Mobile Robot Self-Driving Through Image Classification Using Discriminative Learning of Sum-Product Networks

Undergraduate Thesis

Student: Renato Lui Geh
Advisor: Prof. Denis Deratani Mauá

INSTITUTE OF MATHEMATICS AND STATISTICS
UNIVERSITY OF SÃO PAULO

São Paulo, Brazil

2018

Acknowledgements

I would like to greatly thank my advisor, Prof. Denis Deratani Mauá, for the support and attention, but most of all for the patience of having to read countless reports for both my undergraduate research and undergraduate thesis, many of which were not short.

To my parents, Chen and Luiz, for making sure I wanted for nothing, for giving me the best education possible, and for always assuring me of my abilities.

A special thank you to Maria Clara Cardoso, my best friend and confidant, whose support and company helped me immensely through the last few years. Our conversations are always filled with laughter and I will always hold them dear.

A warm thanks to my friends and colleagues Ricardo and Yan, whose camaraderie and friendship I cherish deeply.

I'd also like to express great gratitude to all professors I had during my undergraduate, whose trade is often overlooked and underappreciated, yet manage to teach us so much and inspire us to always do our best.

This work was partially supported by CNPq grant PIBIC 800585/2016.

Abstract

GEH, L. R. **Mobile robot self-driving through image classification using discriminative learning of sum-product networks**. Institute of Mathematics and Statistics, University of São Paulo, São Paulo, Brazil, 2018.

Driving has proven to be a very difficult task for machines to emulate, not only due to the inherent complexity of the problem but also because of the need for accurate real-time predictions. Nonetheless, recent advances in computer vision and machine learning have shown promising results in the real-world. Mobile robots are low-cost miniature computers with limited processing power and memory. The problem of self-driving can be similarly applied to the mobile robot domain as a down-scaled version of the same task, with an additional hardware constraint. Sum-product networks are probabilistic graphical models capable of representing tractable probability distributions containing a great number of variables. Exact inference is asymptotically linear to the number of edges in the network's graph, and its deep architecture is capable of representing a wide range of distributions. In this work, we attempt to model autonomous driving by using sum-product networks on a small mobile robot. We model this task as an imitation learning problem through image classification. We present accuracy results on an artificial self-driving dataset for different sum-product network learning algorithms, providing a comparative study not only for different network architectures, but also discriminative and generative models. Finally, we provide a real-world mobile robot implementation on a miniature computer.

Keywords: sum-product networks, probabilistic graphical models, machine learning, robotics

Abbreviations

DAG	Directed acyclic graph
EM	Expectation-maximization
GD	Gradient descent
IV	Indicator variable
MAP	Maximum a posteriori probability
MPE	Most probable explanation
MPN	Max-product network
MST	Minimum spanning tree
PGM	Probabilistic graphical model
RV	Random variable
SGD	Stochastic gradient descent
SPN	Sum-product network
SPT	Sum-product tree

Symbols and notations

μ	Gaussian distribution mean
σ	Gaussian distribution standard deviation
$\mathcal{X}$	Sample space of random variables
Z	Partition function
ϕ	Factor (potential)
$[X = x]$	Indicator function for random variable valuation $X = x$

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

In this chapter we first describe the motivations and objectives of this thesis. Next, we describe the structure of this document.

1.1 Motivation and objectives

Self-driving is a challenging computer vision task, mainly due to its inherent complexity and the necessity for real-time decision making. Although there have been many promising results the past few years on autonomous driving, the task still relies on the underlying problem of following a pathway through visual cues (usually road markings). A possible approach to this task is through imitation learning by means of image classification. That is, the agent tasked with driving should be able to reliably mimic human behavior by correctly classifying whether to turn, stop or go straight given an image captured in front of the car.

Mobile robots are low cost machines capable of movement. These robots are usually small, and because of their size and cost often don't have the same performance capabilities as a desktop computer. However, these domain traits make mobile robot self-driving a very similar analogue to real-world autonomous cars. Processing power and memory constraints play a big role in this case, and translate well to embedded systems present in a self-driving car.

Sum-product networks (SPNs) are probabilistic graphical models that are able to represent a wide range of tractable probability distributions of many variables. SPNs have shown impressive results in several domains, and particularly that of image classification. Their deep architecture seems to capture features and contexts well, and since inference is computed in time linear to the network's edges, SPNs are promising models for fast inference in self-driving.

In this work, we attempt to model self-driving of mobile robots through image classification. For the task of classification our objective is to use sum-product networks learned discriminatively, though we also give results for generative SPNs, comparing not only generative and discriminative learning, but also different SPN architectures.

1.2 Thesis structure

This thesis is structured as follows. In chapter 2, we first provide background on sum-product networks, where we formally define an SPN, present key properties on their structure, explain how to compute exact inference and find an approximation of the maximum a posteriori probability (MAP).

In chapter 3, we show how to compute the partial derivatives with respect to a sub-SPN and to its weights, leading on how to perform gradient descent and then on learning the weights of the network through gradient descent both generatively and discriminatively.

Chapter 4 is dedicated to algorithms for learning the structure of an SPN. We explain the two structural learning algorithms that were used in the experiments.

For **??**, we first show how we model self-driving as an image classification problem. We then specify the architecture of the robot used in the experiments, giving specifications on the hardware and software used. Furthermore, we describe some concepts of control we use for navigation.

In **??**, we provide classification results on many image classification datasets from various domains with different learning algorithms. We then describe the self-driving dataset used for training, and give in-dataset accuracies as well as real-world empirical results on the mobile robot itself.

Finally, in **??** we give our conclusions and provide some discussion of the results.

There is an additional section of this thesis in which we give a brief subjective insight on the work done for this thesis. We also list subjects we deemed important for the work done in this thesis.

Furthermore, **??** contains all proofs done in this thesis.

Chapter 2

Sum-product networks

In this chapter we provide some background concepts needed for defining a sum-product network. Once this is covered, we formally define an SPN, list some interesting properties on their structure, and describe how to perform exact inference (i.e. extract the probability of evidence of some valuation) and how to find an approximation of the maximum a posteriori probability.

We leave all proofs in **??**.

2.1 Background

The objective of probabilistic modelling is to compactly represent a probability distribution, be able to find a good approximation to the real function and be able to efficiently compute both the marginals and modes. Probabilistic graphical models (PGMs) attempt to solve this through the use of graphs, representing distributions as a normalized product of factors (Pearl, 1988).

$$P(X = x) = \frac{1}{Z} \prod_k \phi_k(x_{\{k\}})$$

Where $x \in \mathcal{X}$ is a d-dimensional vector valuation of RVs $\mathbf{X}$ on sample space $\mathcal{X}$, and factor (also called a potential) ϕ_k is a function mapping instantiations of X to a non-negative number. Z is the partition function $Z = \sum_{x \in \mathcal{X}} \prod_k \phi_k(x_{\{k\}})$ that sums out all variables and normalizes the term above it to the $[0, 1]$ range.

A downside of this representation is that inference is exponential on the worst case, which makes learning also exponential, as it uses inference as a subroutine. To get around this problem, Darwiche proposed in Darwiche, 2003 the notion of *network polynomial*.

A network polynomial is a function over the probabilities of each instantiation. Let $\Phi(x)$ be a probability distribution. The network polynomial of $\Phi(x)$ is the function $f = \sum_{x \in \mathcal{X}} \Phi(x)\Pi(x)$, where $\Pi(x)$ is the product of the IVs of each variable on instantiation x, where each indicator variable $[Y = y]$ has a value of zero if $Y \neq y$ in x and a value of one otherwise (i.e. if $Y = y$ in x or $Y \notin x$).

