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 more than 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 love and laughter, and I deeply cherish every each one of them.

A warm thanks to my friends and colleagues Ricardo and Yan, whose camaraderie and friendship are invaluable to me.

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 $X = x$
$\mathrm{Pa}(n)$	Set of parent nodes of node n
$\mathrm{Ch}(n)$	Set of child nodes of node n
$\mathrm{Val}(X)$	Set of possible values variable X could take

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 do not 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 Chapter 5, we first show how we model self-driving as an image classification problem. We then give a brief explanation on the dataset used for training and testing. We then formalize how we extract inference for self-driving.

Chapter 6 is dedicated to explaining the hardware aspects of this work. We describe the processing unit used for inference and the unit used for handling the motors. We also explain how communication between the two is done.

How we pre-processed images before learning and inference is explained in Chapter 7. We describe each image transformation and how they affected performance.

In Chapter 8, we show accuracy results and timings on training and inference when using different SPN architectures and weight learning methods.

We finally implement and show results of the self-driving robot on a real world application in Chapter 9.

Finally, in Chapter 10 we give our conclusions and provide some discussion of the results.

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.

2.1 Background

Probabilistic modelling attempts to represent interactions between variables as a probability distribution. The objective of probabilistic models is to compactly represent a distribution, be able to find a good approximation to the real function, and efficiently compute both the marginals and modes. By quantifying uncertainty through data, we are able to predict events by looking at past observations. We model this through random variables (RVs), finding correlations and statistical independencies between variables in order to extract new information from the distribution that encompasses the entire scope of variables. Through Bayesian inference, a probabilistic model is able to infer uncertainties and update its belief accordingly.

Probabilistic graphical models (PGMs) attempt to model 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 indicator variables (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 (Darwiche, 2003). 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 directed acyclic graph. 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 products or weighted sums. Each edge coming out from a sum node n to another*

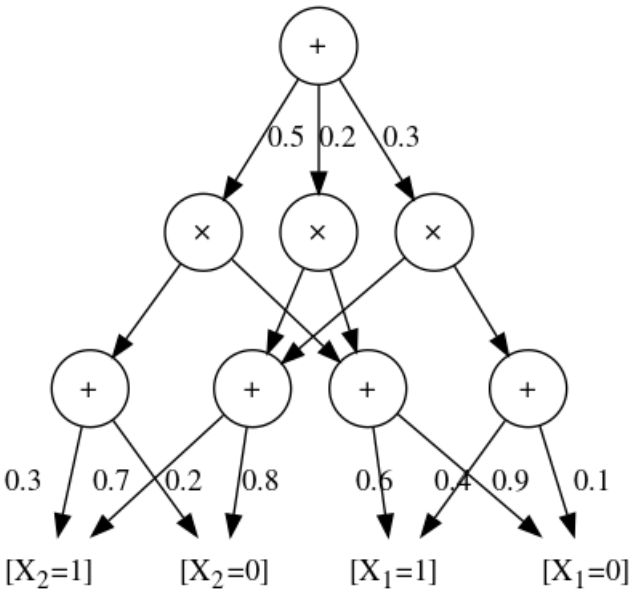

Figure 2.1: *An example of an SPN.*

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.

Throughout this thesis, we denote by $S(X = x)$, or simply by $S(X)$, the value of an SPN S given evidence x. A sub-SPN S_n of S is the subgraph of S rooted at node 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 $\mathrm{Sc}(S)$, 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 be true.

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 consistency, as there can be no inconsistency between product children since scopes are disjoint. Therefore, a complete and decomposable SPN is valid, as it is already consistent. Indeed it is much easier to produce decomposable SPNs than purely consistent ones. Although this condition may seem strong and restrictive, Robert Peharz et al., 2015 showed that a consistent SPN is representable by a polynomially larger decomposable SPN, meaning representability power is not lost on adding decomposability.

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, this represents an unnormalized univariate probability distribution, such as a Gaussian. The value of this integral node n becomes the pdf $p_n(x)$. 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 $\text{Sc}(n) = \{X_j\}$. Let X be some valuation. Assuming the univariate probability distribution of n has pdf $p_n(x)$, then if X has a valuation $X_j = x_j$, 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 value by following the topological order until we reach the root. For sum nodes, we compute the weighted sum $S_n(X) = \sum_{j \in \text{Ch}(n)} w_{n,j} S_j(X)$, and for products $S_n(X) = \prod_{j \in \text{Ch}(n)} S_j(X)$.

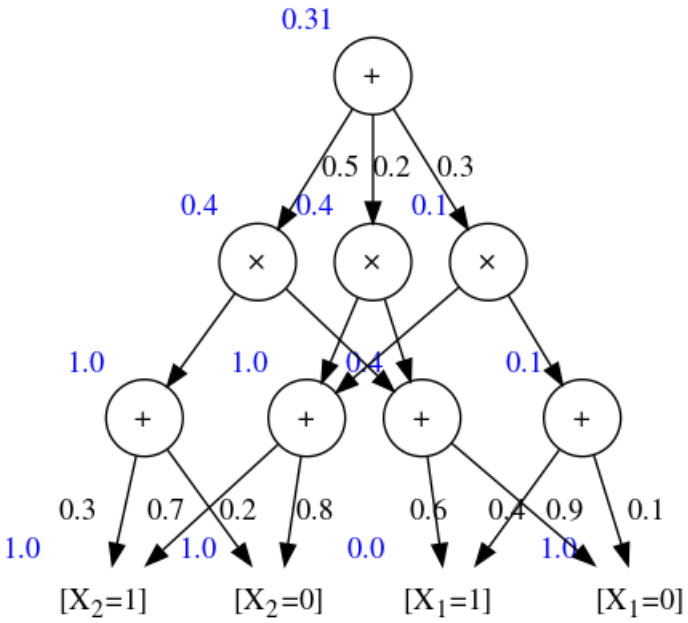

Figure 2.2: *Computing the probability of evidence on a sample SPN.*

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 $M_n(X) = \max_{j \in \text{Ch}(n)} w_{n,j} M_j(X)$. 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 (Robert 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.

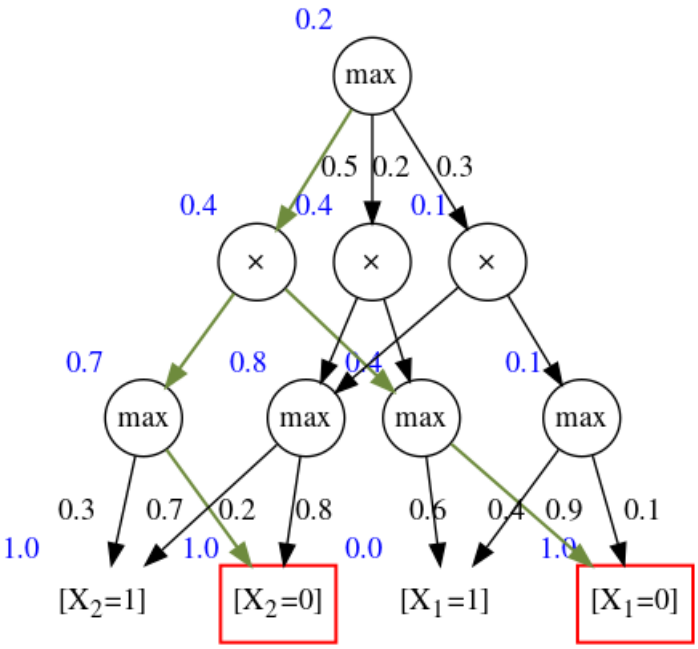

Figure 2.3: *Computing the approximate MAP of an SPN through its MPN.*

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 is 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.

Next, we provide 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 pdf 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 $\text{Sc}(n) = \{X_k\}$, $p_n(x)$ be n's pdf 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 $j \in \text{Ch}(n)$ **do**
            $M_n \leftarrow \max(M_n, w_{n,j} M_j)$
    else
        for all $j \in \text{Ch}(n)$ **do**
            $M_n \leftarrow M_n \cdot M_j$
return each M_n node value

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

$M \leftarrow$ `HardInference`(S, X)
Let Y be a copy of X
Let Q be a queue
Push S into Q
for each node $n \in M$ in Q **do**
    if n is a leaf node **then**
        Let $\text{Sc}(n) = \{X_k\}$ and $p_n(x)$ be n's pdf
        Let $\hat{x} = \arg\max_{x_k} p_n(x_k)$ be p_n's maximum valuation
        if $X_k \notin X$ **then**
            $Y \leftarrow Y \cup \{X_k = \hat{x}\}$
    else if n is sum node **then**
        Push maximum child M_j, $j \in \text{Ch}(n)$ into Q
    else
        Push all children $j \in \text{Ch}(n)$ 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 nodes 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.

The results presented in this chapter follow the derivations from both Poon and Domingos, 2011 for generative gradient descent, and Gens and Domingos, 2012 for discriminative gradient descent. In both, the authors present only a brief sketch of proof for deriving the gradient. In this chapter, we explain these derivations in detail and in a step-by-step manner.

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 $(n \to j, w_{n,j})$, 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 know that S is a multilinear function of X, since in reality S is just a function made out

of sums and products. In particular, if we apply chain rule on $\partial S/\partial S_j$, we have that

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

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.

