UNIVERSIDAD ADOLFO IBAÑEZ

MASTER'S THESIS

Compiling Neural Network Classifiers into Boolean Circuits for Efficient Shap-Score Computation

---

Author:
Jorge E. León

Thesis Supervisor:
Leopoldo Bertossi
SKEMA Business School, Canada
Senior UAI Fellow

Thesis Defense Committee:
Miguel Romero R. (UAI)
Marcelo Arenas S. (PUC)

Thesis carried out in accordance with the requirements for the degree of Master of Science in Data Science

of the

Faculty of Engineering and Sciences

June 2, 2023

UAI | FACULTAD DE INGENIERÍA Y CIENCIAS

UNIVERSIDAD ADOLFO IBAÑEZ

Abstract

Faculty of Engineering and Sciences

Master of Science in Data Science

Compiling Neural Network Classifiers into Boolean Circuits for Efficient Shap-Score Computation

by Jorge E. León

Along with the increasing mass use of machine learning models, there has been an increase in the need to be able to generate explanations for the predictions that they make. In this scenario, an efficient method to calculate the Shap-scores (which serve to define the participation that a variable had in the final result) was noted in a certain type of deterministic and decomposable Boolean circuits. Likewise, a method was found to go from binary neural networks to circuits of this kind. What has been said gave rise to combining these methods and evaluating the convenience of calculating the Shap-scores in this way (as an open-box), compared to the traditional way (as a black-box). We found that it is indeed reliable and more efficient for various scenarios, but future work still needs to be done to reveal the full potential of this technique. Additionally, the conversion method was formalized and the code used was made available for everyone who is interested in this area.

Keywords: Explainable AI, Knowledge compilation, Shap-score, Binary neural network, Boolean circuits.

Contents

List of Abbreviations

BNN	**B**inary **N**eural **N**etwork
BP	**B**inary **P**erceptron
CNF	**C**onjunctive **N**ormal **F**orm
dDBC	**D**ecomposable and **D**eterministic **B**oolean **C**ircuit
dDBCSFi(2)	**D**ecomposable and **D**eterministic **B**oolean **C**ircuit **S**moothed and with **F**an-**i**n **2**
dDNNF	**D**ecomposable and **D**eterministic **N**egation **N**ormal **F**orm
DNF	**D**isjunctive **N**ormal **F**orm
ML	**M**achine **L**earning
NNF	**N**egation **N**ormal **F**orm
OBDD	**O**rdered **B**inary **D**ecision **D**iagram
SDD	**S**entential **D**ecision **D**iagram
vtree	**V**ariable **T**ree

Chapter 1

Introduction

In recent years, as more sophisticated machine learning (ML) models have emerged and become more widely used, there has been an increasing demand for methods to explain and interpret the results or predictions that they generate. Many examples can be found, among them: explanations of why a loan application is rejected, why a medicine is recommended, why a candidate for a job position is selected, etc. The focus of this thesis is on classification models, which assign a label/category to an entity. Being more specific, this research focuses on binary classification models, i.e. those that return one of two possible labels (like 0 or 1).

Explanations come in different forms and can be obtained through different methods. A type of explanation of the results of ML-based models consists of providing **attribution scores**, which quantify the relevance of a feature or a feature value to the result. Here we will concentrate on local scores, i.e. associated with a particular entry at a time, as opposed to a global score that indicates the general relevance of a feature.

One of the popular local scores is Shap (S. M. Lundberg & Lee, 2017), which is based on the Shapley value, used in coalition game theory as well as in practice, and was introduced in (Shapley, 1953; Roth, 1988). Another attribution score that has been recently investigated is Resp (Bertossi, Li, Schleich, Suciu, & Vagena, 2020; Bertossi, 2022), which is based on actual causality (Halpern & Pearl, 2005) and its associated score of actual causality (Chockler & Halpern, 2004). This paper only considers the Shap-score, but the topics investigated here would also be interesting for Resp and other scores.

A score such as Shap, and like Resp, can be computed with either a black-box model or an open-box model (Rudin, 2019); the first being one in which its internal components are not known (or simply not used), but only its input/output relationship. For the open-box, the internal components can be accessed and understood individually. It is common to say that models based on neural networks are black-box models, while, for example, a decision tree model tends to be classified as an open-box model. On this occasion, it seems valid to say that this classification is equivalent to talking about using the internal components of the model (i.e. open-box) when calculating the scores vs. not using them and merely employing the input/output relationship (i.e. black-box). It is common to consider neural-network-based models as black-box models, because their internal gates and structure may be difficult to understand or process when it comes to explaining classification outputs. However, a decision-tree model, due to its much simpler structure and use, is considered to be open-box for the same purpose.

Even for binary classification models, the complexity of calculating Shap-scores is provably difficult; in fact, it is $\#P$-hard for various kinds of binary classification ML-based models

(Bertossi et al., 2020; Arenas, Barceló, Bertossi, & Monet, 2021, 2023). This is valid regardless of whether model internals components are used when calculating Shap or not. However, there are some families of classifiers for which, using the model components and structure, the computational complexity of Shap can be reduced to polynomial time (S. Lundberg et al., 2020; Arenas et al., 2021; Van den Broeck, Lykov, Schleich, & Suciu, 2021). As stated in (Arenas et al., 2021), for **deterministic and decomposable Boolean circuits** (dDBCs), Shap can be computed in polynomial time and an efficient general algorithm was given for dDBCs that are **smoothed and have a fan-in 2** (dDBCSFi(2)). From this result, it becomes clear that it is feasible to compute Shap for a variety of Boolean circuit classifiers and other classifiers that can be represented as (or compiled to) dDBCs. In particular, this is true for ordered binary decision diagrams (OBDDs) (Bryant, 1986) and **sentential decision diagrams** (SDDs) (Darwiche, 2011b). From these, it is possible to obtain the same benefit for decision trees, binary neural networks, and other established classification models that can be compiled (in polynomial time) to OBDDs or SDDs (Shi, Shih, Darwiche, & Choi, 2020; Darwiche & Hirth, 2020; Narodytska, Kasiviswanathan, Ryzhyk, Sagiv, & Walsh, 2018). And it is worth pointing out that in (Van den Broeck et al., 2021), through a different approach, the tractability of Shap computation was obtained for a collection of classifiers that overlaps with that of (Arenas et al., 2021).

In this work, it is shown how to use logic-based knowledge compilation techniques to attack, and -to the best of our knowledge- for the first time, the important and timely problem of efficiently computing explanations scores in ML, which, without these techniques, would stay intractable.

This work focuses on developing this method explicitly for the efficient computation of Shap for **binary neural networks** (BNNs). In support of this, and inspired by (Shi et al., 2020), a BNN is transformed into a dDBCSFi(2) through techniques from **knowledge compilation**; an area that investigates the transformation of (usually) propositional theories into an equivalent one with canonical syntactic form that has some favorable computational property, e.g. tractable model counting. The transformation may incur in a high cost that will be worth it if the particular property is checked frequently (Darwiche & Marquis, 2002; Darwiche, 2011a), as is the case with explanations for the same BNN.

More specifically, this thesis describes in detail how a BNN is first compiled into a propositional formula in Conjunctive Normal Form (CNF), which, in its turn, is compiled into an SDD, which is finally compiled into a dDBCSFi(2). Our method applies at some steps established transformations that are not commonly illustrated or discussed in the context of real applications, which we do here. The whole compilation path so as the application to Shap computation are new. We show how Shap is computed on the resulting circuit via the efficient algorithm in (Arenas et al., 2021). This compilation is performed once, and is independent from any input to the classifier. The final circuit can be used to compute Shap scores for different input entities.

It is worth mentioning that the compilation of binary classifiers into OBDDs was used in (Darwiche & Hirth, 2020) to provide different kinds of explanations for its outputs, but not for the computation of Shap (nor for any other kind of attribution scores). There are several other explanation mechanisms, for ML-based classification and decision systems in general (see (Guidotti et al., 2018)), and some are more specific to neural networks (see (Ras, Xie, van Gerven, & Doran, 2022)).

In this thesis, the real estate as an application domain is considered, where housing prices depend on certain features. The dataset used is *California Housing Prices* (Nugent, 2018) and the problem is to classify blocks of properties, represented as records of thirteen features, as

high price or **low price** block (above or below average, resp.). This is a binary classification problem for which a BNN is used.

As far as we know, this work is the first one to report experiments on the application of polynomial time algorithms for the Shap computation on this class of classifiers. This work confirms a considerable reduction in computation time when a dDBCSFi(2) is used as an open-box model and shows that, for this case, the scores obtained with both methods are perfectly aligned, in the sense that the values are equivalent. However, a couple of relevant points were also found and future lines of research are proposed to address them.

Chapter 2

Problem Definition

Depending on the model, calculating the Shap-scores of an ML-based model may be intractable. However, it has been shown that it is possible to compute them in polynomial time for a certain class of Boolean circuits used as classifiers, more specifically, for circuits in the dDBC class. Furthermore, an algorithm was proposed for this task (Arenas et al., 2021).

Binary Neural Networks (BNNs) used as classification models are common in practice. For them, and until now, the only way to compute Shap-scores as explanations for their outcomes was by naively applying the definition of Shap, which is bound to have high complexity since all the subsets of the features have to be brought into the computation. The efficient techniques mentioned above cannot be directly applied to BNNs, and we addressed the problem of finding a way to apply them.

There are methods for transforming BNNs into Ordered Binary Decision Diagrams (OBDDs) (Shi et al., 2020; Shih, Darwiche, & Choi, 2019; Narodytska et al., 2018). They have been used mainly for high-level model explanation purposes and model verification. We have proposed, investigated and applied a new method to convert BNNs into Sentential Decision Diagrams (SDDs), which in their turn can be easily compiled into dDBCs. In this way, we opened the door to the possibility of efficiently computing Shap-based explanations for BNNs. We applied the above mentioned algorithm, experimenting with real data, and compared the results, in terms of computation cost and score alignment, with Shap-computation directly on the BNN, treated as a black-box classifier.

As we show, the transformation of a BNN into a dDBC may have a high time complexity step along the conversion path. However, on one side, this is a fixed-parameter tractable problem; and this computation is performed once. After that, the same resulting circuit can be used multiple times (and efficiently) to compute Shap-based explanations, for different inputs to the model.

Chapter 3

Preliminaries and State of the Art

The Shapley value is a measure established in coalition game theory. It emerges as the only measure that enjoys certain desirable properties (Roth, 1988). Given a set of players X and a game function $\mathcal{G}$: $\mathcal{P}(X) \to \mathbb{R}$, i.e. a function that maps subsets of players to real numbers, the general form of the Shapley value of a player $x \in X$ is:

$$Shapley(X, \mathcal{G}, x) := \sum_{S \subseteq X \setminus \{x\}} \frac{|S|!(|X| - |S| - 1)!}{|X|!} (\mathcal{G}(S \cup \{p\}) - \mathcal{G}(S)) \tag{3.1}$$

It quantifies the contribution of player x to the wealth function $\mathcal{G}$. Here, all possible permutations of subsets from X and their complements are considered. Given what has been said, this value is the average of the differences between including x and not including it. It should be noted that in order to apply the Shapley value, an appropriate game function $\mathcal{G}$ must be defined.

