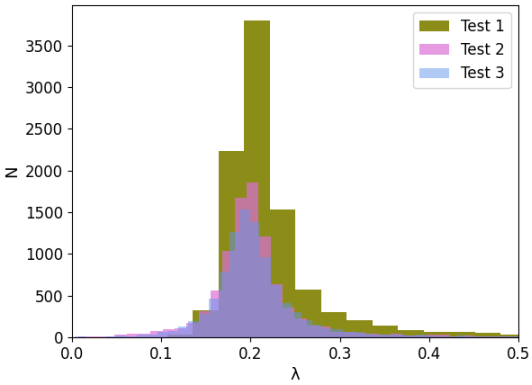
From figure 5.15 we can see the histograms centering around the correct values for



(a) X_3: *Gaussian* μ



(b) X_3: *Gaussian* σ^2



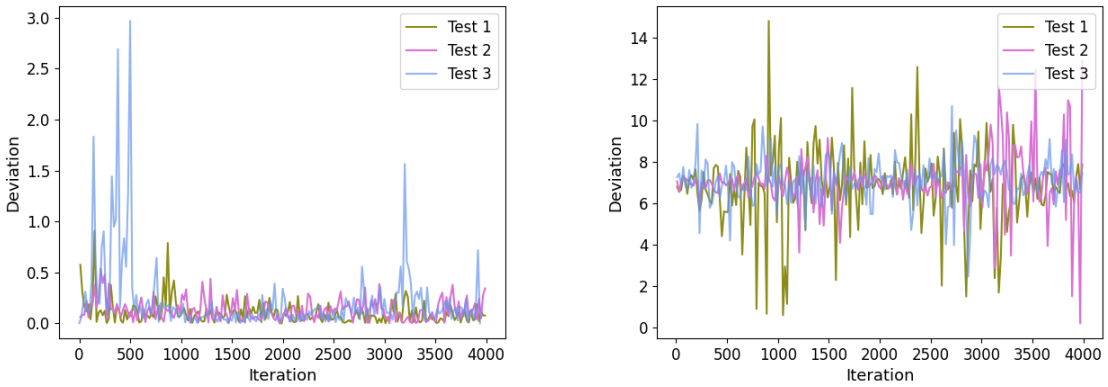
(c) X_3: *Exponential* λ

Figure 5.16: Histograms of *Gaussian* and *Exponential* parameters for dimension X_3 from three repeated runs of the experiment.

the parameters for dimensions X_0, X_1 and X_2. Only Gaussian σ^2 for dimension X_0 is shifted to the right, but it is still close to the correct parameter value, 2. These figures show that the width of these histograms is larger than the width of the corresponding histograms in the previous experiment. The histograms here have also longer tails. We can as well see that all of the histograms between repeated experiments are very similar and center around same values which was the expectation.

Figure 5.16 shows the histograms of the parameter values for dimension X_3 which contains two distributions, *Gaussian* and *Exponential*. In the previous experiment, the outcome for this dimension was not successful and we can see here the same outcome. The parameter values for both *Gaussian* and *Exponential* distributions center around wrong values. These are although the same values which the histograms centered around in the previous experiment.

The deviations of the *Gaussian* distribution parameter samples compared to the parameters of the generating model as a function of iterations for dimensions X_0 and



(a) Dimension X_0 **(b)** Dimension X_3

Figure 5.17: Deviation of *Gaussian* μ parameter samples compared to the parameters of the generating model as a function of iterations for dimension X_0 and X_3 from three repeated runs of the experiment.

X_3 in figure 5.17 are similar compared to the deviations from the previous experiment. For dimension X_0, the deviation has some higher values in the beginning of the learning process and after some iterations it settles into a relatively low value. For dimension X_3, the situation is different, the deviations stay at a high level. This could also be seen from the previous figures where the histograms centered around incorrect values.

Here as well, we calculated average log-likelihood of the SPN which resulted from one of the repeated experiments with a test data set of 2000 data points. The result we got for the average log-likelihood for this set of data using was **-65.5**. This is slightly worse than the average log-likelihood we got when using the algorithm with Metropolis-Hastings MCMC.