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

Here we are computing the derivative $\partial/\partial S_j$ of each weighted sibling of j, counting j itself. Since no sibling i depends on j, its derivative wrt S_j is zero, leaving the case when $i = j$, which is trivially equal to $w_{n,j}$. We do the same for the product case.

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

In this expansion, we simply find the derivative of the product of siblings of j and j itself. This can be seen as the derivative of a variable multiplied by a constant, which gives us the constant, in this case the product of siblings. This brings us to the final form.

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

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 $\frac{\partial S}{\partial S_n} = 0$ except $\frac{\partial S}{\partial S} = 1$
for each node $n \in S$ in top-down order **do**
  if n is sum node **then**
    for all $j \in \mathrm{Ch}(n)$ **do**
      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + w_{n,j} \frac{\partial S}{\partial S_n}$
      $\frac{\partial S}{\partial w_{n,j}} \leftarrow \frac{\partial S}{\partial S_n} S_j$
  else
    for all $j \in \mathrm{Ch}(n)$ **do**
      $\frac{\partial S}{\partial S_j} \leftarrow \frac{\partial S}{\partial S_j} + \frac{\partial S}{\partial S_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} S_k$

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.

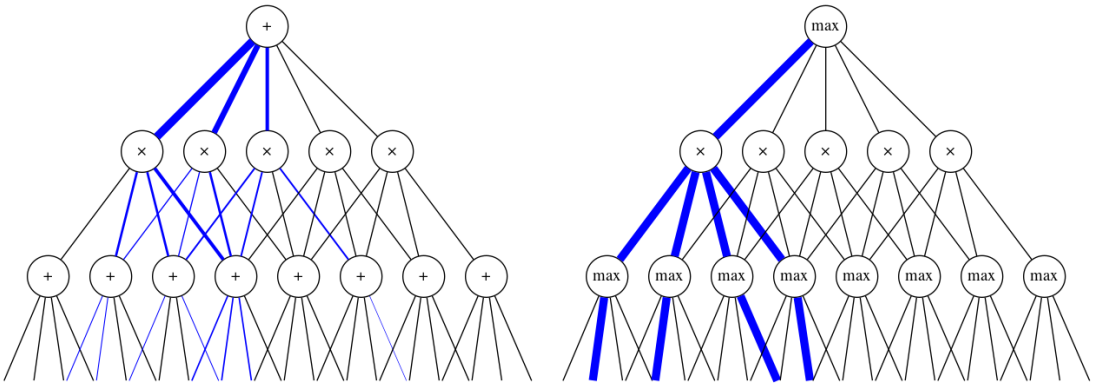

Figure 3.1: *Signal difference between soft and hard derivation.*

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 yields the following expression for 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 $w_{n,j} \in W$.

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

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 $n \in M$ in queue Q **do**
    if n is max node **then**
        Let $j = \arg\max_{i \in \mathrm{Ch}(n)} w_{n,i} M_i(X)$ the maximum weighted child
        $c_{n,j} \leftarrow c_{n,j} + 1$
        Push M_j into Q
    else if n is product node **then**
        Push all children $j \in \mathrm{Ch}(n)$ 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	$\frac{\partial M}{\partial M_j} = \sum_{\substack{n \in \mathrm{Pa}(j) \\ n\text{: max}}} \begin{cases} w_{k,n} \frac{\partial M}{\partial M_k} & \text{if } w_{k,n} \in W, \\ 0 & \text{otherwise.} \end{cases} + \sum_{\substack{n \in \mathrm{Pa}(j) \\ n\text{: product}}} \frac{\partial M}{\partial M_n} \prod_{k \in \mathrm{Ch}(n) \setminus \{j\}} M_k$

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

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

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 is $\partial S/\partial S = 1$ 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 $I \in D$ **do**
    Compute `SoftInference`(S, I)
    Compute `Backprop`(S)
    for each sum node $n \in S$ **do**
      $w_{n,j} \leftarrow \eta \frac{\partial S}{\partial w_{n,j}} - 2\lambda w_{n,j}$
    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 $I \in D$ **do**
    Compute `HardInference`(S, I)
    Compute `HardBackprop`(S)
    for each sum node $n \in S$ **do**
      $w_{n,j} \leftarrow \eta \frac{c_{n,j}}{w_{n,j}} - 2\lambda w_{n,j}$
    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.

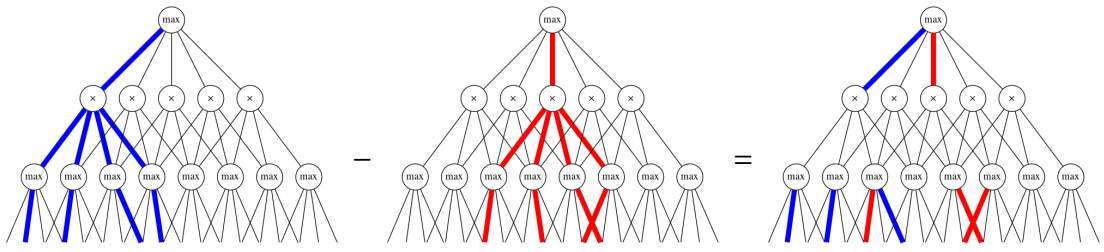

Figure 3.2: *Hard discriminative gradient descent counts visualization.*

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 an $m \times n$ 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 use a *region graph*, which is a simplified representation of an SPN made out of Region nodes and Decomposition nodes.

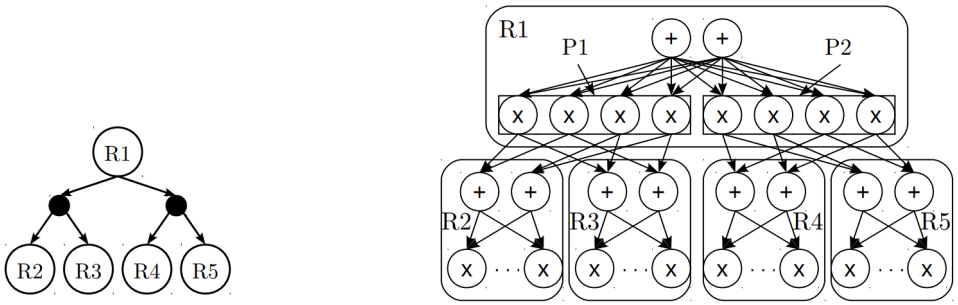

Figure 4.1: *Dennis-Ventura region graph (a) and translated SPN (b) as shown in Dennis and Ventura, 2012.*

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 interpretation that completeness

in a sum node means a child node's variables are similar (and by consequence dissimilar to the variables in other sibling nodes' scopes), and that variables in a product child's scope are dependent of each other (and thus 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 also implemented k-means, k-mode and k-median for clustering and Pearson's χ^2-square for independence testing. We found that DBSCAN yielded the best accuracy results on clustering, but took a long time for training. Using k-means clustering with $k = 2$ yielded worse, but comparable results, but with the upside of being fast to train. The G-test provided the best variable independence results.

Finding independent variable subsets can be done by iteratively comparing variables pairwise. The connected components of the resulting disconnected graph are independent subsets. This brute force approach is intractable, as the G-test already takes $O(|X||Y|)$ time, where X and Y are random variables and $|\cdot|$ indicates the number of categories of the RV, and testing each pair of variables exhaustively takes exponential time.

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.

Our Gens-Domingos implementation 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 initially partitioning data into l clusters 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 $Y = y_l$.

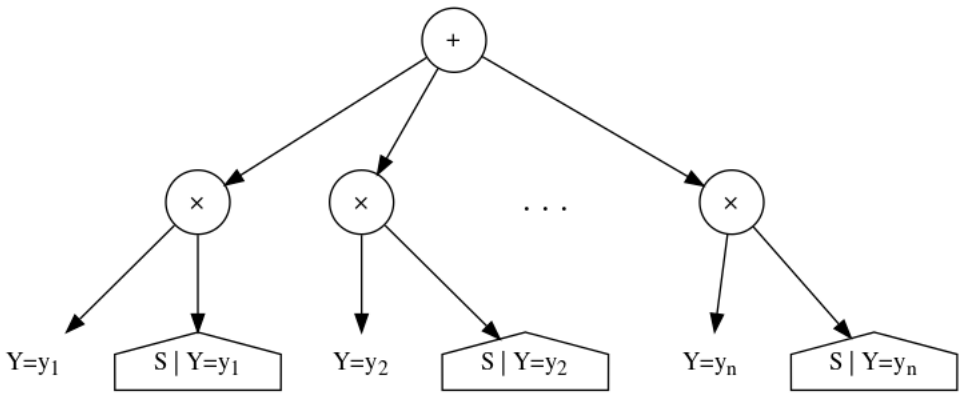

Figure 4.2: *The classification architecture for the Dennis-Ventura structure.*

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

In this chapter we formally define a possible model for the mobile robots self-driving problem. We first present this problem as an imitation learning through image classification problem. We then describe the dataset intended as training set for image classification. Finally, we discuss the problems of computing inference in a mobile robot, and provide a concurrent programming solution for our particular problem.

5.1 The problem

One of the main problems of self-driving is to follow a certain pathway. In real life, a self-driving car should be able to maintain itself centered on a lane, more specifically inbetween lane markings. Our objective is to address this particular subproblem of self-driving. This is achieved by considering this situation as an imitation learning application.