In (S. M. Lundberg & Lee, 2017; S. Lundberg et al., 2020), the Shapley value is applied to a fixed entity $\mathbf{e}$ under classification, defined as $\mathbf{e} = \langle F_1(\mathbf{e}), \ldots, F_N(\mathbf{e}) \rangle$, with values $F_i(\mathbf{e})$ for features in an array $\mathcal{F} = \{F_1, \ldots, F_N\}$. $\mathcal{F}$ becomes the set of players/features, giving rise to the Shap-score, with the game function $\mathcal{G}_\mathbf{e}(S) := \mathbb{E}(L(\mathbf{e}') \mid \mathbf{e}'_S = \mathbf{e}_S)$ in (3.1). This is a game function associated to the specific entity $\mathbf{e}$ under classification. Here, $\mathbb{E}$ denotes an expected value, and $\mathbf{e}_S$ is the projection (or restriction) of $\mathbf{e}$ on (to) the subset of features S, L is the label function associated with the classifier, and $\mathbf{e}'$ is an entity that matches $\mathbf{e}$ in the features of S. Consequently, for feature $F \in \mathcal{F}$, the Shap-score for the entity $\mathbf{e}$ becomes:

$$\mathsf{Shap}(\mathcal{F}, \mathcal{G}_\mathbf{e}, F) := \sum_{S \subseteq \mathcal{F} \setminus \{F\}} \frac{|S|!(|\mathcal{F}| - |S| - 1)!}{|\mathcal{F}|!} [\mathbb{E}(L(\mathbf{e}') | \mathbf{e}'_{S \cup \{F\}} = \mathbf{e}_{S \cup \{F\}}) - \mathbb{E}(L(\mathbf{e}') | \mathbf{e}'_S = \mathbf{e}_S)] \tag{3.2}$$

In addition to the Shap-score, there is another attribution score called Resp that has gained attention in recent research (Bertossi et al., 2020; Bertossi, 2022). Resp is based on actual causality (Halpern & Pearl, 2005) and is associated with the actual causality score (Chockler & Halpern, 2004). Although this paper only focuses on the Shap-score, the findings presented here could also be relevant for other attribution scores, including Resp. Further research could explore the similarities and differences between these scores and their respective applications in different domains.

To compute Shap, all that is needed is the L label function (practically a classifier), without having to use any internal components of the classifier. However, in (Bertossi et al., 2020) was proven that, for an L classifier that only has binary features, the computation of Shap is

#P-hard for a product probability space. This is linked to the fact that the direct computation of Shap (i.e. treating L as a black-box), while considering the product distribution for the entities, may be intractable from a relatively low number of features onwards.

Despite the previous, in (Arenas et al., 2021) is shown that Shap can be computed in polynomial time for dDBCs, used as classifiers, when their structures are used in the computation (i.e. treating them as open-boxes). Particularly, an efficient algorithm was formalized for dDBCSFi(2)s. We proceed to explain the properties of that kind of circuit.

In Figure 3.1 there is a Boolean circuit that can be used as a binary classifier, with input variables x_1, x_2, x_3. The binary values for them, entered at the lower nodes, are propagated up through the Boolean gates, and the final label is read from the upper one. It is **deterministic** in the sense that, for each $\vee$-gate, at most one of its inputs is 1 when the output is 1. It is **decomposable** in the sense that, for each $\wedge$-gate, the inputs do not share variables. The dDBC is also **smoothed** in the sense that subcircuits entering the same $\vee$-gate share the same variables, and it has **fan-in 2** in the sense that each $\wedge/\vee$-gate has at most two inputs. This type of dDBC is called dDBCSFi(2).

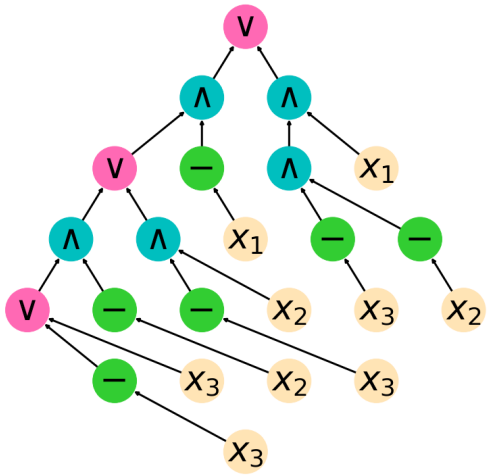

FIGURE 3.1: dDBCSFi(2) example.

In (Arenas et al., 2021), to calculate Shap efficiently in dDBCSFi(2)s, it is assumed that the underlying probability distribution in the entity population is uniform, P^u, or is the product distribution, $P^\times$. These are the following, considering that the features take binary values:

$$P^u(\mathbf{e}) := \frac{1}{2^N} \qquad P^\times(\mathbf{e}) := \Pi_{i=1}^{N} p_i(F_i(\mathbf{e})) \tag{3.3}$$

Here, $p_i(v)$ is the probability assigned to the value $v \in \{-1, 1\}$ for the feature F_i. These distributions require that the features are mutually independent. Unless otherwise stated, in the remainder of this document we will assume that this is true.

The efficient method to compute Shap in dDBCSFi(2)s is already well explained in (Arenas et al., 2021), together with a proof of its correctness. However, for the sake of completeness, it has been included here below as Algorithm 1. It should be noted that the algorithm is designed to work with the product distribution of (3.3), therefore it assumes statistical independence between the features. Additionally, 0 and 1 are used for the features, instead of -1 and 1, resp. This is due to the fact that the algorithm was designed with values 0 and 1 in mind, and working with them does not change the final Shap-scores, with regard of using -1 and 1 on the direct method.

On the other hand, there is the procedure mentioned in (Shih et al., 2019) to convert BNNs into formulas in **conjunctive normal form** (CNF), which later can be compiled as SDDs (that can be converted into dDBCSFi(2)s, as we will formalize in Section 5.3). It is worth saying that a CNF formula is a conjunction of disjunctions of **literals**, i.e. atomic formulas or their negations. The details can be found in (Narodytska et al., 2018) and will be explained later in this document, but the essence is that it considers each layer of neurons in the network as a block to be encoded with *sequential counters* (Sinz, 2005).

Algorithm 1 Efficient Shap-score for dDBCSFi(2)s

Input: A dDBCSFi(2) B over features $\mathcal{F}$, with output gate g_{out}, fixed entity $\mathbf{e}$ in the domain of joint features, fixed feature $F \in \mathcal{F}$, rational probability $p(Y = 1)$ for each $Y \in \mathcal{F}$, and the subsets $\mathcal{V}_g$ of all features feeding g, for each gate g in B.
Output: The $\mathsf{Shap}(\mathcal{F}, \mathcal{G}_\mathbf{e}, F)$ score for the feature F in $\mathbf{e}$, under the product distribution.

for every gate g in B, by bottom-up induction on B **do**
  if g is a constant gate with label $a \in \{0, 1\}$ **then**
    $\gamma_g^0 = a, \quad \delta_g^0 = a$
  else if g is a feature gate with $\mathcal{V}_g = \{F\}$ **then**
    $\gamma_g^0 = 1, \quad \delta_g^0 = 0$
  else if g is a feature gate with $\mathcal{V}_g = \{Y\}$ and $Y \neq F$ **then**
    $\gamma_g^0, \delta_g^0 = p(Y = 1), \quad \gamma_g^1, \delta_g^1 = Y(\mathbf{e})$
  else if g is a negation gate with input gate g' **then**
    for $\ell \in \{0, \ldots, |\mathcal{V}_g \setminus \{F\}|\}$ **do**
      $\gamma_g^\ell = \binom{|\mathcal{V}_g \setminus \{F\}|}{\ell} - \gamma_{g'}^\ell, \quad \delta_g^\ell = \binom{|\mathcal{V}_g \setminus \{F\}|}{\ell} - \delta_{g'}^\ell$
  else if g is a disjunction gate with input gates g_1, g_2 **then**
    for $\ell \in \{0, \ldots, |\mathcal{V}_g \setminus \{F\}|\}$ **do**
      $\gamma_g^\ell = \gamma_{g_1}^\ell + \gamma_{g_2}^\ell, \quad \delta_g^\ell = \delta_{g_1}^\ell + \delta_{g_2}^\ell$
  else if g is a conjunction gate with input gates g_1, g_2 **then**
    for $\ell \in \{0, \ldots, |\mathcal{V}_g \setminus \{F\}|\}$ **do**
      $\gamma_g^\ell = \sum_{\substack{\ell_1 \in \{0,\ldots,|\mathcal{V}_{g_1} \setminus \{F\}|\} \\ \ell_2 \in \{0,\ldots,|\mathcal{V}_{g_2} \setminus \{F\}|\} \\ \ell_1 + \ell_2 = \ell}} \gamma_{g_1}^{\ell_1} \cdot \gamma_{g_2}^{\ell_2}, \quad \delta_g^\ell = \sum_{\substack{\ell_1 \in \{0,\ldots,|\mathcal{V}_{g_1} \setminus \{F\}|\} \\ \ell_2 \in \{0,\ldots,|\mathcal{V}_{g_2} \setminus \{F\}|\} \\ \ell_1 + \ell_2 = \ell}} \delta_{g_1}^{\ell_1} \cdot \delta_{g_2}^{\ell_2}$
$\mathsf{Shap}(\mathcal{F}, \mathcal{G}_\mathbf{e}, F) = \sum_{k=0}^{|\mathcal{F}|-1} \frac{k!\,(|\mathcal{F}|-k-1)!}{|\mathcal{F}|!} \cdot [(F(\mathbf{e}) - p(F = 1)) \cdot (\gamma_{g_{out}}^k - \delta_{g_{out}}^k)]$

Sequential counters are practically a set of instructions that allows rewriting as a Boolean formula the case in which the sum of a set of binary variables is able to surpass a given threshold (like it is implicitly evaluated at every neuron of a BNN), with a relatively low number of clauses, at the cost of adding auxiliary variables (we will talk more about this matter in Section 5.1).

In simple terms, the original method encodes each block, using sequential counters on each neuron (independent from the other neurons in the same block) and taking the encoding of the previous block into account, until the entire network has been encoded. This returns a CNF formula which, after being compiled into an SDD, could be converted to a dDBCSFi(2), except for the addition of auxiliary variables (the removal of which is quite complex). This is why it was necessary to adapt this method to work with just the original variables (we delve into this matter on Section 5.2).

The method that we use is not the only way to get a dDBCSFi(2). For example, (Shi et al., 2020) converts BNNs to OBDDs, and, as OBDDs are practically a special type of SDDs (with a linear order for its variables), these can be converted into dDBCSFi(2)s too. The main reason to adopt the transformation of this thesis is the availability of implementations that can be used for some of the steps, which are mentioned in Chapter 5.

An additional detail is that, following our method, we could also compile a BNN into a mere OBDD, instead of an SDD. Despite this, for speed and conciseness sake (the proof can be found in (Bova, 2016)), it is better to work with an SDD instead.