As an example, take the bayesian network $\mathcal{N} = A \longrightarrow B$ with binary variables. Let λ_a, $\lambda_{\overline{a}}$, λ_b and $\lambda_{\overline{b}}$ be the indicator variables for when $A = 1$, $A = 0$, $B = 1$ and $B = 0$ respectively. The network polynomial of $\mathcal{N}$ is the expression

$$f_{\mathcal{N}} = P(a)P(b|a)\lambda_a\lambda_b + P(a)P(\overline{b}|a)\lambda_a\lambda_{\overline{b}} + P(\overline{a})P(b|\overline{a})\lambda_a\lambda_{\overline{b}} + P(\overline{a})P(\overline{b}|\overline{a})\lambda_{\overline{a}}\lambda_{\overline{b}}.$$

The main advantage of this representation is to avoid recomputing terms. For instance, take an instantiation of $x = \{A = 0\}$. Then, the network polynomial will be as follows.

$$\begin{aligned} f_{\mathcal{N}}(x) &= P(a)P(b|a) \cdot 0 \cdot 1 + P(a)P(\overline{b}|a) \cdot 0 \cdot 1 + P(\overline{a})P(b|\overline{a}) \cdot 1 \cdot 1 + P(\overline{a})P(\overline{b}|\overline{a}) \cdot 1 \cdot 1 = \\ &= P(\overline{a})P(b|\overline{a}) + P(\overline{a})P(\overline{b}|\overline{a}) \end{aligned}$$

Which means we can avoid computing values from the two first terms. We can also compute the network polynomial of some unnormalized probability distribution as long as we divide by the partition function, defined as the network polynomial with all indicators set to one. Although the network polynomial has exponential size in terms of variables, computing the probability of evidence is linear in its size. By representing the network polynomial as an arithmetic circuit of sums and products, one can prove that the cost of inference is indeed polynomial.

2.2 Definitions and properties

Sum-product networks borrow many concepts from network polynomials and arithmetic circuits. There are many definitions of SPNs and in this thesis we present two. The first definition is given by the seminal article Poon and Domingos, 2011, and can be seen as a more low-level approach to defining the network. The second, based on Gens and Domingos, 2013, is a stronger definition, but one which we will use more throughout this thesis, as it lends itself better to continous data.

Let $\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$ be the set of all variables. We shall call this set the root scope. Let G be a graph G. The sets of vertices and edges of G will be denoted by $V(G)$ and $E(G)$. We will call $\mathrm{Ch}(n)$ and $\mathrm{Pa}(n)$ the sets of children and parents of node $n \in V(G)$.

Definition 2.1 (Sum-product network; Poon and Domingos, 2011)**.** *A sum-product network (SPN) over variables $X_1, X_2, \ldots, X_n$ is a DAG whose leaves are indicator variables $[X_1 = x_1^1], [X_2 = x_2^1], \ldots, [X_n = x_n^1], \ldots, [X_1 = x_1^d], [X_2 = x_2^d], \ldots, [X_n = x_n^d]$. Its internal nodes are weighted sums or products. Each edge coming out from a sum node n to another node j has a non-negative weight associated with it. We denote such weight by $w_{n,j}$. The value of a sum node n is $v_n = \sum_{j \in Ch(n)} w_{n,j} v_j$, where v_j is the value of node j. The value of a product node n is $v_n = \prod_{j \in Ch(n)} v_j$. The value of a leaf node is the value of the indicator variable. The value of the SPN is the value of its root node.*

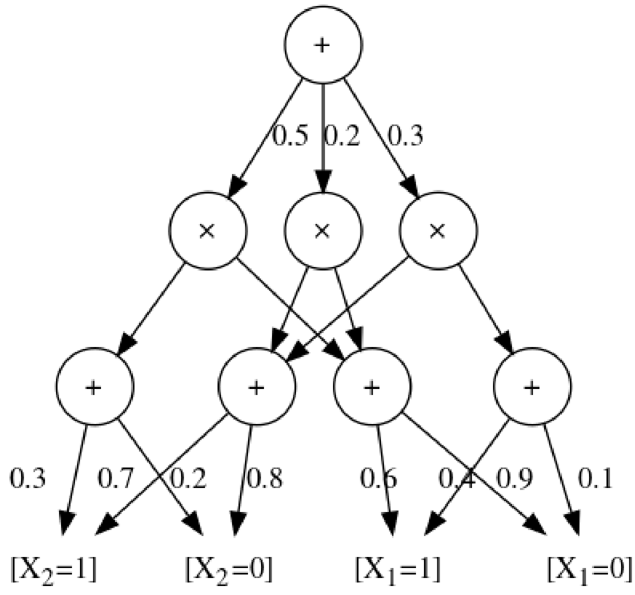


Figure 2.1: *An example of an SPN.*

Throughout this thesis, we denote by $S(X = x)$ the value of an SPN S given evidence x. A sub-SPN S_n of S is the subgraph of S rooted at n. A node in an SPN is itself an SPN. When all indicator variables are set to one, the value of S is denoted by $S(*)$. The scope of an SPN S, denoted by $\text{Sc}(S)$, is the union set of all scopes of its children. A leaf's scope is the scope of its IV.

Definition 2.2 (Validity)**.** *An SPN is valid iff, for all evidence $E = e$, $S(E = e) = \Phi_S(E = e)$, where Φ_S is an unnormalized probability distribution.*

Definition 2.3 (Completeness)**.** *An SPN is complete iff all children of the same sum node have the same scope.*

Definition 2.4 (Consistency)**.** *An SPN is consistent iff no variable appears with a value v in one child of a product node and valued u, with $u \neq v$, in another.*

Validity in an SPN means that the network correctly and efficiently computes the probability of evidence of the distribution it represents. In this document we only work with valid SPNs, as we wish to always compute exact inference. However, non-valid SPNs are an interesting field of research for approximate inference in SPNs.

A sufficient condition for validity is completeness and consistency. Yet whilst this condition is sufficient, it is not necessary, as the converse (i.e. an incomplete and inconsistent valid SPN) can hold.

Theorem 2.1 (Poon and Domingos, 2011)**.** *An SPN is valid if it is complete and consistent.*

When an SPN S is valid, then $S(*)$ is the partition function, and we can extract the probability of evidence from an SPN by computing $P(X = x) = S(x)/S(*)$. If for every sum node all of their weights are non-negative and sum to one, then the partition function is $S(*) = 1$, and the SPN is the distribution itself.

Corollary 2.1 (Validity recursion; Poon and Domingos, 2011)**.** *If an SPN S is valid, then* *all sub-SPN of S is valid.*

Definition 2.5 (Decomposability)**.** *An SPN is decomposable iff no variable appears in more than one child of a product node.*

In other words, an SPN is decomposable if and only if, for every product node, every child node has disjoint scopes with relation to all their other siblings. It is easy to see that decomposability implies in consistency, as there can be no inconsistency between product children since scopes are disjoint. Therefore, a complete and decomposable SPN is valid. Indeed it is much easier to produce decomposable SPNs then only consistent ones, and although this condition may seem too strong and restrictive, Peharz et al., 2015 showed that a consistent SPN is representable by a polynomially larger decomposable SPN.

So far, SPNs are restricted to the discrete domain, as we rely on IVs to define possible valuations to variables. We can generalize SPNs to the continuous by assuming an infinite number of IVs and thus replacing sum nodes whose children are IVs with integral nodes. A leaf node then becomes an integral node with infinite IVs as children. Particularly, it represents an unnormalized univariate probability distribution, such as a Gaussian. The value of this integral node n becomes the fdp $p_n(x)$. This extension brings us to a second definition of SPNs.