Imitation learning consists of an agent accurately mimicking human behavior. In our case, we wish for such an agent to simulate the behavior of keeping a car centered on a single lane. We model this particular situation by use of image classification. The agent, in this case the self-driving car, should reliably identify when to turn and when to go straight by solely "looking" at the road ahead. This can be achieved through the use of image classification, as turns tends to have different visual features then a straight lane.

Whilst this comes naturally to humans, machines have trouble identifying these features by themselves. Noise and object occlusion play a big role in how reliably the agent behaves. A possible obstruction of the agent's view could turn fatal in a real-life scenario. However, with the advent of more complex models in machine learning, modelling this problem through image classification has become a feasible solution.

Our approach to self-driving in mobile robots consists of a very simplified and purely reactive image classification control system. The mobile robot should follow a lane and turn accordingly based on its image input of its front view.

We define a classification variable, which we will denote by Y, as an indicator of what the robot should do. The function $\text{Val}(Y)$ defines the set of all possible values of Y. In our case, $\text{Val}(Y) = \{\text{L}, \text{R}, \text{U}\}$, each meaning that the robot should "go left", "go right" and "go

straight" (or up) respectively.

Let $X = \{X_1, X_2, \ldots, X_n\}$ be the set of variables that compose an image, where each X_i represents a pixel of a flattened image. Our entire scope is defined by the set $W = X \cup \{Y\}$. Our objective is to reliably guess Y's value based solely on the values of X. That is, we wish to find

$$\arg\max_{y \in \text{Val}(Y)} P(Y = y|X) = \arg\max_{y \in \text{Val}(Y)} \frac{P(Y = y, X)}{P(X)} \propto \arg\max_{y \in \text{Val}(Y)} P(Y = y, X). \tag{5.1}$$

Where we assume the existence of an underlying probability distribution that correctly models the classification problem. Ultimately, our goal is to find this distribution by "learning" from data through the learning algorithms described in Chapter 3 and Chapter 4.

Once learned, the SPN is able to find the MAP probabilities and states that correctly predict the most probable values of Y given an image. Note how the LHS of Equation 5.1 can be computed with a single pass with the approximate max-product algorithm. Alternatively, we can also compute the exact max values by computing each possible $y \in \text{Val}(Y)$ with a single forward pass through the SPN.

5.2 The dataset

For training, we used Moraes and Salvatore's self-driving dataset (Moraes and Salvatore, 2018). Every image has dimensions 80×45, with three additional channels for RGB. The dataset is split into three sets, corresponding to training, test and validation data. Each image contains a label indicating whether the robot should go straight, turn left or turn right. These actions are labeled as 0, 1 and 2.

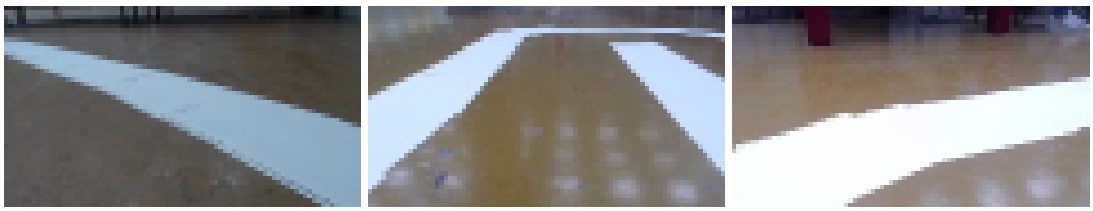

Figure 5.1: *Sample images from training dataset.*

Figure 5.1 showcases sample images from the training dataset. The leftmost image has label L, the one on the middle is U and the one on the right R. It is possible to observe that images do not have uniform lightning and lane markings are irregular. This adds a noise effect to the images.

The original dataset is already reduced in size. However, we can further reduce the number of variables by discarding colors, as its presence is not so important for identifying the correct values of Y. If we compare Figure 5.1 and Figure 5.2, lane markings, the most important features of lane following, are still very much visible.

We can try to further reduce the complexity of the dataset at the same time preserving its most informational features by attempting to reduce the number of possible values

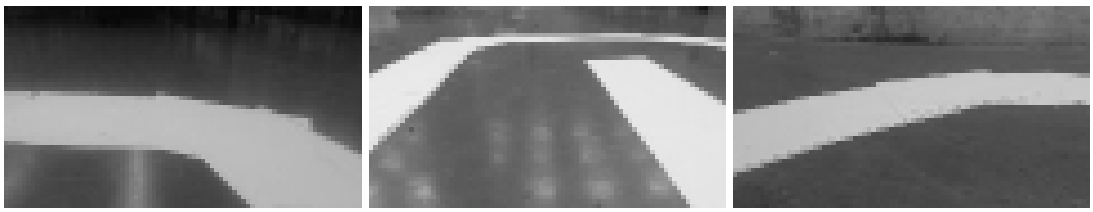

Figure 5.2: *Grayscale sample images from training dataset.*

of each pixel through image quantization. This transformation turned out to be very meaningful in terms of both training speed and accuracy, as we detail in Chapter 8. However, noise is still a problem, as Figure 5.3 shows.

Figure 5.3: *Quantized sample images from training dataset.*

Another possible transformation we can apply on the dataset is binarization. Though this comes with a problem, as traditional "hard" binarization with a fixed k threshold on the image could potentially cause completely black or white images due to a poor choice of k. This can be countered with two possible solutions. Either through adaptive threshold where we choose k either by a mean or gaussian measurement, or by use of Otsu's binarization (Otsu, 1979). We found that Otsu's method, coupled with a prior gaussian blur transformation on each image, proved the most capable of correctly applying binarization in our dataset.

Figure 5.4: *Binarized sample images from training dataset.*

Figure 5.4 shows the final result of applying a combination of gaussian blur and Otsu's binarization.

One last transformation we tested our models on was histogram equalization. Equalization was done in order to reshape the pixel value histogram to an approximately uniform distribution. This was done in an attempt to increase the accuracy of the Gens-Domingos algorithm. As mentioned in Chapter 4, the Gens-Domingos schema attempts to find partitions of independent variables. We use an implementation of the G-test, which works best when there are sufficient samples for every variable category. The original dataset has a skewed histogram that contains almost no pixel values in the extreme ranges

(either too white or too black). This transformation added a lot of noise to the dataset. In Chapter 8 we give more motivation on why we used equalization and its impact on accuracy.

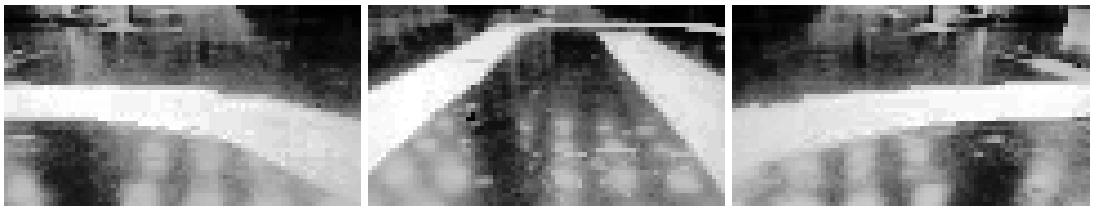

Figure 5.5: *Equalized sample images from training dataset.*

5.3 The model

Our classification model should be able to accurately infer the most probable action to be taken given the front camera feed's image. Although a simple decision model could be used for such a task, in many cases we wish to maintain an uncertainty measurement (e.g. a probability distribution) as a means to quantify error. Error and noise is often a problem in robotics, one which can be mitigated through probabilistic localization algorithms. For our particular case, we discard these problematic issues and focus solely on the problem of computing a valid probability distribution that accurately models our classification problem. SPNs have full probabilistic semantics and are able to represent local and sparse variable interactions due to their deep architecture. This allows for a reliable probabilistic model for modelling our problem.

We use two SPN architectures. We model the first using the Dennis-Ventura algorithm described in section 4.1. This model in particular uses the classification architecture also cited in Chapter 4. The second model uses the Gens-Domingos structural algorithm (section 4.2).

For weight learning, we found that *soft* gradient descent suffered from gradient diffusion, resulting in no change on weight update. Consequently, we showcase only *hard* gradient descent trained networks in Chapter 8.

As we have previously mentioned, we wish to compute the MPE

$$\arg\max_{y \in \text{Val}(Y)} P(Y = y, X) = \arg\max_{y \in \text{Val}(Y)} S(Y = y, X) \approx y|_{M(Y=y,X)}, y \in \text{Val}(Y). \tag{5.2}$$

Where $y|_{M(Y=y,X)}$ signals an MPN forward and backward pass to compute the most probable explanation of variable Y given X as evidence on the underlying SPN.

Recalling Chapter 2, we can compute this probability in two different ways. Either by computing each Y value by means of a forward pass on the SPN $S(Y = y, X)$, or through the approximate MAP $M(Y = y, X)$. These two options both carry disadvantages. On one hand, computing each Y value through a forward pass on each possible valuation can cause inference to become very slow, especially when hardware is as limited as in a mobile robot. On the other hand, the max-product algorithm is a fast inference method, though

approximate. Furthermore, the approximate method tends to favor shallower paths due to its Viterbi-style features.