5.3 Conclusions

The objective of this thesis was to explore sum-product networks and their capabilities in learning from data. An SPN learning algorithm was developed using Metropolis-Hasting Markov chain Monte Carlo and Bayesian inference. The algorithm was based on the algorithm by Trapp et al. [6], the only difference in them being the usage of Metropolis-Hasting Markov chain Monte Carlo instead of Gibbs Sampling. The algorithm was examined by running two experiments with different size SPNs. A simple and small SPN was used in the first experiment, and the learning experiment was run with data sets of three different sizes to explore how the size of the data affects the result of the learning. In the experiment, we observed that having more data increases the SPN's ability for learning the underlying patterns from the data.

In the second experiment, the learning algorithm was used on a larger SPN. There

we saw that the larger the SPN is, the more difficult the learning process becomes as there are possibilities, for example, for the scope function. Also, the computation itself becomes more heavier as there are more node operations to compute. From the results we saw that the learned scope for the SPN was not the same as the original scope, and from this we can conclude that with a large SPN there can be more possibilities for the scope function. It is not guaranteed that the original scope was ideal for the data and this means there is a change for the SPN to learn a different scope which can better describe the data. In this experiment, we saw that having more than one distribution in a dimension brings challenges to the model. The model was not able to identify two different patterns within one dimension, as the learned parameters were from somewhere in the middle of the original parameters. Also, we saw that in this case the deviation of the parameter samples compared to the parameters of the generating model did not decrease and that is stayed on a rather high level compared to the other deviations explored and showed. For the dimensions containing only one distribution, the learning outcome was successful. The experiment was repeated three times which enabled us to compare the results between the repetitions. The results were similar between repeated experiments although we saw some differences in some histograms which indicated a slightly different outcome between the tests.

The algorithm was implemented additionally with Gibbs sampling and the second experiment was repeated using this algorithm. The results were similar compared to the previous experiment. Here we saw the model's ability to learn the parameters from the dimensions containing only one distribution and the inability in learning those from dimensions containing more than one distribution. This experiment was repeated three times and by comparing the results we saw the same results between repeated experiments. In the previous experiment we experienced some differences between different runs of the same test which was not optimal as the learning outcomes should be very similar. In this sense, using Gibbs sampling in the learning algorithm shows better results.

6. Discussion

This thesis gave an overview on sum-product networks and some learning algorithms used for these models. SPNs are graphical models which are able to represent complex data which can be a challenge for other models. A learning algorithm was implemented incorporating Bayesian inference and Metropolis-Hastings MCMC, and we experimented with the said algorithm to gather information on how it performs in a few different settings. The implemented algorithm is based on a learning algorithm by Trapp et al. [6] which is a Bayesian learning algorithm for SPNs which, among other methods, uses, for example, Gibbs sampling. The algorithm implemented for this thesis uses Metropolis-Hastings MCMC instead of Gibbs sampling, but from other aspects the algorithms correspond to each other. The implemented algorithm was examined by running experiments which enabled us to answer some of the research questions based on the results.

When running the experiments, we started from an experiment with a simpler setting, using a small SPN and a simple data set containing only a few dimensions. This made it easier to observe the behavior during the learning as the possibilities in the learning process are limited because of the small number of nodes in the network. In this experiment, three different size data sets were used to gather knowledge on how the data set size affects the learning outcome. The used data set sizes were 1000, 5000 and 20000. The data was generated using an SPN, after which the data sets where fed to another SPN aiming to see how well this second SPN is able to learn the parameters of the data generating SPN. From the results it could be seen that having more data makes the learning results more accurate. This was visible, for example, in the histograms of the deviations of the parameter samples compared to the parameters of the generating model, where the deviations stabilized to lower values when using larger data sets. For all of the three data set sizes, the learned parameter values were very close to the correct values, although with larger data sets the histograms representing the values were narrower and more focused on the correct values. From this experiment we could conclude that having larger data sets helps the learning as it helps the SPN to identify the patterns from the data. From this experiment we could see that the SPN model works in this setting, and is able to detect patterns from

data. In this experiment, we were able to see the effect of data set size on the learning outcomes and overall, how a small SPN performs in a simple learning task. However, this experiment was not able to give us information on how the SPN works in a more complex setting and give comparison on using Metropolis-Hastings MCMC instead of Gibbs sampling.