The final thing that is worth pointing out in this Chapter is that in (Van den Broeck et al., 2021), through a different approach, the tractability of Shap computation was obtained for a collection of classifiers that overlaps with that of (Arenas et al., 2021). However, there is a fundamental difference in the approach taken to show tractability: the reduction of (Van den Broeck et al., 2021) uses multiple oracle calls to the problem of computing expectations,

whereas (Arenas et al., 2021) provides a more direct algorithm to compute the Shap-score on dDBCSFi(2)s. In any case, further research and improvements on both methods will help to define which one is more convenient in a given situation.

Chapter 4

Research Context

4.1 Hypothesis and Objectives

The central hypothesis of this research is that with the new method it is possible to calculate Shap over a dDBCSFi(2) in a more efficient way than with the traditional/direct method (either for the same dDBCSFi(2) or for the BNN from which the dDBCSFi(2) was generated), without sacrificing too much the correctness of the results.

The objectives are the following:

1. Show how to transform BNNs, through SDDs, into dDBCSFi(2)s.
2. Evidence that the resulting dDBCSFi(2)s satisfy the assumptions identified in (Arenas et al., 2021), in which the Shap computation becomes tractable.
3. Compute Shap, using the real estate data, for the classifying BNNs, treated as black-boxes, and for their compilations into dDBCSFi(2)s, treated as both black-boxes, as well as open-boxes.
4. Compare and analyze the experimental results.
5. Leave the entire implementation public, so that anyone interested can access it in the future.

4.2 Methodology

The experiments consisted on training a BNN (with one hidden layer and the same number of neurons as features) and a binary perceptron (BP), using the *California Housing Prices* dataset (Nugent, 2018).

For both trained models, we compiled equivalent dDBCSFi(2)s based on each, and proceeded to calculate the Shap-scores for 100 different entities that were present on the training data subset. As black-boxes, we tried both the original models and the resulting dDBCSFi(2)s, and we also did the calculation using the dDBCSFi(2)s as open-boxes.

In order to compare them, we registered the Shap-scores and the times involved for the Shap computation, as well as for the dDBCSFi(2) conversion.

There are two aspects to explain, which will be addressed in two separate chapters. These are about: conversion from BNNs to dDBCSFi(2)s (Chapter 5), and additional information/details about the experiments (Chapter 6).

We share our results in Chapter 7, and our conclusions on Chapter 8.

Chapter 5

Converting BNNs to dDBCSFi(2)s

To calculate Shap with the efficient method for a BNN, the latter must be converted into a dDBCSFi(2), for which Shap can be computed in polynomial time (Arenas et al., 2021). This transformation follows the next path:

$$\underset{\text{(a)}}{\text{BNN} \longmapsto} \text{ CNF formula } \underset{\text{(b)}}{\longmapsto} \text{ SDD } \underset{\text{(c)}}{\longmapsto} \text{ dDBCSFi(2)} \tag{5.1}$$

This is not the only way to get a dDBCSFi(2). For example, (Shi et al., 2020) converts BNNs to OBDDs, which can also be converted to dDBCSFi(2)s. Some of the steps in (5.1) may not be polynomial-time transformations, which we will discuss in more technical terms at the end of this chapter. However, we can claim at this stage that: (a) Any exponential cost of a transformation is kept under control by a usually small parameter. (b) The resulting dDBCSFi(2) is meant to be used multiple times, to explain different and multiple outcomes; and then, it may be worth taking a one-time, relatively high transformation cost. The main reason to adopt the transformation of this thesis is the availability of implementations that can be used for some of the steps, which are mentioned later along this chapter.

To explain the conversion, we will first describe the original encoding of the BNN into a CNF formula (Section 5.1). Then the modified version will be presented, which does not use auxiliary variables (Section 5.2). This will be followed by the remaining steps to get from a CNF formula to a dDBCSFi(2) (Section 5.3). An example illustrating the entire process will also be shown to ensure its understanding (Section 5.4). Finally, the section will end with a small analysis on the efficiency of the conversion method (Section 5.5).

5.1 BNN to CNF Formula: Auxiliary Variables

This method can be seen in (Shih et al., 2019; Narodytska et al., 2018).

Imagine that we have a dense binary neural network (i.e. the neurons of each layer are connected to all possible inputs from the previous one) that receives ℓ_0 input variables in the form of $\bar{x} = \langle x_1, \ldots, x_{\ell_0} \rangle$. This BNN has m hidden layers, each layer z (from 1 to m) has ℓ_z neurons and a neuron from layer z receives the input vector $\bar{i} = \langle i_1, \ldots, i_{\ell_{z-1}} \rangle$. For the output layer, it has a single neuron. All weights are -1 or 1, biases are real numbers, and the activation functions also return a value of -1 or 1 ($\phi_{\text{hidden layer}}$). The only exception to the latter is the output layer, where a step function is used that returns 0 or 1 ($\phi_{\text{output layer}}$). Formally, the activation functions are:

$$\phi_{\text{hidden layer}}(x) := \begin{cases} 1 & , \quad x \geq 0 \\ -1 & , \quad x < 0 \end{cases} \qquad \phi_{\text{output layer}}(x) := \begin{cases} 1 & , \quad x \geq 0 \\ 0 & , \quad x < 0 \end{cases} \tag{5.2}$$

More commonly, $\phi_{\text{hidden layer}}$ and $\phi_{\text{output layer}}$ are given the names *sign function* and *unit step function*, resp.

As shown in (Narodytska et al., 2018), such BNN can be converted into a CNF formula as follows.

We are going to work with all the layers, one by one, from the first hidden layer to the output layer. For each neuron in each layer, we want to encode the case in which said neuron becomes 1 (*true*). In other words, we want to represent each $\phi(\bar{w} \bullet \bar{i} + b)$ as a CNF formula, where the dot ($\bullet$) denotes the dot product of two vectors. The idea for each neuron is to add clauses incrementally to reflect the cases where our neuron reaches 1. This is done with the help of auxiliary variables $r_{(k,p)}$. The encoding of each neuron is independent of the others in the same layer, but from the second hidden layer, the last auxiliary variable of each neuron is taken as input. In order to avoid complicating the notation, it will be understood that 1 is equivalent to a *true* value, and 0/-1 are equivalent to *false*. It is also relevant to say that $r_{(k,p)}$ symbolizes if $\sum_{j=1}^{k} \frac{w_j \cdot i_j + 1}{2} \geq p$.

Starting with the first hidden layer, we take the first neuron, with weights $\bar{w} = \langle w_1, \ldots, w_{\ell_0} \rangle$ and bias b. As expected, we will take $\bar{i} = \langle x_1, \ldots, x_{\ell_0} \rangle$ as input. We must start by calculating d, our minimum number of inputs that must be conveniently instantiated for the output to be 1. We can deduce d as follows:

$$
\begin{aligned}
\textstyle\sum_{j=1}^{k}(w_j \cdot i_j) + b &\geq 0 &,\\
\textstyle\sum_{j=1}^{k}(w_j \cdot i_j) &\geq -b &,\\
\textstyle\sum_{j=1}^{k}(w_j \cdot (\frac{i_j+1}{2} \cdot 2 - 1)) &\geq -b &,\\
2 \cdot \textstyle\sum_{j=1}^{k}(w_j \cdot \frac{i_j+1}{2}) - \sum_{j=1}^{k}(w_j) &\geq -b &,\\
2 \cdot \textstyle\sum_{j=1}^{k}(w_j \cdot \frac{i_j+1}{2}) &\geq -b + \sum_{j=1}^{k}(w_j) &,\\
\textstyle\sum_{j=1}^{k}(w_j \cdot \frac{i_j+1}{2}) &\geq \left\lceil \frac{-b+\sum_{j=1}^{k} w_j}{2} \right\rceil &,\\
\textstyle\sum_{j=1}^{k}(\frac{w_j+1}{2} \cdot \frac{i_j+1}{2}) - \sum_{j=1}^{k}(\frac{-w_j+1}{2} \cdot \frac{i_j+1}{2}) &\geq \left\lceil \frac{-b+\sum_{j=1}^{k} w_j}{2} \right\rceil &,\\
\textstyle\sum_{j=1}^{k}(\frac{w_j+1}{2} \cdot \frac{i_j+1}{2}) - \sum_{j=1}^{k}(\frac{-w_j+1}{2} - \frac{-w_j+1}{2} \cdot \frac{-i_j+1}{2}) &\geq \left\lceil \frac{-b+\sum_{j=1}^{k} w_j}{2} \right\rceil &,\\
\textstyle\sum_{j=1}^{k}(\frac{w_j+1}{2} \cdot \frac{i_j+1}{2}) + \sum_{j=1}^{k}(\frac{-w_j+1}{2} \cdot \frac{-i_j+1}{2}) &\geq \left\lceil \frac{-b+\sum_{j=1}^{k} w_j}{2} \right\rceil + \textstyle\sum_{j=1}^{k} \frac{-w_j+1}{2} &,\\
\textstyle\sum_{j=1}^{k} \frac{w_j \cdot i_j + 1}{2} &\geq \left\lceil \frac{-b+\sum_{j=1}^{k} w_j}{2} \right\rceil + \textstyle\sum_{j=1}^{k} \frac{-w_j+1}{2}
\end{aligned}
$$

Thus, we can see that d is defined as:

$$
d := \left\lceil \frac{-b + \sum_{j=1}^{\ell_{z-1}} w_j}{2} \right\rceil + \sum_{j=1}^{\ell_{z-1}} \frac{-w_j + 1}{2} \tag{5.3}
$$

Now that we have d, we can use sequential counters (details of which can be found in (Sinz, 2005)) as follows to encode the neuron:

$$SQ(\bar{w},\bar{i},d) := \begin{cases} r_{(\ell_{z-1},d)} & , \ d < 1 \\ (w_1 \cdot i_1 \Leftrightarrow r_{(1,1)}) \wedge & \\ \bigwedge_{j=2}^{d} (-r_{(1,j)}) \wedge & |\bar{i}| \geq d \\ \bigwedge_{l=2}^{\ell_{z-1}} ((r_{(l,1)} \Leftrightarrow (w_l \cdot i_l \vee r_{(l-1,1)})) \wedge & , \ d \geq 1 \\ \quad \bigwedge_{j=2}^{d} (r_{(l,j)} \Leftrightarrow ((w_l \cdot i_l \ \wedge \ r_{(l-1,j-1)}) \vee r_{(l-1,j)}))) & \\ & |\bar{i}| < d \\ -r_{(\ell_{z-1},d)} & , \ d \geq 1 \end{cases} \tag{5.4}$$

It is worth noting that we can convert, for example, $x_1 \Leftrightarrow x_2$ to $(-x_1 \vee x_2) \wedge (x_1 \vee -x_2)$, and we can apply propositional logic to express the encoding of every neuron as a CNF formula.

This same process is repeated for each neuron in the layer. Since the encodings of neurons in the same layer do not interfere with each other, they could well be done in parallel. Consider $r^{(z,k)}$ to refer to the auxiliary variable $r_{(\ell_{z-1},d_{(z,k)})}$ of the k-th neuron from the z-th layer. Now, as anticipated previously, we must give $\bar{i} = \langle r^{(1,1)},\ldots,r^{(1,\ell_1)}\rangle$ as input for the next layer. As before, we encode each neuron by calculating d and $SQ(\bar{w},\bar{i},d)$. We repeat this process until we reach the output layer. At that point, we only need to do the encoding for that final neuron and, for our experiments, we would identify its last auxiliary variable ($r^{(m+1,1)}$), because this is the one that reflects the output of our BNN.

