Master's Thesis

Sum-Product Networks for Efficient Probabilistic Inference

Richard Prüller
Mat.Nr. 1126814

January 10, 2019

Univ. Ass. Dipl.-Ing.
Rene Repp

Ao.Univ.Prof. Dipl.-Ing. Dr.techn.
Franz Hlawatsch

Institute of Telecommunications
Technische Universität Wien

TECHNISCHE UNIVERSITÄT WIEN

institute of telecommunications

Abstract

Sum-product networks (SPNs) are a recently proposed probabilistic graphical model (PGM). Unlike many other PGMs however, many inference tasks are tractable using SPNs. SPNs are typically learned from data, which puts SPNs in the wider field of machine learning. The tractable inference tasks an SPN can perform include the evaluation of marginal and conditional distributions, the computation of the minimum mean square error (MMSE) estimate and variance, as well as conditional sampling. SPNs are interesting because once the network is learned, many different inference tasks, conditioned on different inputs, can be performed. This is in contrast to many other traditional machine learning models like e.g., deep neural networks (NNs), where a new network needs to be trained if the task changes e.g., from classification to regression. SPNs are therefore an interesting topic to study, and the growing number of publications indicates an increasing interest. In this masters thesis, it is discussed how inference with generalized SPNs works and how the structure and parameters of the network can be learned from data, in both an offline and an online manner. At the end some experimental results are presented, based on publicly available datasets and self conducted 4th generation wireless (LTE) datarate measurements.

Kurzfassung

Sum-Product Networks (SPNs) sind ein kürzlich vorgeschlagenes probabilistisches graphisches Modell (PGM). Im Gegensatz zu vielen anderen PGMs können jedoch viele Inferenzaufgaben mithilfe von SPNs effizient gelöst werden. SPNs werden in der Regel aus Daten gelernt, wodurch SPNs in den weiteren Bereich des maschinellen Lernens einbezogen werden. Zu den effizient lösbaren Inferenzaufgaben, die ein SPN durchführen kann, gehören die Evaluierung von Randverteilungen und bedingten Verteilungen, die Berechnung des *minimum mean square error* (MMSE) Schätzers und der Varianz, sowie das Ziehen von bedingten Realisierungen von der gelernten Verteilung. SPNs sind interessant, da, sobald das Netzwerk einmal gelernt ist, viele verschiedene Inferenzaufgaben beantwortet werden können, die auf verschiedenen Eingaben basieren. Dies steht im Gegensatz zu vielen anderen traditionellen maschinellen Lernmodellen, wie z.B. tiefe neuronale Netzwerke (NNs), bei denen ein neues Netzwerk trainiert werden muss, wenn sich die Aufgabe ändert, z.B. von Klassifizierung zu Regression. SPNs sind daher ein interessantes Thema, und die wachsende Anzahl von Publikationen zeigt ein zunehmendes Interesse. In dieser Masterarbeit wird präsentiert, wie Inferenz mit verallgemeinerten SPNs funktioniert und wie Struktur und Parameter des Netzwerks sowohl offline als auch online aus Trainingsdaten gelernt werden können. Am Ende werden einige experimentelle Ergebnisse präsentiert, die auf öffentlich verfügbaren Datensätzen und selbst durchgeführten LTE-Datenratenmessungen basieren.

Contents

List of Figures

List of Tables

Symbols

a, b, c	Scalars
$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$	Vectors
$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$	Matrices
$(\cdot)^\top$	Transpose
$(\cdot)^H$	Conjugate transpose
$\mathbb{A}, \mathbb{B}, \mathbb{C}$	Sets
$\mathbb{Z}$	Set of integers
$\mathbb{R}$	Set of real numbers
$\lvert \cdot \rvert$	Cardinality of a set
$\mathrm{E}\{\cdot\}$	Expectation
$\mathrm{D}_{\mathrm{KL}}(\cdot \parallel \cdot)$	Kullback-Leibler divergence
$\mathsf{x}, \mathsf{y}, \mathsf{z}$	Random variables
$f_{\mathsf{x}}(x)$	Probability density function (PDF) of x
$p_{\mathsf{x}}(x)$	Probability mass function (PMF) of x
$F_{\mathsf{x}}(x), P_{\mathsf{x}}(x)$	Cumulative distribution function (CDF) of x
μ_{x}	Mean of x
P_{x}	Second moment of x
σ_{x}^2	Variance of x
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	Generally mixed (continuous and discrete) random vectors
$p(\boldsymbol{x})$	Distribution (joint PDF and/or PMF) of $\mathbf{x}$
$\boldsymbol{\mu}_{\mathbf{x}}$	Mean of $\mathbf{x}$
$\boldsymbol{C}_{\mathbf{x}}$	Covariance of $\mathbf{x}$
$\boldsymbol{C}_{\mathbf{xy}}$	Cross-covariance of $\mathbf{x}$ and $\mathbf{y}$

Chapter 1

Introduction and Outline

This thesis discusses sum-product networks (SPNs), a recently introduced probabilistic graphical model (PGM). SPNs were first proposed in 2011 [32], and ever since, a growing number of related publications has been released each year. Like many other PGMs, SPNs are learned or trained from data, which puts them into the bigger field of machine learning. The big advantage of SPNs compared to many other PGMs is that they allow for efficient inference. This puts SPNs in an interesting position. On the one hand, we have very specialized deterministic machine learning concepts like e.g. neural networks, which are trained with a very specific task in mind, e.g. classification. Once training is done, however, they can perform very efficiently. On the other hand, we have traditional PGMs, which model the distribution of the data directly and can answer various probabilistic queries, but usually only with a high or even prohibitive computational effort. In a way, SPNs combine the best of these two worlds, and this makes them an interesting research topic.

The probably biggest advantage SPNs have compared to traditional neural networks (NNs), is that SPNs do in general not require a re-training once the task changes. A traditional NN is trained *discriminatively*, i.e., every training sample is divided into an input and a corresponding output, and we want to learn the *function* connecting input with output. After training, the NN can only predict the output for a given input as it was trained, and we can not use the NN for any other task without re-training. SPNs on the other hand are a *generative* model. This means that rather learning a function, SPNs learn the *distribution* underlying the training data. SPNs allow us to access this learned distribution very efficiently, and to answer many questions related to the distribution without the need for re-training. For example, a classical machine learning task is image classification, i.e., estimating the "content" or "label" given a set of pixel values (the image). Training an NN in this context would mean to use the pixel values as input and the label as output and learning the function connecting the two. We can use this trained

NN to classify images, but nothing else. An SPN trained on the same data set allows for much more however, e.g., we could revert the process and generate images for a given label, or we could complete an image if only a part of the image is given. Since SPNs are probabilistic in nature, we can also give an estimation of the variance of our predictions, which is another advantage over traditional machine learning methods.

SPNs are still a rather new topic and because of that the understanding of what SPNs actually are can be different from author to author, and while the general idea is always the same, some aspects can vary. For the most part however, the theoretical foundations of SPNs are established [6], [20], [29], [31] and newer publications are rather concerned with better and more efficient training algorithms [12], [13], [28]. State of the art training algorithms [13], [28] achieve performance that rivals that of NNs in many applications.

The goal of this thesis is to provide an overview of SPNs. This includes the most important theoretical properties and a derivation of commonly used inference tasks. We will also provide the necessary methods to learn SPNs based on mixed, i.e., continuous and discrete, data.

The rest of this Master's thesis is outlined as follows:

Chapter 2: Some fundamentals of graph theory, machine learning and PGMs are presented. We discuss the most important nomenclature and concepts of graph theory in the context of SPNs. We provide a formal definition of graphs and digraphs, a description of trees, which are traditionally found in computer science, and finally computational graphs, which are a practical tool to describe SPNs. A brief introduction to machine learning is given, with special attention to the terminology and common practices of the learning process itself. Finally, PGMs are linked to graph theory and machine learning.

Chapter 3: The different building blocks of SPNs and how they work together is outlined and an interpretation of the inner workings of an SPN is given. The most important inference tasks an SPN can solve exactly are formally described and demonstrated with the help of a simple toy example. At the end, some additional properties of SPNs are described.

Chapter 4: Two algorithms to learn SPNs from data are presented. The first, *LearnSPN* [9] is a very general learning algorithm to learn the structure of an SPN in an *offline* manner, i.e., all the training data is available at once. The second presented learning algorithm is used to learn the structure of an SPN in an *online* manner, i.e., only some training data samples are available at a time. Both algorithms require some additional, mostly exchangeable, internal methods that are separately described at the end of the chapter.

Chapter 5: In the final chapter, we present some experimental results. We compare the performance of the training methods presented in Chapter 4 with each other using some binary data sets commonly found in the SPN literature [9], [13], [32], [33], [37]. The MNIST data set of handwritten digits [17] is used to demonstrate the sampling and image completion capabilities of SPNs. Finally, we use 4th generation mobile datarate measurements for several prediction tasks.

Chapter 2

Some Fundamentals

This chapter reviews some established definitions and concepts with which the reader may be already familiar. The goal here is to create some common ground and introduce the notation and terminology used in later chapters.

2.1 Graph Theory

This section is an overview of some basic terminology and definitions from graph theory that are used to describe SPNs. Much more detailed descriptions of graphs and digraphs can be found in [8] and [2], respectively. The standard terminology for trees in computer science is taken from [15], and a short introduction to computational graphs can be found in [10].

2.1.1 Graphs and digraphs

An *undirected graph* or just *graph* $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ is a pair of sets $\mathbb{V}$ and $\mathbb{E}$. The set $\mathbb{V}$ is a non-empty set of *vertices* or *nodes* and the set of *edges* $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ contains those pairs of

(a) Undirected graph (b) Oriented graph

Figure 2.1: Comparison of an oriented graph and the corresponding undirected graph.

nodes that are connected in the graph. While the term "vertices" is usually used in the mathematical literature, "nodes" is much more common in computer science and in the SPN related literature, which is why it is used in this thesis. An example of a graph can be seen in fig. 2.1a. The number of edges connected to a node $\mathcal{N}$ is called the *degree* of $\mathcal{N}$. It is not uncommon that a value is associated with an edge, usually a real valued scalar called *weight*. A graph containing such weighted edges is called a *weighted graph*.

Two nodes $\mathcal{N}_0$ and $\mathcal{N}_k$ of a graph $\mathcal{G} = (\mathbb{V}, \mathbb{E})$ are said to be connected by a *path* of length k if there exists a set of distinct nodes $\mathbb{V}_\mathrm{P} = \{\mathcal{N}_0, \mathcal{N}_1, \ldots, \mathcal{N}_k\} \subseteq \mathbb{V}$ and a set of edges $\mathbb{E}_\mathrm{P} = \{(\mathcal{N}_0, \mathcal{N}_1), (\mathcal{N}_1, \mathcal{N}_2), \ldots, (\mathcal{N}_{k-1}, \mathcal{N}_k)\} \subseteq \mathbb{E}$. A *cycle* is a path extended by an edge that connects the last node of the path with the first one. A special case of a cycle is a *loop*, which is a single edge that starts and ends in the same node. Graphs containing cycles are said to *cyclic*, in contrast to *acyclic* graphs, which do not contain any cycles. If every node of a graph is connected to every other node by a path, then the graph is said to be *connected*. From here on, all graphs in this thesis are implicitly assumed to be connected.

A different kind of graph is the *directed graph* or *digraph*. A digraph $\mathcal{D} = (\mathbb{V}, \mathbb{A})$ is again a pair of sets. The set of nodes $\mathbb{V}$ is identical to that of an undirected graph, but instead of edges $\mathbb{E}$, there is a set of *arcs* or *directed edges* $\mathbb{A}$. An arc is an ordered pair $(\mathcal{N}_1, \mathcal{N}_2) \in \mathbb{V} \times \mathbb{V}$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are called *tail* and *head*, respectively. The *in-degree* (*out-degree*) of a node of a digraph is the number of arcs that have their head (tail) in that node. If a digraph is depicted like in fig. 2.1b, an arc is usually drawn as an arrow pointing from tail to head. Usually, the number of arcs that can stretch between two nodes of a digraph is limited to maximally one. If we additionally forbid loops, i.e, arcs whose head and tail coincide, then we have an *oriented graph*. The graph in fig. 2.1b is an example of an oriented graph. Concepts like edge weights, paths, cycles, and connectivity can be applied to digraphs as well; however, for paths and cycles the direction of the arcs needs to be taken into consideration. For example, the graph in fig. 2.1a is cyclic, but the digraph in fig. 2.1b is not because the directions of the arcs do not allow for cycles. An acyclic digraph is often abbreviated as DAG (directed acyclic graph). Here, we restrict DAGs to be oriented graphs, i.e., DAGs are digraphs with at most one arc between two nodes and no circles or loops.

2.1.2 Trees and rooted DAGs

A *tree* is a connected acyclic graph, which implies that exactly one path leads from one node to another. In a similar fashion, an *in-branching* (*out-branching*) *oriented tree* or *arborescence* is a connected, acyclic, oriented graph and thus a DAG by our definition,

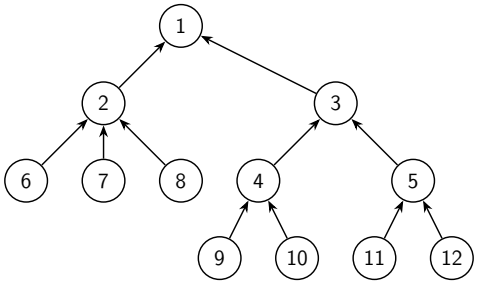


Figure 2.2: In-branching oriented tree

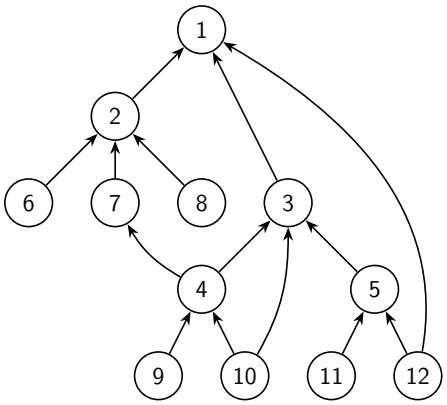


Figure 2.3: Rooted DAG

with a node $\mathcal{R}$ such that exactly one path leads to (from) $\mathcal{R}$ from (to) every other node. This unique node $\mathcal{R}$ is called the *root* of the oriented tree and has out-degree (in-degree) zero in in-branching (out-branching) trees. All nodes of a tree only connected by one edge or arc are called *leaves* of the tree. Figure 2.2 shows an example of an in-branching oriented tree, where node 1 is the root and nodes 6 to 12 are the leaves.

Trees are important in computer science, and a distinct terminology based on family trees can be found there, which is also used throughout this thesis. If two nodes $\mathcal{N}_0$ and $\mathcal{N}_1$ are connected by an edge or arc and $\mathcal{N}_0$ is the node closer to the root, then $\mathcal{N}_0$ is called the *parent* of $\mathcal{N}_1$ and likewise $\mathcal{N}_1$ is a *child* of $\mathcal{N}_0$. We denote the set of all children of a node $\mathcal{N}$ by $\mathbb{C}_\mathcal{N}$. For a node $\mathcal{C}_1 \in \mathbb{C}_\mathcal{N}$, the nodes in $\mathbb{C}_\mathcal{N} \setminus \{\mathcal{C}_1\}$ are called the *siblings* of $\mathcal{C}_1$. In fig. 2.2, node 2 would be the parent of nodes 6, 7 and 8, and nodes 7 and 8 would be the siblings of node 6. In further analogy to a family tree, we have *ancestors* and *descendants*. The ancestors of a node $\mathcal{N}$ are all nodes on the path from the root to $\mathcal{N}$, including the root but excluding the node $\mathcal{N}$ itself, while the descendants of $\mathcal{N}$ are all nodes of the tree which have $\mathcal{N}$ on their path to the root. The descendants of the root are all nodes of the tree except for the root. Again in fig. 2.2, nodes 1 and 3 are the ancestors of nodes 4 and 5 while nodes 4, 5 and 9 through 12 are the descendants of node 3. It is easy to see that a node together with its descendants forms itself a tree, a so-called *subtree* of the whole tree.

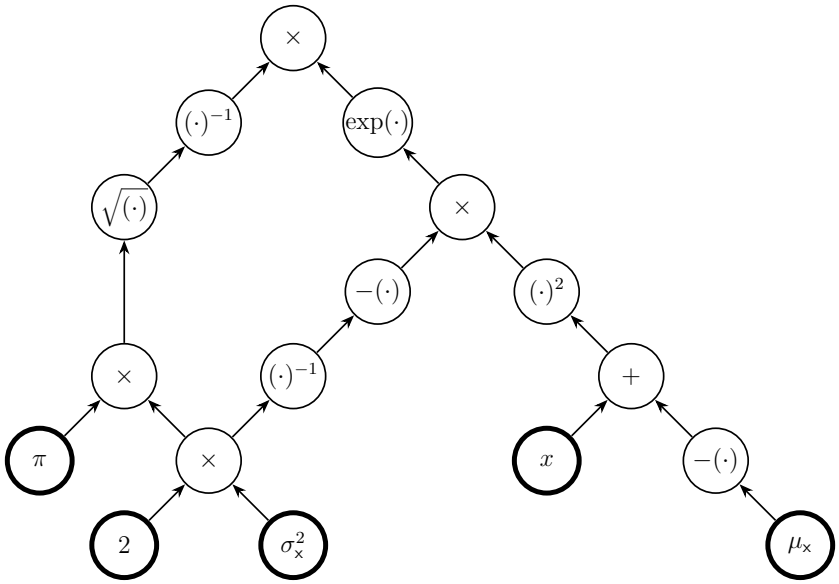


Figure 2.4: A computational graph computing the Gaussian PDF from eq. (2.1) using common math operators and functions as admissable operations.

We can conclude that parents are always ancestors and children are always descendants, and that leaves have no descendants and the root has no ancestors.

The nomenclature of oriented trees can be extended to the more general DAGs. As is shown in [2], every DAG has at least one node with in-degree zero and out-degree zero. Based on this, one of these nodes can be designated as root and the DAG becomes a *rooted DAG*, which is very similar to a tree. In contrast to an oriented tree, the nodes of a rooted DAG can have multiple parents, and thus multiple paths can lead from the root to some other node. The set of parents of a node $\mathcal{N}$ of a rooted DAG is denoted by $\mathbb{P}_{\mathcal{N}}$. Similar to trees, the ancestors of $\mathcal{N}$ are the nodes on the, possibly multiple, paths from $\mathcal{N}$ to the root. The root is an ancestor of every other node in the graph, except of the root node itself. The siblings of a node of a rooted DAG are the union of all the siblings of the multiple parents of the node. An example of a rooted DAG is depicted in fig. 2.3; this is just the tree from fig. 2.2 with a few arcs added. In this figure, for example, the parents of node 10 are nodes 3 and 4, its siblings are nodes 4, 5 and 9, and its ancestors are nodes 1, 2, 3, 4 and 7.

2.1.3 Computational graphs

Computational graphs are not a topic of graph theory per se, but are usually found in the machine learning literature like e.g. [10]. A computational graph represents a computation by an oriented graph. While cycles are permitted in general, we here restrict ourselves to rooted DAGs, because many explanations become much simpler and only computational

graphs corresponding to rooted DAGs are relevant for SPNs. Leaves of a computational graph are the *inputs*, while every other node is one of possibly many admissable *operations*. The results of the children of a node are the operands of the operation represented by the node, and every node forwards its result to its parents. The result of the root is the overall result of the computation, i.e., the output for a given set of inputs. Computational graphs are usually evaluated *bottom-up*, i.e., from the leaves to the root, and every node can be computed as soon as the outputs of its children were computed. Figure 2.4 shows a computational graph that represents the computation of the univariate Gaussian PDF, i.e.,

$$\mathcal{N}\left(x; \mu_{\mathrm{x}}, \sigma_{\mathrm{x}}^2\right) = \frac{1}{\sqrt{2\pi\sigma_{\mathrm{x}}^2}} \exp\left(-\frac{x-\mu_{\mathrm{x}}}{2\sigma_{\mathrm{x}}^2}\right), \tag{2.1}$$

where the inputs (leaves) are highlighted with a thick border.

2.2 Machine Learning

Since SPNs are trained or learned from data, many concepts known from *machine learning* can be applied here as well. This section is intended as a very superficial overview of the most basic principles of machine learning. A much more in-depth discussion of the concepts presented here can be found in [10].

The idea of machine learning, or training programs with collected data, can be traced back to the early days of computers, and by the end of the 1950s, the term machine learning began to be used [35]. A period of renewed interest began in 2006 when some important breakthroughs allowing the efficient training of artificial neural networks (or briefly *neural networks*, NNs) were achieved. NNs are usually represented by a *multi-rooted* computational graph and the nodes are called neurons, in analogy to our understanding of how a brain works, a NN is structured in layers. In a traditional *feed-forward NN*, the outputs from one layer can only be used as inputs for the next layer. NNs typically have one or more dedicated input layers, i.e., layers only containing leaves, and a dedicated output layer containing only the roots. One general result is that *deep* NNs with many layers have better performance than *shallow* models with only very few layers but an identical number of neurons. However, deeper NNs are much more difficult to train. The past decade of machine learning research was dominated by NNs and in particular by training ever deeper models (*deep learning*) and the use of NNs for different applications.

According to [23], machine learning is related to the capability of a program to learn from *experience*, which it does by improving some *performance measure* with respect to a class of *tasks*. The experience generally comes from three possible sources, which in turn characterize how a program learns. One possible source is a labeled data set,

i.e., samples with a known ground truth. Learning techniques utilizing such a data set are called *supervised*; examples include NNs and *support vector machines*. On the other hand, *unsupervised* learning is based on unlabeled samples where a ground truth is not available or too expensive to obtain. *Principal component analysis* as well as clustering algorithms like *k-means* count among the unsupervised learning techniques. The third training paradigm is commonly known as *reinforcement learning*; here a program learns by interacting with its "environment" and from the feedback of its actions.

Some typical tasks of machine learning are classification, regression, i.e., predicting a numerical value, and anomaly detection. The performance measure is usually closely connected to the task. For example, if the task is classification, then a simple ad-hoc performance measure would be the fraction of correctly classified samples from a given set, which is sometimes called *classification accuracy*.

In most cases in machine learning, a *model* is trained, and this model has a number of *parameters* that are subject to optimization during training. Typically, training a model means finding those model parameters that optimize the performance measure. The training procedure itself usually has some parameters as well, called *hyperparameters*, which are in general not optimized along with the model parameters. This means that hyperparameters need to be optimized using another procedure on top of the regular training. Simple yet popular examples of such a procedure are a *grid search* and a *random search*.

It is common to split the available data into two separate data sets. The largest of the two data sets should be the *training set*, which is used as input for the training procedure to find good model parameters. The second data set is the *test set*, a set of data samples not used during training, which is used to measure the performance of the trained model. For hyperparameter training an additional set, the *validation set*, is split from the training set. The validation set is used to measure the performance between the different hyperparameter configurations, and to enable more sophisticated hyperparameter learning procedures, e.g., gradient descent based algorithms. It is important that all three data sets have the same distribution as the real data.

A learned model is said to *generalize well* if it has good performance on previously unseen data. This can be quantified by computing a performance measure for the training set and for the test set, and a subsequent comparison of the two. Generalization is very important in machine learning, and is an aspect distinguishing machine learning from traditional optimization, where the training and test set coincide. Going back to our classification example, let us assume we train a model to achieve the best possible classification accuracy on the training set, i.e., we optimize it with regard to the performance measure. Let us assume that we optimized our model with regard to the training set or

validation set if a hyperparameter search was used. In the context of optimization we are done and we do not care how the trained model performs on unseen data, because unseen data does not exist in the framework of optimization. For machine learning however, a good training set performance is not necessarily required, as long as a good test set performance is achieved.

Generalization is related to the *capacity* of a model, which is roughly speaking the amount of information that can be stored in the model. If the capacity is too low, then the model is *underfitting* as it will never be able to fully store the dependencies within the data and no amount of training can fix this, which also means that the training set performance stays low. On the other hand, if capacity is too high, then the model is prone to *overfitting*, i.e., it stores too much information about the training data, which leads to bad generalization. The usual way around this problem is to take a model with high capacity and then use some form of *regularization*, i.e., some measure aiming to improve the test set performance, possibly at the cost of the training set performance. Most regularization techniques involve some hyperparameters, but those are usually easier to fine-tune than optimizing model capacity directly.

2.3 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) have been a topic of research since the late 1970s, beginning with Markov random fields [14] and Bayesian networks [27]. Based on their underlying graphs, PGMs may be *undirected* like Markov random fields, or *directed* like Bayesian networks. These models aim to efficiently describe the distribution of random variables by modeling their dependencies, and they allow for inference directly based on the model. Typically, inference in these scenarios means drawing samples or computing an expectation from the distribution while conditioning on some *evidence*, i.e., known values of some of the random variables. While the structure of Markov random fields and Bayesian networks is rather simple, inference is usually not and becomes computationally prohibitive for larger and more complex distributions. A lot of research has gone into this still active field, with the goal of improving the quality and speed of inference. A whole range of other PGMs has been developed to tackle different tasks like temporal prediction and decision making, accompanied by many approximate inference techniques to make the computation possible [16].

What all PGMs have in common is that they store the information about the modeled distribution in a combination of their structure and their internal parameters. This information is usually learned from observed samples, which means that many principles from machine learning can be applied in the context of PGMs as well.