Definition 2.6 (Sum-product networks; Gens and Domingos, 2013)**.** *A sum-product network is defined recursively as follows.*

1. *A tractable univariate probability distribution is an SPN.*
2. *A product of SPNs with disjoint scopes is an SPN.*
3. *A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive.*
4. *Nothing else is an SPN.*

This second definition limits our scope to only complete and decomposable SPNs. Note that an IV is also an SPN, as we can assume that an indicator variable is a degenerate tractable univariate distribution, taking a value of one if it agrees with the given evidence and zero otherwise.

2.3 Inference

Throughout this thesis we assume that all sum nodes are normalized and sum to one, meaning the partition function is $S(*) = 1$ and the SPN's value is the probability itself.

Let $X = \{X_1 = x_1, X_2 = x_2, \ldots, X_k = x_k\}$ be a valuation and S be an SPN. We say that X is a complete valuation if $\mathrm{Sc}(X) = \mathrm{Sc}(S)$. That is, X contains a valuation for all variables in S. An incomplete valuation has some variable assignment missing.

Computing the probability of evidence is done through a bottom-up backwards pass through the SPN. To find the value of an SPN, we must know the value of the root node, which depends on all nodes below it. This is done through a topological traversal of the graph.

Finding the value of a leaf node depends on the valuation given. Let n be a leaf node, and $\mathrm{Sc}(n) = \{X_j\}$. Let X be some valuation. Assuming the univariate probability distribution of n has fpd $p_n(x)$, then if X has a valution $X_j = x_j$, the value of node n will be $S_n(X) = p_n(x_j)$. If X has no valuation for variable X_j, then $S_n(X)$ is the distribution's mode. Note that this holds for indicator variables, as if X has a valuation for $X_j = x_j$ and the IV matches with x_j, then $p_n(x_j) = 1$. In case it does not, $p_n(x_j) = 0$. For the incomplete case, the mode of an indicator variable is one, which holds the equivalence.

Once we compute leaf nodes, we can compute each internal node's values by following the topological order until we reach the root. For sum nodes, we compute the weighted sum $S_n(X) = \sum_{j \in \mathrm{Ch}(n)} w_{n,j} S_j(X)$, and for products $S_n(X) = \prod_{j \in \mathrm{Ch}(n)} S_j(X)$.

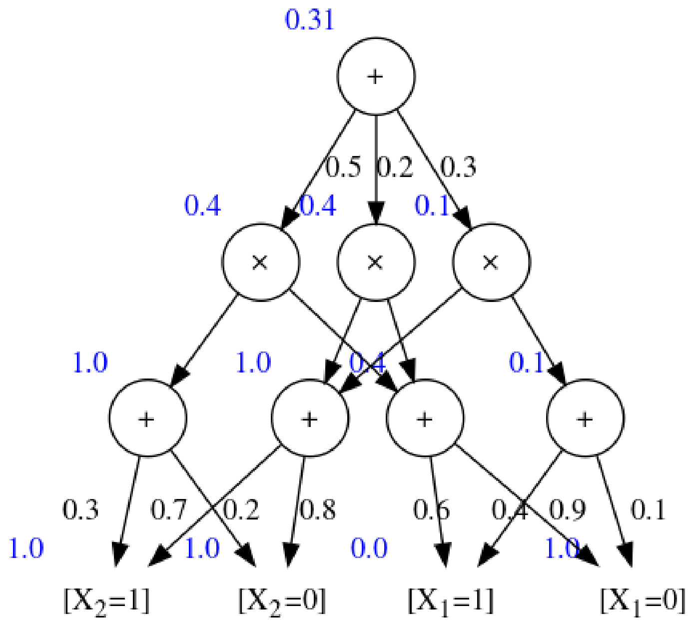


Figure 2.2: *Computing the probability of evidence on a sample SPN.*

Figure 2.2 shows the value of the SPN in Figure 2.1 given a valuation $X = \{X_1 = 0\}$. Values in blue are the values of each sub-SPN. Finding the probability of evidence $P(X = x)$ is fast, as computing the value of an SPN is linear to the number of edges of the graph.

Additionally, we might want to find the probability that maximizes a certain valuation, i.e. the maximum a posteriori probability (MAP). To compute the approximate value of the MAP of some valuation X, we first transform the SPN into a max-product network (MPN) by replacing all sums with max nodes. The value of a max node is the maximum value of its weighted children. More formally, the value of an MPN's max node n is given by $M_n(X) = \max_{j \in \mathrm{Ch}(n)} w_{n,j} M_j(X)$. Other nodes behave identically to an SPN. The computed value of an MPN is an approximation of $\max_y P(X = x, Y = y)$, where X is incomplete and Y is the set of variables that are missing. This is called the max-product algorithm.

In SPNs, computing the exact MAP was shown to be NP-hard (Peharz et al., 2015; Conaty et al., 2017; Mei et al., 2018), and better approximation algorithms were proposed as an alternative to the max-product algorithm described here. However, in this thesis, when we talk about computing the (approximate) MAP, we are referring to the usual max-product algorithm.

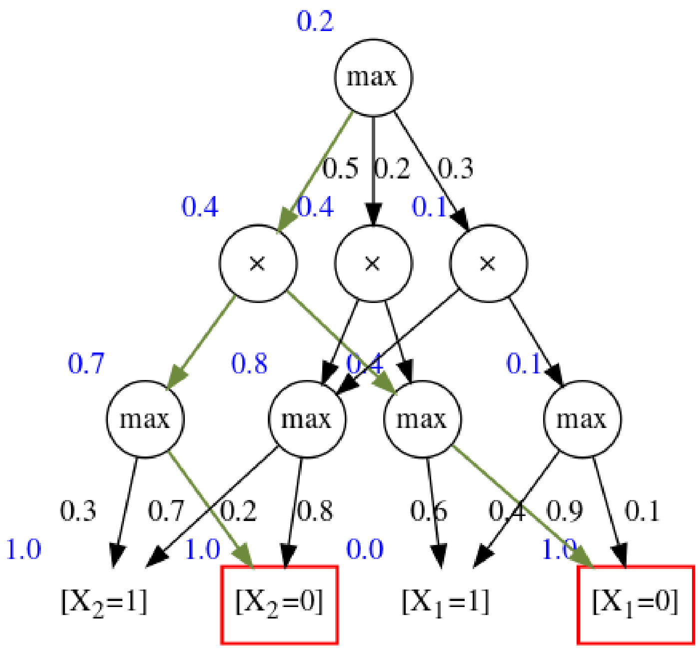


Figure 2.3: *Computing the approximate MAP of an SPN through its MPN.*

Once the MPN values are computed, we can find the most probable explanation (MPE) of the distribution given an evidence. This is done through a top-down forward pass, where we take a maximum sub-circuit path of the MPN by always taking the max path at a max node and taking all paths on a product node. The MPE are the maximum sub-circuit leaves' instantiations.

Figure 2.3 shows the MPN of the SPN shown in Figure 2.1 given $X = \{X_1 = 0\}$, where the numbers in blue represent the MPN values at each node, green arrows indicate the sub-circuit of maximum value and red boxes indicate the most probable valuations given evidence. The resulting MPE $\arg\max_{y \in \mathcal{Y}} P(X = \{X_1 = 0\}, Y = y)$ is the valuation $\{X_1 = 0, X_2 = 0\}$.