We compromise with exact inference, but with an addendum. We take advantage of our hardware's multi-cored CPU by running exact inference in parallel. Since the Gens-Domingos and Dennis-Ventura architectures are very distinct structure-wise, we apply a different implementation for each.

As mentioned in Chapter 4, the Dennis-Ventura structure we implemented follows a particular classification architecture that models each set of images of a certain label as a separate, independent sub-SPN. This allows for an easy parallel programming implementation, as each CPU core can be assigned to a single sub-SPN, and thus to a label. Each core will then be linked to a particular robot command. Once all cores finish inference, we then compare which sub-SPN returned the highest probability. This is only advantageous if the number of labels is low. In our case, $|\text{Val}(Y)| = 3$, meaning this method is feasible for our problem.

This method does not work as well on the Gens-Domingos structure, as we cannot guarantee if an independency or clustering step on the root node has yielded a sufficient number of nodes for each core. However, a similar method is used, where we compute each value of y in parallel. Since $|\text{Val}(Y)| = 3$, we can compute each forward pass concurrently and then compare them.

Chapter 6

Hardware

In this chapter we dive into the hardware specifications of our mobile robot. Our robot is composed of two main units: the Berry and the Brick. The former is composed of a Raspberry Pi and a webcam, and serve as the brain for our robot. The latter is a Lego Mindstorms robot, which contains two differential motors for each wheel and a small processing unit, called the Brick, used for issuing low-level commands to the wheels. We first explore the Berry, and then describe the Brick.

6.1 The Berry

The main processing unit of our robot consists of a Raspberry Pi 3 Model B. With a Broadcom Quad-Core BCM2837 with 64-bit at 1.4 GHz, it is a small, yet powerful microcomputer. Its architecture is based on ARMv7, and has four processing cores.

Through four USB 2.0 ports, we are able to connect the Raspberry Pi with the motor part of our robot and a small portable webcam that will be used for input.

Figure 6.1: *The Berry part of the robot.*

A MicroSD with 16GB provides both the Raspbian operating system, which is based on Debian, as well as an additional 1 GB swap memory space, as the Berry contains only 1GB RAM. The remaining amount is used as storage.

Although the Berry has reasonable processing power for its size, training is done offline in a desktop computer, with only inference done on the micro-computer. On the training-side computer, we generate the SPN and then serialize it into a string of bytes and save it to a binary file. This file is then sent through SSH to the Berry, read, loaded into memory, converting the array of bytes into a full SPN, and then used for inference.

With its connected webcam, the Berry receives each image frame, applies some pre-processing to the image (which will be detailed in Chapter 8), feeds it to the SPN as evidence, computes the most probable classification label, and finally sends this to the external unit, i.e. the Brick, responsible for dealing with the robot's motion. This is all done concurrently, as we can dedicate three cores to classification, and the remaining unit to camera capture, image pre-processing and message passing.

6.2 The Brick

The Lego Mindstorms robot is composed of three main parts: the brick, which is the Lego Mindstorm's processing unit that handles the motors, and two differential motors that are able to give motion to the robot in a somewhat precise manner through the use of tachometers. In this document, when we say the Brick, we are referring to the whole set of brick and motors.

In our experiments we used the Lego Mindstorms NXT. Its main processor is an Atmel AT91SAM7S256 with a 48 MHz clock and 32-bit ARMv4 architecture. It has 64 KB of RAM and 256 KB of flash storage. A USB port allows for local input and output from and to the Brick.

Figure 6.2: *The Brick part of the robot.*

Low-level handling is done on the Brick. Once it receives a command to be executed

(i.e. the classification label passed by the Berry), the Brick needs to interpret it and execute the desired movement. We use the leJOS NXJ API[1] for low-level motor programming. The Brick's cycle is as follows:

Algorithm 11 `Brick`: The Brick's cycle

```
Connect and power up motors
Let UP ⟵ 0x00
Let LEFT ⟵ 0x01
Let RIGHT ⟵ 0x02
Let QUIT ⟵ 0x03
Let NOOP ⟵ 0x04
Let M_L and M_R be the left and right side motors respectively
Let k be some speed constant
while true do
    if input size > 0 then
        Read input byte and store in variable c
        if c is QUIT then
            Disconnect and power down
        else if c is UP then
            Set M_L's power to k
            Set M_R's power to k
        else if c is LEFT then
            Set M_L's power to 2k
            Set M_R's power to 3k
        else if c is RIGHT then
            Set M_L's power to 3k
            Set M_R's power to 2k
```

No complex control is done on the Brick. A standard "do something until told otherwise" is implemented. This is reliant on the Berry's inference speed, but we found that inference was fast enough for this to work well on slow speeds.

Setting the motors' power is straightforward with leJOS. Two function calls are enough for our case. The problem is in choosing k. If k is too big, not only the Berry might not have enough time to compute the labels, but also turns may not be as steep as the lane requires. In the case of a too small k, the challenge of autonomous driving becomes null. We experimented on values of k and chose $k = 150$ for our Lego Mindstorms. We recognize that a better control solution to this would be preferred, but for our case this suffices.

6.3 Bridging the two

Communication between the two modules, Berry and Brick, is done through a USB cable. We use a Go codebase on the Berry, opting to use Google's GoUSB[2] as a low-level

[1] Available at http://lejos.org/nxj.php
[2] Available at https://github.com/google/gousb

interface to handle the Raspberry Pi's data output, sending each predicted label as a byte value.

At each cycle, the Brick checks for new input by reading from an input stream. This is done through Java's `java.io.DataInputStream`, which translates the incoming USB data into Java bytes.

This hierarchization of Berry to Brick allows for the unit with most processing power, i.e. the Berry, to take on the heavy load, leaving only the necessary dedicated low-level motor handling to the Brick, a very limited processing unit in terms of power and memory.

Figure 6.3: *Fully assembled robot.*

Every USB device contains a pair of IDs that are essential for identification. The vendor ID is used to identify which company or organization created the device. Whilst the product ID is used to identify specific products created by the company. In our case, the vendor refers to the Lego Mindstorms company and the product to the NXT version 2, which are `0x0694` and `0x0002` respectively. These IDs allow us to cycle through each active USB device and select exactly the one we need.

USB devices may contain multiple functionalities, such as acting as a power source or as an input/output stream. These are called configurations, and are usually indexed by a number. In our case we are interested in the read and write configuration of the Brick, indexed by the number 1.

Each configuration contains different interfaces that can be seen as virtual devices to the physical USB. We selected interface number 0, with alternative interface 0.

When writing and reading from and to a USB device, we must define an output and input endpoint. There are a total of 30 endpoints, where input endpoints are indexed from

`0x81` to `0x8f`, and output endpoints from `0x01` to `0x0f`. Two additional in/out endpoints `0x80` and `0x00` are control endpoints used internally by the USB device. For our Brick we used the input `0x82` and output `0x01` endpoints.

Chapter 7

Pre-processing

Before training and inference, we apply different image transformations to the dataset. As mentioned in Chapter 5, we use three main transformations: binarization, quantization and equalization. In all cases we first convert the original RGB colored dataset to grayscale.

7.1 Binarization

The binarization process was done by first converting the original dataset to grayscale, followed by applying a gaussian blur with a unit filter kernel (that is, a 1×1 window matrix that blurs a single pixel based on zero values from its neighbors) and standard deviation of one on both axes, and finally using Otsu's binarization (Otsu, 1979). We chose this particular process since standard hard threshold binarization was unable to produce clear images of the track lines.

Figure 7.1: *Binarization using hard threshold.*

Figure 7.1 shows how hard thresholding can produce noisy images, as the applied threshold does not account for local neighborhoods and may end up choosing a bad threshold value. Figure 7.2 shows a sample of the dataset after applying Otsu's binarization, with much clearer tracks. Images are labeled as left, up and right respectively

Figure 7.2: *Binarization using Otsu's threshold.*

7.2 Quantization

The Gens-Domingos algorithm, as mentioned in Chapter 4, has two main steps: a clustering phase and an independency test part. Our independency test implementation uses the standard G-test statistical test based on contingency tables, where each frequency of the categories of each two variables are laid out on a matrix and their likelihood ratios are computed. This test takes only $O(nm)$, where n and m are the number of categories of each variable. However, each variable must be tested pairwise with all others, and although we use a spanning-tree heuristic, time complexity grows fast the more categories and total number of variables there are.

We empirically found that $\max\{n, m\}$ and training set size are directly correlated to the model's accuracy and speed. If n or m are big and the training set size is small, then accuracy will fall. Accuracy increases if the training set is large, but is still dependent on how big the number of categories in variables is. Our best results were with small n and m with a large training set. Another interesting result we found is that the bigger the number of categories, the shallower the SPN when using the Gens-Domingos algorithm, which results in much faster inference.