Now, for the sake of a complete explanation, let us take the case where the output layer has n neurons, so the BNN is for a multi-class problem. One output would be 1 and the rest 0, thanks to a *step softmax* (or an equivalent) activation function. For this output layer, we are taking $\bar{i} = \langle r^{(m,1)},\ldots,r^{(m,\ell_m)}\rangle$ as input. Let us focus on the first output neuron. We need to compute a comparison threshold $\hat{d}_{kl}$ between this neuron and each of the output neurons. For each of these, we must use the following formula:

$$\hat{d}_{kl} := \left\lceil \left\lceil \frac{b_l - b_k + \sum_{j=1}^{\ell_m}(w_{k_j} - w_{l_j})}{2} \right\rceil /2 \right\rceil + \left|\left\{w \in \tfrac{\bar{w}_k - \bar{w}_l}{2} \mid w = -1\right\}\right| \tag{5.5}$$

Where k is the index of our current neuron to encode and l is the index of the neuron to compare (so for our first output neuron, $k = 1$ and we have to iterate with $l \in \{1,\ldots,n\}$). For our selected l, we also need to obtain the subsets $\bar{\omega}_{kl} \subseteq \bar{w}_k$ and $\bar{\imath}_{kl} \subseteq \bar{i}$, taking the elements at the positions where $\bar{w}_k$ differs from $\bar{w}_l$. Formally speaking, they are defined as:

$$\bar{\omega}_{kl} := \left\{w \in \tfrac{\bar{w}_k - \bar{w}_l}{2} \mid w \neq 0\right\} \qquad \bar{\imath}_{kl} := \left\{i \in \tfrac{\bar{i} \odot (\bar{w}_k - \bar{w}_l)}{2} \mid i \neq 0\right\} \tag{5.6}$$

Where $\odot$ denotes the component-wise product of two vectors.

Then, we end up encoding $SQ(\bar{\omega}_{kl},\bar{\imath}_{kl},\hat{d}_{kl})$ for each $l \in \{1,\ldots,n\}$. Now, let us take the final auxiliary variable of all these encodings and put them in a new array $\bar{r}_k$, like this: $\bar{r}_k = \langle r_{(|\bar{\omega}k1|,\hat{d}_{k1})},\ldots,r_{(|\bar{\omega}_{kn}|,\hat{d}_{kn})}$. We finish encoding this neuron with $SQ(\bar{1},\bar{r}^k,n)$, where $\bar{1} = \langle 1,\ldots,1\rangle$. And this is repeated for all the output neurons, so we end up with all of our BNN encoded.

Once the encoding is complete, the final auxiliary variable of any neuron can be selected and perform calculations based on it.

It is important to note that more often than not, the encoding leaves us with an unnecessarily large formula. Because of this, a simplifier can be used to reduce its size, eliminating unimportant auxiliary variables and keeping the variables of interest. For example, for this investigation the SAT solver *Riss* (Manthey, 2017) was used, but it is not able to eliminate all the auxiliary variables when the formula is relatively complex.

The problem with using this encoding is that the efficient method for computing Shap in dDBCSFi(2)s is not intended to work with auxiliary variables. This is why they must be removed. While it is possible to use what is known as "forgetting variables" (Oztok & Darwiche, 2017), this technique is prone to damaging the determinism of the circuit. Although different methods were explored to eliminate these auxiliary variables, in the end it was decided that it would be more convenient to do without them at this time. The adapted method is easy to understand, but it involves working with fewer variables so it does not take too long (if a few more than twenty variables could be used with auxiliary variables, now it seems that the limit is around thirteen, but this may just be a problem of an inefficient implementation).

5.2 BNN to CNF Formula: Only Original Variables

We will consider the same BNN defined above. The basic idea here is the same as for the method using auxiliary variables. We want to encode each neuron, layerwise, in an incremental manner that reflects the case in which for a given number of variables, its instantiation is capable of reaching or surpassing a given threshold.

As before, we start with the first neuron in the first layer and compute its d with (5.3). We can see this process as filling an matrix $M_{|\vec{i}|\times d}$ of Boolean encodings, with $c_{k,t}$ components. M is not a matrix to do operations with, but rather a convenient structure for defining the order in which the encodings are generated. M is filled rowwise and, for each row, columnwise. Row k represents the number of the first variables considered, and column t the threshold to reach or surpass. Note that for any component where $k < t$, the threshold cannot be reached, so any component above the lower triangular matrix will be *false*.

We start with just the first variable. The threshold 1 is reached or surpassed when $w_1 \cdot i_1$ and the other thresholds are impossible to reach or surpass, so our matrix so far would look like this:

$$\begin{bmatrix} w_1 \cdot i_1 & \mathit{false} & \dots & \mathit{false} & \mathit{false} \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

For the first two variables, we now have two cases of thresholds that do not always return *false*. The first is for the threshold 1. Here, $w_1 \cdot i_1$ or $w_2 \cdot i_2$ would suffice. However, for the threshold 2 we would need $w_1 \cdot i_1$ and $w_2 \cdot i_2$. This can be written as:

$$\begin{bmatrix} w_1 \cdot i_1 & \mathit{false} & \mathit{false} & \dots & \mathit{false} \\ w_2 \cdot i_2 \vee c_{1,1} & w_2 \cdot i_2 \wedge c_{1,1} & \mathit{false} & \dots & \mathit{false} \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

Let us see with one more variable. We now have three valid thresholds. For the threshold 1, $w_3 \cdot i_3$ or $w_1 \cdot i_1 \vee w_2 \cdot i_2$ would suffice. For the threshold 2, $w_3 \cdot i_3$ and $w_1 \cdot i_1 \vee w_2 \cdot i_2$ would do, as would $w_1 \cdot i_1 \wedge w_2 \cdot i_2$. And for 3, we would need $w_3 \cdot i_3$ and $w_1 \cdot i_1 \wedge w_2 \cdot i_2$. So now the matrix would be:

$$\begin{bmatrix} w_1 \cdot i_1 & \mathit{false} & \mathit{false} & \mathit{false} & \dots & \mathit{false} \\ w_2 \cdot i_2 \vee c_{1,1} & w_2 \cdot i_2 \wedge c_{1,1} & \mathit{false} & \mathit{false} & \dots & \mathit{false} \\ w_3 \cdot i_3 \vee c_{2,1} & (w_3 \cdot i_3 \wedge c_{2,1}) \vee c_{2,2} & w_3 \cdot i_3 \wedge c_{2,2} & \mathit{false} & \dots & \mathit{false} \\ \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

Now we can imagine how we end up filling the rest of the matrix until we have:

$$M := \begin{bmatrix} w_1 \cdot i_1 & \mathit{false} & \mathit{false} & \dots & \mathit{false} \\ w_2 \cdot i_2 \vee c_{1,1} & w_2 \cdot i_2 \wedge c_{1,1} & \mathit{false} & \dots & \mathit{false} \\ w_3 \cdot i_3 \vee c_{2,1} & (w_3 \cdot i_3 \wedge c_{2,1}) \vee c_{2,2} & w_3 \cdot i_3 \wedge c_{2,2} & \dots & \mathit{false} \\ \dots & \dots & \dots & \dots & \dots \\ w_{\lvert\bar{i}\rvert} \cdot i_{\lvert\bar{i}\rvert} \vee c_{\lvert\bar{i}\rvert-1,1} & (w_{\lvert\bar{i}\rvert} \cdot i_{\lvert\bar{i}\rvert} \wedge c_{\lvert\bar{i}\rvert-1,1}) \vee c_{\lvert\bar{i}\rvert-1,2} & (w_{\lvert\bar{i}\rvert} \cdot i_{\lvert\bar{i}\rvert} \wedge c_{\lvert\bar{i}\rvert-1,2}) \vee c_{\lvert\bar{i}\rvert-1,3} & \dots & (w_{\lvert\bar{i}\rvert} \cdot i_{\lvert\bar{i}\rvert} \wedge c_{\lvert\bar{i}\rvert-1,d-1}) \vee c_{\lvert\bar{i}\rvert-1,d} \end{bmatrix} \tag{5.7}$$

With this, we can better describe the method to fill M. For our first row, the encoding of the first component is $w_1 \cdot i_1$ and *false* for the rest. For each component $c_{k,1}$ in the first column, this is $w_k \cdot i_k \vee c_{k-1,1}$. And for any other component $c_{k,t}$ (with a threshold t), we just use $(w_k \cdot i_k \wedge c_{k-1,t-1}) \vee c_{k-1,t}$. So $c_{|\bar{i}|,d}$ (the bottom right highlighted component) ends up being the encoding of our neuron.

As in Section 5.1, this is repeated for each neuron (i.e. calculate its respective d and generate its M), until we have the encoding of all the neurons in the layer. Since the encoding of each neuron does not influence nor is influenced by the others of the same layer, all the encodings of the neurons of a layer could well be generated in parallel.

For the next layer, we take as inputs the components $c_{\ell_0,d}$ of each neuron from the previous one, and all that remains is to repeat this conversion. That is, with the $c_{\ell_0,d}$s as inputs, the d and M of each neuron in the new layer are computed, which is followed by extracting the $c_{\ell_1,d}$ of each neuron and pass it as input to the next layer. Then it is just a matter of iterating to the last layer. From the last layer, we extract the encoding $c_{\ell_m,d}$ of the single output neuron, since this represents the encoding of the entire BNN.

Practically, we follow the method in (Narodytska et al., 2018) to generate the ds and replicate the order of the encoding, but we differ with the use of M, which helps to obtain a propositional formula without auxiliary variables. The downside of our approach is that it is computationally more expensive, but again, it is a one-time cost. To reinforce and illustrate the idea of this method, Figure 5.1 has been included.

The result, from the output neuron, is a Boolean formula that can be converted to a CNF formula, as well as simplified.