Chapter 3

Sum-Product Networks

SPNs are directed PGMs based on rooted DAGs, and are usually represented with *sum nodes* and *product nodes*. An SPN models a distribution $p(\boldsymbol{x})$ of a generally mixed (discrete and continuous) random vector $\mathbf{x}$. Some restrictions on the structure need to be imposed, in order to guarantee that the SPN is *valid*, i.e., that the represented distribution is always non-negative and integrates or sums to one. Once trained, an SPN can be used for different inference tasks, such as computing the value of the distribution, the mean, the variance, etc. Another strength of SPNs is that all these inference-related queries can be answered for marginal distributions and conditional distributions involving the random variables the SPN was trained on. For example, if an SPN was trained with samples from $p(a, b)$, it can be used to estimate e.g. $p(a)$, $p(a|b)$, $\mathrm{E}\{\mathsf{b}|a\}$, etc.

3.1 Mixture Models and Independence Assumptions

Before formalizing SPNs in the next section, we dedicate this section to an introduction of the mixture models and independence assumptions SPNs are based upon.

A *mixture model* is a structural assumption about a distribution $p(\boldsymbol{x})$ of a random vector $\mathbf{x}$ of generally mixed (discrete and continuous) random variables. If $p(\boldsymbol{x})$ is a mixture, we can express it as

$$p(\boldsymbol{x}) = \sum_{m=1}^{M} w_m p_m(\boldsymbol{x}), \tag{3.1}$$

where the w_m are the mixture weights and the $p_m(\boldsymbol{x})$ are the mixture components, which are themselves distributions. The weights w_m must be non-negative and sum to one, i.e., $w_m \geq 0$ and $\sum_{m=1}^{M} w_m = 1$. Sometimes it is useful to introduce a discrete *latent random variable* $\mathsf{m} \in \{1, 2, \ldots, M\}$ to express the mixture components $p_m(\boldsymbol{x})$ as conditional

distributions, i.e.,

$$p(\boldsymbol{x}) = \sum_{m=1}^{M} p(\boldsymbol{x}|\mathsf{m} = m)p_{\mathsf{m}}(m). \tag{3.2}$$

By comparing eq. (3.1) with eq. (3.2), we can see that $w_m = p_{\mathsf{m}}(m)$ and $p_m(\boldsymbol{x}) = p(\boldsymbol{x}|\mathsf{m} = m)$.

Another kind of assumption one can make about a distribution is an *independence assumption*. If we partition the random vector $\mathbf{x}$ into N random sub-vectors $\mathbf{x}_n$ and we assume that these random sub-vectors are all statistically independent from one another, we obtain

$$p(\boldsymbol{x}) = \prod_{n=1}^{N} p(\boldsymbol{x}_n). \tag{3.3}$$

3.1.1 Illustrative example

We now present an example of a distribution and how to express it in terms of mixture models and independence assumptions, and we will make the link to SPNs. Individually, mixture models and independence assumptions have limited use when modelling more general distributions. We will however see that a combination of both can be quite expressive. To illustrate this we will take a look at a simple example. We assume a distribution $p(a, b, c)$, where a and b are Gaussian mixtures, i.e., mixture models with Gaussian components, and c is an arbitrary discrete random variable. Furthermore, we assume c to be statistically independent from a and b, and according to eq. (3.3) this implies that

$$p(a, b, c) = p(a, b)p_{\mathsf{c}}(c). \tag{3.4}$$

Let the joint distribution $p(a, b)$ be a 2-D mixture model with three components M_1, M_2 and M_3, and let the three mixture components be individually uncorrelated (and thus statistically independent) Gaussians, i.e.,

$$p(a, b|\mathsf{m} = 1) = f_{\mathsf{a}|\mathsf{m}}(a|1)f_{\mathsf{b}|\mathsf{m}}(b|1) = \mathcal{N}\left(a; \mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},1}, \sigma^2_{\mathsf{b},1}\right) \tag{3.5}$$

$$p(a, b|\mathsf{m} = 2) = f_{\mathsf{a}|\mathsf{m}}(a|2)f_{\mathsf{b}|\mathsf{m}}(b|2) = \mathcal{N}\left(a; \mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},2}, \sigma^2_{\mathsf{b},2}\right) \tag{3.6}$$

$$p(a, b|\mathsf{m} = 3) = f_{\mathsf{a}|\mathsf{m}}(a|3)f_{\mathsf{b}|\mathsf{m}}(b|3) = \mathcal{N}\left(a; \mu_{\mathsf{a},2}, \sigma^2_{\mathsf{a},2}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},2}, \sigma^2_{\mathsf{b},2}\right). \tag{3.7}$$

The joint distribution $p(a, b)$ is illustrated in fig. 3.1. If we now use eq. (3.2) and insert eqs. (3.5) to (3.7) into eq. (3.4), we obtain

$$\begin{aligned} p(a, b, c) = p_{\mathsf{c}}(c)\Big[&w_1\mathcal{N}\left(a; \mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},1}, \sigma^2_{\mathsf{b},1}\right) + \\ &+ w_2\mathcal{N}\left(a; \mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},1}, \sigma^2_{\mathsf{b},1}\right) + w_3\mathcal{N}\left(a; \mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)\mathcal{N}\left(b; \mu_{\mathsf{b},1}, \sigma^2_{\mathsf{b},1}\right)\Big]. \end{aligned} \tag{3.8}$$

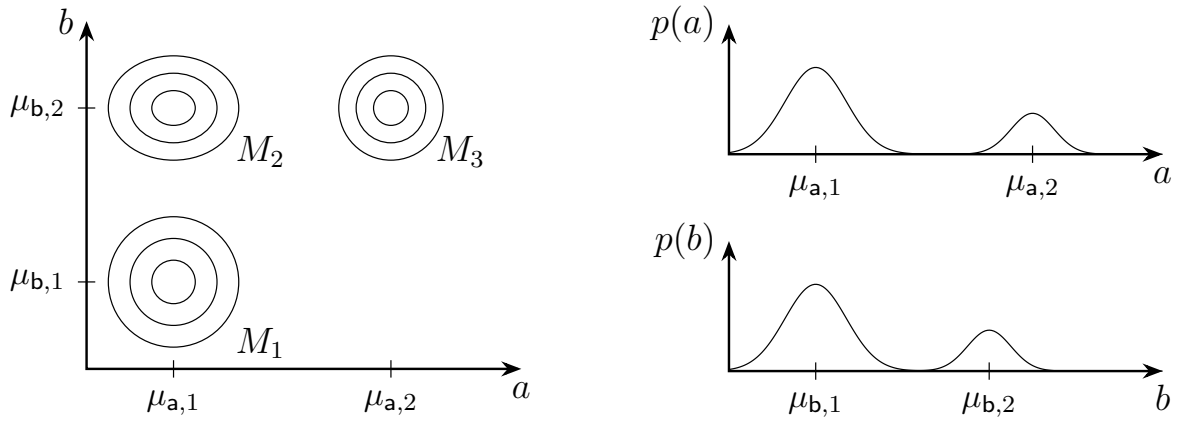


Figure 3.1: 2-D Gaussian mixture with three components M_1, M_2 and M_3.

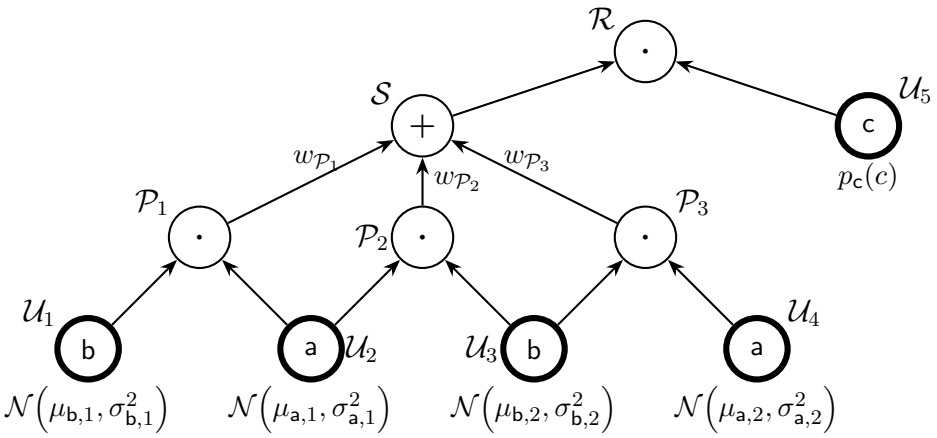


Figure 3.2: SPN representing $p(a, b, c)$, where a and b are the Gaussian mixtures corresponding to fig. 3.1 and c is a discrete random variable.

Based on eq. (3.8), we can make a number of interesting observations. For example, we can obtain a computational graph of eq. (3.8) with *univariate distributions* as inputs, and with *products* and *weighted sums* as admissable operations. The resulting graph is depicted in fig. 3.2, where $w_{\mathcal{P}_m} = w_m$. This computational graph is an SPN, or put differently, SPNs can be interpreted as the computational graphs of equations similar to eq. (3.8). We can further see that the sum $\mathcal{S}$ in the SPN corresponds to a mixture and that the products $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_4$ correspond to independence assumptions. The structure of the SPN is tied to the assumptions we made about the distribution, i.e., when we assumed a mixture and when we assumed independence. Based on the latent variable interpretation of mixture models, every descendant of $\mathcal{S}$ is conditioned on a value of the latent variable m. The latent variable m does not show up at the root $\mathcal{R}$, because the sum node sums it out, as we can see in eq. (3.2). Finally, we can observe that every node in the graph

represents a distribution, and that the root represents the overall distribution $p(a, b, c)$.

3.2 Generalized SPNs

We will now formalize the observations made at the end of the previous section. Originally, SPNs were defined not with univariate leaves, but with *indicator leaves* [32]. This comes from the fact that SPNs originated from *arithmetic circuits* [5], a related PGM. Soon however [9] these indicator leaves were generalized to univariate leaves and the resulting SPNs are called *generalized SPNs*, however, the "generalized" is usually omitted. In this thesis we will focus only on generalized SPNs and we will refer to them as just SPNs from now on.

An SPN is a representation of a distribution $p(\boldsymbol{x})$ and it models this distribution using mixtures and independence assumptions. In general, every distribution $p(\boldsymbol{x})$ of a random vector $\mathbf{x} = (\mathsf{x}_1, \ldots, \mathsf{x}_N)^\top$ can be approximated based on eqs. (3.2) and (3.3) using M mixture components, i.e.,

$$p(\boldsymbol{x}) \approx \sum_{m=1}^{M} p_{\mathsf{m}}(m) \prod_{n=1}^{N} p(\mathsf{x}_n | \mathsf{m} = m). \tag{3.9}$$

The graph of an SPN equals the computational graph of eq. (3.9) with univariate distributions as inputs and products and weighted sums as admissable operations. However, the problem of eq. (3.9) is that it can require a very large M for a good approximation. SPNs allow for an elegant reduction of the actually required number of parameters, i.e., sum weights, by introducing structure to the graph. We have seen this effect already. If we compare fig. 3.2 with eq. (3.9), we see that $p_{\mathsf{c}}(c)$ was effectively pulled out of the sum, thus decreasing the complexity of the sum node and the product nodes below. Generally, it can be proven [6] that a shallow SPN might require up to exponentially more nodes than a *deeper* SPN to represent the same distribution.

SPNs can do more than just compute the value of $p(\boldsymbol{x})$ for a given realization $\boldsymbol{x}$, and while the graph stays the same, the exact operations depend on the inference task. It is thus reasonable to think of an SPN as an *ensemble* of computational graphs, where every computational graph has sum, product, and univariate nodes with varying operations. Rather to view the "sums" and "products" in an SPN as the actual arithmetic operations, it is better to see them as mixture models and independence assumptions, respectively.

Every node $\mathcal{N}$ in an SPN represents a distribution of a random vector $\mathbf{x}$ that is specific to the node, and we will denote this distribution with $p_{\mathcal{N}}(\boldsymbol{x})$. With a more complicated structure of an SPN comes a more complicated equivalent to eq. (3.9), i.e., sums and products might alternate and get nested. Every sum brings with it its own

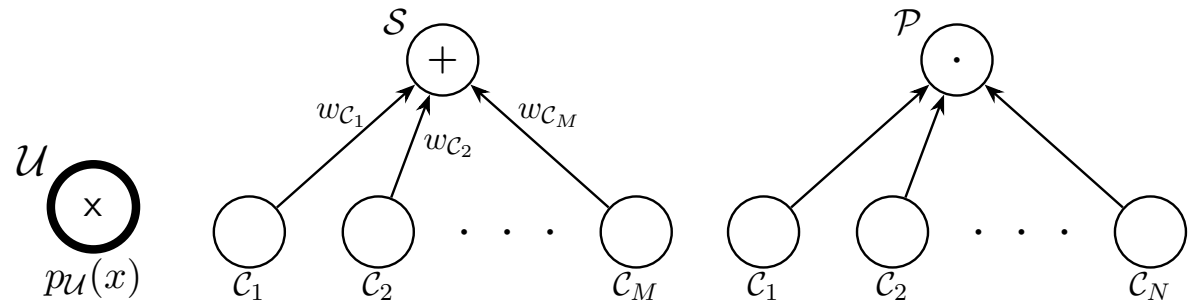
latent variable and thus nodes further away from the root are usually conditioned on many latent variables. As these latent variables are rather a theoretical tool, and practically not relevant in most cases, we will omit them for notational simplicity. The subscript $\mathcal{N}$ in $p_{\mathcal{N}}(\boldsymbol{x})$ serves as a reminder that in general, due to the latent variables, every node represents a conditional distribution with different conditional constraints. The *scope* $\mathbb{S}_{\mathcal{N}}$ of a node $\mathcal{N}$ that represents $p_{\mathcal{N}}(\boldsymbol{x})$ is the set of all random variables in $\mathbf{x}$. For example, if we consider the SPN in fig. 3.2, the scope of the root is $\mathbb{S}_{\mathcal{R}} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, while the scope of the sum is $\mathbb{S}_{\mathcal{S}} = \{\mathsf{a}, \mathsf{b}\}$, and the scope of $\mathcal{U}_1$ is $\mathbb{S}_{\mathcal{U}_1} = \{\mathsf{b}\}$.

An SPN is called *valid* if it represents a valid distribution, i.e., the distribution is non-negative and integrates/sums to one. Considering that the weights $w_m = p_{\mathsf{m}}(m)$ in eq. (3.9) are non-negative and sum to one, and assuming that the $p(\mathsf{x}_n|\mathsf{m} = m)$ are valid themselves, it is clear that the distribution $p(\boldsymbol{x})$ must be valid as well. However, when training an SPN, we usually have a different situation. We have an SPN, with weights and conditional univariate leaves that were trained with data, and we would like to make sure that this SPN is valid. We can guarantee the validity of an SPN by imposing certain restrictions on its individual nodes. The idea is that every node itself represents a valid distribution. If this holds, then the root node must also represent a valid distribution and so the SPN must be valid too. The assumption that every node in an SPN is valid is obviously stronger than the assumption that just the root is valid. However, in general it would require an exhaustive search to prove that an SPN is valid if we allow for invalid nodes within the network. In the following, we will have a look at the different node types and the conditions for their validity.

3.2.1 Univariate nodes

A univariate node $\mathcal{U}$ is a leaf of a generalized SPN. It represents a (conditional) distribution of a random variable x, and thus $\mathbb{S}_{\mathcal{U}} = \{\mathsf{x}\}$. An exemplary univariate node is depicted in fig. 3.3a. We always write $p_{\mathcal{U}}(x)$ for the distribution the node represents, regardless of whether x is discrete, continuous or mixed. A univariate node is valid if $p_{\mathcal{U}}(x)$ is valid.

The distribution $p_{\mathcal{U}}(x)$ itself can be modeled in different ways. For example, in fig. 3.2 the distribution of $\mathcal{U}_1$, $p_{\mathcal{U}}(x)$ is modeled as a Gaussian, i.e., $p_{\mathcal{U}_1}(b) = \mathcal{N}\left(b; \mu_{\mathsf{b},1}, \sigma^2_{\mathsf{b},1}\right)$. The distribution parameters $\mu_{\mathsf{b},1}$ and $\sigma^2_{\mathsf{b},1}$ would usually be estimated during training. Some possible options for univariate distribution estimators are discussed in Section 4.3.



(a) (b) (c)

Figure 3.3: The three types of nodes in an SPN: (a) Univariate node, (b) Sum node, (c) Product node.

3.2.2 Sum nodes

Based on eq. (3.1), a sum node $\mathcal{S}$ represents a *mixture* distribution

$$p_{\mathcal{S}}(\boldsymbol{x}) = \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{x}), \tag{3.10}$$

where $\mathbb{C}_{\mathcal{S}}$ is the set of the children $\mathcal{C}$, and the $w_{\mathcal{C}}$ are the mixture weights. A sum node is shown in eq. (3.10). If $\mathbf{x} = (\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_N)^\top$, then $\mathbb{S}_{\mathcal{S}} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_N\}$. If we assume that the children $\mathcal{C}$ of the sum node are valid, i.e., $p_{\mathcal{C}}(\boldsymbol{x})$ is valid $\forall \mathcal{C} \in \mathbb{C}_{\mathcal{S}}$, then a sum node is valid if two requirements are fulfilled. The first requirement is that the weights $w_{\mathcal{C}}$ are non-negative and sum to one. These are typical restrictions for mixtures and it is easily verified that violations will lead to an invalid distribution. Secondly, it is required that the scope of the children $\mathcal{C}$ is equal to the scope of the sum node $\mathcal{S}$, i.e., $\mathbb{S}_{\mathcal{C}} = \mathbb{S}_{\mathcal{S}}\ \forall \mathcal{C} \in \mathbb{C}_{\mathcal{S}}$. This latter requirement is called *completeness* [32], and sum nodes fulfilling it are called *complete*. For a complete sum node eq. (3.10) directly corresponds to eq. (3.1). If every sum node in an SPN is complete, then the SPN is called complete as well.

However, completeness is not a strictly necessary requirement for validity, which we will demonstrate by means of a simple counterexample. Consider an incomplete sum node $\mathcal{S}$ with distribution $p_{\mathcal{S}}(a, b) = w_{\mathcal{A}} p_{\mathcal{A}}(a) + w_{\mathcal{B}} p_{\mathcal{B}}(b)$, where a, b are continuous and $\mathsf{a} \in \mathbb{A}$, $\mathsf{b} \in \mathbb{B}$. Then,

$$\begin{aligned}
\int_{a\in\mathbb{A}}\int_{b\in\mathbb{B}} p_{\mathcal{S}}(a,b)\,\mathrm{d}a\,\mathrm{d}b &= \int_{a\in\mathbb{A}}\int_{b\in\mathbb{B}} \Big(w_{\mathcal{A}} p_{\mathcal{A}}(a) + w_{\mathcal{B}} p_{\mathcal{B}}(b)\Big)\,\mathrm{d}a\,\mathrm{d}b \\
&= w_{\mathcal{A}} \int_{a\in\mathbb{A}}\int_{b\in\mathbb{B}} p_{\mathcal{A}}(a)\,\mathrm{d}a\,\mathrm{d}b + w_{\mathcal{B}} \int_{a\in\mathbb{A}}\int_{b\in\mathbb{B}} p_{\mathcal{B}}(b)\,\mathrm{d}a\,\mathrm{d}b \\
&= w_{\mathcal{A}} \int_{b\in\mathbb{B}} \mathrm{d}b + w_{\mathcal{B}} \int_{a\in\mathbb{A}} \mathrm{d}a \\
&= w_{\mathcal{A}}|\mathbb{B}| + w_{\mathcal{B}}|\mathbb{A}|,
\end{aligned} \tag{3.11}$$

which can be different from 1. If $w_{\mathcal{A}}|\mathbb{B}| + w_{\mathcal{B}}|\mathbb{A}| = 1$, however, then $\mathcal{S}$ would still be a valid node, despite its incompleteness.

3.2.3 Product nodes

A product node $\mathcal{P}$ models independence assumptions (eq. (3.3)) between the random variables in the subsets of a partition of its scope. An exemplary product node is shown in fig. 3.3c. We divide the $\mathbb{S}_{\mathcal{P}}$ in subsets $\mathbb{S}_{\mathcal{C}}$ and assign to each subset $\mathbb{S}_{\mathcal{C}}$ its own child $\mathcal{C}$. It thus holds that $\mathbb{S}_{\mathcal{C}} \subset \mathbb{S}_{\mathcal{P}}\ \forall \mathcal{C} \in \mathbb{C}_{\mathcal{P}}$. We call a product node *decomposable* if these subsets $\mathbb{S}_{\mathcal{C}}$ constitute a *partition* of $\mathbb{S}_{\mathcal{P}}$, i.e.,

$$\mathbb{S}_{\mathcal{P}} = \bigcup_{\mathcal{C} \in \mathbb{C}_{\mathcal{P}}} \mathbb{S}_{\mathcal{C}}, \quad \text{and} \quad \mathbb{S}_{\mathcal{C}} \cap \mathbb{S}_{\mathcal{C}'} = \emptyset\ \forall \mathcal{C}, \mathcal{C}' \in \mathbb{C}_{\mathcal{P}} \text{ s.t. } \mathcal{C} \neq \mathcal{C}'. \tag{3.12}$$

A product node models the assumption that the variables in one child's scope are statistically independent of the variables in another child's scope. We can express the distribution represented by $\mathcal{P}$ as

$$p_{\mathcal{P}}(\boldsymbol{x}) = \prod_{\mathcal{C} \in \mathbb{C}_{\mathcal{P}}} p_{\mathcal{C}}(\boldsymbol{x}_{\mathcal{C}}), \tag{3.13}$$

where the vector $\boldsymbol{x}_{\mathcal{C}}$ consists of those components in $\boldsymbol{x}$ that correspond to those random variables that are in $\mathbb{S}_{\mathcal{C}}$.

For example, consider the decomposable root node $\mathcal{R}$ from fig. 3.2, which represents the distribution $p_{\mathcal{R}}(a, b, c)$. Since a and b are statistically dependent, but statistically independent of c, we obtain

$$\begin{aligned} \mathbb{S}_{\mathcal{P}} &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \\ \mathbb{S}_{\mathcal{S}} &= \{\mathsf{a}, \mathsf{b}\} \\ \mathbb{S}_{\mathcal{U}_5} &= \{\mathsf{c}\} \\ p_{\mathcal{R}}(a, b, c) &= p_{\mathcal{S}}(a, b) p_{\mathcal{U}_5}(c). \end{aligned}$$

Analogously to completeness, if every product node in an SPN is decomposable, then the SPN is called decomposable as well.

3.3 Exact Inference

In this section we will discuss the most important inference tasks that can be addressed exactly with SPNs. As was mentioned earlier, an SPN can be interpreted as a computational graph, with different operations depending on the inference task. For all inference tasks, we will consider an SPN representing $p(\boldsymbol{x})$, with root $\mathcal{R}$ and root scope $\mathbb{S}_{\mathcal{R}} = \{\mathsf{x}_1, \ldots, \mathsf{x}_N\}$.

Throughout this section, we will make use of fig. 3.2 as a toy example. The distribution represented by the SPN can be expressed as

$$p(a,b,c) = w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b)\, p_{\mathcal{U}_2}(a)\, p_{\mathcal{U}_5}(c) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_2}(a)\, p_{\mathcal{U}_3}(b)\, p_{\mathcal{U}_5}(c) + \\ + w_{\mathcal{P}_3}\, p_{\mathcal{U}_3}(b)\, p_{\mathcal{U}_4}(a)\, p_{\mathcal{U}_5}(c), \quad (3.14)$$

and we can verify, that it conforms to the general structure of eq. (3.9).

Throughout this section we will make use of two disjoint sets of random variables $\mathbb{V}_{\mathsf{y}} \subseteq \mathbb{S}_{\mathcal{R}}$ and $\mathbb{V}_{\mathsf{z}} \subseteq \mathbb{S}_{\mathcal{R}}$. We will call the set $\mathbb{V}_{\mathsf{y}}$ the *query set*, as it consists of the random variables for which we would like to obtain values as a result of the inference task. The set $\mathbb{V}_{\mathsf{z}}$ together with a realization of the random variables in $\mathbb{V}_{\mathsf{z}}$, $\boldsymbol{z}$, will form the *inference task input* and $\boldsymbol{z}$ is called the *evidence*. Consider fig. 3.2, for example, if we would like to know the conditional expectation of b for a specific value of a, a_0, i.e., $\mathrm{E}\{\mathsf{b}|\mathsf{a} = a_0\}$, then we would set $\mathbb{V}_{\mathsf{y}} = \{\mathsf{b}\}$, $\mathbb{V}_{\mathsf{z}} = \{\mathsf{a}\}$ and $\boldsymbol{z} = a_0$. It is furthermore allowed, that either $\mathbb{V}_{\mathsf{y}}$ or $\mathbb{V}_{\mathsf{z}}$ is empty, and that either set equals the entire scope $\mathbb{S}_{\mathcal{R}}$. The union of $\mathbb{V}_{\mathsf{y}}$ and $\mathbb{V}_{\mathsf{z}}$ does not need to equal $\mathbb{S}_{\mathcal{R}}$. If $\mathbb{V}_{\mathsf{z}}$ is empty, then $\boldsymbol{z}$ is not required. All the inference tasks in this section are based on *evaluation*, i.e., for a given inference task input $\mathbb{V}_{\mathsf{z}}$ and $\boldsymbol{z}$, the SPN will compute values for the random variables in the query set $\mathbb{V}_{\mathsf{y}}$ together with the value of the marginal distribution $p(\boldsymbol{z})$. For every task we describe the operation carried out by the univariate node, sum node and product node, which together define the behavior of the complete SPN for that task. As we will see, the sum and product nodes have *recursive* definitions, i.e., they rely on the outputs of other nodes.