The second experiment was done using a larger SPN and a more complicated data set consisting of four dimensions. Out of these four, three of the dimensions contained only one distribution whereas the last dimension contained two distributions in order to raise the difficulty of the learning task. The experiment was run three times to ensure reliable results as the results from repeated runs of the same experiment should be the same. From the results we saw how the outcomes in all repeated runs of the experiment were very similar, although some small differences were spotted. The results showed that for the dimensions containing only one distribution, the SPN was able to successfully learn the model parameters. For the one dimension containing two distributions, the SPN was not able to successfully learn. For this dimension, the deviation of the sampled parameter values compared to the parameters of the generating model stayed at a high level and the outcome of the learned parameter values were incorrect. In this experiment we explored the outcome of the SPN structure learning and compared the results to the structure of the data generating model. The outcome of the structure learning was that the learned SPN structure was not corresponding to the structure of the data generating SPN. Important thing to remember is that beforehand we acknowledged that the learned structure might not be the structure of the generating model as with a larger SPN it is possible that the model learns a structure more suitable. Although, the learned structure was tested by computing average log-likelihood for a test data set and the log-likelihood computed for the SPN was worse than the corresponding value for the generating model. From this we can deduce that the structure learning in this experiment was not successful. To summarize, in this experiment, we were able compare using two different MCMC methods and examine the learning outcomes, concerning parameters and structure, when using a larger SPN.

To be able to answer to one the research questions concerning the comparison of using Metropolis-Hastings MCMC and Gibbs sampling in the learning algorithm, the algorithm used in this thesis was additionally implemented using Gibbs sampling. The same, previously described experiment was ran for this algorithm and the results were presented and compared to the previous results. From these results we saw similar outcomes compared to using Metropolis-Hastings MCMC, here the SPN model was able to learn parameters for the dimensions containing only one distribution but for the dimension containing two distributions the learning was not successful. Compared to the previous experiment, here we saw wider parameter sample histograms

with longer tails. There were no clear benefits between these two variations, using Metropolis-Hastings MCMC or Gibbs sampling. In both cases there were some unsuccessful learning outcomes with more complicated data but with simpler data both of these variations were able to identify the relevant patterns. Both of these experiments were repeated three times to ensure constant results between reruns. When using Metropolis-Hastings MCMC, we could see some differences in the learning outcomes which is not ideal. When using Gibbs sampling, the results between all the runs were similar.

One important aspect to discuss, when discussing the experiments, is the limitations. One most prominent limitation in the experiments done is the size of the used SPNs. Simple and small SPNs have limited ability in learning patterns from data as they have small number of nodes to perform the needed computations. In addition to this, small SPNs have less flexibility in the learning process, as the number of nodes affect the number of possibilities the SPN has, for example, in laying out the scope or in dividing the data across the network and nodes. Although, while having a larger SPN increases the learning capabilities of the network, it also makes the learning process more complex as there are more possibilities for the learning as there are more nodes. For a data set, there can be multiple good ways for the SPN to assemble itself, so the learning outcome can be different when comparing it, for example, to a data generating SPN. Another thing which increases as the SPN size grows, is the computing time. SPNs with more nodes have more computations to perform for each iteration of the learning process.

One limitation in the experiments is the used data. The data used in the experiment was simple, containing only a few dimensions and different distributions. The simplicity of the data affects the learning process, as SPN can learn simpler data easier. By using a more complex data set, more interesting results could have been obtained. From the results of the second experiment, we could already see that the SPN was not able to identify patterns within dimensions containing more than one distribution. Although, when using complex data, the data sets need to be large for a successful learning outcome. The effect of the data set size was shown in the first experiment. This in turn indicates that the learning process becomes more computationally demanding as larger data sets require more computation.

The experiments done covered some aspects, but there were also some aspects left for further experiments. One aspect for exploring would be using larger amount of data. Another aspect for exploring is using real-world data as all of the used data in the experiments here were synthetic. Also, a more fine-grained comparison between the algorithms using Metropolis-Hastings MCMC and Gibbs sampling could be done in order to gather more information on the difference of these methods. In addition to

this, this thesis only considered SPNs with a binary tree structure, hence one aspect to explore would be to investigate using different structures.