Therefore, computing the probability of evidence, which is also called *soft* inference, of an SPN is done through a single bottom-up pass. Similarly, computing the MAP probability, refered to as *hard* inference, is done through a bottom-up pass on the SPN's MPN. On the other hand, finding the MPE valuations requires a bottom-up pass to first compute the MAP, and then a top-down search to find the most probable instantiations.

We provide next pseudocode for computing both soft and hard inference. We assume as input only valid, (weight) normalized SPNs. However, one could easily extend the included algorithms for unnormalized networks.

Algorithm 1 `SoftInference`: Computes the probability of evidence in SPNs

Input A valid SPN S with normalized weights and a valuation X

Output The soft inference values at each node S_n

Initialize $S_n = 0$
Find topological order T of S
for each node $n \in S$ from T **do**
  if n is a leaf node **then**
    Let $\mathrm{Sc}(n) = \{X_k\}$, $p_n(x)$ be n's fdp and $\hat{p}_n$ be p_n's mode
    if $X_k \in X$ **then**
      Let x_k be X_k's value in X
      $S_n \leftarrow p_n(x_k)$
    else
      $S_n \leftarrow \hat{p}_n$
  else if n is sum node **then**
    for all $j \in \mathrm{Ch}(n)$ **do**
      $S_n \leftarrow S_n + w_{n,j} S_j$
  else
    for all $j \in \mathrm{Ch}(n)$ **do**
      $S_n \leftarrow S_n \cdot S_j$
return each S_n node value

Algorithm 2 `HardInference`: Computes an approximation of the MAP in SPNs

Input A valid SPN S with normalized weights and a valuation X

Output The hard inference values at each node M_n

Let M be S's MPN
Initialize $M_n = 0$
Find topological order T of M
for each node $n \in M$ from T **do**
  if n is a leaf node **then**
    Let $\mathrm{Sc}(n) = \{X_k\}$, $p_n(x)$ be n's fdp and $\hat{p}_n$ be p_n's mode
    if $X_k \in X$ **then**
      Let x_k be X_k's value in X
      $M_n \leftarrow p_n(x_k)$
    else
      $M_n \leftarrow \hat{p}_n$
  else if n is sum node **then**
    for all $j \in \mathrm{Ch}(n)$ **do**
      $M_n \leftarrow \max(M_n, w_{n,j} M_j)$
  else
    for all $j \in \mathrm{Ch}(n)$ **do**
      $M_n \leftarrow M_n \cdot M_j$
return each M_n node value

Finally, we show how to algorithmically compute the MPE given some evidence.

Algorithm 3 `ArgMaxSPN`: Finds the MPE of a valuation on an SPN

Input A valid SPN S with normalized weights and a valuation X

Output The arg max values of each variable according to X

```
M ← HardInference(S, X)
Let Y be a copy of X
Let Q be a queue
Push S into Q
for each node n ∈ M in Q do
    if n is a leaf node then
        Let Sc(n) = {X_k} and p_n(x) be n's fdp
        Let x̂ = arg max_{x_k} p_n(x_k) be p_n's maximum valuation
        if X_k ∉ X then
            Y ← Y ∪ {X_k = x̂}
    else if n is sum node then
        Push maximum child M_j, j ∈ Ch(n) into Q
    else
        Push all children j ∈ Ch(n) into Q
return Y
```

Chapter 3

Parameter learning

The objective of this chapter is to expose the ideas behind generative and discriminative gradient descent for parameter learning of sum-product networks. We first show how to derive the SPN with respect to its nods and weights so that we can find the gradient of the SPN wrt its parameters (i.e. weights). This allows us to find the weight updates needed for gradient descent on SPNs. We then describe how to perform generative stochastic gradient descent, and finally discriminative gradient descent.

3.1 Derivatives

Let S be an SPN. We are only interested in finding the derivative of internal nodes, as leaf nodes have no weights to be updated. Our objective is to find the gradient $\partial S/\partial W$ by computing each component $\partial S/\partial w_{n,j}$, allowing us to find each weight update on the SPN.

At each weighted edge $(n \rightarrow j, w_{n,j})$, the derivative $\partial S/\partial w_{n,j}$ takes the form

$$\frac{\partial S}{\partial w_{n,j}}(X) = \frac{\partial S}{\partial S_n}\frac{\partial S_n}{\partial w_{n,j}}(X) = \frac{\partial S}{\partial S_n}\frac{\partial}{\partial w_{n,j}}\left(\sum_{i \in \mathrm{Ch}(n)} w_{n,i}S_i(X)\right) = \frac{\partial S}{\partial S_n}S_j(X). \tag{3.1}$$

The term $\partial S/\partial S_n$ appears because of chain rule, since S_n is a function of S. This can be intuitively interpreted as taking into account the change in all nodes "above" n. So to compute the derivative wrt a weight, we need to find the derivative $\partial S/\partial S_j$ for each internal node j.

Finding $\partial S/\partial S_j$ requires analyzing two possible cases: sum and product parents of j. We now that S is a multilinear function of X, since in reality S is just a function made of sums and products. In particular, if we apply chain rule on $\partial S/\partial S_j$, we have that

$$\frac{\partial S}{\partial S_j}(X) = \underbrace{\sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{sum}}} \frac{\partial S}{\partial S_n}\frac{\partial S_n}{\partial S_j}(X)}_{(*)} + \underbrace{\sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{product}}} \frac{\partial S}{\partial S_n}\frac{\partial S_n}{\partial S_j}(X)}_{(**)}.$$

We expand each term at a time. Starting with the sum parents case, we can substitute the value of $S_n(X)$ with the corresponding expansion.

$$(*) = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{sum}}} \frac{\partial S}{\partial S_n}\frac{\partial}{\partial S_j}\left(\sum_{i \in \mathrm{Ch}(n)} w_{n,i} S_i(X)\right) = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{sum}}} \frac{\partial S}{\partial S_n} w_{n,j}$$

We do the same for the product case.

$$(**) = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{product}}} \frac{\partial S}{\partial S_n}\frac{\partial}{\partial S_j}\left(\prod_{i \in \mathrm{Ch}(n)} S_i(X)\right) = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{product}}} \frac{\partial S}{\partial S_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} S_k$$

Which brings us to the final form.

$$\frac{\partial S}{\partial S_j}(X) = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{sum}}} \frac{\partial S}{\partial S_n} w_{n,j} + \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \mathrm{product}}} \frac{\partial S}{\partial S_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} S_k \tag{3.2}$$

Note how each $\partial S/\partial S_j$ depends on the derivative of its parents. This dependency goes all the way up to the root, where $\partial S/\partial S = 1$. This derivation lends itself neatly to an algorithmic format.

Algorithm 4 `Backprop`: Backpropagation derivation on SPNs

Input A valid SPN S with pre-computed probabilities $S_n(X)$
Output Partial derivatives of S with respect to every node and weight

```
Initialize ∂S/∂S_n = 0 except ∂S/∂S = 1
for each node n ∈ S in top-down order do
    if n is sum node then
        for all j ∈ Ch(n) do
            ∂S/∂S_j ← ∂S/∂S_j + w_{n,j} ∂S/∂S_n
            ∂S/∂w_{n,j} ← ∂S/∂S_n S_j
    else
        for all j ∈ Ch(n) do
            ∂S/∂S_j ← ∂S/∂S_j + ∂S/∂S_n ∏_{k∈Ch(n)\{j}} S_k
```

Computing all derivatives and forward passes is fast, as it takes linear time in the number of edges. However, these values suffer from gradient diffusion, as their signal dwindles the deeper the network, eventually becoming zero.