3.3.1 Marginal and conditional distributions

What all inference tasks have in common is that they rely on the evaluation of marginal distributions. For example, if a node represents the distribution $p(\boldsymbol{y}, \boldsymbol{z})$, we would like to be able to compute the marginal distribution

$$p(\boldsymbol{z}) = \int_{\boldsymbol{y} \in \mathbb{Y}} p(\boldsymbol{y}, \boldsymbol{z})\, \mathrm{d}\boldsymbol{y}. \quad (3.15)$$

Once we can compute the marginal distribution, we have access to the conditional distribution as well since

$$p(\boldsymbol{y}|\boldsymbol{z}) = \frac{p(\boldsymbol{y}, \boldsymbol{z})}{p(\boldsymbol{z})}. \quad (3.16)$$

It is thus sufficient to be able to evaluate the joint distribution $p(\boldsymbol{y}, \boldsymbol{z})$ and the marginal distribution $p(\boldsymbol{z})$.

In general, we have an SPN that represents the distribution $p(\boldsymbol{x})$ with root scope $\mathbb{S}_{\mathcal{R}}$, and we want to compute the value of the marginal distribution $p(\boldsymbol{z})$ for a given evidence $\boldsymbol{z}$,

where $\mathbb{V}_{\mathbf{z}} \subseteq \mathbb{S}_{\mathcal{R}}$. In the case that $\mathbb{V}_{\mathbf{z}} = \mathbb{S}_{\mathcal{R}}$ and thus $\boldsymbol{z} = \boldsymbol{x}$, i.e., that the *full* distribution is evaluated. As this is a special case of the marginalization task described here it is not treated separately. The output of a single node $\mathcal{N}$ is denoted as $p_{\mathcal{N}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$, which is the value of the marginal distribution for the given inference task input $\mathbb{V}_{\mathbf{z}}$ and $\boldsymbol{z}$ for node $\mathcal{N}$. The output of the root node equals the value of the marginal distribution, i.e., $p_{\mathcal{R}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) = p(\boldsymbol{z})$. We will now present the definitions of $p_{\mathcal{N}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$ for the different node types.

Univariate nodes: Let $\mathcal{U}$ be a univariate node that represents the distribution $p_{\mathcal{U}}(x)$ of random variable x. To compute the output $p_{\mathcal{U}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$ of $\mathcal{U}$ for the marginalization task we have to consider the possible outcomes of eq. (3.15) for a univariate distribution in the context of the inference task input $\mathbb{V}_{\mathbf{z}}$ and $\boldsymbol{z}$. There are two possibilities. The first is that $\mathsf{x} \in \mathbb{V}_{\mathbf{z}}$ where we simply proceed with evaluating $p_{\mathcal{U}}(x)$ with the corresponding value z in $\boldsymbol{z}$. The second possibility is that $\mathsf{x} \notin \mathbb{V}_{\mathbf{z}}$, and this means that we integrate over the distribution which will yield 1. This can be summarized with

$$p_{\mathcal{U}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) \coloneqq \begin{cases} p_{\mathcal{U}}(z), & \mathsf{x} \in \mathbb{V}_{\mathbf{z}} \\ 1, & \text{else}, \end{cases} \tag{3.17}$$

where z is the realization in $\boldsymbol{z}$ corresponding to random variable x.

Sum nodes: For a sum node, we can simply re-use eq. (3.10) to compute the value of the marginal distribution for given input $\mathbb{V}_{\mathbf{z}}$ and $\boldsymbol{z}$. For a sum node $\mathcal{S}$, we have

$$p_{\mathcal{S}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) \coloneqq \sum_{\mathcal{C} \in \mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}). \tag{3.18}$$

Product nodes: Analogously to sum nodes, we base the marginalization output of a product node $\mathcal{P}$ on eq. (3.13) and we obtain

$$p_{\mathcal{P}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) \coloneqq \prod_{\mathcal{C} \in \mathbb{C}_{\mathcal{P}}} p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}). \tag{3.19}$$

Example: We want to compute the value of $p(a_0, c_0)$ using the SPN from fig. 3.2. The input is thus $\mathbb{V}_{\mathbf{z},0} = \{\mathsf{a}, \mathsf{c}\}$ and $\boldsymbol{z} = (a_0, c_0)^\top$, and the different node outputs are

$$p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = 1$$
$$p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(a_0)$$

$$
\begin{aligned}
p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= 1 \\
p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_4}(a_0) \\
p_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_5}(c_0) \\
p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(a_0) \\
p_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(a_0) \\
p_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_4}(a_0) \\
p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= w_{\mathcal{P}_1}\, p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) + w_{\mathcal{P}_2}\, p_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) + w_{\mathcal{P}_3}\, p_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \\
&= (w_{\mathcal{P}_1} + w_{\mathcal{P}_2})\, p_{\mathcal{U}_2}(a_0) + w_{\mathcal{P}_3}\, p_{\mathcal{U}_4}(a_0) \\
p_{\mathcal{R}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \\
&= (w_{\mathcal{P}_1} + w_{\mathcal{P}_2})\, p_{\mathcal{U}_2}(a_0)\, p_{\mathcal{U}_5}(c_0) + w_{\mathcal{P}_3}\, p_{\mathcal{U}_4}(a_0)\, p_{\mathcal{U}_5}(c_0).
\end{aligned}
$$

Looking at the root output, we see that it is the same we would get by integrating out b from eq. (3.14).

3.3.2 Sampling

We are also able to draw samples from the distribution represented by the SPN, making SPNs a *generative model* [10]. Additionally to sampling from the distribution, we are able to sample from marginal and conditional distributions as well. In general, we want to draw a sample $\boldsymbol{y}(\boldsymbol{z})$ from the conditional distribution $p(\boldsymbol{y}|\boldsymbol{z})$. We will again use $\mathbb{V}_{\mathbf{z}}$, $\boldsymbol{z}$, and $\mathbb{V}_{\mathbf{y}}$, where $\mathbb{V}_{\mathbf{y}} \subseteq \mathbb{S}_{\mathcal{R}}$ contains the random variables we want to sample. In order to sample from the non-marginalized, unconditional distribution $p(\boldsymbol{x}) = p(x_1, x_2, \ldots)$, we set $\mathbb{V}_{\mathbf{z}} = \emptyset$ and $\mathbb{V}_{\mathbf{y}} = \mathbb{S}_{\mathcal{R}} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots\}$. The output of the sampling operation of a node $\mathcal{N}$, is again the value of the distribution for the given SPN input $p_{\mathcal{N}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$, and additionally a *node sample* $\boldsymbol{y}_{\mathcal{N}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$, which can be scalar-valued in the case of a univariate node. We will now present the operations of the different nodes for the sampling task.

Univariate nodes: Let $\mathcal{U}$ be a univariate node representing the distribution $p_{\mathcal{U}}(x)$ of random variable x. Analogously to the operation when evaluating the marginal distribution value of a univariate node in eq. (3.17), we have to distinguish two cases when sampling. However, this time the set $\mathbb{V}_{\mathbf{y}}$ is relevant, i.e., if $\mathsf{x} \in \mathbb{V}_{\mathbf{y}}$, then we are interested in a sample of x and we draw a sample of the node distribution $p_{\mathcal{U}}(x)$. On the other hand if $\mathsf{x} \notin \mathbb{V}_{\mathbf{y}}$, then we can simply ignore the univariate node. This can be summarized as follows

$$
y_{\mathcal{U}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) \coloneqq \begin{cases} \text{a sample drawn from } p_{\mathcal{U}}(x), & \mathsf{x} \in \mathbb{V}_{\mathbf{y}} \\ \text{no value}, & \text{else.} \end{cases} \tag{3.20}
$$

The second case, with no output, is used if we do not care about the value of the random variable represented by the univariate node, and in an actual implementation, a default value would be used.

Sum nodes: The sampling operation for a sum node $\mathcal{S}$ is twofold. First, the re-scaled and re-normalized weights

$$\tilde{w}_{\mathcal{C}} = \frac{w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})}{\sum_{\mathcal{C}' \in \mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}'}\, p_{\mathcal{C}'}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})}, \quad \forall \mathcal{C} \in \mathbb{C}_{\mathcal{S}} \tag{3.21}$$

are computed. The second step is the standard procedure for drawing samples from a mixture, i.e., interpreting the re-scaled weights as the probabilities of each child and randomly drawing a child $\mathcal{C}^*$ according to these probabilities. The sample of the sum node is the sample of the drawn child, i.e., $\boldsymbol{y}_{\mathcal{S}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) = \boldsymbol{y}_{\mathcal{C}^*}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$.

Product nodes: The sample $\boldsymbol{y}_{\mathcal{P}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$ of a product node $\mathcal{P}$ is simply the vector stacking the outputs of all of the product nodes children, where "no value" outputs are ignored. For simplicity, we did not define the order of the variables within the vector here, but when implementing the sampling procedure described here, care must be taken that the output sample values are not mixed up.

Example: We go back to the SPN in fig. 3.2, and we want to draw a sample from $p(a|\mathsf{b} = b_0)$, for some arbitrary value b_0. The input is formulated as $\mathbb{V}_{\mathbf{y},0} = \{\mathsf{a}\}$, $\mathbb{V}_{\mathbf{z},0} = \{\mathsf{b}\}$, and $\boldsymbol{z}_0 = b_0$. For the univariate nodes, we obtain

$$\begin{aligned}
&p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(b_0), && \\
&p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = 1, && y_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = a_1^* \\
&p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(b_0), && \\
&p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = 1, && y_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = a_2^* \\
&p_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = 1, &&
\end{aligned}$$

where a_1^* and a_2^* are samples drawn from $p_{\mathcal{U}_2}(a) = \mathcal{N}\left(\mu_{\mathsf{a},1}, \sigma^2_{\mathsf{a},1}\right)$ and $p_{\mathcal{U}_4}(a) = \mathcal{N}\left(\mu_{\mathsf{a},2}, \sigma^2_{\mathsf{a},2}\right)$, respectively. In this example, the node samples of the product nodes $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ is again just a scalar sample, because we are only interested in the sample values of a. We thus obtain

$$\begin{aligned}
p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(b_0), \\
p_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(b_0),
\end{aligned}$$

$$p_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(b_0),$$

for the marginal distribution values and

$$\begin{aligned}
y_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= \Big(y_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}), y_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\Big)^\top = a_1^* \\
y_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= \Big(y_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}), y_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\Big)^\top = a_1^* \\
y_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= \Big(y_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}), y_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\Big)^\top = a_2^*
\end{aligned}$$

for the node samples. Continuing with the sum node $\mathcal{S}$, we compute the re-scaled and re-normalized weights according to eq. (3.21)

$$\begin{aligned}
\tilde{w}_{\mathcal{P}_1} &= \frac{w_{\mathcal{P}_1}\, p_{\mathcal{P}_1}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})}{\sum_{i=1}^{3} w_{\mathcal{P}_i}\, p_{\mathcal{P}_i}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})} = \frac{w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0)}{w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)} \\
\tilde{w}_{\mathcal{P}_2} &= \frac{w_{\mathcal{P}_2}\, p_{\mathcal{P}_2}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})}{\sum_{i=1}^{3} w_{\mathcal{P}_i}\, p_{\mathcal{P}_i}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})} = \frac{w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)}{w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)} \\
\tilde{w}_{\mathcal{P}_3} &= \frac{w_{\mathcal{P}_2}\, p_{\mathcal{P}_2}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})}{\sum_{i=1}^{3} w_{\mathcal{P}_i}\, p_{\mathcal{P}_i}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z},0})} = \frac{w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)}{w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)}.
\end{aligned}$$

Now we randomly select a product node $\mathcal{P}^*$ from $\mathcal{P}_1$, $\mathcal{P}_2$ or $\mathcal{P}_3$, according to the probabilities $\tilde{w}_{\mathcal{P}_1}$, $\tilde{w}_{\mathcal{P}_2}$ and $\tilde{w}_{\mathcal{P}_3}$ respectively. Since the root node is again a product node, and we are not interested in a sample of c, the node sample of the selected product node $\mathcal{P}^*$ is the output of the SPN.

3.3.3 Mean and MMSE Estimate

SPNs are furthermore capable of computing conditional means of the form

$$\boldsymbol{\mu}_{\mathbf{y}|\boldsymbol{z}} = \mathrm{E}\{\mathbf{y}|\mathbf{z} = \boldsymbol{z}\} = \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y}\, p(\boldsymbol{y}|\boldsymbol{z})\, \mathrm{d}\boldsymbol{y}, \tag{3.22}$$

which is also known as the minimum mean square error (MMSE) estimate of $\mathbf{y}$ given $\boldsymbol{z}$. Using $\mathbb{V}_{\mathbf{y}}$ and the input $\mathbb{V}_{\mathbf{z}}$ and $\boldsymbol{z}$, we can compute $\mathrm{E}\{\mathbf{y}|\mathbf{z} = \boldsymbol{z}\}$. Here $\mathbf{y}$ and $\mathbf{z}$ are the random vectors of the random variables in $\mathbb{V}_{\mathbf{y}}$ and $\mathbb{V}_{\mathbf{z}}$, respectively. If we set $\mathbb{V}_{\mathbf{y}} = \mathbb{S}_{\mathcal{R}}$, we can compute the unconditional mean $\boldsymbol{\mu}_{\mathbf{x}}$.

The computations carried out by the different node types are based on eq. (3.22), i.e., given a node N we want to compute

$$\mathrm{E}_{\mathcal{N}}\{\mathbf{y}|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\} \coloneqq \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y}\, p_{\mathcal{N}}(\boldsymbol{y}|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})\, \mathrm{d}\boldsymbol{y}. \tag{3.23}$$

The overall MMSE estimate of the SPN is again the output of the root node, i.e., $\mathrm{E}\{\mathbf{y}|\mathbf{z} = \boldsymbol{z}\} = \mathrm{E}_{\mathcal{R}}\{\mathbf{y}|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\}$.

Univariate nodes: Let $\mathcal{U}$ be a univariate node, representing the distribution $p_{\mathcal{U}}(x)$, with node specific mean $\mathrm{E}_{\mathsf{x}\sim p_{\mathcal{U}}(x)}\{\mathsf{x}\}$. Analogously to sampling, we have to distinguish whether $\mathsf{x} \in \mathbb{V}_{\mathbf{y}}$ or not. If $\mathsf{x} \in \mathbb{V}_{\mathbf{y}}$, then eq. (3.23) collapses to $\mathrm{E}_{\mathsf{x}\sim p_{\mathcal{U}}(x)}\{\mathsf{x}\}$, and if $\mathsf{x} \notin \mathbb{V}_{\mathbf{y}}$, then we are not interested in a result per definition, and no value is returned. We obtain

$$\mathrm{E}_{\mathcal{U}}\{\mathbf{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}}\} \coloneqq \begin{cases} \mathrm{E}_{\mathsf{x}\sim p_{\mathcal{U}}(x)}\{\mathsf{x}\}, & \mathsf{x}_i \in \mathbb{V}_{\mathbf{y}} \\ \text{no value}, & \text{else} \end{cases} \tag{3.24}$$

for the operation of a univariate node, which is structurally very similar to eq. (3.20).

Sum nodes: The operation of a sum node $\mathcal{S}$ deserves a little more attention this time. The computation of $\mathrm{E}_{\mathcal{S}}\{\mathbf{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}}\}$ can be derived by inserting eq. (3.16) and eq:spns:sum-node into eq. (3.23), i.e.,

$$\begin{aligned}
\mathrm{E}_{\mathcal{S}}\{\mathbf{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}}\} &\coloneqq \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y}\, p_{\mathcal{N}}(\boldsymbol{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\,\mathrm{d}\boldsymbol{y} \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})} \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y}\, p_{\mathcal{S}}(\boldsymbol{y},\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\,\mathrm{d}\boldsymbol{y} \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})} \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y} \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{y},\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\,\mathrm{d}\boldsymbol{y} \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})} \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y} \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}}) p_{\mathcal{C}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\,\mathrm{d}\boldsymbol{y} \qquad (3.25) \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})} \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}}) \int_{\boldsymbol{y}\in\mathbb{Y}} \boldsymbol{y}\, p_{\mathcal{C}}(\boldsymbol{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\,\mathrm{d}\boldsymbol{y} \qquad (3.26) \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})} \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})\, \mathrm{E}_{\mathcal{C}}\{\mathbf{y}|\boldsymbol{z};\mathbb{V}_{\mathbf{z}}\}. \qquad (3.27)
\end{aligned}$$

Here we can compute $p_{\mathcal{S}}(\boldsymbol{z};\mathbb{V}_{\mathbf{z}})$ using eq. (3.18) and so eq. (3.27) is again a recursive definition only depending on the child outputs.

Product nodes: The output of product nodes is again a stacked vector of the children's conditional expectations, ignoring "no value" outputs. Again, as with sampling, care must be taken that the values in the output are not mixed up.

Example: Again we use the SPN in fig. 3.2. We want to compute the MMSE estimate for a, conditioned on $\mathsf{b} = b_0$, i.e., we want to compute $\mathrm{E}\{\mathsf{a}|\mathsf{b} = b_0\}$. As with the previous example, we set $\mathbb{V}_{\mathbf{y},0} = \{\mathsf{a}\}$, which implies $\mathbf{y}_0 = \mathsf{a}$, $\mathbb{V}_{\mathbf{z},0} = \{\mathsf{b}\}$, and $\boldsymbol{z}_0 = b_0$. We do not care about c. For the univariate nodes and product nodes we obtain

$$p_{\mathcal{U}_1}(\boldsymbol{z}_0;\mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(b_0),$$

$$
\begin{aligned}
p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= 1, & \mathrm{E}_{\mathcal{U}_2}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}\} &= \mu_{\mathsf{a},1} \\
p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_3}(b_0), & & \\
p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= 1, & \mathrm{E}_{\mathcal{U}_4}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}\} &= \mu_{\mathsf{a},2} \\
p_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= 1, & & \\
p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(b_0), & \mathrm{E}_{\mathcal{P}_1}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}\} &= \mu_{\mathsf{a},1} \\
p_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(b_0), & \mathrm{E}_{\mathcal{P}_2}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}\} &= \mu_{\mathsf{a},1} \\
p_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) &= p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(b_0), & \mathrm{E}_{\mathcal{P}_3}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}\} &= \mu_{\mathsf{a},2}
\end{aligned}
$$

We obtain the node MMSE estimate of the sum node $\mathcal{S}$ using eq. (3.27), i.e.,

$$
\begin{aligned}
\mathrm{E}_{\mathcal{S}}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z}}\} &= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})} \sum_{i=1}^{3} w_{\mathcal{P}_i}\, p_{\mathcal{P}_i}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z}})\, \mathrm{E}_{\mathcal{P}_i}\{\mathbf{y}_0|\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z}}\} \\
&= \frac{1}{p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})} \Big[w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0)\, \mu_{\mathsf{a},1} + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0)\, \mu_{\mathsf{a},1} + w_{\mathcal{P}_3}\, p_{\mathcal{U}_3}(b_0)\, \mu_{\mathsf{a},2} \Big],
\end{aligned}
$$

where

$$
p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \sum_{i=1}^{3} w_{\mathcal{P}_i}\, p_{\mathcal{P}_i}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = w_{\mathcal{P}_1}\, p_{\mathcal{U}_1}(b_0) + w_{\mathcal{P}_2}\, p_{\mathcal{U}_3}(b_0) + w_{\mathcal{P}_3}\, p_{\mathcal{U}_3}(b_0).
$$

Finally, since we do not care about c, the last product node $\mathcal{P}$ does not carry out any operation, and thus the sum node output is the final MMSE estimate.

3.3.4 Mean power and variance

We can re-use the procedure we used to compute the mean for different expectations, as long as we can compute the equivalent to eq. (3.27). The limiting factor here is exchanging the sum with the integral in the step from eq. (3.25) to eq. (3.26), which is in general not possible for arbitrary expectations. Fro example, the conditional second moment

$$
P_{\mathsf{y}_i|\boldsymbol{z}} = \mathrm{E}\Big\{\mathsf{y}_i^2|\mathbf{z} = \boldsymbol{z}\Big\} = \int_{y_i \in \mathbb{Y}_i} y_i^2\, p(y_i|\boldsymbol{z})\, \mathrm{d}y_i. \tag{3.28}
$$

of the individual random variables y_i in random vector $\mathbf{y}$ can be computed in this way. The conditional variance $\sigma^2_{\mathsf{y}_i|\boldsymbol{z}}$, on the other hand, cannot be computed using eq. (3.27), because when propagating the variance from the child nodes, we cannot exchange the integral with the sum. However, we can use the well-known relation

$$
\sigma^2_{\mathsf{y}_i|\boldsymbol{z}} = P_{\mathsf{y}_i|\boldsymbol{z}} - \mu^2_{\mathsf{y}_i|\boldsymbol{z}}, \tag{3.29}
$$

to compute the variance from the mean and the second moment. This is useful, because in addition to being able to compute the MMSE estimator, we are also able to compute a

measure of confidence for our estimator, i.e., the variance.

Let us write $\mathrm{E}\{\mathbf{y}^2|\boldsymbol{z}\} = (P_{\mathrm{y}_1|\boldsymbol{z}}, P_{\mathrm{y}_2|\boldsymbol{z}}, \ldots)^\top$ as a shorthand for the stacked individual mean powers. For every node $\mathcal{N}$ we want to compute the node mean powers $\mathrm{E}_{\mathcal{N}}\{\mathbf{y}^2|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\}$, which are defined analogously to eq. (3.23).

Univariate nodes: Let us again assume a univariate node $\mathcal{U}$, representing the distribution $p_{\mathcal{U}}(x)$, and with node specific second moment $\mathrm{E}_{\mathsf{x}\sim p_{\mathcal{U}}(x)}\{\mathsf{x}^2\}$ We have to adapt eq. (3.24) to return the second moment instead of the mean, i.e,

$$\mathrm{E}_{\mathcal{U}}\Big\{\mathbf{y}^2|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\Big\} := \begin{cases} \mathrm{E}_{\mathsf{x}\sim p_{\mathcal{U}}(x)}\{\mathsf{x}^2\}, & \mathsf{x} \in \mathbb{V}_{\mathbf{y}} \\ \text{no value}, & \text{else.} \end{cases} \tag{3.30}$$

Sum nodes: For a sum node $\mathcal{S}$, analogously to eq. (3.27), we obtain

$$\mathrm{E}_{\mathcal{S}}\Big\{\mathbf{y}^2|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\Big\} = \frac{1}{p_{\mathcal{S}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})} \sum_{\mathcal{C}\in\mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})\, \mathrm{E}_{\mathcal{C}}\Big\{\mathbf{y}^2|\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}\Big\}. \tag{3.31}$$

Product nodes: The operation for product nodes is as before, i.e., stacking the child outputs.

3.4 VMAP Estimation and Selective SPNs

In the previous section, we discussed some of the inference tasks that are exactly solvable, including the MMSE estimator. Another popular estimator is the vector maximum a-posteriori (VMAP) estimator, which is defined as the most probable realization of the random vector $\mathbf{y}$ given the input $\boldsymbol{z}$, i.e.,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} p(\boldsymbol{y}|\boldsymbol{z}). \tag{3.32}$$

Contrary to the MMSE estimator, however, there is generally no efficient algorithm to compute the VMAP estimate using SPNs. With SPNs, exact VMAP inference is NP-hard [29], [30], but approximations exist [22], [29], [30], which we will however not present here. Instead we discuss why VMAP is intractable based on this we explain a special class of SPNs, i.e., *selective SPNs*, where VMAP is tractable and the corresponding algorithm is presented.

3.4.1 Intractability of VMAP

We will now investigate why we cannot solve eq. (3.32) with an SPN exactly in general, and we will consider one special case where we can. For univariate nodes representing $p(x)$, there is no problem; we simply assume that we know the *mode* of the distribution, i.e.,

$$\hat{x} = \operatorname*{argmax}_x p(x). \tag{3.33}$$

The product nodes pose no problem either, because we merely stack the results of a product node's children, just as for the MMSE estimator. The intractability of the VMAP estimator is due to the operation of the sum node. Let us consider as a simple example, two functions $f(x)$ and $g(x)$. Then, even if we knew $\hat{x}_f = \operatorname{argmax} f(x)$ and $\hat{x}_g = \operatorname{argmax} g(x)$, in general we would have to exhaustively search to find $\operatorname{argmax}\{f(x) + g(x)\}$. However, in the special case that $f(x)$ and $g(x)$ have *disjoint supports*, i.e., $f(x) \neq 0 \Rightarrow g(x) = 0$ and $g(x) \neq 0 \Rightarrow f(x) = 0$, the solution is simply

$$\operatorname*{argmax}_x \{f(x) + g(x)\} = \operatorname*{argmax}_{x \in \{\hat{x}_f, \hat{x}_g\}} \{f(x) + g(x)\}. \tag{3.34}$$

3.4.2 Selective SPNs and tractable VMAP

Based on the observation from eq. (3.34), we define *selective SPNs* [30] to be SPNs where the supports of the distributions of the children of a sum node are disjoint. We can now use eq. (3.34) to compute the VMAP estimate of sum nodes, which makes the VMAP estimator tractable for selective SPNs. The procedure presented below can also be applied to non-selective SPNs, however then in general the result will not be the VMAP estimator but an approximation.