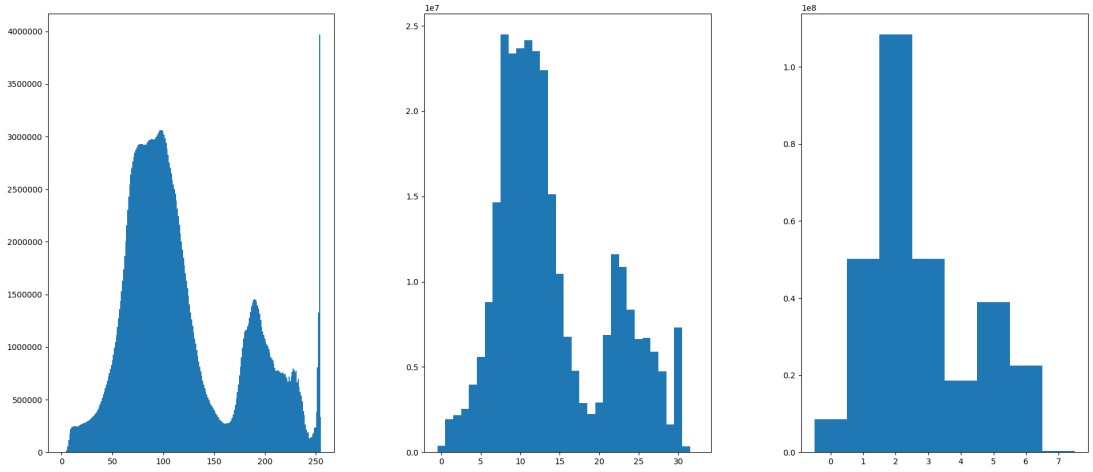

Figure 7.3: *Histogram for dataset pixel values on 8-bit, 5-bit and 3-bit image quantizations.*

A possible explanation for poor results with small training sets is the low number of pixel intensity values for too extreme values, as there are fewer samples where pixels

are either too bright or too dark, as Figure 7.3 shows. This is a problem for the G-test, as chi-squared tests are unreliable when dealing with low frequencies.

Quantizing the dataset resulted in a significant improvement in accuracy to the model when training with a small set of images (≤ 300). We found that when we increased the number of training images, accuracy depended less on quantization, but inference time increased, as the model grew in depth. We thus needed to find a balance between network complexity and inference speed.

We faced two possible solutions to this problem. Either implement an exact independence test, such as the Fisher exact test (Fisher, 1922), or perform histogram equalization on the dataset. The former was unfortunately not an option, as we found that there were no libraries in Go or C that provided a general case implementation of the Fisher exact test, and implementing our own within our time constraints was out of question. We chose the latter, applying histogram equalization on the entire dataset.

7.3 Equalization

Equalization was done using OpenCV. We attempted two methods of histogram equalization: traditional equalization through brightness and contrast normalization, and Contrast Limited Adaptive Histogram Equalization (CLAHE) (Zuiderveld, 1994). We found that the CLAHE method resulted in images that were very similar to the output of the traditional method. Since these transformations are also expected to be applied on-the-fly during inference, we chose the standard equalization for its speed. Figure 7.4 shows how the dataset pixel histogram looks like after equalization.

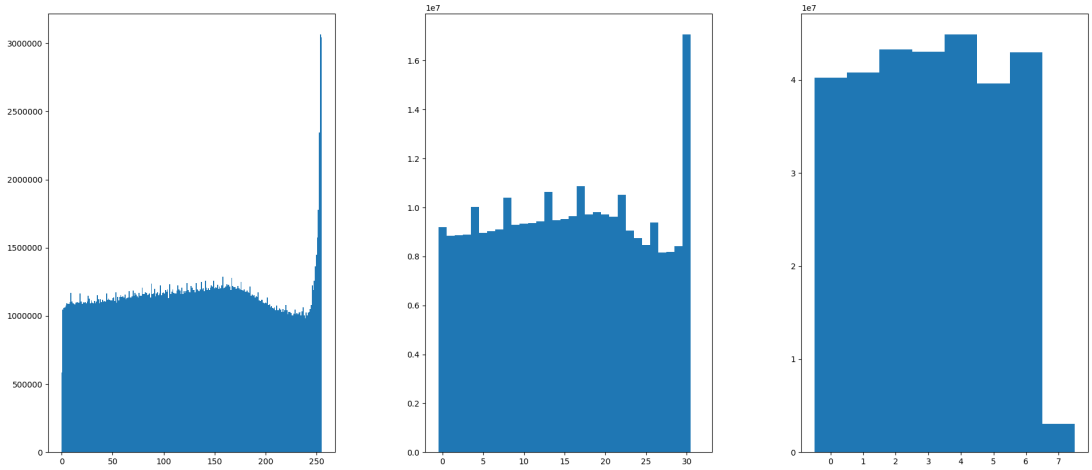

Figure 7.4: *Histogram for equalized dataset with 8-bit, 5-bit and 3-bit image quantizations.*

When coupling quantization and equalization, we slightly increased accuracy, reaching up to a ≈4% accuracy difference. Interestingly, these transformations proved to be harmful for the Dennis-Ventura architecture. In fact, the structure yielded better results without equalization, and quantization had little to no impact on accuracy. We attribute this

phenomenom to the classification architecture we discussed in Chapter 4. Since each sub-SPN is essentially modelling each class as a separate, independent image model to the other classes, the more details in the image, the easier the model can distinguish from other label images. Furthermore, since the Dennis-Ventura algorithm does not need to run an independence test, it does not suffer from its drawbacks and thus does not depend on an equalized histogram.

Chapter 8

Experiments and benchmarks

In this chapter we apply our model on an artificial self-driving dataset, running several experiments and measuring how good our models perform with each setup. The thinking behind this benchmarking is to provide an initial screening to find the best SPN iteration to use on a real-world testing scenario.

We first show results on accuracy for each of the models and pre-processing transformations. We then show how fast each model is, i.e. how long it took for training and how much time it took for the model to predict a label.

All our implementations are free, open-source and publicly available. GoDrive[1] implements all experiments covered in Chapter 8 and Chapter 9. GoSPN[2] implements the learning and inference algorithms.

8.1 Setups

For our experiments, we tested two structure learning algorithms, the Dennis-Ventura and Gens-Domingos architectures, and for each of these models we evaluated accuracy when applying either generative or discriminative weight learning. We additionally ran tests without applying weight learning to serve as reference. In this case, for the Gens-Domingos architecture we set weights proportional to each cluster size, whereas in Dennis-Ventura we randomized weights.

For the Dennis-Ventura algorithm, we discarded pre-clustering, opting to use the classification architecture mentioned in Chapter 4, as it resulted in much better accuracy. We also fixed the number of sums per region and gaussians per pixel to four, and the similarity threshold to 0.975.

For the Gens-Domingos algorithm, we tested two implementations. The first uses k-means for the clustering step with $k = 2$. The second uses DBSCAN, with parameters $\epsilon = 4$ the maximum radius of a point neighborhood, and $m = 4$ the minimum number of points to describe a dense region. Both were set with a p-value of 0.01 for the independence

[1] https://github.com/RenatoGeh/godrive
[2] https://github.com/RenatoGeh/gospn

step. We refer to the k-means implementation as k-GD, and the DBSCAN variation as DBSCAN-GD.

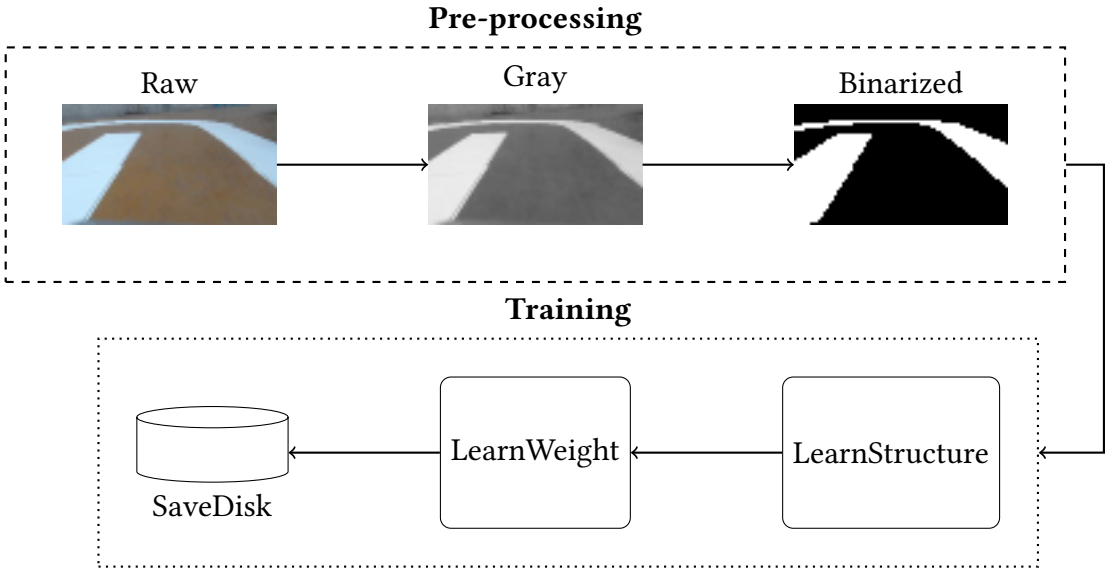

Figure 8.1: *Our training pipeline when using binarization in pre-processing.*

DBSCAN-GD achieved the best scores, but took the longest time for both training and testing. We did not test all possible iterations of pre-processing in this case, as DBSCAN-GD took too long during training (an average of 10 hours for training alone). We found that when using DBSCAN-GD, the resulting structure was too complex (about 32 times bigger than k-GD), causing inference to take too long. For instance, when running inference on k-GD, average prediction took about 0.1 second. When using DBSCAN, the average time of prediction was 19.72 seconds. However, in terms of accuracy, DBSCAN-GD showed impressive results, with all tests achieving a perfect score of 100% accuracy. Despite these numbers, a model that takes too long for inference is not adequate for a self-driving application. For this reason, we discarded the DBSCAN-GD model and decided to only use the k-GD model.