Acknowledgements

I would like to thank my thesis supervisor, Professor Mikko Koivisto, for providing me the topic, for guiding me in writing this thesis and the experiments done, and thoroughly supporting me through the process. I would also like to express my gratitude to my friends and my spouse for supporting me in completing this thesis.

Appendix A. Effective structures with Metropolis-Hastings MCMC

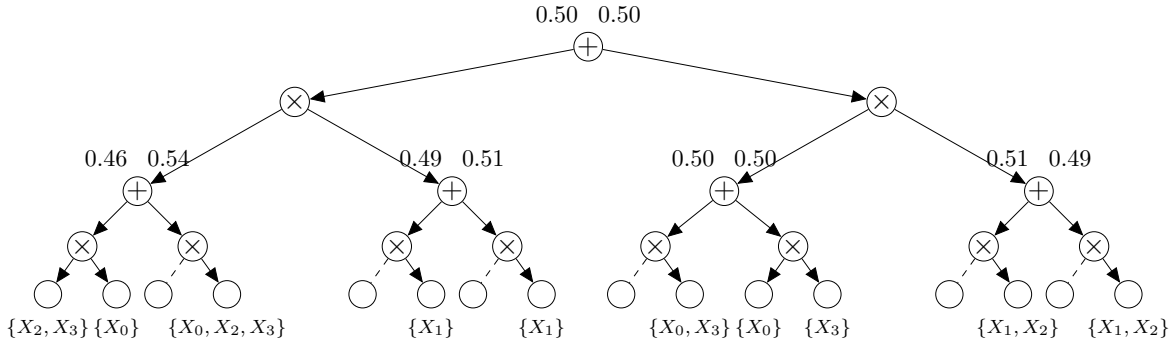


Figure A.1: The learned effective structure at iteration 9000.

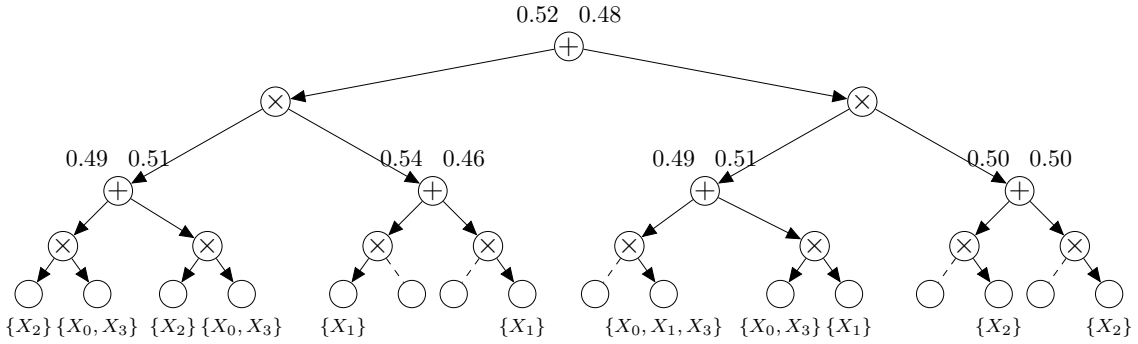


Figure A.2: The learned effective structure at iteration 8000.

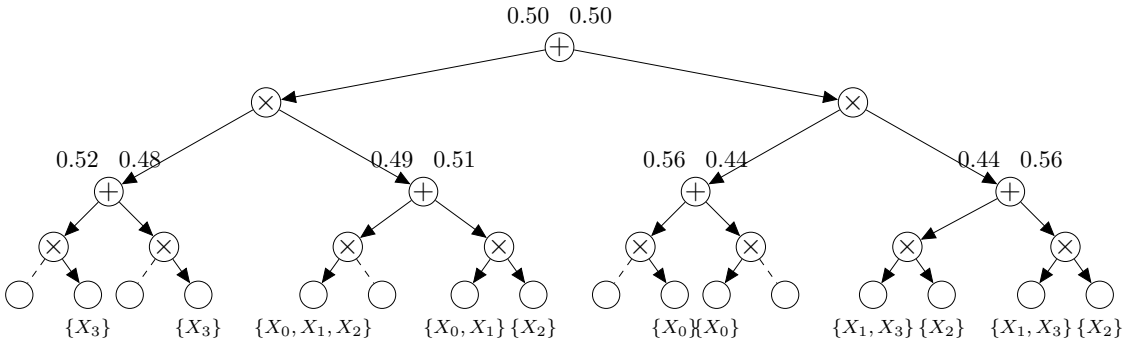


Figure A.3: The learned effective structure at iteration 7000.