Analogously to the operation definitions in the previous section, we want to compute the VMAP estimate $\hat{\boldsymbol{y}}_{\mathcal{N}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}})$ for node a $\mathcal{N}$ given $\mathbb{V}_{\mathbf{y}}$, $\mathbb{V}_{\mathbf{z}}$ and $\boldsymbol{z}$. The output of the root $\mathcal{R}$ is the overall VMAP estimate, i.e., $\hat{\boldsymbol{y}}_{\mathcal{R}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) = \hat{\boldsymbol{y}} = \operatorname{argmax}_{\boldsymbol{y}} p(\boldsymbol{y}|\boldsymbol{z})$. We can again use $\mathbb{V}_{\mathbf{y}} = \mathbb{S}_{\mathcal{R}}$ to compute the unconditional mode $\hat{\boldsymbol{x}}$ of the distribution $p(\boldsymbol{x})$ the root, and thus the SPN, represents.

Univariate nodes: The output of a univariate node $\mathcal{U}$ representing distribution $p_{\mathcal{U}}(x)$ with node specific mode $\hat{x}_{\mathcal{U}} = \operatorname{argmax}_x p_{\mathcal{U}}(x)$ is defined as

$$\hat{\boldsymbol{y}}_{\mathcal{U}}(\boldsymbol{z}; \mathbb{V}_{\mathbf{z}}) := \begin{cases} \hat{x}_{\mathcal{U}}, & \mathsf{x} \in \mathbb{V}_{\mathbf{y}} \\ \text{no value}, & \text{else.} \end{cases} \tag{3.35}$$

Additionally, for the VMAP estimator we have to change the evaluation of the marginal

distribution of $\mathcal{U}$ (eq. (3.17)) to

$$p_{\mathcal{U}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}) \coloneqq \begin{cases} p_{\mathcal{U}}(z), & \mathsf{x} \in \mathbb{V}_{\mathsf{z}} \\ \max_x p_{\mathcal{U}}(x) = p_{\mathcal{U}}(\hat{x}_{\mathcal{U}}), & \mathsf{x} \in \mathbb{V}_{\mathsf{y}} \\ 1, & \text{else.} \end{cases} \tag{3.36}$$

Sum nodes: The operation of a sum node $\mathcal{S}$ is similar to the sampling operation. First we have to find the child $\hat{\mathcal{C}}$ that maximizes the product of weight and marginal distribution, i.e.,

$$\hat{\mathcal{C}} = \operatorname*{argmax}_{\mathcal{C} \in \mathbb{C}_{\mathcal{S}}} w_{\mathcal{C}}\, p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}). \tag{3.37}$$

Since we defined selective SPNs to have disjoint sum child supports, only $\hat{\mathcal{C}}$ can have a non-zero marginal distribution value, i.e., $p_{\hat{\mathcal{C}}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}) \geq 0$ and $p_{\mathcal{C}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}) = 0 \quad \forall \mathcal{C} \in \mathbb{C}_{\mathcal{S}} \setminus \{\hat{\mathcal{C}}\}$. According to eq. (3.34), the VMAP estimate of $\mathcal{S}$ is now defined as the VMAP estimate of $\hat{\mathcal{C}}$, i.e.,

$$\hat{\boldsymbol{y}}_{\mathcal{S}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}) \coloneqq \hat{\boldsymbol{y}}_{\hat{\mathcal{C}}}(\boldsymbol{z}; \mathbb{V}_{\mathsf{z}}). \tag{3.38}$$

Product nodes: Product nodes, again, simply stack the values of their children.

Example: To demonstrate the VMAP procedure discussed above we will again use the SPN in fig. 3.2. However, due to the Gaussian leaves, which have infinite support, this SPN is not selective, but as we can see in fig. 3.1, the mixture components are well separated and we can at least assume approximate selectivity. We want to compute $\operatorname{argmax}_{a,c} p(a, c | \mathsf{b} = \mu_{\mathsf{b},1})$ and thus we set $\mathbb{V}_{\mathbf{y},0} = \{\mathsf{a}, \mathsf{c}\}$, which implies $\mathbf{y}_0 = (\mathsf{a}, \mathsf{c})^\top$, and the task input is $\mathbb{V}_{\mathbf{z},0} = \{\mathsf{b}\}$ and $\boldsymbol{z}_0 = \mu_{\mathsf{b},1}$. For all the univariate nodes and the first three product nodes we obtain

$$\begin{aligned}
&p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(\mu_{\mathsf{b},1}), && \\
&p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(\mu_{\mathsf{a},1}), && \hat{\boldsymbol{y}}_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},1} \\
&p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(\mu_{\mathsf{b},1}), && \\
&p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_4}(\mu_{\mathsf{a},2}), && \hat{\boldsymbol{y}}_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},2} \\
&p_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_5}(\hat{c}), && \hat{\boldsymbol{y}}_{\mathcal{U}_5}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \hat{c} \\
&p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(\mu_{\mathsf{a},1})\, p_{\mathcal{U}_1}(\mu_{\mathsf{b},1}), && \hat{\boldsymbol{y}}_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},1} \\
&p_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_4}(\mu_{\mathsf{a},1})\, p_{\mathcal{U}_3}(\mu_{\mathsf{b},1}), && \hat{\boldsymbol{y}}_{\mathcal{P}_2}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},1} \\
&p_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\, p_{\mathcal{U}_4}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_4}(\mu_{\mathsf{a},2})\, p_{\mathcal{U}_3}(\mu_{\mathsf{b},1}), && \hat{\boldsymbol{y}}_{\mathcal{P}_3}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},2}
\end{aligned}$$

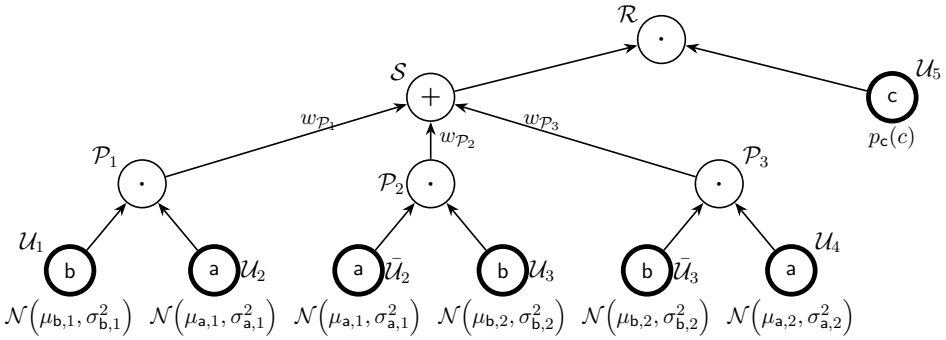


Figure 3.4: The SPN from fig. 3.2 converted to an SPT. Copies of nodes are indicated with a bar, e.g., $\mathcal{U}_2$ and $\bar{\mathcal{U}}_2$.

We now continue with the sum node $\mathcal{S}$, and we have to select the product node $\hat{\mathcal{P}}$ that maximizes $w_{\hat{\mathcal{P}}} p_{\hat{\mathcal{P}}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})$. Based on visual inspection of fig. 3.1 and the fact that we conditioned on $\mathsf{b} = \mu_{\mathsf{b},1}$, we can conclude that only mixture component M_1 is relevant and thus $\hat{\mathcal{P}} = \mathcal{P}_1$. And thus the output of the sum node $\mathcal{S}$ is

$$p_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = p_{\mathcal{U}_2}(\mu_{\mathsf{a},1})\, p_{\mathcal{U}_1}(\mu_{\mathsf{b},1}), \quad \hat{\boldsymbol{y}}_{\mathcal{S}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \hat{\boldsymbol{y}}_{\mathcal{P}_1}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \mu_{\mathsf{a},1}.$$

Finally, we stack the outputs of the children of the root node, and for the overall VMAP estimate we obtain

$$\hat{\boldsymbol{y}}_{\mathcal{R}}(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}) = \left(\hat{\boldsymbol{y}}_{\mathcal{S}}^\top(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0}), \hat{\boldsymbol{y}}_{\mathcal{U}_5}^\top(\boldsymbol{z}_0; \mathbb{V}_{\mathbf{z},0})\right)^\top = (\mu_{\mathsf{a},1}, \hat{c})^\top.$$

3.5 Sum-Product Trees

Sum-product trees (SPTs) are a special kind of SPN, where the graph is not just a rooted DAG but an oriented tree, i.e., every node can have at most one parent. SPTs are important in practice since most structure learning algorithms, including those presented in chapter 4, learn SPTs and not the more general SPNs. An exception to this is, e.g., [13], a newly proposed structure learning algorithm capable of directly learning SPNs.

In general, SPTs are not less expressive than SPNs. However, they may require exponentially more nodes than SPNs to represent the same distribution. For *deterministic* inference, where the same input will always yield the same output, SPNs can be converted to equivalent SPTs. The only non-deterministic inference task we have discussed is sampling, since it involves a random sampling operation at every sum node. For example, the SPN from fig. 3.2 converted to an SPT is shown in fig. 3.4.

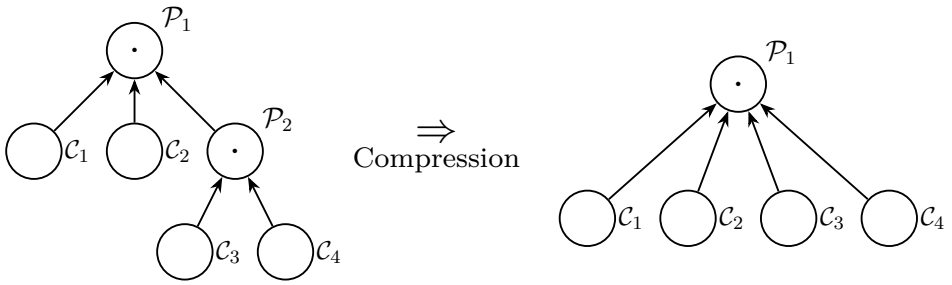


Figure 3.5: Model compression: Product node $\mathcal{P}_2$ can be removed because its only parent is another product node $\mathcal{P}_1$.

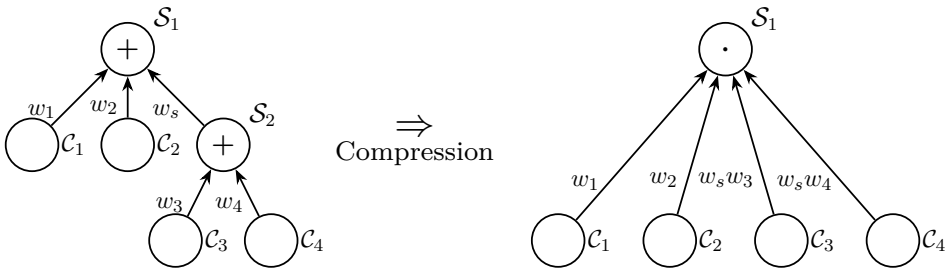


Figure 3.6: Model compression: Sum node $\mathcal{S}_2$ can be removed because its only parent is another sum node $\mathcal{S}_1$.

3.6 Further properties of SPNs

3.6.1 Compression

We can reduce the number of nodes in an SPN by exploiting the structure of eqs. (3.10) and (3.13). Whenever there is a product node $\mathcal{P}_2$ with only one parent, and this parent is again a product node $\mathcal{P}_1$, then we can replace $\mathcal{P}_2$ with the children of $\mathcal{P}_2$. An example for the compression of one product node into another is shown in fig. 3.5. The same can be done with sum nodes, but the weights of the parent sum node need to be adjusted after merging, this is shown in fig. 3.6. As a consequence of this, every SPT can be compressed such that no sum node has a sum node as parent and that no product node has another product node as parent.

3.6.2 Compatibility with other models

Every node in an SPN represents a distribution. The node distributions are distinguished by their scope and the conditions imposed by the latent variables of the sum nodes on the path to the root. This allows us to model the distribution represented by any node in the SPN by different means, e.g. by a different PGM. As long as such a model "behaves" like a "typical" SPN node, we can insert it in the SPN and use it as a replacement for a node

and its descendants.

We will now formalize this. We want to replace a node $\mathcal{N}$ whose scope consists of the variables of the random vector $\mathbf{x} = (\mathsf{x}_1, \ldots, \mathsf{x}_N)^\top$. In the following, $\mathbf{y}$ and $\mathbf{z}$ are two random vectors consisting of arbitrary random variables from $\mathbf{x}$, with the constraint that variables appearing in $\mathbf{y}$ cannot also appear in $\mathbf{z}$. If we want to be able to use the SPN for the inference tasks presented in Sections 3.3 and 3.4, the model, or method, replacing $\mathcal{N}$ needs to be able to (approximately) compute:

1. the value of the joint distribution $p(\boldsymbol{x})$;
2. the value of the marginal distribution $p(\boldsymbol{z})$;
3. the individual conditional expectations $\mathrm{E}\{\mathbf{y}_i | \mathbf{z} = \boldsymbol{z}\}$ and $\mathrm{E}\{\mathbf{y}_i^2 | \mathbf{z} = \boldsymbol{z}\}$;
4. the most probable $\hat{\boldsymbol{y}}$ conditioned on $\boldsymbol{z}$, i.e., $\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} p(\boldsymbol{y}|\boldsymbol{z})$;
5. samples from $p(\boldsymbol{x})$ and $p(\boldsymbol{y}|\boldsymbol{z})$.

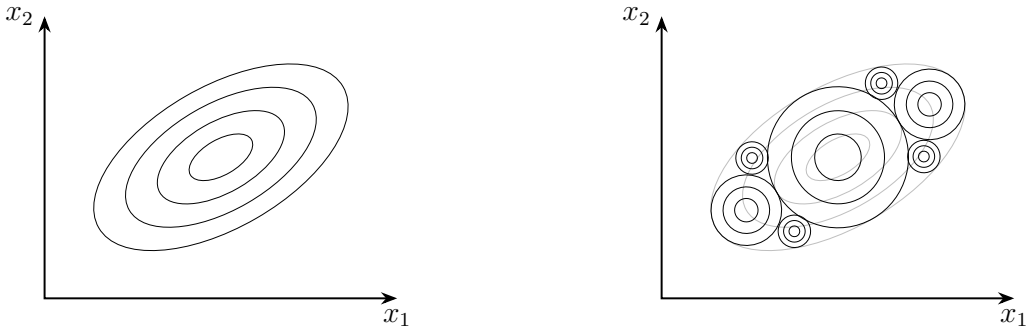
A special case of this "model insertion" are univariate nodes, where requirements 1 to 4 can be simplified. A model or method that is used as a univariate node representing the distribution $p(x)$ of a random variable x needs to be able to (approximately) compute:

1. the value of the distribution $p(x)$;
2. the mean μ_{x} and second moment P_{x};
3. the mode $\hat{x} = \operatorname*{argmax}_x p(x)$;
4. samples from $p(x)$.

3.6.3 Limitations

So far we discussed only complete and decomposable SPNs. We know that these SPNs represent valid distributions, but neither completeness nor decomposability are necessary for validity. The question is, whether these conditions limit the representative power of SPNs. In [31], it was shown that the gain in simplicity obtained by restricting ourselves to complete and decomposable SPNs is usually worth the loss of representative power, which is not much.

However, in [20] it was shown that there exist distributions an SPN cannot represent without requiring an infinite number of nodes, but other PGMs could model these distributions efficiently. This is mainly due to the fact that SPNs can only model indirect variable dependencies, but not direct ones. Consider, for example, a Gaussian random vector $\mathbf{x} = (\mathsf{x}_1, \mathsf{x}_2)^\top \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{x}}, \boldsymbol{C}_{\mathbf{x}})$ with a non-diagonal covariance matrix $\boldsymbol{C}_{\mathbf{x}}$. Figure 3.7a shows an illustration of the distribution of such a random vector, and we can see that when



(a) (b)

Figure 3.7: (a) Illustration of the distribution of a 2-D jointly Gaussian random vector $\mathbf{x} = (\mathsf{x}_1, \mathsf{x}_2)^\top \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{x}, \boldsymbol{C}_\mathbf{x})$ with non-diagonal $\boldsymbol{C}_\mathbf{x}$. (b) A possibility how an SPN could approximate the distribution from (a).

x_1 grows, x_2 tends to grow as well. This is an example of a direct variable dependency, and SPNs are unable to capture this behavior efficiently. Because SPNs can represent distributions only with mixtures and independence assumptions, we have to "divide" the domain $\mathbf{x}$ into parts where x_1 and x_2 are *approximately* independent. In fig. 3.7b, we can see a possible approximation of the distribution of $\mathbf{x}$ by means of mixture components of uncorrelated Gaussian distributions. The figure illustrates that an SPN would require an infinite number of mixture components to fully capture direct variable dependencies.

A possible solution for this problem would be to allow for multivariate leaves, involving a limited number of random variables, and model them by different means. We could for example use other PGMs that can model direct variable dependencies, to model distributions of five random variables or less [33]. However, this approach requires that the plugged-in model fulfills the requirements stated in Section 3.6.2, especially marginalization. Without the possibility of marginalization, many inference tasks would not be available.

Chapter 4

Learning the Structure of SPTs

In this chapter two methods to learn SPTs from data are presented. Both are so called *structure learning* methods. This means that the graph is built during learning and that no prior assumptions about depth, node counts, connections, etc., are required. Another common approach is *weight learning*, where a structure is assumed prior to learning and only the sum weights and univariate distributions are learned. The advantage of structure learning is that, in general, fewer hyperparameters are required. The downside is that the structures learned are typically SPTs and not more general SPNs. This leads to larger graphs than necessary for a given training data set.

The next two sections discuss *offline* and *online structure learning* respectively. In this context offline structure learning means that all the training data is available at the beginning of the learning procedure, and the learning algorithm can make use of the complete data. Online structure learning on the other hand only uses a limited number of training samples, a so called *minibatch*, at a time. This allows online algorithms to handle much larger data sets that would not fit into memory. The challenge is to still learn the dependencies between the different variables without a complete view of the data.

Both algorithms depend on several sub-algorithms that are generally interchangeable. These are presented in the later sections of this chapter.

It is assumed that every training sample $\boldsymbol{x}_n$ comes from the same distribution $p(\boldsymbol{x})$ of random vector $\mathbf{x} = (\mathsf{x}_1, \mathsf{x}_2, \ldots)^\top$, and we define $\mathbb{V}_{\mathbf{x}} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots\}$, the set of all random variables in $\mathbf{x}$. The exact size of $\mathbf{x}$ is of minor importance. The individual random variables x_i can either be discrete, continuous or mixed. The general goal of every learning algorithm is to learn an SPT with root $\mathcal{R}$ that represents distribution $p_{\mathcal{R}}(\boldsymbol{x})$, such that $p_{\mathcal{R}}(\boldsymbol{x})$ is as *close* to the true, unknown $p(\boldsymbol{x})$ as possible. A practical measure for *closeness* of $p_{\mathcal{R}}(\boldsymbol{x})$ to $p(\boldsymbol{x})$ is presented in Section 5.1.

4.1 Offline Structure Learning

Almost every offline structure learning algorithm for SPTs is derived from *LearnSPN* [9], which in turn extends the concept presented in [7]. Algorithm 4.1 is based on LearnSPN, with adaptations from [37]. The rest of this section is used to explain algorithm 4.1 in detail, and from here on, LearnSPN refers to the specific algorithm presented here.

4.1.1 LearnSPN

LearnSPN is a recursive algorithm and with every iteration a new node is created. The inputs are a set of training samples $\mathbb{T}$ and a set of random variables $\mathbb{V}$. The goal is to create a node $\mathcal{N}$ that represents a distribution $p_{\mathcal{N}}(\boldsymbol{v})$ of the variables in $\mathbb{V}$, conditioned such that $p_{\mathcal{N}}(\boldsymbol{v})$ equals the distribution of the samples in $\mathbb{T}$. Of course we never have access to the true distribution of the samples in $\mathbb{T}$ directly, which is why we have to resort to approximations. To initialize the recursion we call LearnSPN with the set containing all training samples $\mathbb{T}_0 = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and the set of all considered random variables $\mathbb{V}_{\mathbf{x}}$. The returned node $\mathcal{R}$ is the root of our learned SPT and the distribution it represents $p_{\mathcal{R}}(\boldsymbol{x})$ should, by our assumptions, approximate the true distribution $p(\boldsymbol{x})$.

Let us now inspect the different parts of algorithm 4.1. At the beginning we check if only one random variable is in $\mathbb{V}$ and if this is the case, we create a univariate node and estimate its distribution. This corresponds to lines 4 to 8 in algorithm 4.1. Some possible distribution estimators for this purpose are discussed in Section 4.3. It is possible to use a different distribution estimator for each random variable, which makes sense, because some estimators work better for discrete random variables, while others are intended for continuous random variables. The choice which distribution estimator to use for a given random variable is an important one and it is a major way to incorporate prior assumptions into the learning procedure.

If there are multiple random variables in $\mathbb{V}$ we continue. On line 9 we decide whether further clustering of the samples makes sense. If not, we assume that all variables in $\mathbb{V}$ are independent from each other and create a product node. To decide this, we first check if there are less than N_{M} samples in $\mathbb{T}$, where N_{M} is the minimum number of samples required for clustering. The hyperparameter N_{M} can be chosen arbitrarily, and it is used to increase the average number of samples that are used to learn the univariate distributions. We also check if all samples are equal, because in this case the best clustering outcome is trivially one cluster containing all samples, and nothing was gained. If any of the above two conditions is met, we create a product node of univariate nodes, where every univariate node is based on on the samples in $\mathbb{T}$ and a different variable in $\mathbb{V}$.

If we have multiple random variables in $\mathbb{V}$, equal or more than N_{M} samples in $\mathbb{T}$, and

at least some of these samples are different, then we continue with line 18 in algorithm 4.1. At this point it must be decided, whether a product node, or a sum node should be created. We use the IndependentVariables sub-routine to partition $\mathbb{V}$ into L subsets of random variables $\mathbb{V}_l$. The random variables in one subset $\mathbb{V}_l$ are at least approximately statistically independent from the random variables in the other subsets.
A new product node is created, and the L children of this new product node are learned by recursively calling LearnSPN for every subset $\mathbb{V}_l$.

Typically, statistical dependency is decided *pairwise*, i.e., any variable within subset $\mathbb{V}_l$ is dependent on at least *one* other random variable in $\mathbb{V}_l$. This is in general different from *joint* statistical dependency between *all* the random variables within subset $\mathbb{V}_l$, which would be more desirable. The reason why the pairwise approximation is used is twofold. For one, some dependency measures presented in Section 4.4 do not scale well to measure joint dependency, i.e., mutual information and G-test; and even if joint dependency can be measured efficiently, the count of set comparisons would be exponential in $|\mathbb{V}|$. All the dependency measures in Section 4.4 have in common, that the output is a real value and we have to use a threshold T_{D} to decide whether two variables are dependent or not. Given a set of samples $\mathbb{T}$ and a dependency measure D, we say that two random variables v_1 and v_2 are approximately statistically dependent, if $\mathrm{D}(\mathsf{v}_1, \mathsf{v}_2, \mathbb{T}) \geq T_{\mathrm{D}}$.

If there are no independent subsets of random variables, i.e., $L = 1$, we use a clustering algorithm to partition the sample set $\mathbb{T}$ into K clusters $\mathbb{T}_k$. A new sum node is created and for every cluster $\mathbb{T}_k$ a new child is learned by calling LearnSPN recursively. The weight of a new child $\mathcal{C}$ is given by the fraction of samples in the cluster $\mathbb{T}_k$ compared to the whole sample set $\mathbb{T}$, i.e., $w_{\mathcal{C}} = |\mathbb{T}_k|/|\mathbb{T}|$. Two possible clustering algorithms are presented in Section 4.5, but every unsupervised clustering algorithm can be used for this purpose. The requirement is that $K \geq 2$, but typically a value of K needs to be chosen beforehand and a common choice is $K = 2$. The reason for this is that $K = 2$ is rather general and subsequent sum nodes can be compressed after learning, to form a sum node with more than two children.

LearnSPN builds on the assumption that samples within a cluster show similar dependencies, and that samples of fewer random variables are easier to cluster. At the end, the outcome is comparable to fig. 3.7b; differently sized clusters, with locally independent random variables. Going back to the latent variable model, we can interpret the latent variable as assignment to a specific cluster, i.e., $h = k$. Every leaf is conditioned on the latent variables of the sum nodes on the path connecting the leaf with the root.