In both training and testing, we first apply image transformations (e.g. quantizing, binarization or equalization) to the dataset and then train or perform prediction with a particular model. The applied image transformation is always identical in training and testing. For example, a valid training pipeline would be choosing to apply 3-bit quantization and equalization to a training dataset, train an SPN structure using k-GD, perform discriminative gradient descent on the resulting structure to learn its weights, and finally save the model for testing. Its then equivalent testing pipeline would be applying the same image transformation, in our case 3-bit quantization and equalization, and for each image find the $\arg\max_{y \in Y} P(Y = y|X)$. Figure 8.1 shows a visualization of the training pipeline for binarization, whereas Figure 8.2 displays its equivalent inference pipeline.

Both Gens-Domingos and Dennis-Ventura algorithms generate a structure as deep as the number of training samples. The deeper the structure, the more expressive it is. We found that accuracy with models trained with 1000 samples had much better accuracies

than with 500. However, a more complex network means inference will take longer. We attempted to keep inference at less than a second per prediction. A training set size of 500 was chosen as the generated SPNs had decent prediction time and accuracy, and did not take too long in training. The size of the testing dataset was also 500 images.

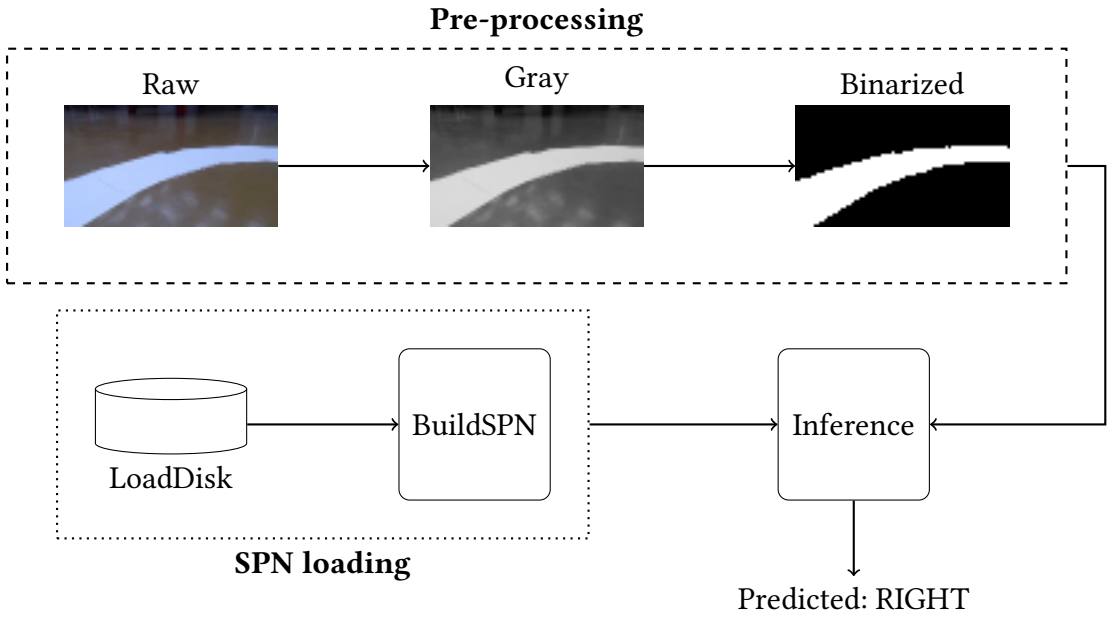

Figure 8.2: *Our inference pipeline when using binarization in pre-processing.*

We use a particular set of notations for our experiments. For image transformations, we denote by Q_n as applying an n-bit quantization of the dataset. An E means the dataset was equalized, and a B means it was binarized. Any combination of image transformation is signalled with a + sign. A $\emptyset$ means there were no image transformations done to the dataset apart from grayscale conversion.

For learning algorithms, a GD means we are using k-means Gens-Domingos and DV Dennis-Ventura. This is then followed by the weight learning algorithm used. The letters g, d and s mean we either applied generative gradient descent, discriminative gradient descent or no weight learning.

Parameters for weight learning were fixed to learning rate $\eta = 1.0$, L2 regularization constant $\lambda = 0.001$, mini-batch size $b = 50$ and number of epochs $N = 15$. In all setups weight learning suffered heavily from gradient diffusion when using soft gradient descent, forcing us to only use hard gradient descent for all tests.

8.2 Accuracy

In this section we show accuracy results in each setup. All values are in percentage of hits.

Table 8.1 shows some interesting results. The first is that generative gradient descent on the Dennis-Ventura architecture had a negative impact on the performance of the network. Discriminative learning on the other hand seemed to not impact accuracy at

Accuracy (%)	DV+g	DV+d	DV+s	GD+g	GD+d	GD+s
B	78.8	78.8	78.8	82.8	83.8	85.0
Q_2	78.6	78.0	78.0	78.6	80.4	79.4
$Q_2 + E$	76.6	76.6	76.8	79.6	82.8	81.8
Q_3	77.4	77.4	77.4	77.6	80.2	79.8
$Q_3 + E$	70.4	76.6	76.6	79.2	81.2	77.4
Q_4	78.2	78.4	78.2	76.0	78.2	76.4
$Q_4 + E$	76.6	76.6	76.8	76.0	74.6	80.6
Q_5	77.8	78.4	78.4	77.6	74.0	73.8
$Q_5 + E$	76.6	76.6	76.6	72.0	72.8	72.0
Q_6	77.4	78.4	78.4	75.2	74.4	72.0
$Q_6 + E$	76.0	76.4	76.4	73.0	75.0	73.6
Q_7	78.2	78.4	78.4	62.8	72.2	71.4
$Q_7 + E$	76.2	76.4	76.4	70.6	71.4	71.6
$\emptyset$	78.0	78.4	78.4	62.4	62.4	62.4
E	76.4	76.4	76.4	60.4	60.0	61.2

Table 8.1: *Accuracy values for each possible model permutation.*

all. Not only that, quantization seems to have had no impact on the DV structure, with equalization always deteriorating accuracy.

The Gens-Domingos architecture, on the other hand, achieved better results, but showed a more unpredictable behavior. When resolution was low, as is the case of B, Q_2 and Q_3, the GD architecture seems to always outclass DV. In fact, our best results were, as expected, when binarization was used, as we are only tracking the most significant feature: lane bounds. In some cases generative learning improved accuracy, but in most it had a negative impact. Discriminative learning, on the other hand, seemed to almost always provide a minor boost to the network's accuracy.

Our main takeaway from these accuracy results is that the structure of an SPN is much more meaningful than its weights. Generative learning in our use case had a negative impact, and discriminative learning slightly improved accuracy.

8.3 Speed

In this section we try to quantify both training and inference of our models in each setup. We ran all tests on an Intel Core i7-4500U CPU @ 1.8GHz, a 2-core 4-threaded processor. For memory we had 16GB RAM and 16GB swap space, though training did not use more than 4GB and testing did not exceed 100MB.

Training time for DV was very regular, as Table 8.2 shows. As expected, discriminative gradient descent took the longest time, as we require two passes through the network for each mini-batch. Training time with gradient descent depends on the depth of the network. We found that the DV structure had very regular training times for only generating the network. Not only that, regardless of pre-processing the network had very similar network depth and size. This was in clear distinction with the Gens-Domingos structure. In fact,

Training (mins)	DV+g	DV+d	DV+s	GD+g	GD+d	GD+s
B	22m36s	34m23s	01m03s	85m58s	200m53s	01m50s
Q_2	22m07s	34m59s	00m23s	52m57s	125m17s	01m28s
$Q_2 + E$	22m01s	35m05s	00m24s	98m51s	232m20s	09m36s
Q_3	22m15s	32m42s	00m25s	34m24s	96m18s	04m56s
$Q_3 + E$	22m25s	32m35s	00m25s	44m47s	102m03s	13m01s
Q_4	22m16s	31m56s	00m25s	31m25s	41m46s	08m15s
$Q_4 + E$	22m23s	31m49s	00m31s	34m21s	45m42s	11m35s
Q_5	22m24s	35m06s	00m29s	23m10s	23m18s	07m24s
$Q_5 + E$	22m25s	36m44s	00m29s	20m13s	29m38s	08m36s
Q_6	22m20s	36m34s	00m27s	49m22s	50m35s	21m37s
$Q_6 + E$	22m12s	37m13s	00m27s	54m53s	43m30s	21m17s
Q_7	22m13s	33m14s	00m30s	78m05s	72m03s	43m44s
$Q_7 + E$	22m22s	37m40s	00m29s	95m30s	79m48s	65m19s
$\emptyset$	22m10s	36m56s	00m33s	166m53s	107m44s	90m22s
E	22m10s	34m58s	00m33s	174m55s	116m41s	186m49s