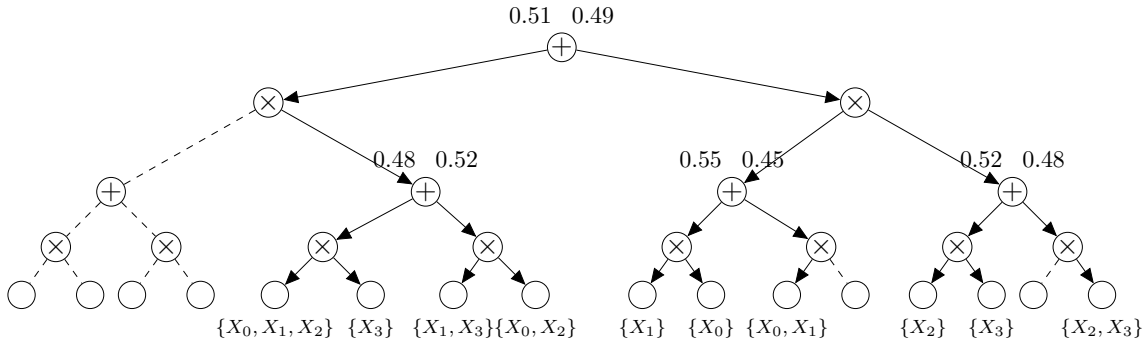


Figure A.4: The learned effective structure at iteration 6000.

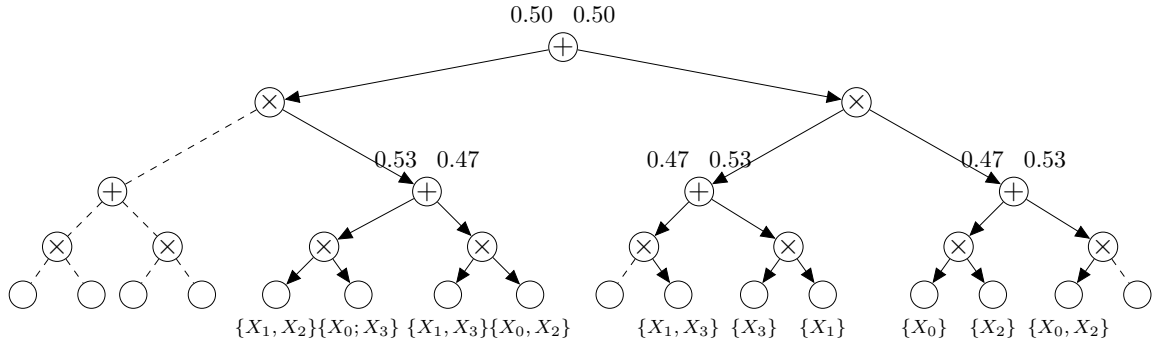


Figure A.5: The learned effective structure at iteration 5000.

Appendix B. Effective structures with Gibbs Sampling

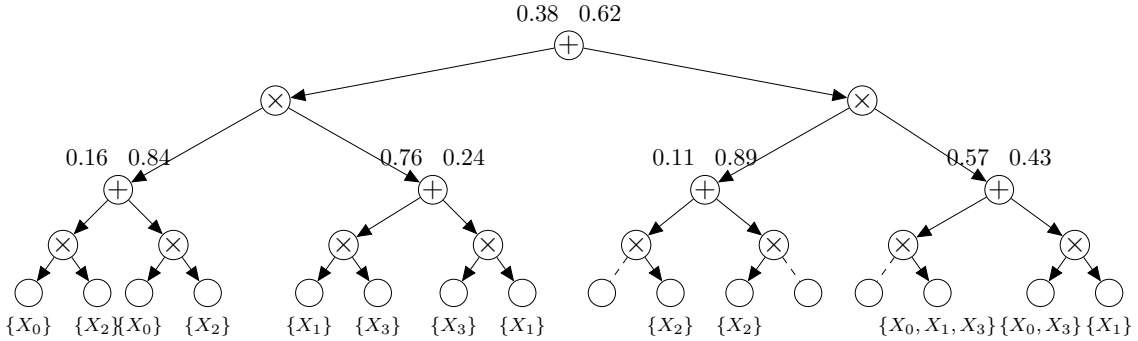


Figure B.1: The learned effective structure at iteration 9000.