Algorithm 4.1 LearnSPN

Inputs: Set of training samples $\mathbb{T}$ and set of variables $\mathbb{V}$
Output: A learned node
procedure LearnSPN($\mathbb{T}, \mathbb{V}$)
    if $|\mathbb{V}| = 1$ **then**
        $\mathsf{v} \leftarrow \mathbb{V}$ ▷ v is the only variable in $\mathbb{V}$
        Create univariate node $\mathcal{U}$
        $p_{\mathcal{U}}(v) \coloneqq$ EstimateDistribution($\mathbb{T}, \mathsf{v}$)
        return $\mathcal{U}$
    else if $|\mathbb{T}| < N_{\mathrm{M}}$ **or** AllSamplesEqual($\mathbb{T}$) **then**
        Create product node $\mathcal{P}$ with $\mathbb{C}_{\mathcal{P}} \coloneqq \emptyset$
        for each $\mathsf{v}_i \in \mathbb{V}$ **do**
            Create univariate node $\mathcal{U}_i$
            $p_{\mathcal{U}_i}(v_i) \coloneqq$ EstimateDistribution($\mathbb{T}, \mathsf{v}_i$)
            $\mathbb{C}_{\mathcal{P}} \leftarrow \mathcal{U}_i$ ▷ Add $\mathcal{U}_i$ to $\mathbb{C}_{\mathcal{P}}$
        end for
        return $\mathcal{P}$
    else
        $\{\mathbb{V}_l\}_{1:L} \coloneqq$ IndependentVariables($\mathbb{T}, \mathbb{V}$)
        if $|\{\mathbb{V}_l\}_{1:L}| > 1$ **then**
            Create product node $\mathcal{P}$ with $\mathbb{C}_{\mathcal{P}} \coloneqq \emptyset$
            for each $\mathbb{V}_l \in \{\mathbb{V}_l\}_{1:L}$ **do**
                $\mathcal{C}_l \coloneqq$ LearnSPN($\mathbb{T}, \mathbb{V}_l$) ▷ Recursively learn child $\mathcal{C}_l$
                $\mathbb{C}_{\mathcal{P}} \leftarrow \mathcal{C}_l$ ▷ Add $\mathcal{C}_l$ to $\mathbb{C}_{\mathcal{P}}$
            end for
            return $\mathcal{P}$
        else
            $\{\mathbb{T}_k\}_{1:K} \coloneqq$ ClusterSamples($\mathbb{T}, \mathbb{V}$)
            Create sum node $\mathcal{S}$ with $\mathbb{C}_{\mathcal{S}} \coloneqq \emptyset$
            for each $\mathbb{T}_k \in \{\mathbb{T}_k\}_{1:K}$ **do**
                $\mathcal{C}_k \coloneqq$ LearnSPN($\mathbb{T}_k, \mathbb{V}$) ▷ Recursively learn child $\mathcal{C}_k$
                $\mathbb{C}_{\mathcal{S}} \leftarrow \mathcal{C}_k$ ▷ Add $\mathcal{C}_k$ to $\mathbb{C}_{\mathcal{S}}$
                $w_{\mathcal{C}_k} \coloneqq \frac{|\mathbb{T}_k|}{|\mathbb{T}|}$ ▷ Sum weight of child $\mathcal{C}_k$
            end for
            return $\mathcal{S}$
        end if
    end if
end procedure

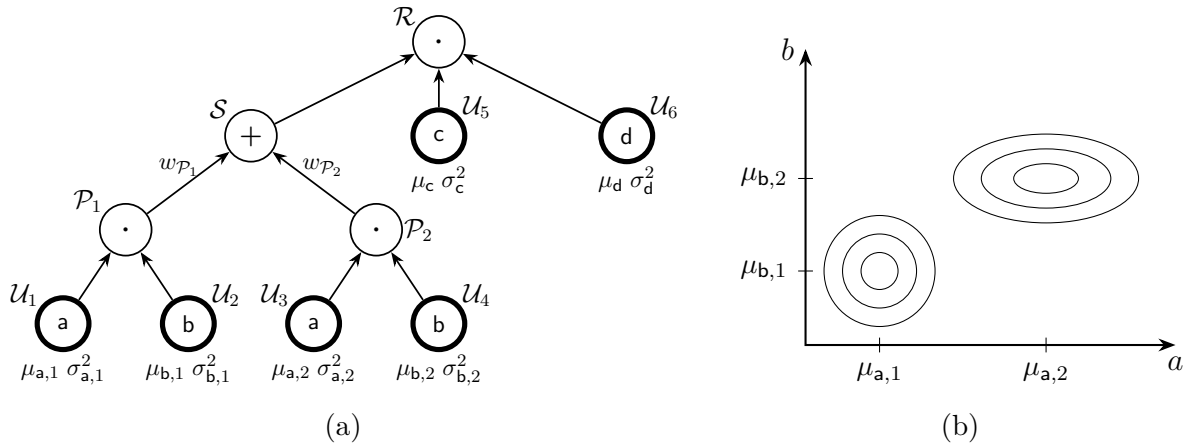


Figure 4.1: (a) An exemplary SPN with only Gaussian leaves. Every leaf is defined by its variable and the associated mean and variance. (b) An illustration of the marginal distribution $p(a, b)$.

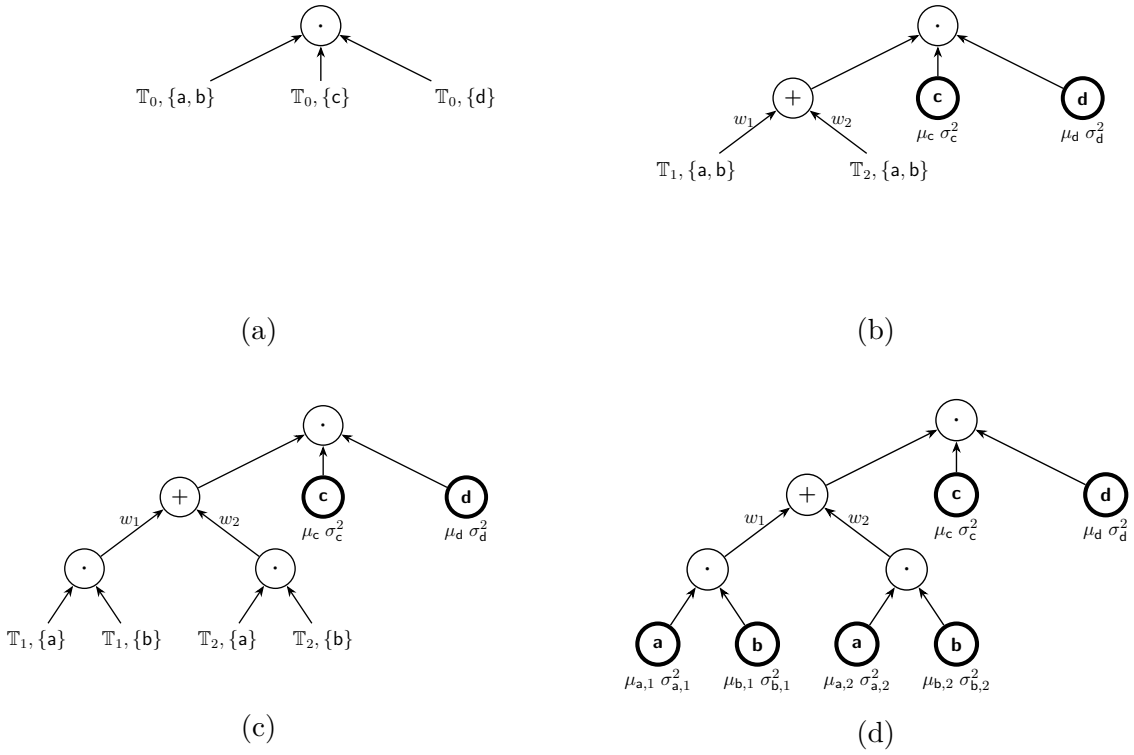


Figure 4.2: Illustration of how LearnSPN learns the structure of an SPT.

4.1.2 Example

To illustrate how LearnSPN learns an SPT, we will discuss a simple example. We want to learn the SPN from fig. 4.1, and we assume that the four random variables a, b, c, d are distributed as in fig. 4.1. This means that c and d are independent and Gaussian, and

that a and b have a jointly Gaussian mixture structure with two mixture components. We further assume that we have a set of samples $\mathbb{T}_0$ from the joint distribution $p(a, b, c, d)$. Throughout this example we will assume ideal hyperparameters.

We learn the root node by initializing algorithm 4.1 with $\mathbb{T}_0$ and $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ as inputs. Since we consider more than one variable and assume that we have more than N_{M} samples that are not all equal, we have to find sets of approximately independent variables. The random variables c and d are independent and the used dependency measure should reflect this. The output of IndependentVariables should therefore be $\{\{\mathsf{a}, \mathsf{b}\}, \{\mathsf{c}\}, \{\mathsf{d}\}\}$. This means that we have to learn a product node with three children. The current state we are in can be seen in fig. 4.2a.

Because the root has three children, algorithm 4.1 is called three times recursively. Considering the recursive calls with only one random variable, i.e., with c and d, we immediately see that the condition on line 4 is true, which means that we learn a univariate leaf for c and d respectively. Since we assume that c and d have a Gaussian distribution, we can use the sample mean eq. (4.9) and sample variance eq. (4.10) to estimate the univariate distributions. The third call to algorithm 4.1, with $\mathbb{T}_0$ and $\{\mathsf{a}, \mathsf{b}\}$ as inputs, is a little bit more difficult. We follow the different decisions through algorithm 4.1 again, and this time IndependentVariables should return just $\{\{\mathsf{a}, \mathsf{b}\}\}$, i.e., no independent sets of variables could be found, which means that we resort to clustering. A good clustering algorithm will now partition the samples from $\mathbb{T}_0$ into two subsets, $\mathbb{T}_1$ and $\mathbb{T}_2$, containing the samples from mixture component one and component two respectively. This results in a sum node with two children and weights $w_1 = |\mathbb{T}_1|/|\mathbb{T}_0|$ and $w_2 = |\mathbb{T}_2|/|\mathbb{T}_0|$. We can see this in fig. 4.2b.

Conditioned on a cluster, a and b are independent which leads to two product nodes again, and the learned SPT at this stage is shown in fig. 4.2c. At this point we need to learn the conditional univariate nodes for a and b for each cluster. The final SPT can be seen in fig. 4.2d, and if we compare it with fig. 4.1, we see that the correct SPT was learned.

One final remark. The procedure described here does not correspond to algorithm 4.1 directly, because we ignored the recursive nature of the algorithm, i.e., the root would actually be the last node that is finished. It is however not advisable to implement LearnSPN as a recursive function, but rather based on a loop with a queue. If a queue is used to avoid recursive function calls, then the procedure described here would be the actual behavior of the program.

4.2 Online Structure Learning

In this section an algorithm is presented that builds on the online structure learning algorithm presented in [12]. The algorithm in [12] uses multivariate Gaussians as leaf distributions, and the correlation coefficient (Section 4.4.2) to measure the dependency between two variables. Here we extend this algorithm to arbitrary leaf distributions and dependency measures. We do not use multivariate leaves however, because this would make it generally impossible to exactly compute marginal distributions, and in turn most inference scenarios would not be available in their exact versions.

We will refer to algorithms 4.2 and 4.3 together as online structure learning (OSL). OSL is a heuristic approach to structure learning and the reason for this is, that the procedure presented in algorithm 4.3 has no theoretical foundation, but rather an intuition behind it. In practice however, OSL learns SPTs with size, depth and performance, comparable to LearnSPN.

In order to use OSL, we need to adapt our model, or more specifically the nodes. Every node $\mathcal{N}$ needs to keep track of the count of samples $N_{\mathcal{N}}$, that were used to train this node. Additionally, every product node $\mathcal{P}$ has a *node dependency measure* $\mathrm{D}_{\mathcal{P}}$ associated with it. This node dependency measure is learned from data in an online manner and it is used to quantify the dependency between two of the product node's children. For a given dependency measure D, the dependency between two child nodes $\mathcal{C}_1$ and $\mathcal{C}_2$ is given by

$$\mathrm{D}_{\mathcal{P}}(\mathcal{C}_1, \mathcal{C}_2) \coloneqq \max_{\mathsf{v}_1 \in \mathbb{S}_{\mathcal{C}_1}} \max_{\mathsf{v}_2 \in \mathbb{S}_{\mathcal{C}_2}} \mathrm{D}(\mathsf{v}_1, \mathsf{v}_2), \tag{4.1}$$

i.e., the maximum dependency between any two random variables from the children's scope. Finally, it is required that the univariate distribution estimators can be learned in an online manner as well.

Since OSL is an online learning algorithm, it does not require all training samples at once, which makes it possible to learn from more samples than fit into memory. The complete sample set is partitioned into smaller subsets (minibatches), and algorithm 4.2 is invoked for every such subset sequentially. Before using OSL for the first minibatch however, a root node $\mathcal{R}$ needs to be initialized. First, a product node is created that will serve as the root. For every random variable $\mathsf{x}_n \in \mathbb{V}_{\mathbf{x}}$ a univariate node is created, and all univariate node together form the children of the root product node. We now have a root node with $\mathbb{S}_{\mathcal{R}} = \mathbb{V}_{\mathbf{x}}$ and $N_{\mathcal{R}} = 0$ and we can start learning with OSL.

An OSL update is initialized by using algorithm 4.2 with the root node and a new minibatch as inputs. With every minibatch the sum weights, dependency measures and univariate distribution estimates are updated using algorithm 4.2. After a few iterations the univariate nodes start to learn their respective marginal distributions and the dependencies

between the variables start to emerge. Based on a dependency threshold T_{SU} and a minimum number of required samples N_{SU}, a *structure update* (algorithm 4.3) is triggered. A mixture is created with the dependent children in one component and a new product node with unlearned univariate leaves in another component. The hope is that this new component can now learn a new aspect of the distribution. Over time the tree grows and ever more specialized sub-trees are learned.

4.2.1 Parameter update

Every OSL iteration consists of a parameter update and structure updates if necessary. We will now discuss the parameter update procedure shown in algorithm 4.2, which recursively traverses the SPT. Every node $\mathcal{N}$ is updated based on a set of samples $\mathbb{T}$, which can be either the complete minibatch, or a sub-set of it. The first step is to increment the sample counter of the node $N_{\mathcal{N}}$ by the number of samples in $\mathbb{T}$. The sample count is required to compute the sum node weights and check whether a structure update is allowed or not. After this, the parameter update depends on the type of node $\mathcal{N}$ is.

In case $\mathcal{N}$ is a univariate node, the distribution estimator is updated using the samples in $\mathbb{T}$.

If $\mathcal{N}$ is a sum node, then the sample set is partitioned into sub-sets $\mathbb{T}_{\mathcal{C}}$. A sample $\boldsymbol{x}_i$ is put in $\mathbb{T}_{\mathcal{C}}$, if the likelihood $p_{\mathcal{C}}(\boldsymbol{x}_i)$ is maximum among the children of the sum node, i.e., if

$$\hat{\mathcal{C}} = \operatorname*{argmax}_{\mathcal{C} \in \mathbb{C}_{\mathcal{N}}} p_{\mathcal{C}}(\boldsymbol{x}_i), \tag{4.2}$$

then $\boldsymbol{x}_i$ is put in $\mathbb{T}_{\hat{\mathcal{C}}}$. Now every child of the sum node is recursively updated with the samples assigned to it, and the weights are adjusted based on the new child sample counts, i.e., $w_{\mathcal{C}} = N_{\mathcal{C}}/N_{\mathcal{N}}$. The reasoning behind sample partitioning is that we cannot use a clustering algorithm with OSL directly. LearnSPN uses a clustering algorithm to group similar samples together and mixtures are created based on these clusters. With OSL we create sum nodes with structure updates, but they still represent mixtures and we update the parameters of each sum node child only with samples that likely originated from the child's mixture component.

Finally, if $\mathcal{N}$ is a product node, the first step is to update the parameters of all children. If the product node is marked as *finished*, or has only one child, then the parameter update is done. The algorithm allows that a product node can lose all but one of its children after several structure updates. These product nodes with only one child can be replaced by their child either at at specific points during training, or after training is finished. Product nodes might be marked as finished during a structure update, which means that further structure updates make no sense for this product node. If the product node is

not marked as finished and has multiple children, then the node dependency measure $\mathrm{D}_{\mathcal{N}}$ associated with the product node is updated. Now it is checked whether $N_{\mathcal{N}} \geq N_{\mathrm{SU}}$, i.e., if the product node was learned with the minimum number of samples required for a structure update. The hyperparameter N_{SU} is used to make sure that the learning of $\mathrm{D}_{\mathcal{N}}$ has converged such that false positives are avoided. If $N_{\mathcal{N}} \geq N_{\mathrm{SU}}$, we find the two children $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ with the highest node dependency, i.e.,

$$\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2 = \underset{\mathcal{C}_1, \mathcal{C}_2 \in \mathbb{C}_{\mathcal{N}}}{\operatorname{argmax}} \mathrm{D}_{\mathcal{N}}(\mathcal{C}_1, \mathcal{C}_2). \tag{4.3}$$

If $\mathrm{D}_{\mathcal{N}}(\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2) > T_{\mathrm{SU}}$, then a structure update is triggered. Similar to LearnSPN, T_{SU} is a dependency threshold used to determine when a structure update is required.

4.2.2 Structure update

The structure update procedure algorithm 4.3 takes three inputs, a product node $\mathcal{P}$ and two approximately dependent children $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$.

First, the two children are removed from the product node and a new product node $\mathcal{P}_1$, with $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ as its children, is created. The sample count of $\mathcal{P}_1$, $N_{\mathcal{P}_1}$, is set to the sample count of $\mathcal{P}$, which in turn is equal to the sample counts of $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$. Since we already know that $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ are dependent, further structure updates of $\mathcal{P}_1$ would make no sense and the new product node $\mathcal{P}_1$ is marked as finished.

Secondly, a new product node $\mathcal{P}_2$ is created, and its children are new univariate nodes $\mathcal{U}_i$, one for every random variable in the joint scope of $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$. The sample counts of the newly created nodes are set to zero, since no samples were used to train the new product node and its univariate children so far. This is very similar to the creation of the root node, and the intention is the same, i.e., to make "space", to learn new aspects of the distribution.

Finally, a sum node $\mathcal{S}$ with children $\mathcal{P}_1$ and $\mathcal{P}_2$ is created. We set $N_{\mathcal{S}}$ equal to $N_{\mathcal{P}}$ and the sum weights are one and zero for $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, which is consistent with the sample counts of $\mathcal{P}_1$ and $\mathcal{P}_2$. The new sum node is added to the children of $\mathcal{P}$ and now replaces the previously removed $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$.

The intention behind the structure update is as follows. We know that a part of the data seen so far indicates that $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ are dependent, but we assume that there might be a different data cluster with different behavior. So a mixture is created with the already learned $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ as one component, and "fresh model space" as the other. The overall distribution of $\mathcal{P}$ is still the same right after the update, since the already known mixture component has weight one.

Algorithm 4.2 Online Structure Learning: Parameter Update

Inputs: Current node $\mathcal{N}$ and set of samples $\mathbb{T} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_I\}$
procedure ParameterUpdate($\mathcal{N}, \mathbb{T}$)
    $N_{\mathcal{N}} \coloneqq N_{\mathcal{N}} + |\mathbb{T}|$
    if $\mathcal{N}$ **is** *Univariate node* **then**
        UpdateUnivariate($\mathcal{N}, \mathbb{T}$)
    else if $\mathcal{N}$ **is** *Sum node* **then**
        $\mathbb{T}_{\mathcal{C}} \coloneqq \emptyset$ for each $\mathcal{C} \in \mathbb{C}_{\mathcal{N}}$
        for each $\boldsymbol{x}_i \in \mathbb{T}$ **do** ▷ Assign the samples $\boldsymbol{x}_i$
            $\hat{\mathcal{C}} \coloneqq \operatorname{argmax}_{\mathcal{C} \in \mathbb{C}_{\mathcal{N}}} p_{\mathcal{C}}(\boldsymbol{x}_i)$ ▷ to the children with
            $\mathbb{T}_{\hat{\mathcal{C}}} \leftarrow \boldsymbol{x}_i$ ▷ the highest likelihood
        end for
        for each $\mathcal{C} \in \mathbb{C}_{\mathcal{N}}$ **do**
            ParameterUpdate($\mathcal{C}, \mathbb{T}_{\mathcal{C}}$) ▷ Recursive update with sub-set $\mathbb{T}_{\mathcal{C}}$
            $w_{\mathcal{C}} \coloneqq \frac{N_{\mathcal{C}}}{N_{\mathcal{N}}}$
        end for
    else if $\mathcal{N}$ **is** *Product node* **then**
        for each $\mathcal{C} \in \mathbb{C}_{\mathcal{N}}$ **do**
            ParameterUpdate($\mathcal{C}, \mathbb{T}$) ▷ Recursive update of all children
        end for
        if $|\mathbb{C}_{\mathcal{N}}| > 1$ **and** $\mathcal{N}$ **is not** *Finished* **then**
            UpdateDependencyMeasure($\mathrm{D}_{\mathcal{N}}, \mathbb{T}$)
            if $N_{\mathcal{N}} \geq N_{\mathrm{SU}}$ **then**
                $\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2 = \operatorname{argmax}_{\mathcal{C}_1, \mathcal{C}_2 \in \mathbb{C}_{\mathcal{N}}} \mathrm{D}_{\mathcal{N}}(\mathcal{C}_1, \mathcal{C}_2)$ ▷ Find the two most dependent children
                if $\mathrm{D}_{\mathcal{N}}(\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2) > T_{\mathrm{SU}}$ **then**
                    StructureUpdate($\mathcal{N}, \hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2$)
                end if
            end if
        end if
    end if
end procedure

There is however one practical issue. During the parameter update, a sum node assigns samples to its children based on the likelihood. A sum node child thus needs to be able to compute a likelihood in order to get samples assigned to it, but all the univariate nodes of $\mathcal{P}_2$ are unlearned so far. We want that $\mathcal{P}_2$ receives samples that are unlikely for $\mathcal{P}_1$, because we assume that these samples are from a yet unlearned mixture component. The

solution is to assign every distribution $p_{\mathcal{U}_i}(v_i)$ of the new univariate nodes a constant value P_I, which is replaced once the distribution estimators are trained with samples. This has the effect that the new $\mathcal{P}_2$ has $p_\mathcal{P}(\boldsymbol{x}) = |\mathbb{C}_{\mathcal{P}_2}|P_\text{I}$ for every sample $\boldsymbol{x}$, and this is a threshold to initially detect samples that are not of the other mixture component.

Algorithm 4.3 Online Structure Learning: Structure Update

Inputs: Product node $\mathcal{P}$ and two children of the product node, $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$
procedure StructureUpdate($\mathcal{P}, \hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2$)
    $\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2 \leftarrow \mathbb{C}_\mathcal{P}$ ▷ Remove $\hat{\mathcal{C}}_1$ and $\hat{\mathcal{C}}_2$ from $\mathbb{C}_\mathcal{P}$
    Create product node $\mathcal{P}_1$ with $\mathbb{C}_{\mathcal{P}_1} \coloneqq \{\hat{\mathcal{C}}_1, \hat{\mathcal{C}}_2\}$ and $N_{\mathcal{P}_1} \coloneqq N_\mathcal{P}$
    Mark $\mathcal{P}_1$ as *Finished*
    Create product node $\mathcal{P}_2$ with $\mathbb{C}_{\mathcal{P}_2} \coloneqq \emptyset$ and $N_{\mathcal{P}_2} \coloneqq 0$
    for each $\mathsf{v}_i \in \mathbb{S}_{\hat{\mathcal{C}}_1} \cap \mathbb{S}_{\hat{\mathcal{C}}_2}$ **do**
        Create univariate node $\mathcal{U}_i$ with $N_{\mathcal{U}_i} \coloneqq 0$
        $p_{\mathcal{U}_i}(v_i) \coloneqq P_\text{I}$ ▷ Initialize $p_{\mathcal{U}_i}(v_i)$ to constant P_I
        $\mathbb{C}_{\mathcal{P}_2} \leftarrow \mathcal{U}_i$ ▷ Add $\mathcal{U}_i$ to $\mathbb{C}_{\mathcal{P}_2}$
    end for
    Create sum node $\mathcal{S}$ with $\mathbb{C}_\mathcal{S} \coloneqq \{\mathcal{P}_1, \mathcal{P}_2\}$ and $N_\mathcal{S} \coloneqq N_\mathcal{P}$
    $w_{\mathcal{P}_1} \coloneqq 1$
    $w_{\mathcal{P}_2} \coloneqq 0$
    $\mathbb{C}_\mathcal{P} \leftarrow \mathcal{S}$ ▷ Add $\mathcal{S}$ to $\mathbb{C}_\mathcal{P}$
end procedure

4.2.3 Example

To illustrate OSL, we use the same setup from Section 4.1.2, i.e., we want to learn the SPT from fig. 4.1 with the four random variables a, b, c and d. We initialize our SPT with a product node with four children, one univariate node for every random variable. Since we know that our random variables are Gaussians, we use Gaussian distributions for the leaves. This effectively means that we estimate the online sample mean eq. (4.20) and online sample variance eq. (4.21) at every univariate leaf. At the beginning however, all univariate distributions did not receive any training samples and are thus uninitialized, as can be seen in fig. 4.3a.