Table 8.2: *Average time in minutes and seconds for training each model.*

Inference (secs)	DV+g	DV+d	DV+s	GD+g	GD+d	GD+s
B	0.23	0.25	0.25	0.38	0.37	0.31
Q_2	0.22	0.24	0.23	0.28	0.34	0.16
$Q_2 + E$	0.22	0.23	0.23	0.38	0.30	0.27
Q_3	0.22	0.23	0.22	0.22	0.32	0.17
$Q_3 + E$	0.22	0.23	0.22	0.34	0.32	0.31
Q_4	0.22	0.22	0.23	0.16	0.17	0.13
$Q_4 + E$	0.23	0.27	0.29	0.13	0.14	0.13
Q_5	0.22	0.26	0.28	0.07	0.05	0.02
$Q_5 + E$	0.22	0.29	0.25	0.05	0.05	0.02
Q_6	0.23	0.24	0.23	0.04	0.05	0.01
$Q_6 + E$	0.22	0.24	0.28	0.03	0.04	0.02
Q_7	0.23	0.23	0.26	0.03	0.01	0.01
$Q_7 + E$	0.22	0.26	0.24	0.01	0.01	0.01
$\emptyset$	0.22	0.26	0.23	0.02	0.01	0.01
E	0.23	0.23	0.22	0.01	0.01	0.02

Table 8.3: *Average time in seconds to predict a single image.*

binarization generated the deepest network when applying the GD learning algorithm, whilst training the structure with no pre-processing resulted in the shallowest. Interestingly, on average, deeper networks took the least time to train and had the best accuracy results. This is probably due to better variable partitioning during the independence step, resulting in a more balanced and more expressive tree.

Table 8.3 shows how much time it took, on average, to predict a single image. Computation of the average time was done by timing all 500 predictions on the test dataset, and then dividing by the number of total predictions. Again, the DV architecture

had very regular times. The GD structure, on the other hand, took the longest on deeper networks (where lower resolution quantization was used) and the shortest on shallower SPNs.

Note that although values in Table 8.3 look promising, they were extracted from a desktop computer. We quantify actual mobile robot inference speeds in Chapter 9.

Chapter 9

Real world

In this chapter we finally apply our model to a real-world implementation of mobile robot lane following. We give reasons as to why we chose the models and pre-processing used in our real-world experiment. Next, we describe how each model performed in each track. A video demonstration of the robots in action is available online[1]. Finally, we give some brief comments on the performance of the robots in general.

9.1 Control and inference

Our main goal in this project was to be able to provide a fast, accurate and robust model for self-driving that was capable of real-time measurement of uncertainty. Emphasis must be placed on real-time, as a self-driving car ought to react at least as fast as a human. For this reason, we chose to use a very simple control system that relied on fast and accurate inference to perform well.

In Moraes and Salvatore, 2018, the robot's control system employed a command queue. In their implementation, the robot does not move itself until it receives a command. Once it does, it then moves a fixed distance. If the next command is not ready to be processed, it stops and waits for inference (i.e. the next command). Otherwise, the queued command is executed. These steps are then repeated. Although this method of self-driving evaluation nicely quantifies accuracy, it fails to account for inference speed. Indeed, Moraes and Salvatore achieved ≈80% accuracy with deep feedforward neural networks and ≈85% with convolutional neural networks, but inference took about 1.35 seconds, with binarized results taking faster average speeds of 0.6 seconds, even with GPU support. In the real-world, a self-driving car cannot afford to run a few meters, stop for inference, and then continue for a few more. We account for this in our implementation.

For this reason, a much more reactive control system was built for this project. Input was based solely on commands given by an inference model. Not only that, the speed of which these commands were given impacted heavily on its performance. As mentioned in Chapter 6, the Brick receives a command and applies power to its motors accordingly. Until another command is given by the Berry, the Brick repeats the same command

[1] https://youtu.be/vhpWQDX2cQU

previously given. For instance, if the Brick is given a command to go straight, it will continue to do so without stopping until a different order is otherwise given. This sort of control is able to quantify how reliable inference is based not only on accuracy, but also speed of inference.

In Chapter 8, we showed inference speed results based on a desktop computer with a powerful CPU. When running the same models on the Berry, we found that although the binarized GD architecture achieved 85% accuracy and took about 0.3 seconds on a desktop PC, this speed went up to 3.5 seconds on the Berry. This high increase is not only due to the power difference between the two CPUs, but also by the need of real-time image pre-processing (in this case binarization) directly from a live video feed.

This speed discrepancy forced us to discard several models due to slow inference. In fact, we found that a model that took more than 0.15 seconds would take more than 1.0 second on the Berry. Additionally, histogram equalization proved to severely decrease speeds, causing us to discard any of its use in our real-world experimentations. We ultimately decided to use

1. Q_4 and GD+d;
2. Q_6 and GD+d;
3. no pre-processing and GD+d.

Inference times were about 0.7, 0.150 and 0.075 seconds for item 1, item 2 and item 3 respectively. All our implementations used solely the CPU, with no GPU support. Our experiments show that a high accuracy but slow inference speed performs poorly on our setup. A low accuracy but fast prediction speed model can behave erratically, but is able to correct itself because of its speed. We found that a good balance between accuracy and speed performed the best. We will refer as Model 1, Model 2 and Model 3 the model and pre-processing done in item 1, item 2 and item 3 respectively.

9.2 Tracks

We built three different tracks in order to evaluate how well models behaved in different environments. We assembled tracks in the same way as done in Moraes and Salvatore, 2018. In each track, A4 paper sheets were placed in order to form road markings on both sides of the "road". In this section, we describe each track and discuss how each model performed.

9.2.1 Track 0

The first track is a standard square shaped track. It is the simplest track we built in order to evaluate turns and lines. The robot's objective is to complete a lap without going out of bounds.

All models were able to drive through the whole lap. We noticed that Model 1 was much slower in terms of updating its state when compared to the other models. It particularly had trouble with straight lines, as the model's slow inference speed prevented it from making

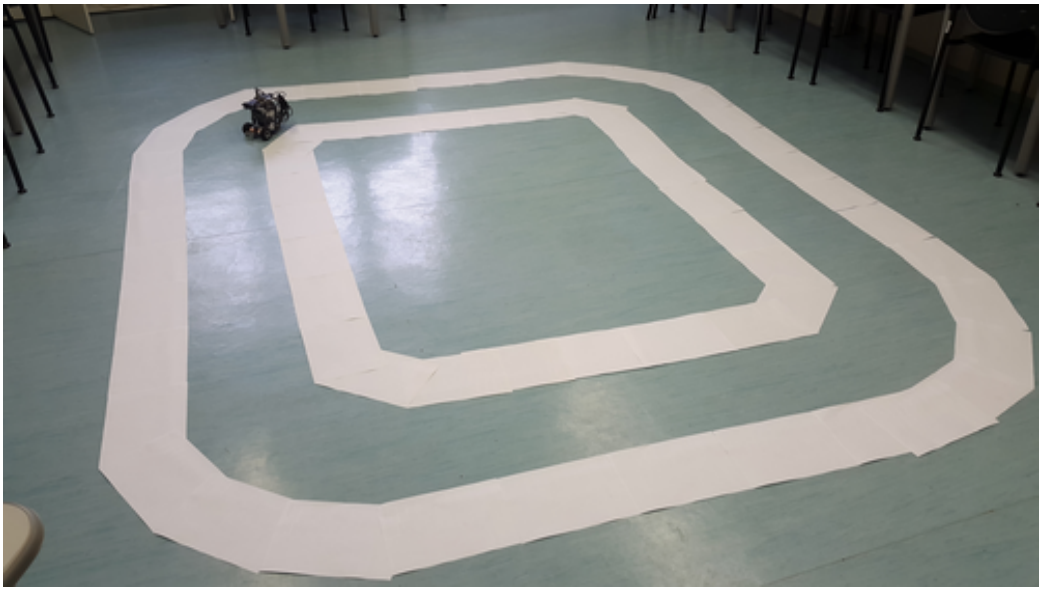

Figure 9.1: *Track 0 is a simple square shaped circuit.*

small corrections and thus propagated a sort of accumulated error at each attempted correction. Model 3 seemed to avoid moving in a straight line, opting to zig-zag instead. This model's behavior was a constant in every track. Model 2 was able to correctly turn and run lines with no apparent issue. Small corrections when travelling on straight lines were handled well, as its fast inference speed allowed for fine modifications to its direction.

9.2.2 Track 1

In this ∞-shaped track, the robot is supposed to travel through both conjoined circles, alternating between the two of them without going out of bounds.

Figure 9.2: *Track 1 is an infinity shaped circuit.*

Model 1 was unable to complete the lap, as once on the intersection of the two circles, the robot kept its turning course, repeating the same circle instead of the other. This is due to its slowness in updating its prediction, as the intersection point between the two circles could be predicted as both a turn to the left, right or, as intended, a go straight command. All other models were able to complete the two circles.

Model 2 completed the track perfectly, having no issue with traversing the intersection point. Model 3 zig-zagged throughout the whole course, but was able to complete the track as well.

9.2.3 Track 2

The hardest of the tracks, it features sharp turns and irregular paths. One may map this artificial scenario to the real-world challenge of roads in steep terrains, such as going down a mountain. An additional challenge in this circuit is the proximity and otherwise intersection between road markings that do not belong to the track the robot is currently on. This confuses the robot, as depending on its position relative to the marks, it may incorrectly predict its next action. Indeed, all models failed in this track directly or indirectly due to this.