Giving more detail about our implementation, we transform each generated propositional formula to CNF. After it is generated (i.e. each component in each M matrix), some basic simplifications are applied to avoid an excessive growth (see Section 5.5 for the detailed explanation on why this is needed). In theory, for each neuron g, both the method with and without auxiliary variables have a time complexity around $O(d_g \cdot |\bar{i}_g|)$, but in practice the time of our method without auxiliary variables grows exponentially. This means that the

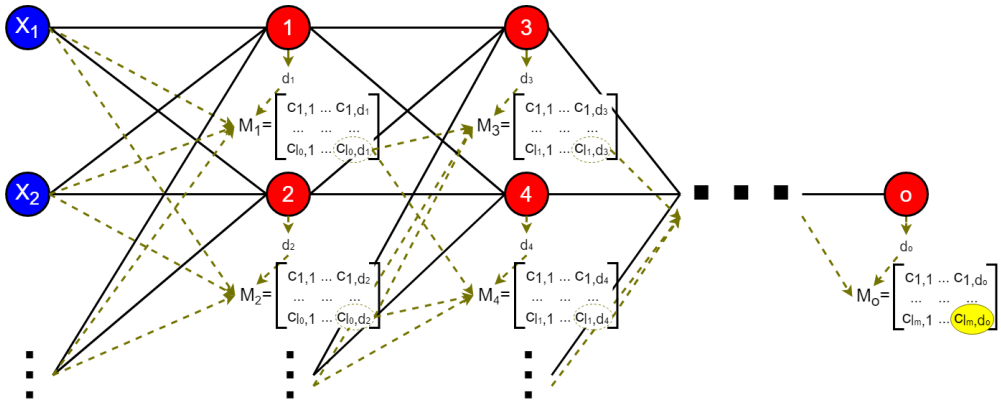

FIGURE 5.1: Conversion from a BNN to a CNF formula. The inputs for the first layer are the respective variables (blue nodes). Each neuron (red nodes) g has a d_g and a M_g, of which the component $c_{|\bar{i}|,d_g}$ is the final encoding of g. These final encodings are given as inputs to the M_gs of the next layer and the process is repeated until the last layer. c_{ℓ_m,d_o} of M_o represents the encoding of all the BNN.

main issue seems to be in these simplifications, where several inefficient cycles through all clauses are done at each neuron conversion (for reference, in our experiments we end up having thousands of clauses).

It is important to mention that the efficiency of this method and its implementation are not at all ideal. In fact, it there seems to be much room for improvement with both for more complex BNNs. Nevertheless, it is good enough for the experiments in this investigation.

5.3 CNF Formula to dDBCSFi(2)

As noted in (Darwiche, 2011b; Oztok & Darwiche, 2014), a CNF formula can be converted into a *Sentential Decision Diagram* (SDD) (Darwiche, 2011b; Van den Broeck & Darwiche, 2015), while keeping logical equivalence.[1] An SDD, as a particular kind of *decision diagram* (Bollig & Buttkus, 2019), is a directed acyclic graph. So as the popular OBDDs (Bryant, 1986), that SDDs generalize, they can be used to represent general Boolean formulas; in particular, propositional formulas (but without necessarily being *per se* propositional formulas).

Every SDD is made up of *decision nodes* and *elements*. Decision nodes are numbered circles ((k)) that practically function as disjunctions pointing to two or more inputs, which are always elements. The elements are pairs of rectangles ($[j|k]$) that operate as conjunctions, where j is named *prime* and k *sub*. Each rectangle can have a literal, a truth value, or a pointer ($\bullet$) with an edge to a decision node. If a rectangle of an element does not have a $\bullet$, it is called a *terminal*. At the end, these SDDs can be used as Boolean functions that take a given instantiation for its variables (according to an entity) and return its corresponding label by following the path downwards.

Before generating the respective SDD, one must choose a so-called *vtree* (for "variable tree"), which is a tree-based structure that represents a way of recursively partitioning the variables in a Boolean function into subsets, and with which the orders of occurrence of variables in

[1] The algorithm for compilation has not been published per se, but is that of the official library for SDDs in *Python* (Meert & Choi, 2018; Choi & Darwiche, 2018). We also know that it works in a bottom-up fashion, using the *apply* operations described in (Darwiche, 2011b) and commented in (Choi & Darwiche, 2013).

the diagram must be compliant with.[2] More precisely, a vtree for a set of variables $\mathcal{V}$ is a binary tree that is full (i.e. every node has 0 or 2 children), ordered (i.e. the children of a node are totally ordered), and there is a bijection between the set of leaves and $\mathcal{V}$ (i.e. every variable is depicted by a single leaf) (Pipatsrisawat & Darwiche, 2008; Bova, 2016; Bollig & Buttkus, 2019; Nakamura, Denzumi, & Nishino, 2020). Additionally, the total order on all of its nodes is obtained by an inorder traversal of the vtree nodes (i.e. left subtree, node, right subtree) and all nodes are labeled with a number for ID.[3]

Let η be an NNF formula, circuit or binary decision diagram with input variables $\mathcal{V}$. We say that η, with fan-in 2 in every $\wedge$-gate or any equivalent, *respects* a vtree $\mathcal{T}$ if for every $\wedge$-gate g in η (with left input gate g_1 and right g_2), there exists an internal gate τ in $\mathcal{T}$ (with left child τ_1 and right child τ_2) such that the input gates of the subcircuit g_1 mention only variables in τ_1 and the input gates of the subcircuit g_2 mention only variables in τ_2. The connection between a vtree and an SDD is tied to this, in the sense that every SDD must respect the vtree on which it is based on.

A vtree is linear if for every internal node one child is a leaf. The reason why it is said that SDDs generalize OBDDs is because an SDD based on a linear vtree can represent an OBDD respecting the same variable ordering (Bollig & Buttkus, 2019). Depending on the chosen vtree, substructures of an SDD can be better reused when representing a Boolean function, e.g. a propositional formula, which becomes important to obtain a compact representation. An important feature of SDDs is that they can easily be combined via propositional operations, resulting in a new SDD (Darwiche, 2011b).

In other terms, a vtree represents a set of variable partitions to follow in order to generate the desired SDD. It is important to note that for a given vtree there is a unique SDD trimmed (i.e. it does not have decompositions of the form $\{(\top, \alpha)\}$ and $\{(\alpha, \top), (\neg\alpha, \bot)\}$) and compressed (i.e. for each partition, there are no repeated *sub*s) that will be generated from it. With this in mind, the goal would be to find a vtree which ideally would allow us to compile the smallest SDD possible. The details are beyond the scope of this document, but this search can be performed using *swap* and *rotate* operations (Choi & Darwiche, 2013).[4]

SDDs can also be translated into propositional formulas, which always have negation normal form (NNF), which is characterized by the exclusive use of disjunctions, conjunctions, and negations, with negations only being applied to atomic propositions. More importantly, these SDDs formulas feature structured decomposition and strong determinism (Darwiche, 2011b), which means that they are a strict subset of d-DNNF (i.e. NNF formulas that are deterministic and decomposable). As for the decision of why this type of d-DFNN was chosen, one could also work with an OBDD, but, for reasons of speed and conciseness (Van den Broeck & Darwiche, 2015; Bova, 2016; Bollig & Buttkus, 2019), it is more advisable to work with SDDs.

Although SDDs can also be generated based on formulas in disjunctive normal form (DNF) and the algorithm used could be adapted to convert BNNs into DNF formulas, for these there is no well-defined upper bound of complexity like for the CNF (Darwiche, 2011b), so it seemed best to stick with the latter option.

[2] Extending OBDDs, which have special kinds of vtrees that capture the condition that variables in a path must always appear in the same order. This generalization makes SDDs much more succinct than OBDDs (Van den Broeck & Darwiche, 2015; Bova, 2016; Bollig & Buttkus, 2019).

[3] For the program that we use, the numbering of the nodes is given by a left-to-right traversal of the vtree nodes (Choi & Darwiche, 2018), but this numbering is arbitrary and non-essential, as long as it stays consistent with the respective SDDs based on it.

[4] For our experiments, this vtree search is also automatically handled by *PySDD*.

Algorithm 2 Transformation from dDBC to dDBCSFi(2)

Input: A *dDBC*, with output node *output_node*.
Output: A *dDBCSFi(2)* equivalent to the given *dDBC*.

conjunction($circuit_1$, $circuit_2$): Conjoins $circuit_1$ and $circuit_2$, simplifying the *true*s.
disjunction($circuit_1$, $circuit_2$): Disjoins $circuit_1$ and $circuit_2$, simplifying the *false*s.
negation(*circuit*): Removes or adds a negation on *circuit*, depending on whether its output node is a negation or not, resp. If applied to a truth value, it is inverted (i.e. *false* becomes *true* and *true* becomes *false*).

```
function FIX_NODE(dDBC_node)
    if dDBC_node is a disjunction then
        new_circuit = false
        for each subcircuit in dDBC_node do
            fixed_subcircuit = FIX_NODE(subcircuit)
            if fixed_subcircuit is a true value or is equivalent to −new_circuit then
                return true
            else if fixed_subcircuit is not a false value then
                for each variable v in new_circuit and not in fixed_subcircuit do
                    fixed_subcircuit = conjunction(fixed_subcircuit, disjunction(v, −v))
                for each variable v in fixed_subcircuit and not in new_circuit do
                    new_circuit = conjunction(new_circuit, disjunction(v, −v))
                new_circuit = disjunction(new_circuit, fixed_subcircuit)
        return new_circuit
    else if dDBC_node is a conjunction then
        new_circuit = true
        for each subcircuit in dDBC_node do
            fixed_subcircuit = FIX_NODE(subcircuit)
            if fixed_subcircuit is a false value or is equivalent to −new_circuit then
                return false
            else if fixed_subcircuit is not a true value then
                new_circuit = conjunction(new_circuit, fixed_subcircuit)
        return new_circuit
    else if dDBC_node is a negation then
        return negation(FIX_NODE(negation(dDBC_node)))
    else                                   ▷ (dDBC_node is a literal or a truth value)
        return dDBC_node
dDBCSFi(2) = FIX_NODE(output_node)
```

The SDD returned should be converted to a dDBC. The process is quite simple, we just have to take each node of the graph as its equivalent in a Boolean tree. This means that a decision node becomes a disjunction node, an element becomes a conjunction node, and literals and truth values become input nodes. All maintaining the connections between the nodes. Because of the triviality of this process, its mention is omitted on the path (5.1).

The final step would be to ensure that the resulting circuit is smoothed and has an fan-in 2. That is, we have to convert the dDBC to a dDBCSFi(2).

It is possible to transform an arbitrary dDBC into a dDBCSFi(2), as follows. In a bottom-up fashion, similar to what is suggested in (Arenas et al., 2023), for each conjunction or disjunction gate with fan-in $m > 2$, it must be rewritten as a chain of $m - 1$ gates of the same type, with fan-in 2.

On the other hand, to ensure smoothness, for each disjunction gate (now with fan-in 2), fed by subcircuits C_1 and C_2, we must find the set of all variables present in C_1 and not in C_2 (V_{1-2}), together with all those that are in C_2 and not in C_1 (V_{2-1}). For each variable $v \in V_{2-1}$, we redefine C_1 as $C_1 \wedge (v \vee -v)$. As you might expect, for each variable $v \in V_{1-2}$, C_2 is redefined as $C_2 \wedge (v \vee -v)$. For example, for $(x_1 \wedge x_2 \wedge x_3) \vee (x_2 \wedge -x_3)$, we would get $[(x_1 \wedge x_2) \wedge x_3] \vee [(x_2 \wedge -x_3) \wedge (x_1 \vee -x_1)]$. The formal method can be found in the

Algorithm 2 and, since it requires going through all the nodes of the circuit just once, we can notice that its complexity is linear, with respect to the number of nodes.

This completes the compilation of the path (5.1), giving us a dDBCSFi(2) with the properties we need. Using the resulting dDBCSFi(2)s is fairly straightforward. If the value of a variable x_i is equal to 1, then it is interpreted as a *true* value and all x_i gates are replaced by *true*. Otherwise, if the value of x_i is -1, then it is interpreted as a *false* value and all x_i gates are replaced by *false*. The rest is classical propositional logic.