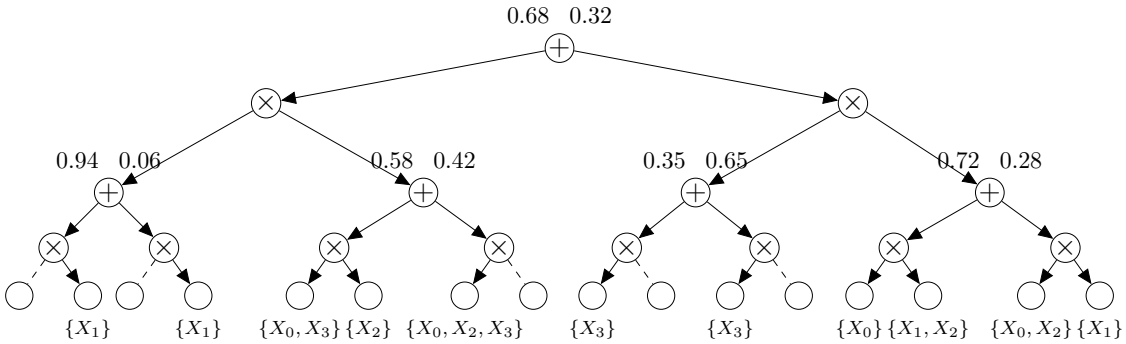


Figure B.2: The learned effective structure at iteration 8000.

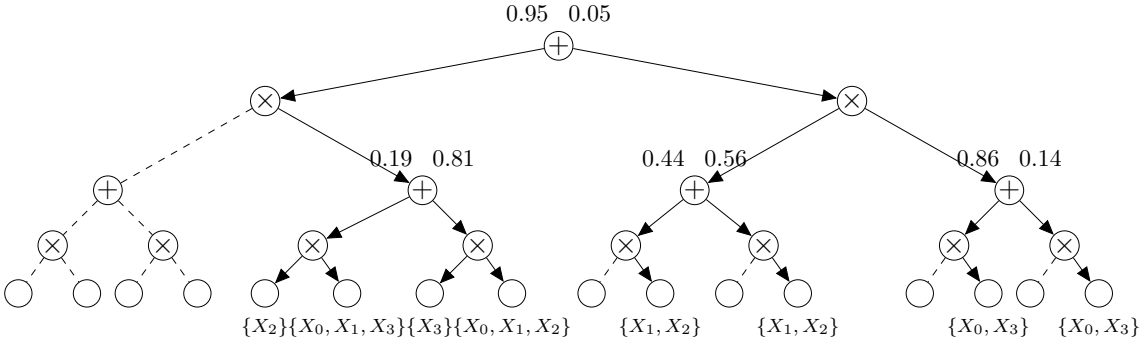


Figure B.3: The learned effective structure at iteration 7000.

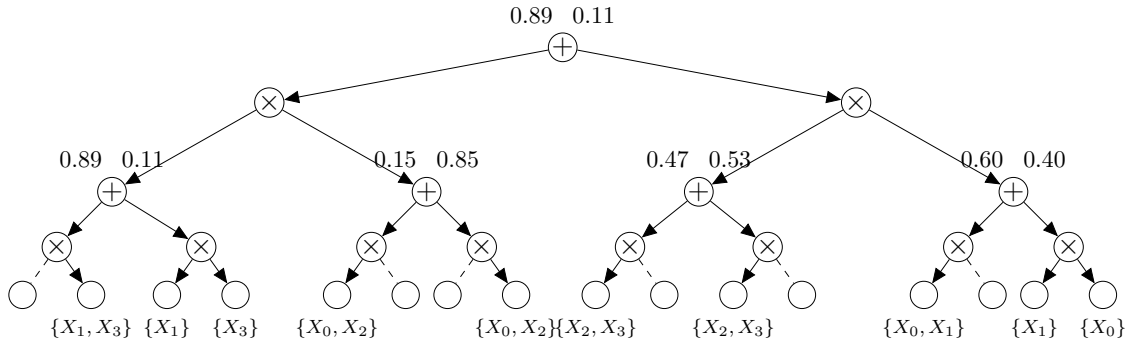


Figure B.4: The learned effective structure at iteration 6000.