A possible solution to this issue is replacing soft derivation with hard derivation. This is done by finding the derivatives of the MPN of the network instead of the SPN. This guarantees that the signal remains constant throughout the structure, at the cost of slower convergence rate. We call this hard inference derivation, as opposed to the regular soft inference derivation we covered earlier.

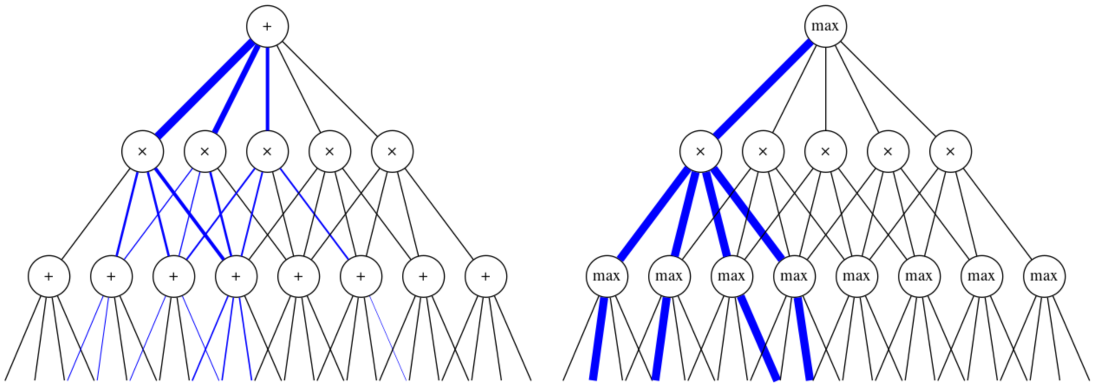


Figure 3.1: *Signal difference between soft and hard derivation.*

Figure 3.1 gives a visual representation of the difference between soft and hard derivation in gradient descent. MPNs preserve the signal, as the resulting gradient is constant.

To compute the hard derivatives of an SPN, we take its MPN and find its derivatives in a similar way as in soft derivation. Let M be an MPN. We shall call W the multiset of weights that a forward pass through M visits. The value of M is $M(X) = \prod_{w_i \in W} w_i^{c_i}$, where c_i is the number of times w_i appears in W. We can then take the logarithm of the MPN to end up with a friendlier expression.

$$\frac{\partial \log M}{\partial w_{n,j}} = \frac{\partial}{\partial w_{n,j}} \log \left(\prod_{w_i \in W} w_i^{c_i} \right) = \frac{1}{\prod_{w_i \in W} w_i^{c_i}} \cdot c_{n,j} w_{n,j}^{c_{n,j}-1} \cdot \prod_{w_i \in W \setminus \{w_{n,j}\}} w_i^{c_i}$$

If we assume that weights are strictly positive, the resulting expression results in the final hard derivative

$$\frac{\partial \log M}{\partial w_{n,j}} = c_{n,j} \frac{w_{n,j}^{c_{n,j}-1}}{w_{n,j}^{c_{n,j}}} = \frac{c_{n,j}}{w_{n,j}}. \tag{3.3}$$

Although not needed for the gradient, we can also compute the derivative in each internal node. The process is similar to soft derivation. There is no change for parent product nodes. For parent max nodes, we sum only contributions where $w_{n,j} \in W$.

$$\frac{\partial M}{\partial M_j} = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \max}} \begin{cases} w_{k,n} \frac{\partial M}{\partial M_k} & \text{if } w_{k,n} \in W \\ 0 & \text{otherwise} \end{cases} + \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \text{product}}} \frac{\partial M}{\partial M_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} M_k \tag{3.4}$$

Computing each derivative $\partial \log M / \partial w_{n,j}$ means finding each $c_{n,j}$ count at each weight. This is done through an initial forward pass on M to find each MAP, and then finding each maximal edge in W through a backwards pass. Algorithm 5 computes the total number of occurences $c_{n,j}$ for each maximal edge $w_{n,j}$.

Algorithm 5 `HardBackprop`: Hard backpropagation derivation on SPNs

Input A valid SPN S with pre-computed MAP probabilities $M_n(X)$
Output Counts $c_{n,j}$ of each derivative $\frac{\partial \log M}{\partial w_{n,j}}$

```
Initialize c_{n,j} = 0
Let Q be a queue
Push M into Q
for each node n ∈ M in queue Q do
    if n is max node then
        Let j = arg max_{i∈Ch(n)} w_{n,i} M_i(X) the maximum weighted child
        c_{n,j} ← c_{n,j} + 1
        Push M_j into Q
    else if n is product node then
        Push all children j ∈ Ch(n) into Q
return all counts c_{n,j}
```

In summary, the derivatives of an SPN with respect to its internal nodes take values according to Table 3.1. The gradient components are shown in Table 3.2.

Inference	Partial derivatives wrt internal node j
Soft	$\frac{\partial S}{\partial S_j} = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \text{sum}}} w_{n,j} \frac{\partial S}{\partial S_n} + \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \text{product}}} \frac{\partial S}{\partial S_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} S_k$
Hard	$\frac{\partial M}{\partial M_j} = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \max}} \begin{cases} w_{k,n} \frac{\partial M}{\partial M_k} & \text{if } w_{k,n} \in W, \\ 0 & \text{otherwise.} \end{cases} + \sum_{\substack{n \in \mathrm{Pa}(j) \\ n:\ \text{product}}} \frac{\partial M}{\partial M_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} M_k$

Table 3.1: *Partial derivatives for the SPN wrt internal nodes.*

Inference	Partial derivatives wrt weight $w_{n,j}$
Soft	$\dfrac{\partial S}{\partial w_{n,j}} = S_j \dfrac{\partial S}{\partial S_n}$
Hard	$\dfrac{\partial M}{\partial w_{n,j}} = M_j \dfrac{\partial M}{\partial M_n}$

Table 3.2: *Partial derivatives for the SPN wrt weights.*

3.2 Generative gradient descent

Once computed all derivatives, we update each node with the resulting gradient component. For generative gradient descent, where we are learning a joint probability distribution $P(X, Y)$, our objective is to find the gradient of the log-likelihood

$$\frac{\partial}{\partial W} \log P(X, Y) = \frac{\partial}{\partial W} \log S(X, Y) = \frac{1}{S(X, Y)} \frac{\partial S}{\partial W}(X, Y) \propto \frac{\partial S}{\partial W}(X, Y).$$

Since the gradient is proportional to the derivative of the weights, our weight update becomes

$$\Delta w_{n,j} = \eta \frac{\partial S}{\partial w_{n,j}}(X, Y),$$

where η is the learning rate. An L2 regularization factor can be added to the expression above, leaving us with the final generative gradient descent weight update

$$\Delta w_{n,j} = \eta \frac{\partial S}{\partial w_{n,j}}(X, Y) - 2\lambda w_{n,j}, \tag{3.5}$$

where λ is the regularization constant. We call this soft generative gradient descent. It is now easy to visualize why gradient diffusion occurs with soft derivation. Component $\partial S/\partial w_{n,j}$ depends on partial derivative $\partial S/\partial S_n$. Assuming normalized weights, the root node derivative $\partial S/\partial S = 1$ and each subsequent descendant node becomes smaller and smaller.

Weight update for hard derivation comes directly from Equation 3.3. Since we are interested in the log-likelihood of the joint distribution

$$\frac{\partial}{\partial W} \log P(X, Y) = \frac{\partial}{\partial W} \log M(X, Y),$$

we get, for each component $w_{n,j}$, the weight update

$$\Delta w_{n,j} = \eta \frac{c_{n,j}}{w_{n,j}}.$$

In a similar fashion to soft generative gradient descent, we can apply L2 regularization to each weight update.

$$\Delta w_{n,j} = \eta \frac{c_{n,j}}{w_{n,j}} - 2\lambda w_{n,j} \tag{3.6}$$

So for generative gradient descent we get the following weight updates.