It is interesting to note that, for a binary perceptron, the compilation of the BNN to a CNF formula can be adapted to simply return a dDBCSFi(2). This is because the encoding of a single neuron is already decomposable and can be easily modified to enforce the other properties. It may seem like a good alternative, but an additional experiment of ours proved otherwise; since doing this for a 13-variable binary perceptron returned a circuit with 15,439 nodes, in contrast to 4,571 nodes by using the full path. This can be seen at the end of Appendix A.

5.4 A Complete Example of the Conversion

Some of the steps in (5.1) may not be polynomial-time transformations, which we will discuss in more technical terms later in this section, as well as in the next one. However, we can claim at this stage that: (a) Any exponential cost of a transformation is kept under control by a usually small parameter. (b) The resulting dDBCSFi(2) is meant to be used multiple times, to explain different and multiple outcomes; and then, it may be worth taking a one-time, relatively high transformation cost. A good reason for our transformation path is the availability of implementations we can take advantage of.

We will describe, explain and illustrate the conversion path (5.1) by means of a running example with a simple BNN.

Example 1 The BNN in Figure 5.2 has hidden neuron gates h_1, h_2, h_3, an output gate o, and three input gates, x_1, x_2, x_3, that receive binary values.

The latter represent, together, an input entity $\bar{x} = \langle x_1, x_2, x_3 \rangle$ that is being classified by means of a label returned by o. Each gate g is activated by means of a *step function* (5.2). For technical, non-essential reasons, for the gates, we use 1 and -1. However, the output gate employs an activation function that returns instead 1 or 0, for *true* or *false*, resp. For example, h_1 is *true*, i.e. outputs 1, for an input $\bar{x} = (x_1, x_2, x_3)$ iff $\bar{w}_{h_1} \bullet \bar{x} + b_{h_1} = (-1) \times x_1 + (-1) \times x_2 + 1 \times x_3 + 0.16 \geq 0$. Otherwise, h_1 is *false*, i.e. it returns -1.

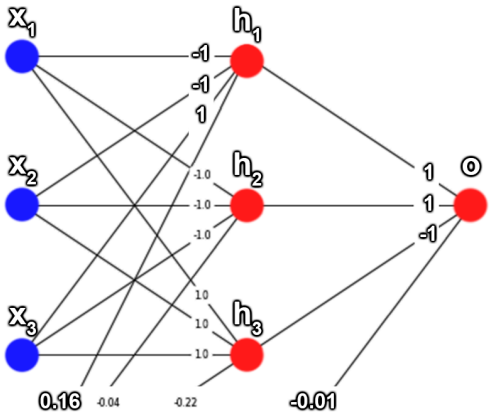

FIGURE 5.2: A BNN.

Similarly, output gate o is *true*, i.e. returns label 1 for a binary input $\bar{h} = (h_1, h_3, h_3)$ iff $\bar{w}_o \bullet \bar{h} = 1 \times h_1 + 1 \times h_2 + (-1) \times h_3 - 0.01 \geq 0$, and 0 otherwise. □

The first step, (a) in (5.1), consists in representing the BNN as a CNF formula. For this, we adapt the approach in (Narodytska et al., 2018), in their case, to verify properties of BNNs. Contrary to them, we avoid the use of auxiliary variables since their posterior elimination conflicts with our need for determinism.

Each gate of the BNN is represented by a propositional formula, initially not necessarily in CNF, which, in its turn, is used as one of the inputs to gates next to the right. In this way, we eventually obtain a defining formula for the output gate. The formula is converted into CNF. The participating propositional variables are logically treated as *true* or *false*, even if they take numerical values 1 or -1, resp.

Example 2 (example 1 cont.) Consider gate h_1, with parameters $\bar{w} = \langle -1, -1, 1\rangle$ and $b = 0.16$, and input $\bar{i} = \langle x_1, x_2, x_3\rangle$. An input x_j is said to be *conveniently instantiated* if it has the same sign as w_j, and then, contributing to the activation function to return 1. E.g., this is the case of $x_1 = -1$. In order to represent as a propositional formula its output variable, also denoted with h_1, we first compute the number, d, of conveniently instantiated inputs that are necessary and sufficient to make $\bar{w} \bullet \bar{i} + b$ greater than or equal to 0. This is the (only) case when h_1 becomes *true*; otherwise, it is *false*. This number can be computed in general by (5.3). In the case of h_1, with 2 negative weights: $d = \lceil(-0.16 + (-1 - 1 + 1))/2\rceil + 2 = 2$. With this, we can impose conditions on two input variables with the right sign at a time, considering all possible convenient pairs. For h_1 we obtain its condition to be true:

$$h_1 \longleftrightarrow (-x_1 \wedge -x_2) \vee (-x_1 \wedge x_3) \vee (-x_2 \wedge x_3). \tag{5.8}$$

This is a DNF formula, directly obtained from considering all possible convenient pairs (which is already better that trying all cases of three variables at a time). However, there is a more expedite, iterative method that still uses the number of convenient inputs (cf. Section 5.2). Using this algorithm, we obtain an equivalent formula defining h_1:

$$h_1 \longleftrightarrow (x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1). \tag{5.9}$$

Similarly, we obtain defining formulas for gates h_2 and h_3, and o: (for all of them, $d = 2$)

$$\begin{aligned} h_2 &\longleftrightarrow (-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1), \\ h_3 &\longleftrightarrow (x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1), \\ o &\longleftrightarrow (-h_3 \wedge (h_2 \vee h_1)) \vee (h_2 \wedge h_1). \end{aligned} \tag{5.10}$$

Replacing the definitions of h_1, h_2, h_3 into (5.10), we finally obtain:

$$\begin{aligned} o \longleftrightarrow\ & (-[(x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1)] \wedge \\ & ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)] \vee \\ & [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)])) \vee \\ & ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)] \wedge \\ & [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)]). \end{aligned} \tag{5.11}$$

The final part of step (a) in path (5.1), requires transforming this formula into CNF. In this example, it can be taken straightforwardly into CNF.[5] The resulting CNF formula is, in its turn, simplified into a shorter and simpler new CNF formula by means of the SAT solver *Riss* (Manthey, 2017). For this example, the simplified CNF formula is as follows:

$$o \longleftrightarrow (-x_1 \vee -x_2) \wedge (-x_1 \vee -x_3) \wedge (-x_2 \vee -x_3). \tag{5.12}$$

Having a CNF formula will be convenient for the next conversion steps along path (5.1). □

[5] For our experiments, we programmed a simple algorithm that does this job, while making sure the generated CNF does not grow too much (see Section 5.2).

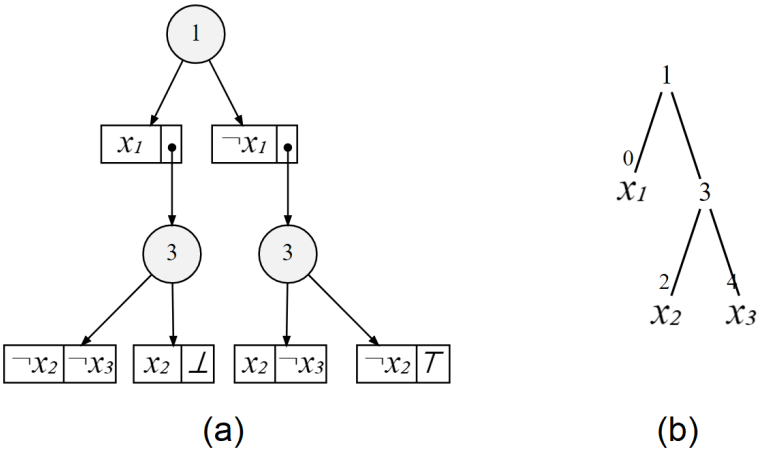

FIGURE 5.3: An SDD (a) and a vtree (b).

Following with step (b) along path (5.1), the resulting CNF formula is transformed into an SDD.

Example 3 (example 2 cont.) Figure 5.3(a) shows an SDD, $\mathcal{S}$, to be used for illustration. (See (Bova, 2016; Nakamura et al., 2020) for more definitions than the one from Section 5.3.) As previously said in Section 5.3, an SDD has different kinds of nodes. Those represented with encircled numbers are decision nodes (Van den Broeck & Darwiche, 2015), e.g. ① and ③, that consider alternatives for the inputs (in essence, disjunctions). There are also nodes called elements. They are labeled with constructs of the form $[\ell_1|\ell_2]$, where ℓ_1, ℓ_2, called the *prime* and the *sub*, resp., are Boolean literals, e.g. x_1 and $\neg x_2$, including $\top$ and $\bot$, for 1 or 0, resp. E.g. $[\neg x_2|\top]$ is one of them. Either the *prime* or the *sub* can also be a pointer, $\bullet$, with an edge to a decision node. $[\ell_1|\ell_2]$ represents two conditions that have to be satisfied simultaneously (in essence, a conjunction). A rectangle of an element without $\bullet$ is a terminal.

An SDD represents (or defines) a total Boolean function $F_{\mathcal{S}} : \langle x_1, x_2, x_3\rangle \in \{0,1\}^3 \mapsto \{0,1\}$. For example, $F_{\mathcal{S}}(0,1,1)$ is evaluated by following the graph downwards. Since $x_1 = 0$, we descent to the right; next via node ③ underneath, with $x_2 = 1$, we reach the instantiated leaf node labeled with $[1|0]$, a "conjunction", with the second component due to $x_3 = 1$. We obtain $F_{\mathcal{S}}(0,1,1) = 0$.

Figure 5.3(b) shows a vtree, $\mathcal{T}$, for $\mathcal{V} = \{x_1, x_2, x_3\}$. (See (Bova, 2016; Bollig & Buttkus, 2019; Nakamura et al., 2020) for more definitions than the one from Section 5.3.) Its leaves, $0, 2, 4$, show their associated variables in $\mathcal{V}$. We can see that nodes 1 and 3 convey the partitions $\{x_1\}|\{x_2, x_3\}$ and $\{x_2\}|\{x_3\}$, resp. We can also note that the $\mathcal{S}$ respects $\mathcal{T}$. Intuitively, the variables at the terminals of $\mathcal{S}$, when they go upwards through decision nodes ⓝ, also go upwards through the corresponding nodes n in $\mathcal{T}$.

$\mathcal{S}$ can be obtained from the RHS of (5.12), ψ, by generating and combining SDDs from it, following the partitions in $\mathcal{T}$, from the simplest partition to the most complex one. This means first generating the SSDs that respect the partition $\{x_2\}|\{x_3\}$, for all operation between x_2 and x_3 in ψ, and then the ones that respect $\{x_1\}|\{x_2, x_3\}$, combining with the previously generated SDDs.