The individual leaf distributions should converge after some minibatches are learned, however the mean and variance of the leaves of a and b will be those of the whole mixture, indicated by $\tilde{\mu}$ and $\tilde{\sigma}^2$ in fig. 4.3b. At some point, the dependency measure associated with the root product node should detect, that a and b are dependent. After the subsequent

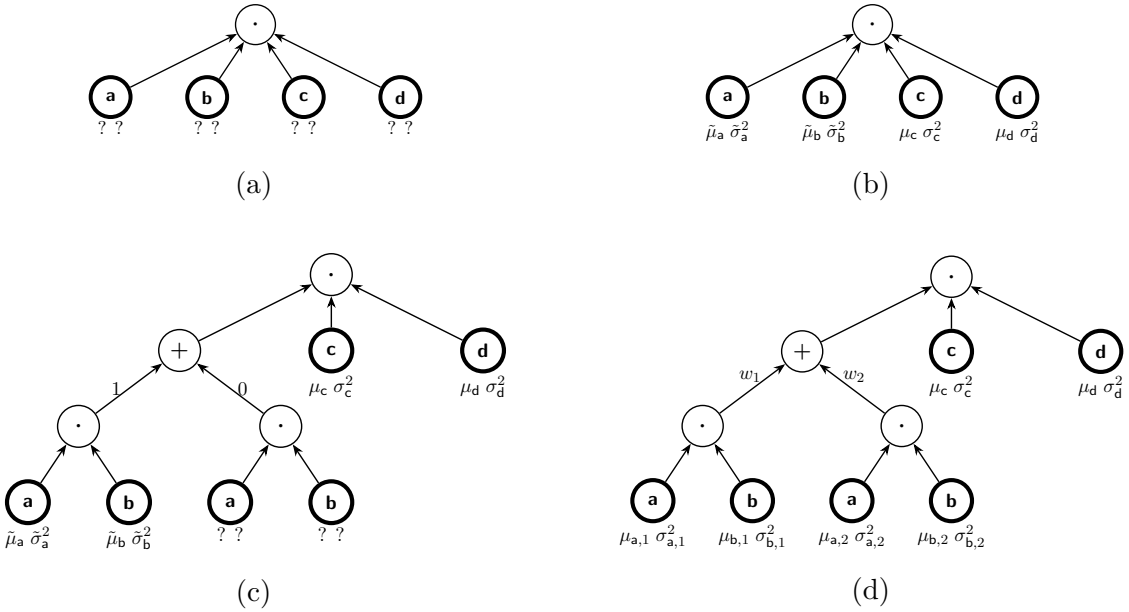


Figure 4.3: Illustration of how OSL learns the structure of an SPT.

structure update the SPT looks like fig. 4.3c. The left child product node of the sum node is marked as finished and no further structure updates are possible. The right child product node however, is uninitialized so far and has yet to receive any training samples.

What will happen now, is that in a future update, depending on P_{I} (line 9 in algorithm 4.3), a sample from either cluster will achieve higher likelihood for the right product node than for the left one. Once this happens, the new univariate leaves get initialized and the sample means of the right product node are now near the cluster the sample was from. It is now likely that all future training samples from this cluster will get assigned to this node and the weights of the sum node will approach their true values. Since a and b are independent given a cluster, no further structure updates will happen and the structure of the SPT in fig. 4.3d will not change any further. We can again compare fig. 4.3d to fig. 4.1 and see that both structures are identical. However, because some of the first few samples got assigned to, what is the wrong node in the end, the weights of the sum node, the sample mean and the sample variance will be off by some amount.

4.3 Univariate Distribution Estimators

Both learning algorithms, LearnSPN and OSL, require univariate distribution estimators. If OSL is used, then the distribution estimator must be capable of online learning as well. Every distribution estimator capable of online learning can be used for offline learning as well, by partitioning the complete data in minibatches. In this section two possible

options for online capable distribution estimators are presented, one for discrete and one for continuous distributions. Both estimators are *non-parametric*, i.e., no assumptions about the real data distribution are made, and both have a *smoothing* parameter that is used for regularization.

4.3.1 Discrete

The discrete distribution estimator presented here, is a straight forward counting of occurrences together with *Laplace smoothing*. Every sample x_n can take on one from the d possible values from set $\mathbb{X}_d$, which is the set of possible outcomes or classes. The count of samples observed of one class $x \in \mathbb{X}_d$ during learning is denoted by N_x. The estimated probability of class x is given by

$$\hat{P}_x = \hat{p}_{\mathsf{x}}(x) \coloneqq \frac{N_x + \alpha}{\sum_{x' \in \mathbb{X}_d} N'_x + \alpha |\mathbb{X}_d|}, \tag{4.4}$$

where $\alpha \geq 0$ is the Laplace smoothing parameter. Laplace smoothing can be interpreted as a prior uniform assumption and the higher α, the higher this assumption is valued compared to the observed samples. The set $\mathbb{X}_d$ can be learned as well, but the estimate will be different for $\alpha > 0$, if not all possible classes are present in the observed samples.

A special case of a discrete distribution is the *Bernoulli distribution*, where the random variable can take on the values zero or one. In case of a Bernoulli distributed random variable x_B, the estimator from eq. (4.4) is

$$\hat{P}_0 = \hat{p}_{\mathsf{x}_\mathrm{B}}(0) = \frac{N_0 + \alpha}{N + 2\alpha}, \tag{4.5}$$

where N_0 is the count of observed zeros and N is the total training sample count. The probability of a one is trivially $\hat{P}_1 = 1 - \hat{P}_0$.

4.3.2 Online KDE

Kernel density estimators (KDEs) are well known non-parametric methods to estimate continuous distributions [26], [36]. The usual setup of a KDE is as follows. Given N training samples x_n, then the estimated distribution is

$$\hat{f}_{\mathsf{x}}(x) = \frac{1}{Nh} \sum_{n=1}^{N} K\left(\frac{x - x_n}{h}\right), \tag{4.6}$$

where $K(\cdot)$ is the *kernel* and h is the *bandwidth*. The kernel must be non-negative and must integrate to one. Typical choices for the kernel are uniform, triangular, Gaussian,

etc. The bandwidth is usually dependent on the samples, and generally it should $h \to 0$ as $N \to \infty$. Here we will use Gaussian kernels, i.e.,

$$\frac{1}{h}K\Big(\frac{x - x_n}{h}\Big) = \frac{1}{h}\mathcal{N}\Big(\frac{x - x_n}{h}; 0, 1\Big) = \mathcal{N}\big(x; x_n, h^2\big). \tag{4.7}$$

We see that for Gaussian kernels, the bandwidth equals the standard deviation of the individual components.

In the following, *Silverman's rule of thumb* [36], a simple and common bandwidth estimator is presented and we will denote it with h_{S}. Assuming that the true distribution $f_{\mathsf{x}}(x)$ and the kernels $K(\cdot)$ are both Gaussians, h_{S} minimizes the *mean integrated squared error*

$$\mathrm{MISE}(h) \coloneqq \mathrm{E}\Big\{\int_{-\infty}^{\infty}\Big(\hat{f}_{\mathsf{x}}(x) - f_{\mathsf{x}}(x)\Big)^2 \,\mathrm{d}x\Big\}. \tag{4.8}$$

Together with the *sample mean*

$$\hat{\mu}_{\mathsf{x}} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{4.9}$$

and the *sample variance*

$$\hat{\sigma}_{\mathsf{x}}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \hat{\mu}_{\mathsf{x}})^2, \tag{4.10}$$

the estimated bandwidth is given by

$$h_{\mathrm{S}} \coloneqq \operatorname*{argmin}_h \mathrm{MISE}(h) = \hat{\sigma}_{\mathsf{x}}\left(\frac{4}{3N}\right)^{\frac{1}{5}}. \tag{4.11}$$

Because h_{S} is optimum for $f_{\mathsf{x}}(x)$ Gaussian, it tends to over-estimate the true bandwidth, resulting in a too smooth $\hat{f}_{\mathsf{x}}(x)$, should $f_{\mathsf{x}}(x)$ not be Gaussian. To avoid this and to allow for tunable regularization, we introduce a *smoothing parameter* α, and the final estimation of the bandwidth is

$$h_\alpha \coloneqq \alpha h_{\mathrm{S}}. \tag{4.12}$$

We now have to solve another practical problem. While the distribution estimator based on eq. (4.6) can be easily learned in an online manner by just adding additional components to the sum, this requires that all past samples need to be stored. A way to counteract this problem, is to reduce the number of components by merging similar components together. An algorithm for reducing the number of components of a Gaussian mixture by merging is presented in [34], and here we adapt it to work with KDE. In order to accommodate merged components, we need to change eq. (4.6) to

$$\hat{f}_{\mathsf{x}}(x) = \sum_{i=1}^{N_{\mathrm{C}}} w_i \mathcal{N}\big(x; \mu_i, \sigma_i^2 h_\alpha^2\big), \tag{4.13}$$

where N_C is the maximum component count and w_i, μ_i, σ_i^2 are the parameters of the ith mixture component.

If an update occurs with N_N new samples x_n, a new component with variance one is added to eq. (4.13) and we initialize the merging algorithm with

$$\hat{f}_\mathsf{x}(x) = \sum_{i=1}^{N_\mathrm{C}} w_i \mathcal{N}\Big(x; \mu_i, \sigma_i^2 h_\alpha^2\Big) + \sum_{n=1}^{N_\mathrm{N}} w_{N_\mathrm{C}+n}\, \mathcal{N}\Big(x; \mu_{N_\mathrm{C}+n}, \sigma_{N_\mathrm{C}+n}^2\, h_\alpha^2\Big). \tag{4.14}$$

The overall sample count N is increased by N_N and the other parameters are set as follows

$$\begin{aligned} w_i &= \frac{N - N_\mathrm{N}}{N} w_{i,old} & w_{N_\mathrm{C}+n} &= \frac{1}{N} \\ \mu_i &= \mu_{i,old} & \mu_{N_\mathrm{C}+n} &= x_n \\ \sigma_{i,old}^2 &= \sigma_i^2 & \sigma_{N_\mathrm{C}+n}^2 &= 1. \end{aligned} \tag{4.15}$$

After this initialization, we have a new mixture with $N_\mathrm{C} + N_\mathrm{N}$ components. The mixture components are now iteratively merged, two components at a time, until only N_C components are left. If components i and j are merged, then the new component has the moment preserving parameters

$$\begin{aligned} w_{ij} &= w_i + w_j \\ \mu_{ij} &= \frac{w_i}{w_i + w_j}[w_i\mu_i + w_j\mu_j] \\ \sigma_{ij}^2 &= \frac{w_i}{w_i + w_j}\left[w_i\sigma_i^2 + w_j\sigma_j^2 + w_i w_j \frac{(\mu_i - \mu_j)^2}{h_\alpha^2}\right]. \end{aligned} \tag{4.16}$$

The components that are merged are selected based on the *Kullback-Leibler divergence* between the mixture before merging $\hat{f}_\mathsf{x}(x)$ and the mixture after merging the two components i and j, $\hat{f}_\mathsf{x}^{(ij)}(x)$, i.e.,

$$\mathrm{D}_\mathrm{KL}\Big(\hat{f}_\mathsf{x}(x) \parallel \hat{f}_\mathsf{x}^{(ij)}(x)\Big) = \int_{-\infty}^{\infty} \hat{f}_\mathsf{x}(x) \ln \frac{\hat{f}_\mathsf{x}(x)}{\hat{f}_\mathsf{x}^{(ij)}(x)}\, \mathrm{d}x. \tag{4.17}$$

This is however not exactly computable, but for this special case there exists an upper bound, i.e.,

$$\mathrm{D}_\mathrm{KL}\Big(\hat{f}_\mathsf{x}(x) \parallel \hat{f}_\mathsf{x}^{(ij)}(x)\Big) \leq B_{ij} = \frac{1}{2}\Big[(w_i + w_j)\ln\sigma_{ij}^2 - w_i \ln\sigma_i^2 - w_j \ln\sigma_j^2\Big]. \tag{4.18}$$

We now compute B_{ij} for all potential mergers, and merge those two components that minimize B_{ij}.

Now to summarize our online KDE. We use eq. (4.13) to model our distribution estimator with a maximum of N_C components and smoothing α. To update the estimator

with N_N new samples x_n, we update the sample mean and sample variance using

$$N = N_\mathrm{old} + N_\mathrm{N} \tag{4.19}$$

$$\hat{\mu}_\mathrm{x} = \frac{N - N_\mathrm{N}}{N}\hat{\mu}_\mathrm{x,old} + \frac{1}{N}\sum_{n=1}^{N_\mathrm{N}} x_n \tag{4.20}$$

$$\hat{\sigma}_\mathrm{x}^2 = \frac{N - N_\mathrm{N}}{N}\left(\hat{\sigma}_\mathrm{x,old}^2 + \hat{\mu}_\mathrm{x,old}^2\right) - \hat{\mu}_\mathrm{x}^2 + \frac{1}{N}\sum_{n=1}^{N_\mathrm{N}} x_n^2. \tag{4.21}$$

The bandwidth is updated accordingly, i.e.,

$$h_\alpha = \alpha\hat{\sigma}_\mathrm{x}\left(\frac{4}{3N}\right)^{\frac{1}{5}}. \tag{4.22}$$

Then we initialize the iterative merging using eqs. (4.14) and (4.15). We continue merging those two components minimizing eq. (4.18), using eq. (4.16), until there are only N_C components left.

4.4 Dependency Measures

This section discusses some dependency measures usable for both, LearnSPN and OSL. The first two, *mutual information* and the *G-test* [21], are closely related and in here only used for discrete data. The *correlation coefficient* and its generalization to higher dimensions, the *canonical correlation analysis* (CCA) [11], are simple dependency measures for both discrete, continuous or mixed data. Finally the *randomized dependence coefficient* (RDC) [18] is an extension to the CCA with a better ability to detect dependencies. All dependency measures presented here have in common that a higher value indicates a higher dependency between the random variables in question. The values of the mutual information and G-test are positive but in general unbounded, while the correlation coefficient, CCA and RDC deliver results from the interval $[0, 1]$.

4.4.1 Mutual information and G-test

One of the theoretically most appealing measures for dependency is the mutual information, as it is defined as the Kullback-Leibler divergence between the joint and marginal distributions of random variables. While the mutual information is not restricted to discrete random variables, we will only consider the discrete mutual information. The reason for this is that in practice the mutual information is not exactly computable in general, and estimations need to be used and the simple approximation presented here only works for discrete random variables.

For two discrete random variables $\mathsf{x} \in \mathbb{X}$ and $\mathsf{y} \in \mathbb{Y}$ the mutual information is given by

$$\mathrm{I}(\mathsf{x};\mathsf{y}) = \mathrm{D}_{\mathrm{KL}}(p(x,y) \parallel p_{\mathsf{x}}(x)p_{\mathsf{y}}(y)) = \sum_{x\in\mathbb{X}}\sum_{y\in\mathbb{Y}} p(x,y)\ln\frac{p(x,y)}{p_{\mathsf{x}}(x)p_{\mathsf{y}}(y)}. \tag{4.23}$$

The problem is, that the *true* distributions $p_{\mathsf{x}}(x)$, $p_{\mathsf{y}}(y)$ and $p(x,y)$ are not available and need to be approximated. We will use the distribution estimator from Section 4.3.1 without smoothing and extend it to two dimensions for $p(x,y)$. Given N samples $(x_n, y_n)^\top$ from $p(x,y)$, then we approximate $p_{\mathsf{x}}(x)$, $p_{\mathsf{y}}(y)$ and $p(x,y)$ by

$$\hat{p}_{\mathsf{x}}(x) = \frac{N_x}{N} \tag{4.24}$$

$$\hat{p}_{\mathsf{y}}(y) = \frac{N_y}{N} \tag{4.25}$$

$$\hat{p}(x,y) = \frac{N_{x,y}}{N} \tag{4.26}$$

respectively, where $N_{x,y}$ is the count of samples where x and y jointly appeared. Our approximation of the mutual information thus becomes

$$\hat{\mathrm{I}}(\mathsf{x};\mathsf{y}) = \sum_{x\in\mathbb{X}}\sum_{y\in\mathbb{Y}} \hat{p}(x,y)\ln\frac{\hat{p}(x,y)}{\hat{p}_{\mathsf{x}}(x)\hat{p}_{\mathsf{y}}(y)} = \frac{1}{N}\sum_{x\in\mathbb{X}}\sum_{y\in\mathbb{Y}} N_{x,y}\ln\frac{N_{x,y}N}{N_x N_y}. \tag{4.27}$$

The *G-test of goodness of fit* [21], or simply G-test, was developed as a hypothesis test, but it is a common dependency measure when learning SPNs. Because the G-test is closely related to the approximation of the mutual information from eq. (4.27), we can simply write it as

$$G = 2N\hat{\mathrm{I}}(\mathsf{x};\mathsf{y}) = 2\sum_{x\in\mathbb{X}}\sum_{y\in\mathbb{Y}} N_{x,y}\ln\frac{N_{x,y}N}{N_x N_y}. \tag{4.28}$$

The advantage of the G-test, compared to the mutual information when learning SPNs, is that the count of samples is factored into the result, and this avoids *false positives* at low sample counts to a certain extent.

4.4.2 Correlation coefficient and CCA

In this section two methods are presented to measure the *correlation* between two random variables or vectors. While correlation is in general different from dependence, in practice it can deliver a good estimation for the dependency of two random variables or vectors. The correlation coefficient of two random variables x and y is given by

$$\rho(\mathsf{x},\mathsf{y}) := \frac{\mathrm{E}\{(\mathsf{x}-\mu_{\mathsf{x}})(\mathsf{y}-\mu_{\mathsf{y}})\}}{\sqrt{\sigma_{\mathsf{x}}^2\sigma_{\mathsf{y}}^2}}. \tag{4.29}$$

The CCA [11] can be seen as an extension to this for higher dimensions, however the actual goal of CCA is something different. For two random vectors $\mathbf{x}$ and $\mathbf{y}$, the CCA seeks the two vectors $\hat{\boldsymbol{a}}$, $\hat{\boldsymbol{b}}$ which maximize the correlation between $\hat{\boldsymbol{a}}^\top \mathbf{x}$ and $\hat{\boldsymbol{b}}^\top \mathbf{y}$, i.e.,

$$\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}} \coloneqq \underset{\boldsymbol{a},\boldsymbol{b}}{\operatorname{argmax}} \rho(\boldsymbol{a}^\top \mathbf{x}, \boldsymbol{b}^\top \mathbf{y}). \tag{4.30}$$

Here we are however not interested in $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{b}}$ directly, but we can use them to define the *maximum canonical correlation coefficient*

$$\rho(\mathbf{x}, \mathbf{y}) \coloneqq \max_{\boldsymbol{a},\boldsymbol{b}} \rho(\boldsymbol{a}^\top \mathbf{x}, \boldsymbol{b}^\top \mathbf{y}) = \rho(\hat{\boldsymbol{a}}^\top \mathbf{x}, \hat{\boldsymbol{b}}^\top \mathbf{y}). \tag{4.31}$$

In order to compute $\rho(\mathbf{x}, \mathbf{y})$, without finding $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{b}}$ first, we define

$$\boldsymbol{K} \coloneqq \boldsymbol{C}_{\mathbf{x}}^{-\frac{1}{2}} \boldsymbol{C}_{\mathbf{xy}} \boldsymbol{C}_{\mathbf{y}}^{-\frac{1}{2}} \tag{4.32}$$

and then perform a *singular value decomposition* on $\boldsymbol{K}$, i.e.,

$$\boldsymbol{K} = \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{V}^H, \tag{4.33}$$

where $\boldsymbol{\Sigma} = \operatorname{diag}(\sqrt{\lambda_1}, \sqrt{\lambda_2}, \ldots)$ is a diagonal matrix with the square roots of the eigenvalues of $\boldsymbol{K}$ on its main diagonal. The maximum canonical correlation coefficient is then given as

$$\rho(\mathbf{x}, \mathbf{y}) = \max_i \sqrt{\lambda_i}, \tag{4.34}$$

i.e., as the square root of the maximum eigenvalue of $\boldsymbol{K}$. In the case of one dimensional random vectors, the CCA from eqs. (4.32) to (4.34) equals the correlation coefficient from eq. (4.29). The CCA would allow us to estimate the correlation between *groups* of random variables, but as was mentioned above, this would cause a high number of comparisons when learning SPNs.

So far we have not discussed how to compute the CCA (and thus the correlation coefficient) from samples. In the context of learning SPNs, we usually have samples from a random vector $\mathbf{x}$ at any given state of the learning process and we want to make a statement on the dependency of random variables within $\mathbf{x}$. For CCA it is sufficient to have an estimate of the covariance matrix of $\mathbf{x}$, $\hat{\boldsymbol{C}}_{\mathbf{x}}$, and we can generalize eq. (4.20) and eq. (4.21) to higher dimensions to get such an estimate. To update the *sample covariance matrix* with N_N new samples $\boldsymbol{x}_n$ we compute

$$N = N_\mathrm{old} + N_\mathrm{N} \tag{4.35}$$

$$\hat{\boldsymbol{\mu}}_{\mathbf{x}} = \frac{N - N_\mathrm{N}}{N} \hat{\boldsymbol{\mu}}_{\mathbf{x},\mathrm{old}} + \frac{1}{N} \sum_{n=1}^{N_\mathrm{N}} \boldsymbol{x}_n \tag{4.36}$$

$$\hat{\boldsymbol{C}}_{\mathbf{x}} = \frac{N - N_{\mathrm{N}}}{N}\left(\hat{\boldsymbol{C}}_{\mathbf{x},\mathrm{old}} + \hat{\boldsymbol{\mu}}_{\mathbf{x},\mathrm{old}}\hat{\boldsymbol{\mu}}_{\mathbf{x},\mathrm{old}}^{\top}\right) - \hat{\boldsymbol{\mu}}_{\mathbf{x}}\hat{\boldsymbol{\mu}}_{\mathbf{x}}^{\top} + \frac{1}{N}\sum_{n=1}^{N_{\mathrm{N}}} \boldsymbol{x}_n \boldsymbol{x}_n^{\top}. \tag{4.37}$$

Based on the sample covariance matrix, we can rewrite eq. (4.29) to get an estimate of the correlation coefficient between the ith and jth variable of $\mathbf{x}$, i.e.,

$$\hat{\rho}(\mathsf{x}_i, \mathsf{x}_j) = \frac{(\hat{\boldsymbol{C}}_{\mathbf{x}})_{ij}}{\sqrt{(\hat{\boldsymbol{C}}_{\mathbf{x}})_{ii}(\hat{\boldsymbol{C}}_{\mathbf{x}})_{jj}}}. \tag{4.38}$$

To get an estimate of the maximum canonical correlation coefficient between two groups of random variables within $\mathbf{x}$, we can build the matrix $\boldsymbol{K}$ from eq. (4.32) by using the corresponding entries in $\hat{\boldsymbol{C}}_{\mathbf{x}}$.

4.4.3 Randomized dependence coefficient

The RDC was first used to train SPNs in [24]. The argument for using the RDC is that it allows to efficiently measure the dependency between discrete and continuous random variables. Another advantage of the RDC is that it is a provably [18] good estimation for, not just correlation, but the dependency of two random vectors. In the following an adaptation of the RDC is presented that allows online learning in the context of SPNs.

We have samples from a random vector $\mathbf{x} = (\mathsf{x}_1, \mathsf{x}_2, \ldots)^{\top}$, which are in general provided in an online manner, and we want to estimate the dependency between single random variables of that vector. The overall idea of the RDC is to put every sample through a non-linear, random transformation and then compute the correlation of these transformed samples. The transformation happens in three stages, which we will discuss now with the example of a single sample $\boldsymbol{x}$.

The first stage is, to compute the *copula transformation*

$$v_i = \begin{cases} P_{\mathsf{x}_i}(x_i), & \mathsf{x}_i \text{ is discrete} \\ F_{\mathsf{x}_i}(x_i), & \mathsf{x}_i \text{ is continuous} \end{cases} \tag{4.39}$$

of the values of $\boldsymbol{x}$, where $P_{\mathsf{x}_i}(\cdot)$ is the *cumulative distribution function* (CDF) of x_i. The resulting random variables $\mathsf{v}_i = P_{\mathsf{x}_i}(\mathsf{x}_i)$ are continuous and uniformly distributed on the interval $[0, 1]$. Of course the true CDFs of the random variables in $\mathbf{x}$ are not available, but need to be estimated. While there are many different CDF estimators available, we will re-use the distribution estimators from Section 4.3 here. Based on eq. (4.4), for discrete random variables we use

$$\hat{P}_{\mathsf{x}_i}(x_i) = \sum_{x=-\infty}^{x_i} \hat{p}_{\mathsf{x}_i}(x_i) = \frac{\sum_{x=-\infty}^{x_i}(N_x + \alpha)}{\sum_{x' \in \mathbb{X}_d} N'_x + \alpha|\mathbb{X}_d|}, \tag{4.40}$$

which can be efficiently computed if the cumulative sample counts are stored. Similarly for continuous random variables, we re-use the online KDE and based on eq. (4.13) and we get

$$\hat{F}_{\mathsf{x}_i}(x_i) = \int_{x=-\infty}^{x_i} \hat{f}_{\mathsf{x}_i}(x)\,\mathrm{d}x = \sum_{j=1}^{N_\mathrm{C}} \frac{w_j}{2}\left[1 + \operatorname{erf}\left(\frac{x_i - \mu_j}{\sqrt{2\sigma_j^2 h_\alpha^2}}\right)\right], \tag{4.41}$$

where $\operatorname{erf}(\cdot)$ is the well known error function.