Inference	**Weight updates**
Soft	$\Delta w_{n,j} = \eta \frac{\partial S}{\partial w_{n,j}}(X, Y) - 2\lambda w_{n,j}$
Hard	$\Delta w_{n,j} = \eta \frac{c_{n,j}}{w_{n,j}} - 2\lambda w_{n,j}$

Table 3.3: *Generative gradient descent weight updates with L2 regularization.*

Algorithm 6 and Algorithm 7 show pseudocode for both soft and hard generative stochastic gradient descent, though it is easy to extend both to mini-batch versions. From now on we denote soft generative gradient descent and hard generative gradient descent as SGGD and HGGD for short.

Algorithm 6 `SoftGenGD`: Soft generative stochastic gradient descent for SPNs

Input A valid SPN S, learning rate η, regularization constant λ and a dataset D
Output S with learned weights

```
repeat
    for each instance I ∈ D do
        Compute SoftInference(S, I)
        Compute Backprop(S)
        for each sum node n ∈ S do
            w_{n,j} ← η ∂S/∂w_{n,j} − 2λ w_{n,j}
        Normalize weights
until convergence
```

Algorithm 7 HardGenGD: Hard generative stochastic gradient descent for SPNs

Input A valid SPN S, learning rate η, regularization constant λ and a dataset D
Output S with learned weights

```
repeat
    for each instance I ∈ D do
        Compute HardInference(S, I)
        Compute HardBackprop(S)
        for each sum node n ∈ S do
            w_{n,j} ← η c_{n,j}/w_{n,j} − 2λ w_{n,j}
        Normalize weights
until convergence
```

3.3 Discriminative gradient descent

The goal of discriminative learning is optimizing the conditional probability distribution $P(Y|X)$, where Y and X are query and evidence variables. To compute the gradient of this distribution we maximize the conditional log-likelihood (Gens and Domingos, 2012).

$$\frac{\partial}{\partial W}\log P(Y|X) = \frac{\partial}{\partial W}\log\left(\frac{P(Y,X)}{P(X)}\right) = \frac{\partial}{\partial W}\log P(Y,X) - \frac{\partial}{\partial W}\log P(X)$$

Through chain rule, we get the form

$$\begin{aligned}\frac{\partial}{\partial W}\log P(Y,X) - \frac{\partial}{\partial W}\log P(X) &= \frac{1}{P(Y,X)}\frac{\partial}{\partial W}P(Y,X) - \frac{1}{P(X)}\frac{\partial}{\partial W}P(X)\\ &= \frac{1}{S(Y,X)}\frac{\partial}{\partial W}S(Y,X) - \frac{1}{S(X)}\frac{\partial}{\partial W}S(X).\end{aligned}$$

We can update our weights discriminatively by taking each gradient component

$$\Delta w_{n,j} = \eta\left(\frac{1}{S(Y,X)}\frac{\partial S(Y,X)}{\partial w_{n,j}} - \frac{1}{S(X)}\frac{\partial S(X)}{\partial w_{n,j}}\right).$$

With L2 regularization, soft discriminative gradient descent has the following form.

$$\Delta w_{n,j} = \eta\left(\frac{1}{S(Y,X)}\frac{\partial S(Y,X)}{\partial w_{n,j}} - \frac{1}{S(X)}\frac{\partial S(X)}{\partial w_{n,j}}\right) - 2\lambda w_{n,j} \tag{3.7}$$

For hard inference we want to optimize the following expression.

$$\frac{\partial}{\partial W} \log \tilde{P}(Y|X) = \frac{\partial}{\partial W} \log \left(\frac{\tilde{P}(Y,X)}{\tilde{P}(X)} \right) = \frac{\partial}{\partial W} \log \left(\frac{M(Y,X)}{M(X)} \right)$$

Where $\tilde{P}$ is the MAP probability of the distribution. As usual, we apply chain rule, yielding

$$\frac{\partial}{\partial W} \log \left(\frac{M(Y,X)}{M(X)} \right) = \frac{\partial}{\partial W} \log M(Y,X) - \frac{\partial}{\partial W} \log M(X).$$

But we know from Equation 3.3 that the derivatives of the logs have a particular expression based on the counts of visited weights. We substitute the equation above with the earlier results from hard derivation, giving us the following equation for each gradient component.

$$\frac{\partial}{\partial w_{n,j}} \log \left(\frac{M(Y,X)}{M(X)} \right) = \frac{\partial}{\partial w_{n,j}} \log M(Y,X) - \frac{\partial}{\partial w_{n,j}} \log M(X) = \frac{\Delta c_{n,j}}{w_{n,j}}$$

Where $\Delta c_{n,j}$ is the difference between the first counting, restricted to (Y,X), and the second restricted to only X.

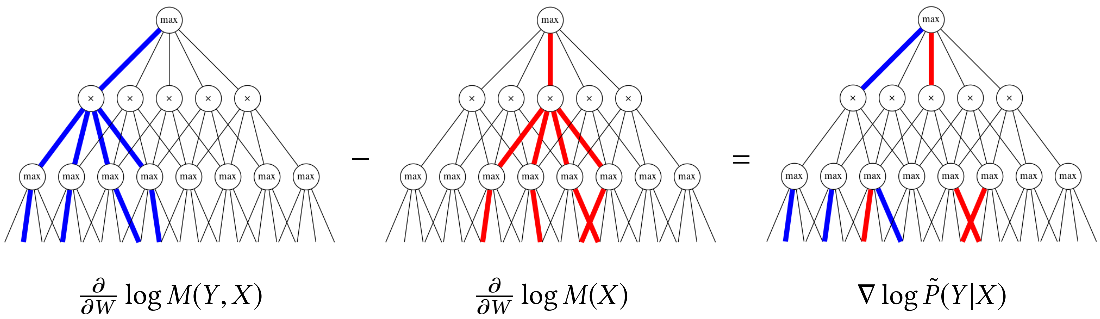


Figure 3.2: *Hard discriminative gradient descent counts visualization.*

Figure 3.2 shows the hard discriminative gradient descent difference derived from $\frac{\partial}{\partial W} \log \tilde{P}(Y|X)$. The first pass, shown in the image with blue edges, counts the maximum edges given the Y, X valuation. The second pass, in red, is the evidence pass on X. The gradient is then computed by finding the difference between the two countings. On the right-hand side of the expression portrayed in Figure 3.2, blue edges mean a positive count $c_{n,j}$ and red edges represent a negative count. Edges coming out from product nodes are not colored, as they are not weighted.

The actual weight update has a similar form to hard gradient descent.

$$\Delta w_{n,j} = \eta \frac{\Delta c_{n,j}}{w_{n,j}}$$

With L2 regularization we get

$$\Delta w_{n,j} = \eta \frac{\Delta c_{n,j}}{w_{n,j}} - 2\lambda w_{n,j}. \tag{3.8}$$

In a similar fashion to generative gradient descent, we denote by HDGD and SDGD hard discriminative gradient descent and soft discriminative gradient descent respectively.

We now build a discriminative gradient descent table for each inference type. Just like in generative gradient descent, we add an L2 term to it.