The SDD S can be straightforwardly represented as a propositional formula by interpreting decision nodes as disjunctions, and elements as conjunctions, obtaining:

$$[x_1 \wedge ((-x_2 \wedge -x_3) \vee (x_2 \wedge \bot))] \vee [-x_1 \wedge ((x_2 \wedge -x_3) \vee (-x_2 \wedge \top))] \tag{5.13}$$

Which is logically equivalent to the formula ψ that represents the BNN. Accordingly, the BNN is represented by the SDD in Figure 5.3(a). □

In our experiments and in the running example, we used the *PySDD* system (Meert & Choi, 2018), which, given a CNF formula ψ, produces a vtree and a compliant SDD, both optimized in size, that represents ψ (Choi & Darwiche, 2013, 2018).

This compilation takes space and time that are exponential only in the *tree-width*, $TW(\psi)$, of ψ, which is the tree-width of the graph $\mathcal{G}$ associated to ψ (Darwiche, 2011b; Oztok & Darwiche, 2014). $\mathcal{G}$ contains the variables as nodes, and undirected edges between any of them when they appear in a same clause. The tree-width measures how close the graph is to being a tree. The exponential upper-bound on the tree-width is a positive *fixed-parameter tractability* result (Flum & Grohe, 2006) in that $TW(\psi)$ is in general much smaller $|\psi|$.

For example, the graph $\mathcal{G}$ for the formula ψ on the RHS of (5.12) has x_1, x_2, x_3 as nodes, and edges between any pair of variables, which makes $\mathcal{G}$ a complete graph. Since every complete graph has a tree-width equal to the number of nodes minus one, we have $TW(\psi) = 2$.

Our final transformation step consists in obtaining a dDBC from the resulting SDD and a dDBCSFi(2) from said dDBC. An SDD turns out to correspond to a d-DNNF Boolean circuit, for which decomposability and determinism hold, and has only variables as inputs to negation gates (Darwiche, 2011b). The class d-DNNF is contained in dDBC, so the first part is pretty straight forward.

We must remember that the reason why we want a dDBCSFi(2) is that the algorithm for Shap computing (Algorithm 1) requires the dDBC to be a dDBCSFi(2). Every dDBC can be transformed in linear time into a dDBCSFi(2) (Arenas et al., 2023). More details can be found in Section 5.3.

Example 4 (example 3 cont.) By interpreting decision nodes and elements as disjunctions and conjunctions, resp., the SDD in Figure 5.3(a) can be easily converted into d-DNNF circuit. Notice that only variables are affected by negations. However, due to the children of node ③, that do not have the same variables, the directly resulting dDBC is not smooth (it has fan-in 2 though). This can be solved by taking our equivalent formula (5.13) and applying Algorithm 2 on it. We now proceed to show the steps of the final transformation for the formula (5.13), highlighting in bold the added elements and changes of each iteration.

1: $(\_) \mid (\_)$
2: $(\boldsymbol{x_1}\ \boldsymbol{\&}\ (\_)) \mid (\_)$
3: $(x_1\ \&\ ((\_) \mid (\_))) \mid (\_)$
4: $(x_1\ \&\ ((\boldsymbol{-x_2}\ \boldsymbol{\&}\ (\_)) \mid (\_))) \mid (\_)$
5: $(x_1\ \&\ ((-x_2\ \&\ \boldsymbol{-x_3}) \mid (\_))) \mid (\_)$
6: $(x_1\ \&\ ((-x_2\ \wedge -x_3) \mid (\_))) \mid (\_)$
7: $(x_1\ \&\ ((-x_2 \wedge -x_3) \mid (\boldsymbol{x_2}\ \boldsymbol{\&}\ (\_)))) \mid (\_)$
8: $(x_1\ \&\ ((-x_2 \wedge -x_3) \mid (x_2\ \&\ \bot))) \mid (\_)$
9: $(x_1\ \&\ ((-x_2 \wedge -x_3) \mid \bot)) \mid (\_)$
10: $(x_1\ \&\ (\boldsymbol{-x_2} \wedge \boldsymbol{-x_3})) \mid (\_)$
11: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (\_)$
12: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (\boldsymbol{-x_1}\ \boldsymbol{\&}\ (\_))$
13: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((\_) \mid (\_)))$
14: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((\boldsymbol{x_2}\ \boldsymbol{\&}\ (\_)) \mid (\_)))$
15: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((x_2\ \&\ \boldsymbol{-x_3}) \mid (\_)))$
16: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((x_2 \wedge -x_3) \mid (\_)))$

17: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \;\&\; ((x_2 \wedge -x_3) \mid (-\boldsymbol{x_2} \;\&\; (\_))))$
18: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \;\&\; ((x_2 \wedge -x_3) \mid (-x_2 \;\&\; \top)))$
19: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \;\&\; ((x_2 \wedge -x_3) \mid -\boldsymbol{x_2}))$
20: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \;\&\; ((x_2 \wedge -x_3) \mid (-\boldsymbol{x_2} \wedge (\boldsymbol{x_3} \vee -\boldsymbol{x_3}))))$
21: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \;\&\; ((x_2 \wedge -x_3) \vee (-x_2 \wedge (x_3 \vee -x_3))))$
22: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1 \wedge ((x_2 \wedge -x_3) \vee (-x_2 \wedge (x_3 \vee -x_3))))$
23: $(x_1 \wedge (-x_2 \wedge -x_3)) \vee (-x_1 \wedge ((x_2 \wedge -x_3) \vee (-x_2 \wedge (x_3 \vee -x_3))))$

And, as can be seen, the algorithm gave us:

$$[x_1 \wedge (-x_2 \wedge -x_3)] \vee [-x_1 \wedge ((x_2 \wedge -x_3) \vee [-x_2 \wedge (x_3 \vee -x_3)])] \tag{5.14}$$

This formula (5.14) is equivalent to the dDBCSFi(2) shown in Figure 3.1. □

Figure 5.4 summarizes this example with the central diagrams.

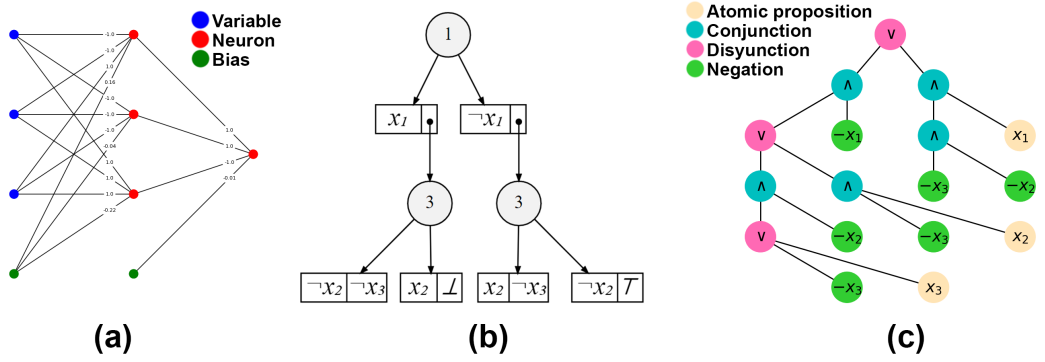

FIGURE 5.4: Real example of the conversion of a BNN (a) to an SDD (b) and to a dDBCSFi(2) (c).

An additional example, for a binary perceptron, is presented in Figure 5.5. As stated at the end of Section 5.3, the algorithm for converting a BNN into a CNF formula can be adapted to directly transform a perceptron to a dDBCSFi(2), as shown in the illustration. Although, in this case, the number of nodes in the resulting dDBCSFi(2)s are equivalent, it must be remembered that this is not always the case. Anticipating to the analysis in this regard, since

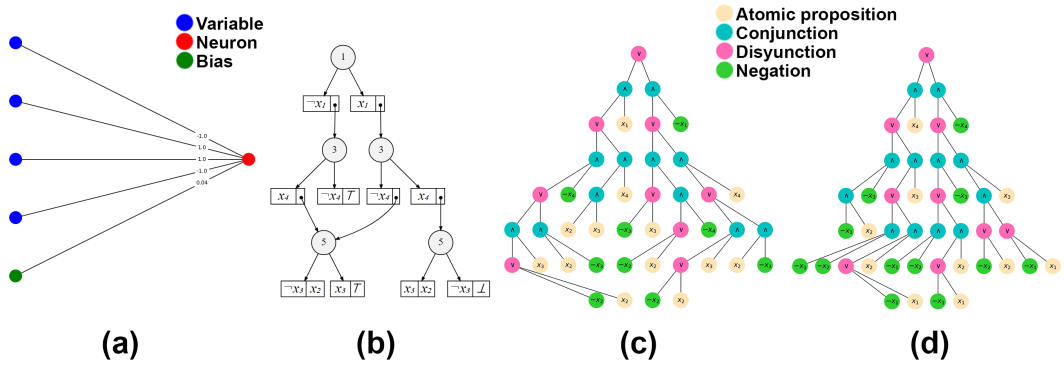

FIGURE 5.5: Real example of the conversion of a binary perceptron (a) to an SDD (b) and to a dDBCSFi(2) (c). (d) shows the dDBCSFi(2) obtained directly from the perceptron, by adjusting the algorithm.

it is not a central matter on this research, it can be commented that perhaps further refinement in this alternative path could considerably narrow this gap in the number of nodes that is seen when scaling the number of variables.

5.5 On the Efficiency of the Method

The following assumptions will be made: (a) The weights have the same probability of being 1 or -1. (b) The biases will all be 0. Like before, we are also considering a BNN that receives ℓ_0 input variables in the form of $\bar{x} = \langle x_1, \ldots, x_{\ell_0} \rangle$. As in Section 5.1, this BNN has m hidden layers, each layer z (from 1 to m) has ℓ_z of neurons and the neurons from layer z receive the input vector $\bar{i} = \langle i_1, \ldots, i_{\ell_{z-1}} \rangle$. For the output layer, it has a single neuron. Similarly, all weights are -1 or 1 and the activation functions also return a value of -1 or 1. The only exception to the latter is the output layer, where a step function is used that returns 0 or 1.

With these assumptions, the threshold d (5.3) for a given neuron with $|\bar{i}|$ inputs can be estimated as $d = \left(\left\lceil \frac{b+\sum_{j=1}^{|\bar{i}|} w_j}{2} \right\rceil + \sum_{j=1}^{|\bar{i}|} \frac{-w_j+1}{2} \right) \approx \left(\left\lceil \frac{0+0}{2} \right\rceil + \frac{0+|\bar{i}|}{2} \right) = \left(0 + \frac{|\bar{i}|}{2} \right) \leq \left\lceil \frac{|\bar{i}|}{2} \right\rceil$. Now, to dimension the number of clauses for this neuron, we can observe how this number grows for some variables and thresholds, just by applying distribution on each disjunction of CNF formulas to generate a single CNF formula (based on our algorithm for the encoding and assuming that the neuron receives only variables as input, not Boolean formulas). This can be seen in the following matrix C, with components $c_{k,t}$, where the indices of the rows represent the number k of variables and those of the columns the threshold t to reach or surpass (with $k \in \mathbb{N}$ and $t \in \mathbb{N}$):

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\ 1 & 2 & 1 & 1 & 1 & 1 & 1 & \\ 1 & 4 & 3 & 1 & 1 & 1 & 1 & \\ 1 & 8 & 15 & 4 & 1 & 1 & 1 & \ldots \\ 1 & 16 & 135 & 64 & 5 & 1 & 1 & \\ 1 & 32 & 2{,}295 & 8{,}704 & 325 & 6 & 1 & \\ 1 & 64 & 75{,}735 & 19{,}984{,}384 & 2{,}829{,}125 & 1{,}956 & 7 & \\ & & & \ldots & & & & \end{bmatrix} \tag{5.15}$$

The growth in the number of clauses follows this rule for the lower triangular matrix: $c_{k,t} = (c_{k-1,t-1} + 1) \cdot c_{k-1,t}$, $k > t > 1$. Since we define our d to be $\left\lceil \frac{|\bar{i}|}{2} \right\rceil$, it seems that we are in an interval where the number of clauses grows at an exponential rate if is left untreated.[6] And all this does not even consider neurons that do not receive mere variables, but Boolean formulas. This is why our code does some basic simplifications during the encoding and a big one at the end with a specialized SAT solver *Riss*. However, the question of how much it really helps and how much can be reduced remains unanswered.