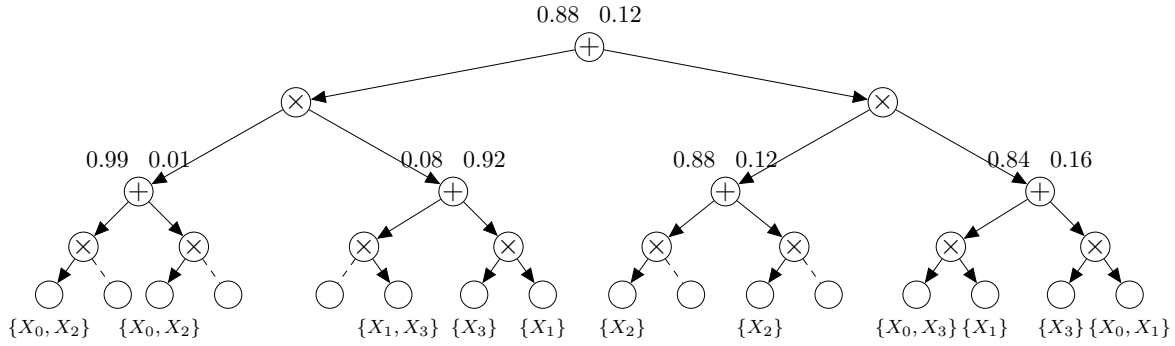


Figure B.5: The learned effective structure at iteration 5000.

Bibliography

[1] Yagang Zhang. *Machine Learning*. InTech, 2010.

[2] Herve Abdi. *Neural networks*. Sage university papers series. Quantitative applications in the social sciences ; 07-0124. SAGE, 1999.

[3] A. Dennis and D. Ventura. Learning the architecture of sum-product networks using clustering on variables. 3:2033–2041, 2012.

[4] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. *Proceedings of the IEEE International Conference on Computer Vision*, 2012.

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

[6] Martin Trapp, Robert Peharz, Hong Ge, Franz Pernkopf, and Zoubin Ghahramani. Bayesian learning of sum-product networks. *33rd Conference on Neural Information Processing Systems, NeurIPS 2019*, 2019.

[7] John F. Monahan. *Numerical methods of statistics*. Cambridge series in statistical and probabilistic mathematics ; [32]. Cambridge University Press, Cambridge ;, 2nd ed. edition, 2011.

[8] Christian Robert. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer New York, New York, NY, 1st ed. 1999. edition, 1999.

[9] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass., 2013.

[10] Iqbal H. Sarker. Machine learning: Algorithms, real-world applications and research directions. 2(3), 2021.

[11] Bing Liu. *Supervised Learning*, pages 63–132. 2011.

[12] Salim Dridi. Unsupervised learning - a systematic literature review. 2021.

[13] Leonhard Held. *Applied statistical inference : likelihood and Bayes*. Springer, Berlin, 2014.

[14] Luis Enrique Sucar. *Probabilistic Graphical Models: Principles and Applications*. Springer Publishing Company, Incorporated, 2015.

[15] Kenji Doya, Shin Ishii, Alexandre Pouget, and Rajesh Rao. *Bayesian brain : probabilistic approaches to neural coding*. Computational neuroscience. MIT Press, Cambridge, Mass, 2007.

[16] Karl-Rudolf Koch. *Introduction to Bayesian Statistics*. Springer Publishing Company, Incorporated, 2nd edition, 2007.

[17] Fahim Faizi, Pedro Buigues Jorro, George Deligiannidis, and Edina Rosta. Simulated tempering with irreversible gibbs sampling techniques. 2020.

[18] William L. (William Lee) Dunn. *Exploring Monte Carlo methods*. Elsevier, Amsterdam, 2012.

[19] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[20] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[21] Robert Nishihara, Iain Murray, and Ryan Adams. Parallel mcmc with generalized elliptical slice sampling. *Journal of Machine Learning Research*, 15, 2012.