Figure 9.3: *Track 2 simulates a road going down a mountain.*

Turns are identified by their order of appearance when travelling the pathway. Turns are increasingly tighter as the road progresses. Figure 9.4 marks these four turns with visual aids colored in blue. We do the same for the three ambiguous road markings, coloring them with red.

Model 1 failed due to both ambiguous road markings (particularly on Mark 1) and the first sharp turn (Turn 1). Because of its slow inference, it was unable to correct itself. Model 2 managed to go all the way down to Turn 3, however it got confused on Mark 3, mistaking it for a right turn. Model 3 failed on Mark 2 due to the same reasons.

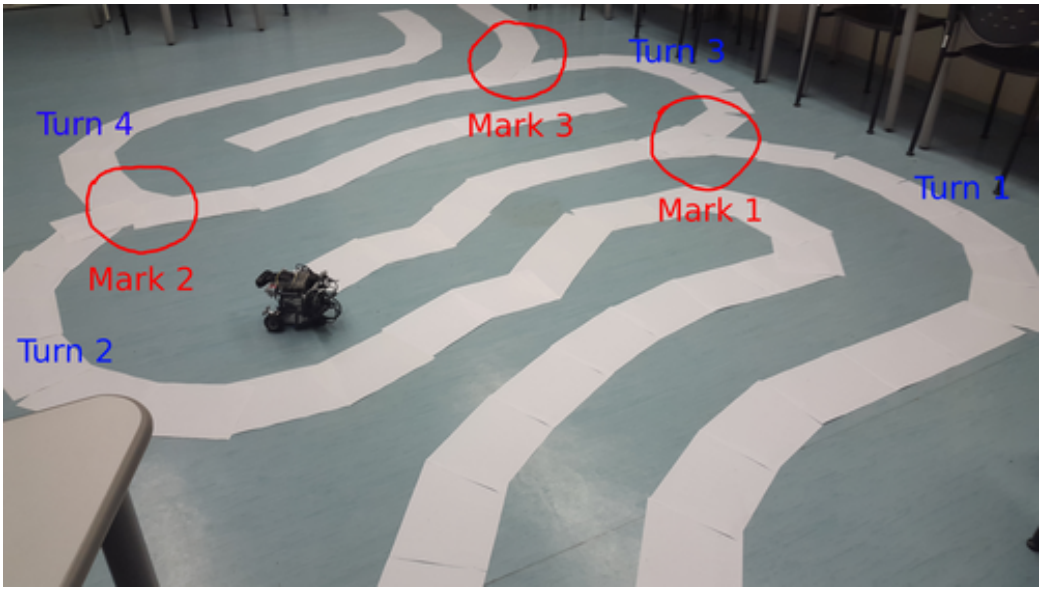

Figure 9.4: *Track 2 ambiguous markings and sharp turns.*

Track 2 was designed with the flaws of our system in mind. We knew beforehand our approach with image classification does not take into account previous classifications. This causes ambiguous markings to be misclassified when they could have been inferred from previous predictions. Likewise, sharp turns prove to be a challenge for a robot that is only allowed to go forward, as one misclassification may prove fatal. Nevertheless, Model 2 exceeded our expectations, managing to correctly travel through Turn 1 and Turn 2, and correctly evading Mark 1 and Mark 2.

9.3 Comments

Although our experiments have shown that image classification for self-driving is not perfect, it is worthy to mention that the robots were able to correctly extrapolate from data. Even though the materials used to build the tracks were identical to the dataset's, the tracks themselves were very distinct from what was used in the original's. This shows how SPNs were able to generalize from their training data. Model 2 especially was able to correctly classify both Marks 1 and 2 despite never seeing this kind of road marking in its training dataset. It is also worthy to mention that, although Model 3 failed to go straight when presented with a straight line, the probabilities computed by the model to go UP when on a line were very often high, meaning the model had some sense of uncertainty between turning and going straight.

We also noticed that calibration seemed to have played a major role in how well the models performed. In fact, the correct positioning of the camera, i.e. its angle with relation to the ground and its height, was crucial for making the robot work. Self-driving image classification seems to be very sensitive with respect to this. The training dataset had a specific camera angle and height that we attempted to replicate as best as possible. In spite of this, as the robot is in constant movement, and because of small vibrations made by the Brick's motors, the camera angle was often disturbed, causing the robot to misclassify.

However this could easily be solved by affixing the camera to the robot itself.

Another recurring issue we had were CPU processing interruptions. Often, the Raspberry Pi would completely halt the program's runtime. We are unsure if this was an OS interruption or if it was related to the power source, as we were using several peripherals for input and output whilst using an underpowered battery source. This can be viewed in the demonstration video, where Model 2's front video feed in Track 0 skips several frames during recording.

Overall, the models were able to perform fairly well, although with some minor technical issues.

Chapter 10

Conclusion and discussion

In which we conclude our thesis, provide some discussion on the topics covered by pointing the advantages and the flaws in using image classification in self-driving, and finally give a brief overview on possible future work.

10.1 Conclusion

In this thesis we argued the feasibility of lane following via image classification. We showed that, with a fast and accurate model, it is possible to obtain reasonable results with such a simple approach. SPNs were fast enough to be able to provide both accuracy and speed, even on such a limited miniature computer as the Raspberry Pi.

Having said that, we identified a few flaws in this approach, as documented by the previous chapter. First and foremost, the choice of which model to use (i.e. to find the fine balance between network complexity and inference speed) is still unquantifiable. At which point is a fast model accurate enough? Should self-driving be more focused on being accurate or fast?

Second, although current deep models have proven to be able to extract very meaningful features and reach impressive accuracy levels, it is still not completely foolproof. Ambiguous markings such as the ones mentioned in Chapter 9 can easily fool any model, as they are perfectly valid classifications. A possible solution to this would be to account for previous classifications through a temporal model, such as a markov-chain-like network or a recurrent neural network.

Finally, the choice of how to model the control system can play a big part in how well the robot is going to perform on a real case scenario. We chose a very simple control system that only turned left, right or went straight with no degree of continuity between commands. Furthermore, our implementation had a fixed turn ratio, meaning sharp turns were a problem from the start.

Despite all this, our attempt was reasonably successful at modelling self-driving on a low-budget mobile robot. The robot was able to correct itself before going off tracks, identified turns correctly, and in its best iteration was able to follow long lines smoothly.

It is also worthy to mention that training was much faster and required fewer training samples than other deep models. We were also able to accurately quantify uncertainty very fast due to linear time exact inference in SPNs.

10.2 Further work

This section is dedicated to possible future work related to this thesis. We give brief suggestions on how to improve the training and inference model as well as the robot's control system.

In our thesis, we only implemented the Gens and Domingos, 2013 and Dennis and Ventura, 2012 structure learning algorithms. There have been many other architectures that have achieved better results since. Future work on a comparison between these more recent state-of-the-art networks would be a very interesting path to take.

For weight learning, we only applied hard generative and discriminative gradient descent. There have been many advances on weight learning, including Rashwan et al., 2018 through the use of Extended Baum-Welch. Accuracy increased significantly though this new weight learning method, and we speculate whether this would provide a boost to accuracy.

Nowadays, deep models make use of GPUs to accelerate both learning and inference. Our implementation made use of only the CPU. GPU parallelization in SPNs is still in its early infancy, but implementations such as R. Peharz et al., 2018 have shown that a more connectionist approach similar to deep neural networks could potentially increase its performance whilst maintaining its probabilistic semantics.

Our robot control system was based on a very simplistic approach that does not mirror a real-world implementation. A more complex controller could potentially enhance self-driving as a whole. However an excedingly complex system could put too much load on the robot, decreasing its performance.

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. 8).

[Darwiche 2003] Adnan Darwiche. "A Differential Approach to Inference in Bayesian Networks". In: (2003) (cit. on pp. 3, 4).

[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, 56).

[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).

[Fisher 1922] Ronald A. Fisher. "On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P". In: *Journal of the Royal Statistics Society* 85.1 (1922), pp. 87–94 (cit. on p. 41).

[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. 11, 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, 22, 56).

[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. 8).

[Moraes and Salvatore 2018] Paula Moraes and Felipe Salvatore. *Self Driving Data*. 2018. url: https://github.com/felipessalvatore/self_driving_data (cit. on pp. 28, 49, 50).

[Otsu 1979] Nobuyuki Otsu. "A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66 (cit. on pp. 29, 39).

[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. 4–6, 11, 21).

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

[Robert 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, 8).

[R. Peharz et al. 2018] R. Peharz et al. "Probabilistic Deep Learning using Random Sum-Product Networks". In: *ArXiv e-prints* (2018). arXiv: 1806.01910 (cit. on p. 56).

[Rashwan et al. 2018] Abdullah Rashwan, Pascal Poupart, and Chen Zhitang. "Discriminative Training of Sum-Product Networks by Extended Baum-Welch". In: *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*. Vol. 72. Proceedings of Machine Learning Research. 2018, pp. 356–367 (cit. on p. 56).

[Zuidervald 1994] Karel Zuidervald. "Contrast limited adaptive histogram equalization". In: *Graphics gems IV* 4 (1994), pp. 474–485 (cit. on p. 41).