Inference	Weight updates
Soft	$\Delta w_{n,j} = \eta \left(\frac{1}{S(Y,X)} \frac{\partial S(Y,X)}{\partial w_{n,j}} - \frac{1}{S(X)} \frac{\partial S(X)}{\partial w_{n,j}} \right) - 2\lambda w_{n,j}$
Hard	$\Delta w_{n,j} = \eta \frac{\Delta c_{n,j}}{w_{n,j}} - 2\lambda w_{n,j}$

Table 3.4: *Discriminative gradient descent weight updates with L2 regularization.*

We now finally show an algorithmic form to HDGD and SDGD. Note how in discriminative gradient descent we have two passes through the network. We can avoid recomputing node values by memoizing nodes that have no query variables in descendant's scopes (Gens and Domingos, 2012).

Algorithm 8 `SoftDiscGD`: Soft discriminative stochastic gradient descent for SPNs

Input A valid SPN S, query variables Y, learning rate η, regularization constant λ and a dataset D

Output S with learned weights

repeat
    for each instance $I \in D$ **do**
        Compute `SoftInference`(S, I) and store them in S_n^+ for each node n
        Compute `Backprop`(S^+) and store them in $\frac{\partial S_n^+}{\partial w_{n,j}}$
        Compute `SoftInference`$(S, I \setminus Y)$ and store them in S_n^- for each node n
        Compute `Backprop`(S^-) and store them in $\frac{\partial S_n^-}{\partial w_{n,j}}$
        for each sum node $n \in S^- \cup S^+$ **do**
            $w_{n,j} \leftarrow \eta \left(\frac{1}{S^+} \frac{\partial S^+}{\partial w_{n,j}} - \frac{1}{S^-} \frac{\partial S^-}{\partial w_{n,j}} \right) - 2\lambda w_{n,j}$
        Normalize weights
until convergence

Algorithm 9 `HardDiscGD`: Hard discriminative stochastic gradient descent for SPNs

Input A valid SPN S, query variables Y, learning rate η, regularization constant λ and a dataset D

Output S with learned weights

repeat
    for each instance $I \in D$ **do**
        Compute `HardInference`(S, I) and store them in M_n^+ for each node n
        Compute `HardBackprop`(M^+) and store them in $c_{n,j}^+$
        Compute `HardInference`$(S, I \setminus Y)$ and store them in M_n^- for each node n
        Compute `HardBackprop`(M^-) and store them in $c_{n,j}^-$
        for each sum node $n \in S$ **do**
            $w_{n,j} \leftarrow \eta \left(\frac{c_{n,j}^+ - c_{n,j}^-}{w_{n,j}} \right) - 2\lambda w_{n,j}$
        Normalize weights
until convergence

Chapter 4

Structure learning

In this chapter we cover two structure learning algorithms we use for image classification. The first is based on Dennis and Ventura, 2012. The second is a variation of Gens and Domingos, 2013's structure learning schema. Once we cover both algorithms, we explain how we add a slight modification to the first architecture. We have empirically found that this change increased image classification accuracy significantly. We call this the "classification architecture".

4.1 The Dennis-Ventura architecture

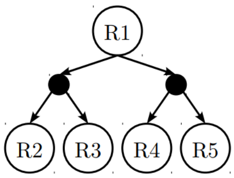
Let us first formalize the notion of dataset. We call a dataset a set of *instances*, where each instance is a set we call *valuation* or *instantiation*. As we have mentioned before, a valuation may be incomplete, meaning that an instantiation of some random variable may be missing from the instance. In this thesis we assume complete data, as both structure learning algorithms do not admit incomplete datasets.

Having said that, let D be a complete dataset. Since D is complete, for each instance I we can map each random variable X from I to a number, yielding an ordered vector $(X_1, X_2, \ldots, X_m)$ equivalent to I. We do the same for each instance I. The vector $(I_1, I_2, \ldots, I_n)$ is a representation of D. This way, D could be seen as a $m \times n$ matrix. We denote by $D^\intercal$ the transpose of the matrix representation of the dataset D. Let T be a subscope, that is, a subset of the set of all variables in the SPN. We use the notation D_T to represent the matrix of all instances from D but restricted only to elements from random variables in T.

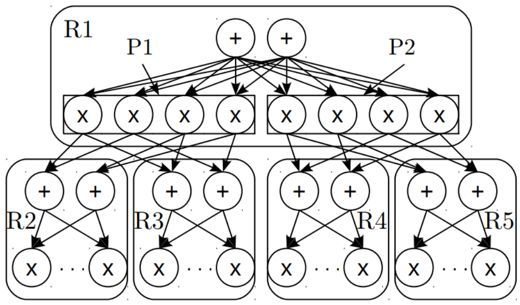
Since we are restricted to the image classification domain, we give some semantic meaning to datasets. If D is a dataset, then each instance $I \in D$ can be seen as a vector containing all the pixel values of an image plus a classification label. Each variable is a pixel from the image, and each variable value is the pixel's color intensity. If the dataset D is a vector of images and their labels, the transpose $D^\intercal$ is a vector of variables and their values in each image.

Just like in Poon and Domingos, 2011, the Dennis-Ventura algorithm uses the notion of similarity between local variables. This local neighborhood is called a Region. A Region represents a cluster of pixels that has some semantic value when grouped together. Contrastingly, a Decomposition represents independence between variables. In an SPN, a Region is graphically represented by a set of sum nodes, whilst a Decomposition is a set of products.

To learn an SPN structure from data, Dennis and Ventura, 2012 uses a *region graph*, which is a simplified representation of an SPN made out of Region nodes and Decomposition nodes.



(a)



(b)

Figure 4.1: *Dennis-Ventura region graph and translated SPN as shown in* Dennis and Ventura, 2012.

The region graph is generated by recursively finding two subregions from a parent region through the use of k-means clustering. Let R be a region, and $D_R^\intercal$ the transposed dataset restricted to R's scope. We partition R into two subregions R_1 and R_2 by k-means clustering $D_R^\intercal$, yielding two subclusters $D_{R_1}^\intercal$ and $D_{R_2}^\intercal$. We then apply recursion on the two subregions. At each clustering step, we connect regions R to a decomposition node P, which is then connected to each R_i node. Our stop criteria is when $\mathrm{Sc}(R_i) = 1$. The root node is a special case. We run k-means cluster on D, and for each D_i cluster, we construct a sub-SPN for each root child with D_i.

Once created, the region graph is then translated to a valid SPN. Each region node R is translated to a set of SPN nodes. If $\mathrm{Sc}(R) = 1$, then these nodes are g univariate gaussian distributions, where each gaussian is a different quantile of the distribution of the pixel. Else, m sum nodes are created. Partition nodes are translated to product nodes. Edges are added such that every product child node of a region is connected to all sum nodes in the region. Each of these product nodes are then connected to a distinct pair of sum nodes from both region's children subregions, meaning that each decomposition node contains 2^m products.

With the architecture done, we apply parameter learning on the SPN to learn weights. This is done either through gradient descent or EM. In this thesis we apply generative and discriminative gradient descent to the architecture.

4.2 The Gens-Domingos schema

In Gens and Domingos, 2013, Gens and Domingos describe a flexible schema for structure learning of SPNs. The schema is based on the idea that the scope in a sum node's child mean that these variables are similar (and by consequence dissimilar to the variables in other sibling nodes' scopes), and that variables in a product's child mean that variables are dependent of each other (and analogally to sums, independent of other siblings).

This interpretation of SPNs yields an adaptable and open schema of learning. Sum nodes are created through clustering, as each cluster has some similarity aspect given some metric. Meanwhile, product nodes are created through statistical variable independence algorithms. When the scope of this partitioning of variables is one, we create a univariate distribution over the partitioned dataset.