At the second stage, for every v_i, J affine transformations

$$v_{ij} = \tilde{w}_{ij} v_i + \tilde{b}_{ij} \tag{4.42}$$

are computed, where J is a parameter of the RDC. The *weights* $\tilde{w}_{ij}$ and *biases* $\tilde{b}_{ij}$ are randomly chosen during the initialization of the RDC and then stay constant during the entire learning procedure. It is reported [18] that,

$$\begin{aligned} \tilde{w}_{ij} &\sim \mathcal{N}\left(0, \tilde{\sigma}^2\right) \\ \tilde{b}_{ij} &\sim \mathcal{U}(0, 2\pi) \end{aligned} \tag{4.43}$$

are good choices the distributions of the weights and biases, where $\tilde{\sigma}^2$ is another parameter of the RDC and $\mathcal{U}(0, 2\pi)$ is the continuous uniform distribution on the interval $[0, 2\pi]$.

Finally the v_{ij} are transformed with L non-linear functions $\tilde{f}_l(\cdot)$, i.e., for every v_{ij} we compute

$$v_{ijl} = \tilde{f}_l(v_{ij}). \tag{4.44}$$

The count of non-linear functions L and the functions $\tilde{f}_l(\cdot)$ themselves, are the final parameters of the RDC. According to [18], good choices are

$$\begin{aligned} \tilde{f}_1(v) &= \sin(v) \\ \tilde{f}_2(v) &= \cos(v). \end{aligned} \tag{4.45}$$

All transformed values v_{ijl} for a given i, are now put into a vector $\boldsymbol{v}_i$, which can be seen as a sample of the random vector $\mathbf{v}_i$. For example, let x_i be a continuous random variable, and we use the choices from eqs. (4.43) and (4.45) with $J = 2$, then we have

$$\boldsymbol{v}_i = \begin{pmatrix} \sin(\tilde{w}_{i1}\hat{F}_{\mathsf{x}_i}(x_i) + \tilde{b}_{i1}) \\ \cos(\tilde{w}_{i1}\hat{F}_{\mathsf{x}_i}(x_i) + \tilde{b}_{i1}) \\ \sin(\tilde{w}_{i2}\hat{F}_{\mathsf{x}_i}(x_i) + \tilde{b}_{i2}) \\ \cos(\tilde{w}_{i2}\hat{F}_{\mathsf{x}_i}(x_i) + \tilde{b}_{i2}) \end{pmatrix}. \tag{4.46}$$

The vectors $\boldsymbol{v}_i$ are now stacked to form a large vector $\boldsymbol{v}$. We now keep track of the sample covariance of $\boldsymbol{v}$ using eqs. (4.35) to (4.37). The value of the RDC between x_i and x_j is now

defined as the estimated maximum canonical correlation coefficient between the random vectors $\mathbf{v}_i$ and $\mathbf{v}_j$, i.e.,

$$\text{rdc}(\mathsf{x}_i, \mathsf{x}_j) \coloneqq \hat{\rho}(\mathbf{v}_i, \mathbf{v}_j). \tag{4.47}$$

4.5 Clustering Algorithms

In this section two common clustering algorithms, k-means and Gaussian mixture model (GMM) expectation maximization (EM) are discussed [3]. Both can be used as part of LearnSPN. Both algorithms share a very similar structure which comes from the fact that k-means can be interpreted as a special case of GMM EM [3]. Both algorithms are iterative and every iteration consists of two phases, the expectation and the maximization phase. At the end of every iteration an objective function is computed and if the change, compared to the last iteration, is below a threshold, the clustering is stopped. The initialization, i.e., the initial cluster centers, play a major role in the final outcome, which is why at the end of the section a widely used initialization technique called *k-means++* [1] is presented. Another straight-forward way to reduce the influence of the initial cluster centers is to repeat the algorithm several times with different initial cluster centers, and select the clustering that achieves the best performance with respect to the objective function.

4.5.1 *k*-means

A very basic clustering algorithm is k-means. It works reasonably well in most scenarios, is numerically stable and due to its simple structure, k-meansis fast, compared to other clustering techniques. The downside is that k-means is highly dependent on its initialization, which makes a good initialization algorithm necessary.

The goal of k-means is to group N samples $\boldsymbol{x}_n$ into K clusters, based on a distance measure $d(\cdot, \cdot)$. A common choice for the distance measure is the squared L_2 norm, i.e., $d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$. Every cluster defined is by its *center* $\boldsymbol{\mu}_k$. These cluster centers are, what the initialization needs to provide. A simple method to get the initial cluster centers is to randomly choose K samples, but k-means++ (Section 4.5.3) usually leads to better results.

Every iteration of k-means starts with a reassignment of the N samples to one of the K clusters. We have an assignment variable β_{nk} that is either one, if sample $\boldsymbol{x}_n$ is assigned to cluster k, or zero if not. A sample is assigned to that cluster, that has the minimum

distance from its center to the sample, i.e.,

$$\beta_{nk} = \begin{cases} 1, & \text{argmin}_{k'} \, d(\boldsymbol{x}_n, \boldsymbol{\mu}_{k'}) = k \\ 0, & \text{else.} \end{cases} \tag{4.48}$$

After this reassignment, the new cluster centers are computed using

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \beta_{nk} \boldsymbol{x}_n}{\sum_{n=1}^{N} \beta_{nk}}. \tag{4.49}$$

The objective function of k-means is usually

$$J = \sum_{k=1}^{K} \sum_{n=1}^{N} \beta_{nk} d(\boldsymbol{x}_n, \boldsymbol{\mu}_k), \tag{4.50}$$

which is sometimes called the *distortion measure* [3], and we want to minimize it. If the change of the distortion measure between two iterations is smaller than a threshold ε, i.e., $|J_{new} - J_{old}| < \varepsilon$, then the clustering is stopped and the final assignments β_{nk} are the result of the clustering algorithm.

4.5.2 Gaussian mixture model EM

Another very popular clustering algorithm is GMM EM [3]. The traditional task of GMM EM is to fit a GMM to a given set of N samples x_n using the EM algorithm, but this can be re-purposed as a clustering algorithm. The GMM considered here consists of K mixture components, i.e.,

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_k, \boldsymbol{C}_k), \tag{4.51}$$

where w_k, $\boldsymbol{\mu}_k$ and $\boldsymbol{C}_k$ are the component weights, means and covariance matrices respectively.

The initial component means are provided using the same techniques applicable for $k-means$, e.g. k-means++. The initial weights are usually set to $w_k = 1/K$ and the covariances are typically set to identity. The E-step consists of recomputing the *responsibilities* γ_{nk}, which are comparable to the β_{nk} of k-means, but instead of *hard* assignments, GMM EM works with *soft* assignments, i.e., $\gamma_{nk} \in [0, 1]$. We compute the responsibilities using

$$\gamma_{nk} = \frac{w_k \mathcal{N}(\boldsymbol{x}_n; \boldsymbol{\mu}_k, \boldsymbol{C}_k)}{p(\boldsymbol{x}_n)} = \frac{w_k \mathcal{N}(\boldsymbol{x}_n; \boldsymbol{\mu}_k, \boldsymbol{C}_k)}{\sum_{k'=1}^{K} w_{k'} \mathcal{N}(\boldsymbol{x}_n; \boldsymbol{\mu}_{k'}, \boldsymbol{C}_{k'})}. \tag{4.52}$$

During the M-step, the component parameters are re-estimated with

$$w_k = \frac{N_k}{N} \tag{4.53}$$

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} \boldsymbol{x}_n \tag{4.54}$$

$$\boldsymbol{C}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} (\boldsymbol{x}_n - \boldsymbol{\mu}_k)(\boldsymbol{x}_n - \boldsymbol{\mu}_k)^\top, \tag{4.55}$$

where $N_k = \sum_{n=1}^{N} \gamma_{nk}$ is the *effective* number of samples assigned to component k. The objective function in case of GMM EM is the *log-likelihood* of the samples, i.e.,

$$L = \ln\left(\prod_{n=1}^{N} p(\boldsymbol{x}_n)\right) = \sum_{n=1}^{N} \ln\left(\sum_{k=1}^{K} w_k \mathcal{N}(\boldsymbol{x}_n; \boldsymbol{\mu}_k, \boldsymbol{C}_k)\right). \tag{4.56}$$

As is usual for EM algorithms, L increases from iteration to iteration and we stop if the change of L between two iterations is below a threshold ε.

Once the EM algorithm has finished, the final γ_{nk} can be used as soft cluster assignments, but for LearnSPN we require hard assignments

$$\beta_{nk} = \begin{cases} 1, & \operatorname{argmax}_{k'} \gamma_{nk'} = k \\ 0, & \text{else.} \end{cases} \tag{4.57}$$

4.5.3 Initialization

Both, k-means and GMM EM, require initial cluster centers or component means $\boldsymbol{\mu}_k$. A provably good way for choosing these is k-means++ [1], which is presented here.

Given are N samples $\boldsymbol{x}_n$ that we want to cluster in K clusters. The first center $\boldsymbol{\mu}_1$ is chosen by drawing a sample uniformly from the given $\boldsymbol{x}_n$. For the remaining centers, we compute a probability P_n for every sample $\boldsymbol{x}_n$ using

$$P_n = \frac{d_{\mathrm{S}}(\boldsymbol{x}_n)}{\sum_{n'=1}^{N} d_{\mathrm{S}}(\boldsymbol{x}_{n'})}, \tag{4.58}$$

where $d_{\mathrm{S}}(\boldsymbol{x}_n)$ is the distance of $\boldsymbol{x}_n$ to the nearest already chosen center $\boldsymbol{\mu}_k$, i.e.,

$$d_{\mathrm{S}}(\boldsymbol{x}_n) = \min_k \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|_2^2. \tag{4.59}$$

The next center is randomly chosen from the samples $\boldsymbol{x}_n$ based on probability P_n. The computation of P_n and random selection of the next $\boldsymbol{\mu}_k$ is repeated until all K centers are chosen.

Chapter 5

Experimental Results

In this final chapter we will present some experimental results. In Section 5.1 we compare the different methods presented in Chapter 4 using the binary data sets that are also used in [9], [13], [32], [33], [37]. Section 5.2 is about the MNIST data set of handwritten digits [17] and the sampling and completion capabilities of SPNs. Finally, in Section 5.3 we present results based on real world 4th generation mobile wireless datarate measurements.

5.1 Binary Data Sets

In this section we compare the performance of the various SPN learning methods presented in Chapter 4. As performance measure we will use the *test set cross entropy*, i.e., the *cross entropy* [10] approximated with a set of test samples $\mathbb{T}_\mathrm{T}$. We can motivate the usage of the test set cross entropy as follows. Let $p(\boldsymbol{x})$ be the true, generally unknown, distribution of the training and test set, and let $\hat{p}(\boldsymbol{x})$ be the approximation our learned SPN represents. A perfect approximation is achieved if the Kullback-Leibler divergence between $p(\boldsymbol{x})$ and $\hat{p}(\boldsymbol{x})$ is zero, i.e., $\mathrm{D}_\mathrm{KL}(p(\boldsymbol{x}) \parallel \hat{p}(\boldsymbol{x})) = 0$. However, because we do not know $p(\boldsymbol{x})$ in general, $\mathrm{D}_\mathrm{KL}(p(\boldsymbol{x}) \parallel \hat{p}(\boldsymbol{x}))$ cannot be computed. We can express the Kullback-Leibler divergence in terms of expectations, i.e.,

$$\mathrm{D}_\mathrm{KL}(p(\boldsymbol{x}) \parallel \hat{p}(\boldsymbol{x})) = \mathrm{E}_{\mathbf{x}\sim p(\boldsymbol{x})}\{-\ln \hat{p}(\boldsymbol{x})\} - \mathrm{E}_{\mathbf{x}\sim p(\boldsymbol{x})}\{-\ln p(\boldsymbol{x})\}, \tag{5.1}$$

where the right expectation is the entropy of $\mathbf{x} \sim p(\boldsymbol{x})$ and the left expectation is the cross entropy of our approximation $\hat{p}(\boldsymbol{x})$. Using our assumption that the samples in the test set $\mathbb{T}_\mathrm{T}$ are distributed according to $p(\boldsymbol{x})$, we can do a *Monte Carlo integration* to approximate the cross entropy in eq. (5.1), i.e.,

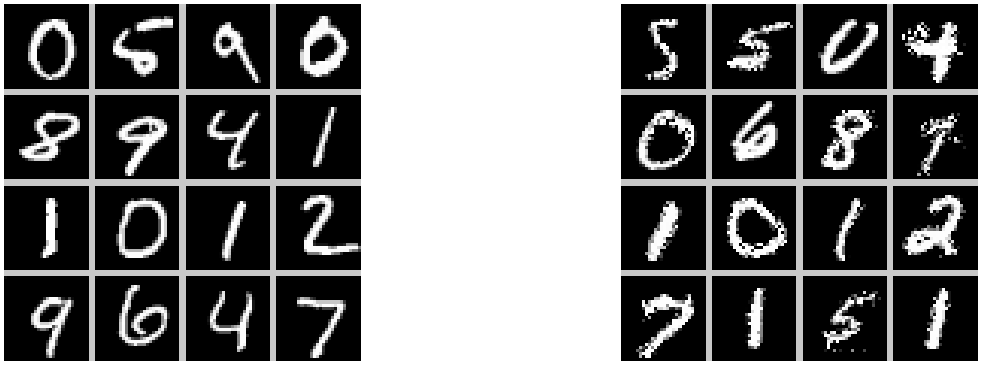
$$\mathrm{E}_{\mathbf{x}\sim p(\boldsymbol{x})}\{-\ln \hat{p}(\boldsymbol{x})\} = -\int_{\mathbb{X}} p(\boldsymbol{x}) \ln \hat{p}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \approx -\frac{1}{|\mathbb{T}_\mathrm{T}|} \sum_{\boldsymbol{x}\in\mathbb{T}_\mathrm{T}} \ln \hat{p}(\boldsymbol{x}). \tag{5.2}$$

Dataset				LearnSPN								OSL	
Name	# Var	# Train	# Test	G/GMM	G/k	MI/GMM	MI/k	ρ/GMM	ρ/k	RDC/GMM	RDC/k	ρ	RDC
NLTCS	16	16181	3236	6.07	**6.05**	6.37	6.73	6.24	6.05	6.13	6.06	6.30	6.28
MSNBC	17	291326	58265	6.05	**6.04**	6.77	6.77	6.13	6.06	6.05	6.05	6.22	6.35
KDDCup2K	64	180092	34955	2.16	**2.14**	2.45	2.45	2.17	2.17	2.18	2.18	2.16	2.16
Plants	69	17412	3482	13.25	**12.93**	13.58	17.46	13.50	13.00	14.69	13.03	19.66	19.95
Audio	100	15000	3000	40.81	**40.08**	47.15	49.21	40.93	40.22	40.67	40.28	44.55	45.71
Netflix	100	15000	3000	58.40	56.81	62.79	64.37	60.26	56.85	58.72	**56.80**	61.69	62.09
Jester	100	9000	4116	53.94	**53.04**	60.11	58.96	56.14	53.13	54.76	53.08	53.40	53.56
Accidents	111	12758	2551	30.11	30.32	32.93	35.56	32.89	**29.59**	39.76	35.50	40.75	40.22
Retail	135	22041	4408	11.42	**10.94**	11.32	11.32	10.98	10.97	11.32	12.09	11.02	11.23
Pumsb-star	163	12262	2452	75.86	**23.86**	31.30	26.27	71.23	26.84	75.86	30.94	31.31	34.11
DNA	180	1600	1186	99.20	82.75	100.39	82.18	99.22	**82.17**	99.20	97.84	92.42	100.10
Kosarek	190	33375	6675	13.33	**10.81**	13.02	13.02	12.08	11.00	13.33	12.15	10.92	11.29
MSWeb	294	29441	5000	11.36	**9.97**	11.10	11.10	10.70	10.21	11.36	11.24	10.29	10.42
EachMovie	500	4524	591	83.60	**52.58**	74.46	65.62	83.60	53.59	83.60	54.53	63.09	76.54
Book	500	8700	1739	41.12	**34.45**	41.12	41.12	39.69	34.82	41.12	40.21	35.47	36.59
WebKB	839	2803	838	179.98	**153.96**	173.71	173.51	169.89	159.28	179.98	161.05	170.73	178.73
Reuters-52	889	6532	1540	108.10	**82.97**	103.21	103.13	99.00	87.65	108.10	95.59	98.66	106.54
20Newsgroup	910	11293	3764	172.39	**151.97**	171.74	171.73	170.57	157.04	172.39	157.84	158.36	160.31
BBC	1058	1670	330	277.08	**249.59**	272.20	272.29	267.38	252.75	277.10	253.58	270.07	276.62
Ad	1556	2461	491	59.73	**19.01**	49.56	48.71	49.60	19.41	63.82	24.93	57.84	61.90

Table 5.1: Comparison of the best achieved test set cross-entropy for various binary data sets and different learning methods. The number of binary variables per sample, the number of training and test samples of each data set are listed on the left. The results achieved with LearnSPN for various dependency measures and clustering algorithms are presented in the middle and the OSL results for different dependency measures are presented on the right. Legend: G = G-test, MI = Mutual information, ρ = Correlation coefficient, GMM = GMM EM, k = k-means.

This is the test set cross entropy and while it is no longer an absolute performance measure, i.e., we do not know the best achievable value, it can be used to compare different methods to each other. Lower values of the test set cross entropy indicate a better approximation of $\hat{p}(\boldsymbol{x})$ with respect to $p(\boldsymbol{x})$.

A common benchmark in the SPN literature [9], [13], [32], [33], [37] is the *average log likelihood*, which is the negative test set cross entropy, of certain binary data sets. The data sets have a various numbers of binary variables per sample, and different training set and test set sizes. The results shown in table 5.1 compare the methods presented in Chapter 4. Here, we compare the performance of LearnSPN using the G-test, mutual information, correlation coefficient and RDC as dependency measures, and k-means and GMM EM as clustering algorithms. We further compare the OSL performance using the correlation coefficient and the RDC as dependency measures. Table 5.1 shows the best test set cross entropies that we achieved with a grid search over the hyperparameter space. The results are mostly within variance of what can be found in other publications [9], [13], [32], [33], [37]. For these binary data sets LearnSPN with the G-test and k-means performs the best overall, and OSL usually performes similarly to LearnSPN.



(a) (b)

Figure 5.1: (a) Samples from the MNIST training set. (b) Samples computed with an SPN.

5.2 MNIST Data Set of Handwritten Digits

We now present some results based on the MNIST data set of handwritten digits [17]. Every sample in the data set consists of a 28 by 28 pixel gray-scale image of a handwritten digit together with the corresponding label, i.e., the digit that is depicted. The training set consists of 60000 samples and the test set of 10000. Figure 5.1a shows 16 examples from the training set.

The typical task for this data set is to classify the images in the test set based on their pixel values. We achieved 93% accuracy with our LearnSPN implementation. This is far behind the performance of NNs, which are capable of achieving more than 99% accuracy [4]. However, SPNs learned by different methods are reported to achieve over 98% accuracy [13], [28].

Additionally to the classification accuracy, we conducted experiments to highlight the sampling and image completion capabilities of SPNs. For example, the images shown in fig. 5.1b were generated by sampling from an SPN. While not indistinguishable from the samples in the training set, the generated samples are well readable and show the typical style many images in the training set have. Figure 5.2 depicts some exemplary right half completions performed by an SPN. The task was to predict the pixels of the right half of the image based on the knowledge of the pixels of the left half. We additionally investigated the effect the knowledge (or the lack thereof) of the label, i.e., the label, would have on the completion. We can see in fig. 5.2 for example, that the "9" in the 5th column is completed to a "4" if the label is not known. Further, it can be observed that the MMSE estimation tends to smoother images, while the images generated with VMAP estimation are usually sharp.

Figure 5.2: Right half completions of samples from the MNIST test set, i.e., the pixel values of the left half is given as input. We compare the results of MMSE/VMAP estimation with known/unknown label (i.e., depicted digit). From top to bottom, the five rows depict: 1. The left half of the test set images, i.e., the input of the completion. 2. MMSE and known label. 3. MMSE and unknown label. 4. VMAP and known label. 5. VMAP and unknown label.

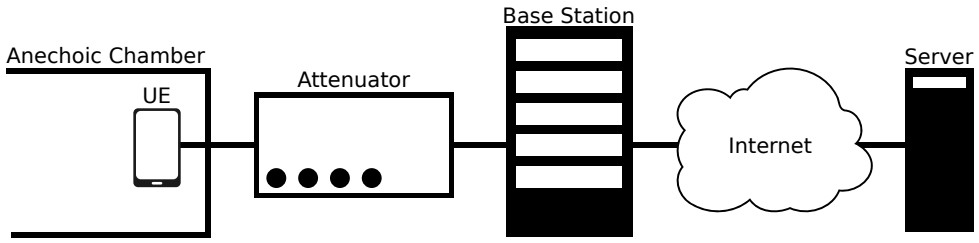


Figure 5.3: The LTE data rate measurement setup.

5.3 LTE Data Rate Measurements

Finally, we will present results based on Long Term Evolution (LTE) [19] data rate measurements.

5.3.1 Measurement setup

The measurements were conducted in a controlled environment and with the goal of better understanding how different parameters influence the maximum possible datarate. The setup is shown in fig. 5.3. The *user equipment* (UE), i.e., a cell phone, is located in an anechoic chamber to avoid outside interference with the measurement. The UE is directly connected to a tunable attenuator, which in turn is connected to a base station that only serves the measured UE. From the base station a measurement server is reachable. We had two different subscriber identity modules (SIMs) in use, one unlimited and the other limited to a maximum download data rate of 10 Mbit/s.

All measurements were conducted without interference, which means that the only

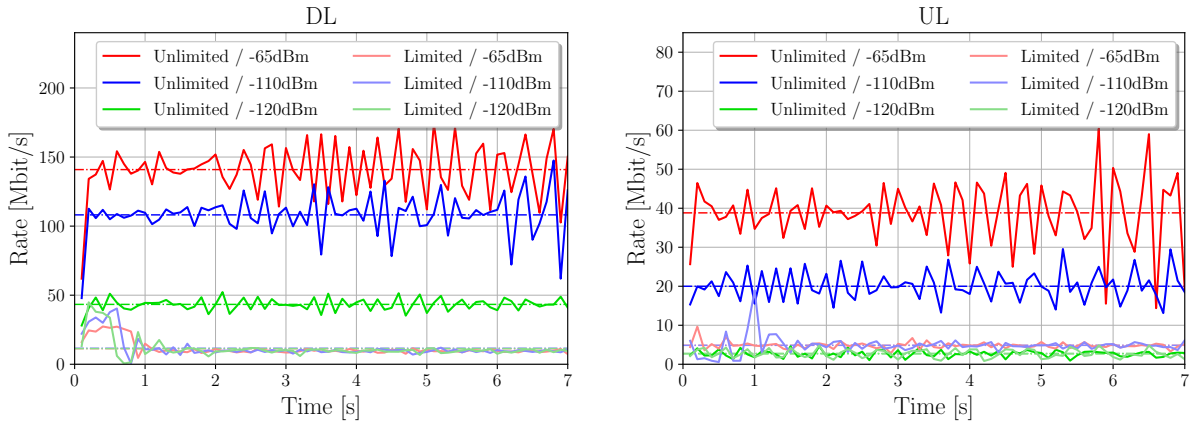


Figure 5.4: Examples of test set measurement time series for different combinations of SIM card limitations and RSRP values. The horizontal lines indicate the overall data rates.

factor limiting the datarate is the *signal to noise ratio* (SNR). In our controlled environment, we can substitute the difficult to measure SNR with the *reference signal received power* (RSRP), i.e., the signal power received from a reference signal. The reference signal and RSRP are defined in the LTE standard [19]. The RSRP is measured in dBm and the currently measured value is available to the software of modern cell phones.

The program used for testing was Open-RMBT [25]. A test consists of a download and an upload (UL), both with several parallel streams, in our case 10. At the end of a test, additionally to the overall computed datarate and the RSRP, the cumulative downloaded volume of every stream as time series is stored. The time series of the different streams were combined and re-sampled to form a new 7 seconds long time series with 0.1 seconds sample rate. We obtained two data sets, one for the download or downlink (DL), and one for the upload or uplink (UL). Both were randomly partitioned into a training set and a test set with 2777 and 694 sample vectors respectively. Every sample vector has 73 entries, i.e., SIM limitation, RSRP, overall rate, and the 70 points long combined re-sampled time series. Some exemplary time series are shown in fig. 5.4 and scatter plots of the RSRP values and overall data rates is shown in fig. 5.5.