[22] F. (Faming) Liang. *Advanced Markov chain Monte Carlo methods : learning from past samples*. Wiley Series in Computational Statistics. Wiley, Chichester, England, 2010 - 2010.

[23] Dirk P. Kroese. *Handbook of Monte Carlo methods*. Wiley series in probability and statistics ; 706. Wiley, Hoboken, N.J, 2011.

[24] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distribution, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6:721–741, 1984.

[25] Jun Liu. The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. *Journal of The American Statistical Association - J AMER STATIST ASSN*, 89:958–966, 1994.

[26] R. Gens and Pedro Domingos. Discriminative learning of sum-product networks. *Advances in Neural Information Processing Systems*, 4:3239–3247, 2012.

[27] Han Zhao, Mazen Melibari, and Pascal Poupart. On the relationship between sum-product networks and bayesian networks. 2015.

[28] Kiran R. Karkera. *Building probabilistic graphical models with Python : solve machine learning problems using probabilistic graphical models implemented in Python with real-world applications*. Community experience distilled. Packt Publishing, Birmingham, 2014.

[29] Amirmohammad Rooshenas and Daniel Lowd. Learning sum-product networks with direct and indirect variable interactions. *31st International Conference on Machine Learning, ICML 2014*, 2:1070–1080, 2014.

[30] Andrew Blake, Pushmeet Kohli, and Carsten Rother. *Markov random fields for vision and image processing*. MIT Press, Cambridge, Mass, 2011.

[31] Daniel Lowd and Pedro M. Domingos. Learning arithmetic circuits. *CoRR*, abs/1206.3271, 2012.

[32] Amirmohammad Rooshenas and Daniel Lowd. Discriminative structure learning of arithmetic circuits. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 1506–1514, Cadiz, Spain, 09–11 May 2016. PMLR.

[33] Arthur Choi and Adnan Darwiche. On relaxing determinism in arithmetic circuits. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 825–833, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[34] Daniel Lowd and Amirmohammad Rooshenas. Learning markov networks with arithmetic circuits. 2013.

[35] 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*, PP, 2016.

[36] Cory Butz, Jhonatan Oliveira, Andre Santos, and Andre Teixeira. Deep convolutional sum-product networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:3248–3255, 2019.

[37] Priyank Jaini, Amur Ghose, and Pascal Poupart. Prometheus : Directly learning acyclic directed graph structures for sum-product networks. In Vaclav Kratochvil and Milan Studeny, editors, *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, volume 72 of *Proceedings of Machine Learning Research*, pages 181–192, Prague, Czech Republic, 11–14 Sep 2018. PMLR.

[38] Olivier Delalleau and Y. Bengio. *Shallow vs. Deep Sum-Product Networks*, volume 24, pages 666–674. 2011.

[39] R. Gens and Pedro Domingos. Learning the structure of sum-product networks. *30th International Conference on Machine Learning, ICML 2013*, 28:1910–1917, 2013.

[40] Martin Ratajczak, Sebastian Tschiatschek, and Franz Pernkopf. Sum-product networks for sequence labeling. 2018.

[41] B.C.G. Peharz and Franz Pernkopf. Greedy part-wise learning of sum-product networks. *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery ECML/PKDD*, 8189:612–627, 2013.

[42] Kaiyu Zheng, Andrzej Pronobis, and Rajesh P. N. Rao. Learning graph-structured sum-product networks for probabilistic semantic maps, 2017.

[43] Andrzej Pronobis, Francesco Riccio, and Rajesh P. N. Rao. Deep Spatial Affordance Hierarchy: Spatial knowledge representation for planning in large-scale environments. In *ICAPS 2017 Workshop on Planning and Robotics*, Pittsburgh, PA, USA, June 2017.

[44] M. R. Amer and S. Todorovic. Sum product networks for activity recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(4):800–813, 2016.

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

[46] Antonio Vergari, Alejandro Molina, Robert Peharz, Zoubin Ghahramani, Kristian Kersting, and Isabel Valera. Automatic bayesian density analysis. 2018.

[47] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Floriana Esposito, and Kristian Kersting. Mixed sum-product networks: A deep architecture for hybrid domains. 2018.

[48] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019.

[49] Sergey I. Nikolenko. Synthetic data for deep learning, 2019.