Algorithm 10 `GensArch`: Gens-Domingos structure learning schema

Input Set of instances D and scope X
Output SPN structure learned from D and X

if $|X| = 1$ **then**
  return univariate distribution over D_X
else
  Partition X into $P_1, P_2, \ldots, P_m$ such that every P_i is independent of P_j, $i \neq j$
  if $m > 1$ **then**
    Let π be a new product node
    for $i \leftarrow 1, \ldots, m$ **do**
      $p_i \leftarrow$ `GensArch`(D_{P_i}, P_i)
      π.`AddChild`(p_i)
    return π
  else
    Cluster D such that $Q_1, Q_2, \ldots, Q_n$ are D's clusters
    Let σ be a new sum node
    for $i \leftarrow 1, \ldots, n$ **do**
      $s_i \leftarrow$ `GensArch`(Q_i, X)
      $w \leftarrow |Q_i|/|D|$
      σ.`AddChild`(s_i, w) ▷ w is edge $\sigma \rightarrow s_i$'s weight
    return σ

Our implementation was done by using DBSCAN, a density based clustering algorithm that automatically decides the number of clusters to generate (Ester et al., 1996), for clustering and the traditional G-test for variable independence. We additionally implemented k-means, k-mode and k-median for clustering and Pearson's χ^2-square for independence testing. However, we found that DBSCAN yielded the best clustering results and G-test the best variable independence.

For variable independence, instead of testing every variable pairwise, we constructed a dependency graph. Each vertex from the dependency graph represents a variable. An edge between two vertices means the two variables are dependent. To find independent partitions in a dataset it suffices to find a spanning tree of the dependency graph. We do this through Kruskal's MST union-find algorithm. The resulting connected components of the spanning tree are the partitions we wish to find. This significantly reduces complexity. However, we have found that it still accounts for approximately 90% of the algorithm's runtime.

We speculate that a better approach to variable indepedence would be finding approximate spanning trees on the graph. Many independence tests resulted in a completely dependent graph, but with cuts that could possibly yield better accuracy and runtime performance.

Gens-Domingos learned structures generated very deep and expressive SPNs, resulting in good accuracy results. However, the algorithm only generates SPTs, as once each step (either clustering or variable independence) is concluded, the function never returns to the same node.

4.3 The classification architecture

The Dennis-Ventura structure learning algorithm is able to model classification problems by partitioning data into l clusters, where l is the total number of labels, and assigning a sub-SPN for each cluster. One can interpret each sub-SPN as a model of each label. However, clustering may not select the right classification instances for each label, as we have no control over which labels fit each cluster. This effect is intensified on datasets containing a large number of data.

We try to solve this problem by simplifying the model. Instead of generating sub-SPNs through clustering, we restrict each label to its own SPN. In our architecture, we create a single sum node as root, representing the image and its classification label. For each label l, we construct a sub-SPN S_l such that the SPN is still valid. This is done by assigning a product node as S_l's root. Let Y be the classification variable. Each of these products are then connected to an indicator variable $[Y = y_l]$ and a sub-SPN restricted to only data where $Y = y_l$.

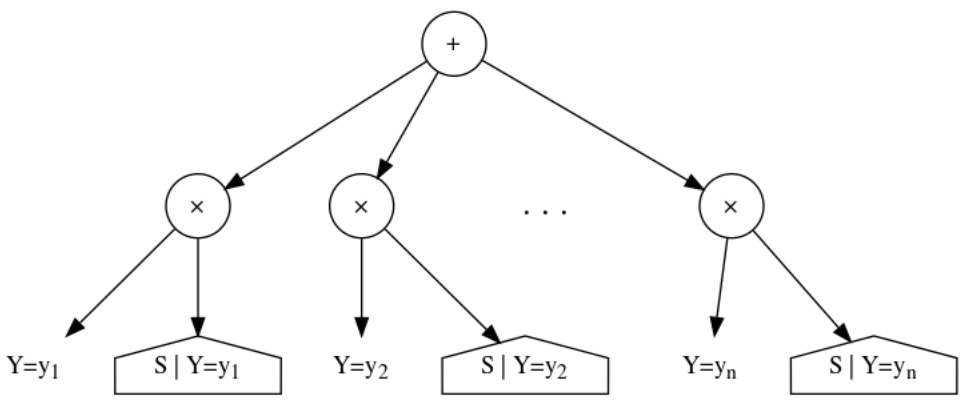


Figure 4.2: *The classification architecture for the Dennis-Ventura structure.*

Figure 4.2 shows the graphical representation of the classification architecture. The SPN is still valid, as the product node guarantees decomposability and the root sum node is complete. Each sub-SPN $S|Y = y_i$ is then constructed with the Dennis-Ventura algorithm, but restricted to data with the $Y = y_i$ valuation.

In practice, this architecture yielded much better results than the original clustering method. Furthermore, it is possible to easily parallelize each $S|Y = y_i$ learning procedures for faster learning runtime. Similarly, since each $S|Y = y_i$ is independent of other restricted SPNs, we can compute SPN values concurrently, allowing for faster inference. We cover this more thoroughly in chapter 5.

Interestingly, when this architecture model was applied to the Gens-Domingos algorithm, accuracy decreased. This is possibly due to the limitations of trees in SPNs. Another possible reason is that the Gens-Domingos better captures interactions between labels and pixels than only between pixels.

Chapter 5

Modelling the problem

Bibliography

[Conaty et al. 2017] Diarmaid Conaty, Denis D. Mauá, and Cassio P. de Campos. "Approximation Complexity of Maximum A Posteriori Inference in Sum-Product Networks". In: *Uncertainty in Artificial Intelligence 2017 (UAI 2017)* (2017) (cit. on p. 7).

[Darwiche 2003] Adnan Darwiche. "A Differential Approach to Inference in Bayesian Networks". In: (2003) (cit. on p. 3).

[Dennis and Ventura 2012] Aaron Dennis and Dan Ventura. "Learning the Architecture of Sum-Product Networks Using Clustering on Variables". In: *Advances in Neural Information Processing Systems* 25 (2012) (cit. on pp. 21, 22).

[Ester et al. 1996] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD-96 Proceedings* (1996) (cit. on p. 23).

[Gens and Domingos 2012] Robert Gens and Pedro Domingos. "Discriminative Learning of Sum-Product Networks". In: *Advances in Neural Information Processing Systems 25 (NIPS 2012)* (2012) (cit. on pp. 17, 19).

[Gens and Domingos 2013] Robert Gens and Pedro Domingos. "Learning the Structure of Sum-Product Networks". In: *International Conference on Machine Learning* 30 (2013) (cit. on pp. 4, 6, 21, 23).

[Mei et al. 2018] Jun Mei, Yong Jiang, and Kewei Tu. "Maximum a Posteriori Inference in Sum-Product Networks". In: *Association for the Advancement of Artificiall Intelligence 32 (AAAI 2018)* (2018) (cit. on p. 7).

[Poon and Domingos 2011] Hoifung Poon and Pedro Domingos. "Sum-Product Networks: A New Deep Architecture". In: *Uncertainty in Artificial Intelligence* 27 (2011) (cit. on pp. 4, 5, 21).

[Pearl 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988 (cit. on p. 3).

[Peharz et al. 2015] Robert Peharz, Sebastian Tschiatschek, Frank Pernkopf, and Pedro Domingos. "On Theoretical Properties of Sum-Product Networks". In:

International Conference on Artificial Intelligence and Statistics 18 (AISTATS 2015) (2015) (cit. on pp. 6, 7).