5.3.2 Time series and overall rate prediction

The first interesting problem is to predict the overall rate and the complete time series based on the RSRP and the first few values of the time series. This is relevant, because as the possible data rates increase, so does the downloaded (or uploaded) data volume during a 7 second test, which usually has to be paid by the end user. Another problem of long tests is, that the cell the test is conducted in is put under heavy load for the duration of

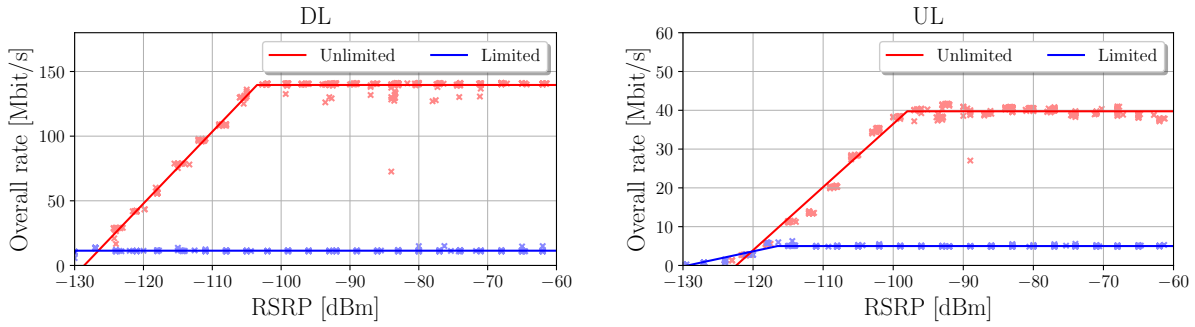


Figure 5.5: Scatter plots of the data rates vs. the RSRPs of the test sets. The solid lines are least squares fits of a piece-wise, linear and constant, function.

the test. Both problems can be mitigated by reducing the duration of the test, and thus a prediction of the overall rate after 0.5 s or 1 s is desirable. All results in this section are obtained by selecting the best SPN from a grid search over the various hyperparameters, i.e., the SPN minimizing the predicted error given the first 5 samples averaged over the test set sample vectors. Examples for predicted series are shown in fig. 5.6.

A problem poses the SIM limitation, because this information is usually not available to the measurement software and the limiting algorithm only starts after about 0.5 s in our tests, as is shown in fig. 5.4. We can jointly estimate the SIM limitation together with the series and the overall rate and because of our idealized environment, high accuracies can be achieved. Since the information whether the SIM is limited or not is present in our test data, we can compare the prediction results as is done in fig. 5.7 and table 5.2. The relative error in percent $e_{\%}$ that is reported in the histograms and table is computed with

$$e_{\%} = \frac{\hat{y} - y}{y} \cdot 100\%, \tag{5.3}$$

where y is the "true" overall data rate from the test set and $\hat{y}$ the MMSE estimate from the SPN.

We can notice, that the histograms in fig. 5.7 show not much difference depending on whether the SIM limitation is known or not. If we take a look at table 5.2 however, we can see that the average error increases. This is due to the fact that a wrongly estimated SIM limitation will cause a very large error for the predicted overall rate. We can also observe, that the average error for the UL set is lower than for the DL set, despite the fact that it is easier to predict the SIM limitation for the DL set. This can be explained by taking a look at fig. 5.4, where we can see that limited and unlimited series are hardly distinguishable for the UL set in the low RSRP regions, and thus a miss-classification has less effect on the estimation of the overall data rate.

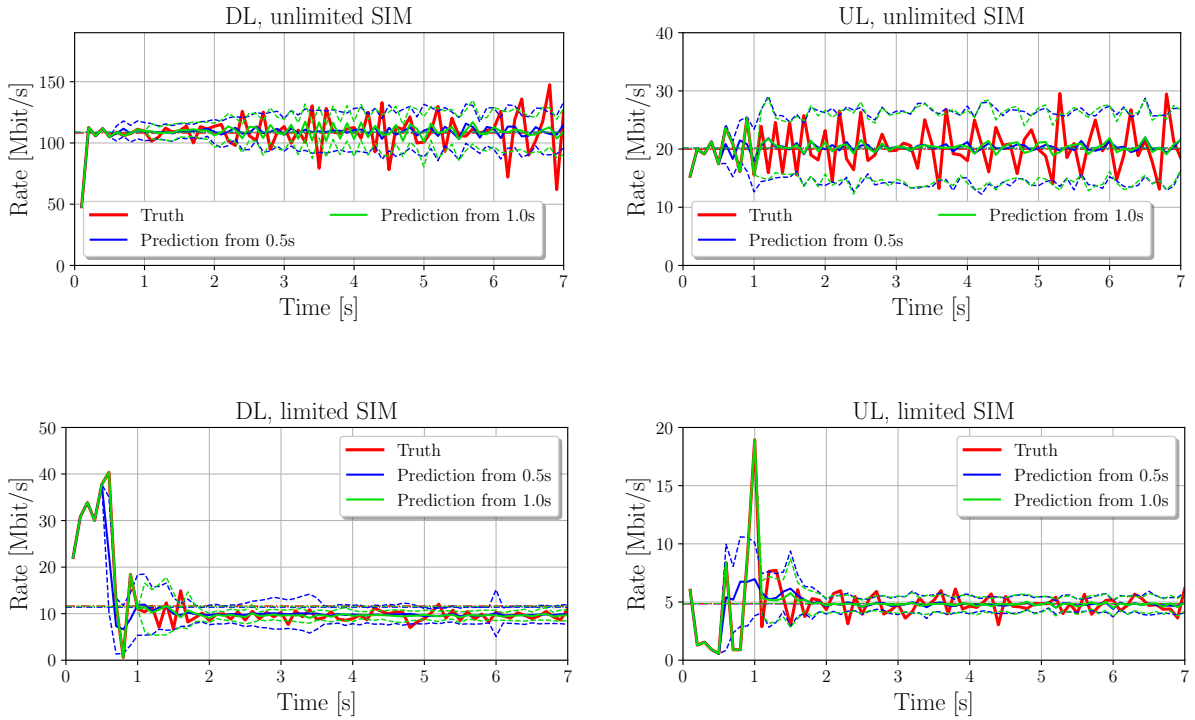


Figure 5.6: MMSE estimations of the time series with −110 dBm RSRP from fig. 5.4. The predictions start after 0.5 s and 1 s and are based on the RSRP, SIM limitation and the first few unpredicted values. The dashed lines are $\pm\sigma$ away from the MMSE estimate, where σ is the square root of the estimated variance. The horizontal lines indicate the overall data rates, or their MMSE estimates in case of the predicted series.

5.3.3 Prediction of unlimited series and rates

The last task we want to investigate is the prediction of an unlimited series if the first few samples are from a limited series. The reason behind this is, that this is a first step towards a more realistic scenario, where we would be limited by interference. The results for this experiment can be seen in fig. 5.8, fig. 5.9 and table 5.3. These results look not to promising however, and it seems that the provided time series samples cause more harm than they help since the errors increase if more samples are provided.

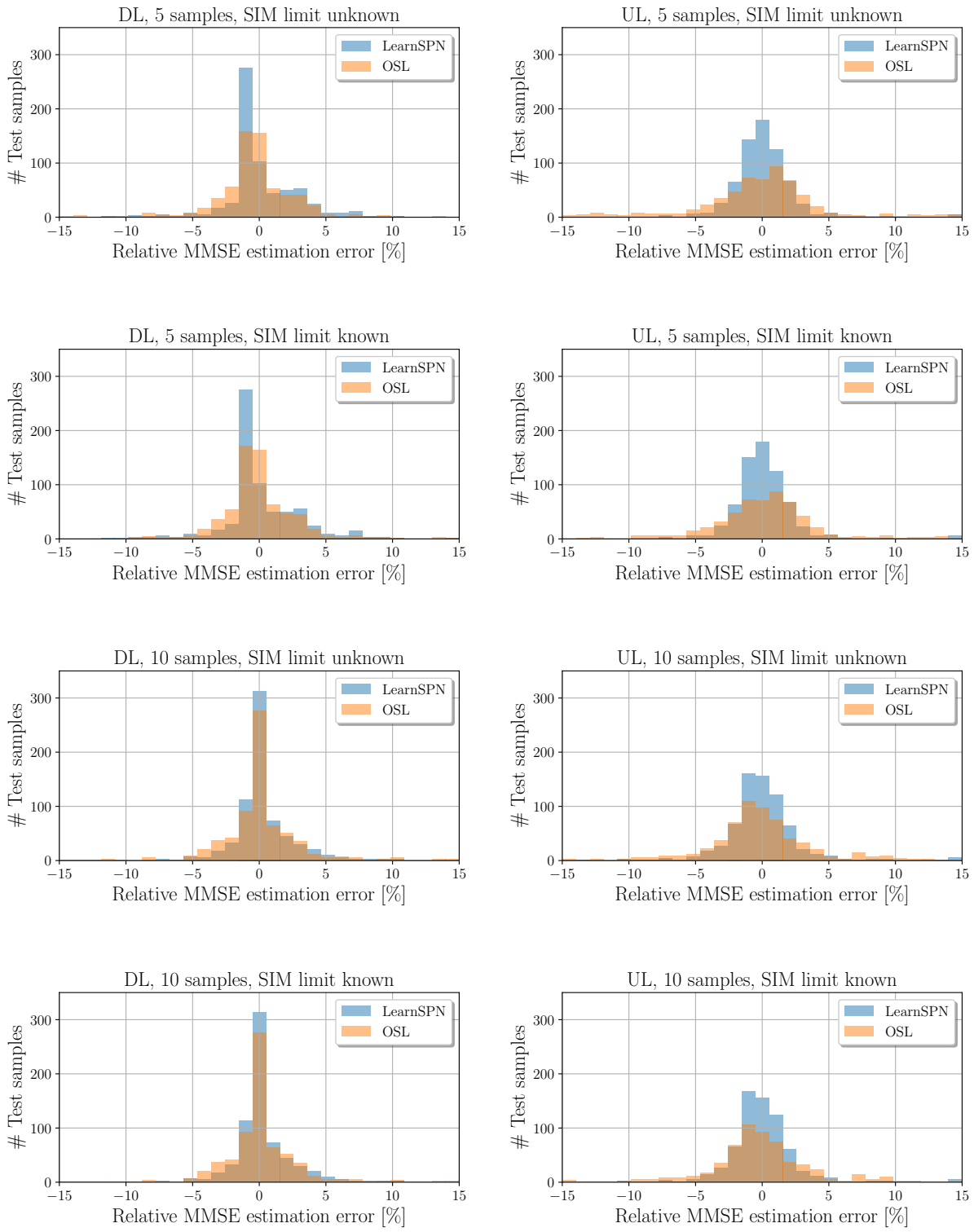


Figure 5.7: Relative error histograms of the overall rate MMSE estimates for all test set sample vectors and different inputs. The estimates are based on the RSRP, the first 5 or 10 samples from the time series and in some cases the knowledge of the SIM limitation.

				SIM limit. classification counts / Avg. errors				Overall results	
Set	Samples	SIM limit.	Learner	U + Est. U	L + Est. L	U + Est. L	L + Est. U	Lim. acc.	Avg. error
DL	5	known	LearnSPN	415 / 2.5	279 / 2.8				**2.6**
DL	5	known	OSL	415 / 1.9	279 / 3.6				**2.6**
DL	5	unknown	LearnSPN	413 / 2.4	276 / 3.3	2 / 23.8	3 / 97.3	**99.28**	**3.3**
DL	5	unknown	OSL	413 / 2.1	261 / 6.3	2 / 38.4	18 / 221.9	**97.12**	**9.5**
DL	10	known	LearnSPN	415 / 1.0	279 / 2.5				**1.6**
DL	10	known	OSL	415 / 1.4	279 / 5.5				**3.1**
DL	10	unknown	LearnSPN	414 / 1.0	278 / 2.5	1 / 18.1	1 / 131.6	**99.71**	**1.8**
DL	10	unknown	OSL	415 / 1.4	274 / 3.2	0 / –	5 / 146.0	**99.28**	**3.2**
UL	5	known	LearnSPN	406 / 1.6	288 / 2.6				**2.0**
UL	5	known	OSL	406 / 77.6	288 / 62.4				**71.3**
UL	5	unknown	LearnSPN	402 / 1.6	281 / 2.7	4 / 2.3	7 / 2.9	**98.41**	**2.1**
UL	5	unknown	OSL	344 / 5.0	287 / 63.1	62 / 16.3	1 / 124.4	**90.92**	**30.2**
UL	10	known	LearnSPN	406 / 1.6	288 / 2.7				**2.0**
UL	10	known	OSL	406 / 21.8	288 / 109.3				**58.1**
UL	10	unknown	LearnSPN	400 / 1.6	279 / 2.7	6 / 1.8	9 / 3.3	**97.84**	**2.1**
UL	10	unknown	OSL	344 / 3.3	288 / 109.5	62 / 42.4	0 / –	**91.07**	**50.8**

Table 5.2: Comparison of the overall rate MMSE estimates for all test set sample vectors and different inputs. The estimates are based on the RSRP, the first 5 or 10 samples from the time series and in some cases the knowledge of the SIM limitation. The average estimation errors and the overall limitation classification accuracy are given in percent.

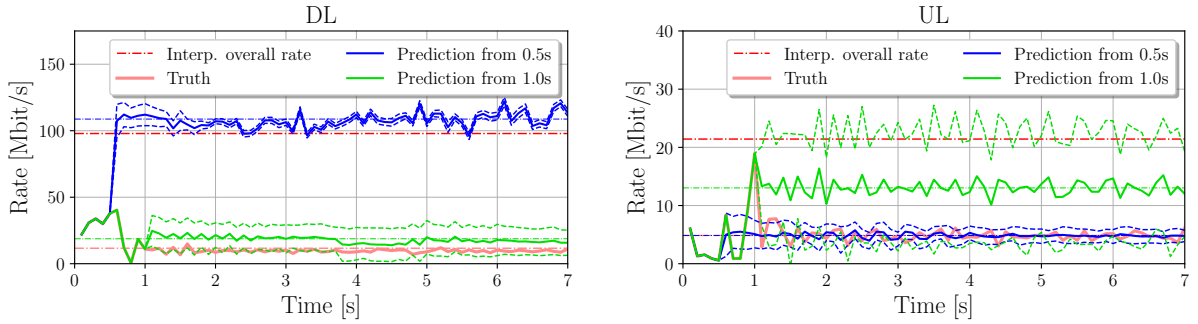


Figure 5.8: MMSE estimations conditioned on an unlimited SIM of the limited time series with −110 dBm RSRP from fig. 5.4. The predictions start after 0.5 s and 1 s and are based on the RSRP and the first few unpredicted values. The dashed lines are $\pm\sigma$ away from the MMSE estimate, where σ is the square root of the estimated variance. The horizontal lines indicate the overall data rates, or their MMSE estimates in case of the predicted series.

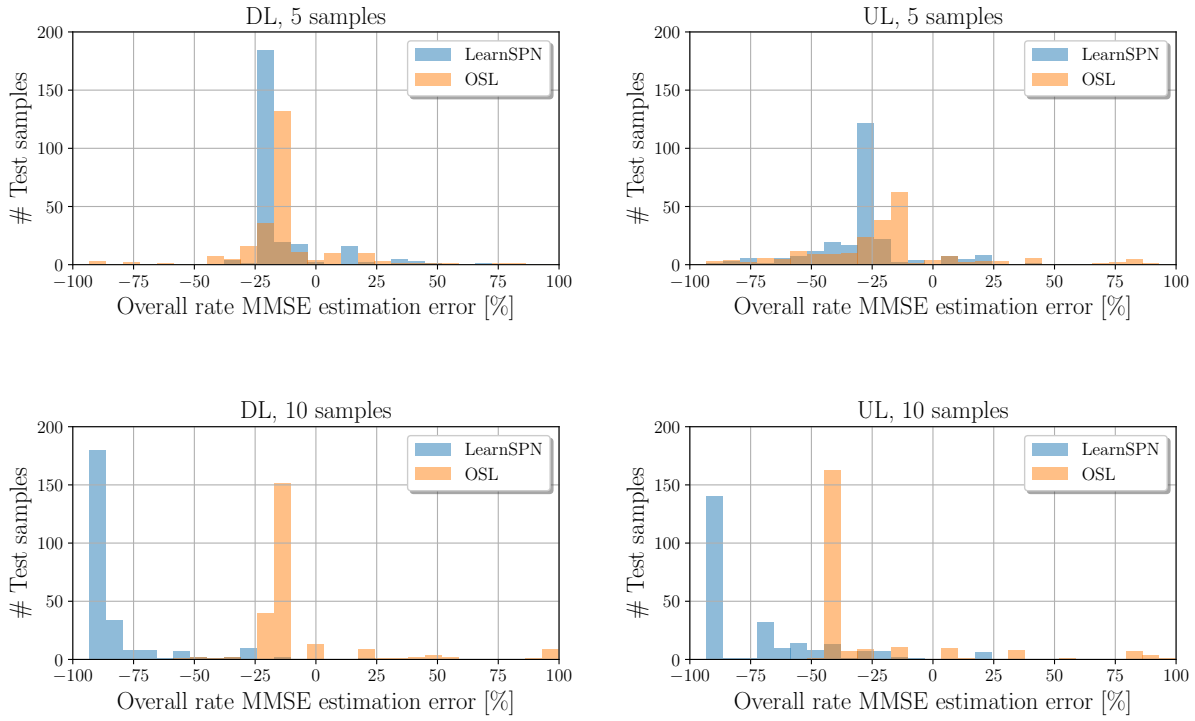


Figure 5.9: Relative error histograms of the overall rate MMSE estimates conditioned on an unlimited SIM for all limited test set sample vectors and different inputs. The estimates are based on the RSRP and the first 5 or 10 samples from the time series.

Set	Samples	Learner	Avg. error
DL	LearnSPN	5	**47.1**
DL	OSL	5	**82.7**
DL	LearnSPN	10	**90.7**
DL	OSL	10	**87.6**
UL	LearnSPN	5	**45.9**
UL	OSL	5	**68.6**
UL	LearnSPN	10	**79.2**
UL	OSL	10	**66.4**

Table 5.3: Comparison of the overall rate MMSE estimates conditioned on an unlimited SIM for all limited test set sample vectors and different inputs. The estimates are based on the RSRP and the first 5 or 10 samples from the time series. The average estimation errors are given in percent.

Chapter 6

Conclusion

We have presented SPNs, a recently introduced PGM that allows for efficient inference. Like NNs, SPNs are based on computational graphs, while still being interpretable in a probabilistic way, i.e., SPNs represent a distribution. We are able to evaluate and sample from marginals and conditionals of this distribution and we can further compute MMSE and approximate VMAP estimates. Unlike many other machine learning methods, once trained, an SPN can naturally be used for many different tasks and inputs, and is not limited to a specific combination of task and input. Some selected theoretical properties of SPNs were discussed, including validity, completeness, decomposability, the model limitation to indirect variable dependencies, and the ability to plug-in other models.

Two structure learning algorithms were presented, i.e., LearnSPN for offline learning and OSL for online learning. Both learning algorithms depend on other methods to model univariate distributions, to estimate the dependency between random variables and to cluster data samples. For all these methods we have presented several possibilities with the goal that we can use SPNs for mixed, i.e., continuous and discrete, data.

Finally, we presented some experimental results. We compared the different learning methods and showed results based on the widely used MNIST data set and real world LTE datarate measurements.

There are many SPN related topics this master's thesis did not discuss, including:

- weight learning, i.e., the other big learning paradigm, next to structure learning, used to train SPNs;
- the relation of SPNs to other graphical models, especially Bayesian networks;
- better approximations of the VMAP estimator;
- a well founded theoretical examination of the model limitations;
- an efficient implementation of SPNs and the training algorithms.

Bibliography

[1] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proc. 18th annual ACM-SIAM Symposium on Discrete Algorithms, SODA'07*, New Orleans, LA, USA, Jan. 2007, pp. 1027–1035.

[2] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, 2nd ed. London: Springer-Verlag, 2008.

[3] C. Bishop, *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer-Verlag, 2006.

[4] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI*, Barcelona, Spain, Jul. 2011, pp. 1237–1242.

[5] A. Darwiche, "A differential approach to inference in Bayesian networks," *J. ACM*, vol. 50, no. 3, pp. 280–305, May 2003.

[6] O. Delalleau and Y. Bengio, "Shallow vs. deep sum-product networks," in *Advances in Neural Information Processing Systems 24*, Granada, Spain, Dec. 2011, pp. 666–674.

[7] A. W. Dennis and D. Ventura, "Learning the architecture of sum-product networks using clustering on variables," in *Proc. 25th Conference on Neural Information Processing Systems, NIPS'12*, Lake Tahoe, NV, USA, Dec. 2012, pp. 2042–2050.

[8] R. Diestel, *Graph Theory*, 5th ed. Heidelberg, Germany: Springer-Verlag, 2016.

[9] R. Gens and P. M. Domingos, "Learning the structure of sum-product networks," in *Proc. ICML 2013*, Atlanta, GA, USA, Jun. 2013, pp. 873–880.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, 2016.

[11] W. K. Härdle and L. Simar, *Applied Multivariate Statistical Analysis*, 3rd ed. Heidelberg, Germany: Springer-Verlag, 2012.

[12] W. Hsu, A. Kalra, and P. Poupart, "Online structure learning for sum-product networks with Gaussian leaves," in *5th International Conference on Learning Representations, ICLR 2017 - Workshop Track*, Toulon, France, Apr. 2017.

[13] P. Jaini, A. Ghose, and P. Poupart, "Prometheus: Directly learning acyclic directed graph structures for sum-product networks," in *Proc. Machine Learning Research, PGM 2018*, vol. 72, Prague, Czech, Sep. 2018, pp. 181–192.

[14] R. Kindermann and J. L. Snell, *Markov Random Fields and Their Applications*, ser. Contemporary Mathematics. Providence, RI, USA: American Mathematical Society, 1980, vol. 1.

[15] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997, vol. 1.

[16] D. Koller and N. Friedman, *Probabilistic Graphical Models – Principles and Techniques*, ser. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, 2009.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[18] D. Lopez-Paz, P. Hennig, and B. Schölkopf, "The randomized dependence coefficient," in *Proc. 26th Conference on Neural Information Processing Systems, NIPS'13*, Lake Tahoe, NV, USA, Dec. 2013, pp. 1–9.

[19] (Dec. 2018). LTE standard, 3GPP, [Online]. Available: http://www.3gpp.org/LTE/.

[20] J. Martens and V. Medabalimi, "On the expressive efficiency of sum product networks," *arXiv - CoRR*, Nov. 2014, arXiv/1411.7717 [cs.LG].

[21] J. H. McDonald, *Handbook of Biological Statistics*, 3rd ed. Baltimore, MD, USA: Sparky House Publishing, 2014.

[22] J. Mei, Y. Jiang, and K. Tu, "Maximum A posteriori inference in sum-product networks," in *Proc. 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, New Orleans, LA, USA, Feb. 2018, pp. 1923–1930.

[23] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997.

[24] A. Molina, A. Vergari, N. D. Mauro, S. Natarajan, F. Esposito, and K. Kersting, "Mixed sum-product networks: A deep architecture for hybrid domains," in *Proc. 32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, New Orleans, LA, USA, Feb. 2018, pp. 3828–3835.

[25] (Dec. 2018). Open-RMBT, [Online]. Available: https://github.com/alladin-IT/open-rmbt.

[26] E. Parzen, "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 1956.

[27] J. Pearl, "Bayesian networks: A model of self-activated memory for evidential reasoning," University of California L.A., Computer Science Department, Tech. Rep., Jun. 1985.

[28] R. Peharz, A. Vergari, K. Stelzner, A. Molina, M. Trapp, K. Kersting, and Z. Ghahramani, "Probabilistic deep learning using random sum-product networks," *arXiv - CoRR*, Jun. 2018, arXiv/1806.01910 [cs.LG].

[29] R. Peharz, "Foundations of sum-product networks for probabilistic modeling," PhD thesis, Graz University of Technology, 2015.

[30] R. Peharz, R. Gens, F. Pernkopf, and P. Domingos, "On the latent variable interpretation in sum-product networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 10, pp. 2030–2044, Oct. 2017.

[31] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos, "On theoretical properties of sum-product networks," in *Proc. AISTATS 2015*, San Diego, CA, USA, May 2015, pp. 744–752.

[32] H. Poon and P. M. Domingos, "Sum-product networks: A new deep architecture," in *IEEE ICCV 2011 Workshops*, Barcelona, Spain, Nov. 2011, pp. 689–690.

[33] A. Rooshenas and D. Lowd, "Learning sum-product networks with direct and indirect variable interactions," in *Proc. ICML 2014*, Beijing, China, Jun. 2014, pp. 710–718.

[34] A. R. Runnalls, "Kullback-Leibler approach to Gaussian mixture reduction," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 43, no. 3, pp. 989–999, Jul. 2007.

[35] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, Jul. 1959.

[36] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. London, UK: Chapman & Hall, 1986.

[37] A. Vergari, N. D. Mauro, and F. Esposito, "Simplifying, regularizing and strengthening sum-product network structure learning," in *Proc. ECML PKDD 2015*, Sep. 2015, pp. 343–358.

Abbreviations

CCA, Canonical Correlation Analysis, 55
CDF, Cumulative Distribution Function, 8, 58
DAG, Directed Acyclic Graph, 13
DL, Download, Downlink, 67
DL, Upload, Uplink, 67
EM, Expectation Maximization, 60
GMM, Gaussian Mixture Model, 60
KDE, Kernel Density Estimator, 52
LTE, Long Term Evolution, 66
MMSE, Minimum Mean Square Error, 30
NN, Neural Network, 9, 16
OSL, Online Structure Learning, 46
PDF, Probability Density Function, 8
PGM, Probabilistic Graphical Model, 9
PMF, Probability Mass Function, 8
RDC, Randomized Dependence Coefficient, 55
RSRP, Reference Signal Received Power, 67
SIM, Subscriber Identity Module, 66
SNR, Signal to Noise Ratio, 67
SPN, Sum-Product Network, 9
SPT, Sum-Product Tree, 36
UE, User Equipment, 66
UL, Upload, 67
VMAP, Vector Maximum A-Posteriori, 33

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, January 10, 2019

Richard Prüller