For reference, one can consider that a truth table of a model, for all combinations of $|\bar{x}|$ variables, can be converted to a CNF formula by taking all combinations that return 0, negating/inverting their values and concatenating all the combinations with conjunctions, taking each one as a disjunction of variables. So, in the worst case, you could be having up to $2^{|\bar{x}|}$

[6] Thinking on literals as inputs, it is possible that the smallest number of clauses to represent a neuron is equal to $\binom{|\bar{i}|}{d}$, but we have not studied this with enough detail to be sure.

clauses; number that is much lower than what would be obtained for the mentioned neuron, considering that it has 7 inputs, for example.

Given the above, it is noteworthy that simplifications are really important in the algorithm, but that does not mean that the number of clauses will not grow exponentially with the number of variables.

The previous is just considering the size growth, but, as said in Section 5.2, these simplifications imply a great time cost. This is because, to perform said simplifications in our implementation, at every neuron several cycles are done through all clauses, which raises the time complexity from polynomial to exponential. Nevertheless, this probably could be fixed with a better optimization of the code.

Added to this, as shown in (Darwiche, 2011b), SDDs have a well-defined upper bound of complexity, but worrying when compiled from a CNF formula. This is determined by its number of variables $|\bar{x}|$ and the width of the tree ω of the CNF formula.[7] Focusing again on the worst case, we can expect $\omega = |\bar{x}| - 1$. Thus, the upper bound for an SDD generated from a CNF formula could have a complexity of $O(|\bar{x}|2^{|\bar{x}|-1})$. Again, we would be up against an exponential size. This means that no matter how much we simplify a CNF formula, if enough variables influence the result, we still risk not being able to compile its SDD. In any case, the exponential upper bound with the width of the tree is a positive result of fixed parameters (Flum & Grohe, 2006), considering that ω is generally smaller than $|\bar{x}| - 1$.

Both aspects limit the number of variables and the complexity of the BNN that could be used. However, it is still feasible to work with at least 13 variables, as it will be demonstrated in the experiments.

[7] This tree is around the graph that is generated by ℓ_0 nodes that represent each of the variables in the formula, where every possible edge exists if the two connected variables appear at the same time in any of the clauses of the formula.

Chapter 6

Description of the Experiments

As stated on Chapter 1 and Section 4.2, the dataset used for the experiments, to train the models and calculate the Shap-scores, was *California Housing Prices* (Nugent, 2018) (which was first introduced in (Pace & Barry, 1997)). Its numeric features were binarized according to whether their values were greater than the average (1) or less (-1), while one-hot encoding

Feature	Description	Original values	Binarization
#1 *ocean_proximity* (predictor)	Label of the location of the house w.r.t. the sea/ocean	Labels *1h_ocean* (#1$_a$), *inland* (#1$_b$), *island* (#1$_c$), *near_bay* (#1$_d$) and *near_ocean* (#1$_e$)	Five new features (one for each label), for which 1 means it is the value of *ocean_proximity*, and -1 means it is not
#2 *households* (predictor)	Total number of households (group of people residing within a housing unit) in a block	Integer numbers from 1 to 6,082	1 (above average of the feature) or -1 (below the average)
#3 *housing_median_age* (predictor)	Average age of a house within a block (lower numbers mean newer buildings)	Integer numbers from 1 to 52	1 (above average of the feature) or -1 (below the average)
#4 *latitude* (predictor)	Angular measure of how far north a block is (a higher value means it is further north)	Real numbers from 32.54 to 41.95	1 (above average of the feature) or -1 (below the average)
#5 *longitude* (predictor)	Angular measure of how far west a block is (a higher value means it is further west)	Real numbers from -124.35 to -114.31	1 (above average of the feature) or -1 (below the average)
#6 *median_income* (predictor)	Median household income within a block (measured in tens of thousands of US dollars)	Real numbers from 0.50 to 15.00	1 (above average of the feature) or -1 (below the average)
#7 *population* (predictor)	Total number of people residing within a block	Integer numbers from 3 to 35,682	1 (above average of the feature) or -1 (below the average)
#8 *total_bedrooms* (predictor)	Total number of bedrooms within a block	Integer numbers from 1 to 6,445	1 (above average of the feature) or -1 (below the average)
#9 *total_rooms* (predictor)	Total number of rooms within a block	Integer numbers from 2 to 39,320	1 (above average of the feature) or -1 (below the average)
#10 ***median_house_value*** (target)	Median house value for households within a block (measured in US dollars)	Integer numbers from 14,999 to 500,001	1 (above average of the feature) or 0 (below the average)

TABLE 6.1: Features of the *California Housing Prices* dataset.

was used for the categorical one. 13 predictive features were obtained with an additional one (the target or output) that represented whether the price of each block was high or low (i.e. above or below the average). A summary can be found in Table 6.1. This dataset was used to train a BNN (with one hidden layer, with the same number of neurons as features) and a binary perceptron (BP). Both are used in the experiments.

Preliminarily, for the BNN three models were trained (using the *Tensorflow* library):

1. One with real-valued parameters, rectifier activation function for the hidden layer and sigmoid for the output.
2. Another with real-valued parameters, hyperbolic tangent activation function for the hidden layer and sigmoid for the output.
3. And a true binary one, which used a specialized library for binary layers (called *Larq*).

For the BP only the last two types of models (2 and 3) were trained. The weights of these models were converted to binary values (1 and -1), the activation functions of the hidden layers were changed to sign functions and the output functions to unit step functions (both shown in (5.2)). With the objective of defining which is the best type of model for the central experiments, the variations in accuracies and losses after the change were recorded to compare them. The loss function chosen was the binary cross-entropy, which is defined as:

$$\text{Binary cross-entropy}(\bar{y}, \hat{\bar{y}}) := -\frac{1}{|\bar{y}|}\sum_{i=1}^{|\bar{y}|}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{6.1}$$

In (6.1), $\hat{\bar{y}}$ represents the labels estimated by the model and $\bar{y}$ are the true labels.

Type of model		**Phase**	**Binary cross-entropy**	**Accuracy**	**Training time (in seconds)**
BNN	Non-binary model	Before the binarization	0.4633	0.7879	14.72
		After the binarization	2.9138	0.7773	
	Pseudo binary model	Before the binarization	0.4705	0.7865	10.59
		After the binarization	1.2153	0.7827	
	Binary model (with Larq)	Before the binarization	0.9041	0.6580	6.41
		After the binarization	0.9041	0.6580	
BP	Pseudo binary model	Before the binarization	0.5525	0.7416	15.56
		After the binarization	0.5884	0.7269	
	Binary model (with Larq)	Before the binarization	0.7437	0.7093	10.17
		After the binarization	0.7437	0.7093	

TABLE 6.2: Performance information for the BNNs and BPs trained on the *California Housing Prices* dataset.

The results of this preliminary experiment are found on Table 6.2. In it, we can observe that not only the results of the binary models (with *Larq*) remain consistent after the binarization, but they seem to give good enough results (based on the loss). Because of this, we chose to use the models with *Larq* for the following steps. Graphical representations, in the form of Boolean trees, of the generated dDBCSFi(2)s from each model are shown in Appendix A, which we suspect that may be helpful in some way in future research.

According to the transformation path (5.1), both trained models were first represented as a CNF formula with 2,391 clauses for the BNN and 1,716 for the BP. Both have a tree-width of 12. These CNF formulas were transformed, via the SDD conversion, into dDBCSFi(2)s which ended up having 18,671 nodes for the BNN and 4,571 nodes for the BP (without counting the negations affecting only input gates). The initial transformation into CNF took 1.3 hrs with the BNN and 5.2389 s with the BP. This is *practically* the most expensive step, due to the reasons given in Chapter 5. The conversion of the simplified CNF into the dDBCSFi(2)s took 0.8276 s for the BNN and 0.3150 s for the BP.

The experiments consisted of calculating the Shap-scores for 100 different entities that were present on the training data subset. For the direct method, we tried both the original models and the resulting dDBCSFi(2)s and we also did the calculation using the efficient algorithm on the dDBCSFi(2)s.

Taking $\text{ent}(\mathcal{F})$ as the set of all entities over $\mathcal{F}$, for all 13 features and assuming a uniform probability distribution (i.e. all entities have $\frac{1}{2^{13}}$ chances of occurrence, as P^u of (3.3)), Shap was calculated directly (like a black-box) with the following formula, which is derived from (3.2):

$$\mathsf{Shap}(\mathcal{F}, \mathcal{G}_{\mathbf{e}}, F) \quad = \tag{6.2}$$

$$\sum_{S \subseteq \mathcal{F} \setminus \{F\}} \frac{|S|!(|\mathcal{F}| - |S| - 1)!}{|\mathcal{F}|!} \left(\sum_{\substack{\mathbf{e}' \in \text{ent}(\mathcal{F}) \\ \mathbf{e}'_{S \cup \{F\}} = \mathbf{e}_{S \cup \{F\}}}} \frac{L(\mathbf{e}')}{2^{12-|S|}} - \sum_{\substack{\mathbf{e}' \in \text{ent}(\mathcal{F}) \\ \mathbf{e}'_S = \mathbf{e}_S}} \frac{L(\mathbf{e}')}{2^{13-|S|}} \right)$$

And for the open-box approach, we simply followed the efficient algorithm described in Chapter 3 (i.e. the Algorithm 1), using a probability of 0.5 for all features.

For the training we used *Google Colab* (with an *NVIDIA Tesla T4* enabled) and employed a train-test split of 50% on the data, as well as 200 epochs, with an early stopping that allowed 40 epochs of patience. The version of *Tensorflow* used was 2.9.2. All the code is on a *Jupyter Notebook*, which is available at the following link for anyone interested in the details or on doing similar experiments: https://github.com/Jorvan758/dDBCSFi2 (it is important to note that it was designed to run on *Google Colab*). See Appendix B for more details on the repository, including an overview on the code and available files.

Chapter 7

Experimental Results and Analysis

Presenting all Shap-scores would have been excessive, so the average of each feature were included in Figure 7.1 instead. An exact match was seen in the Shap-scores of both methods (i.e. black-box and open-box), on each pair of values in the 100 entities considered, for all the features. For this reason, only four plots of averages are presented: two that contain the averages of the original Shap-scores based on each model, and another two that were

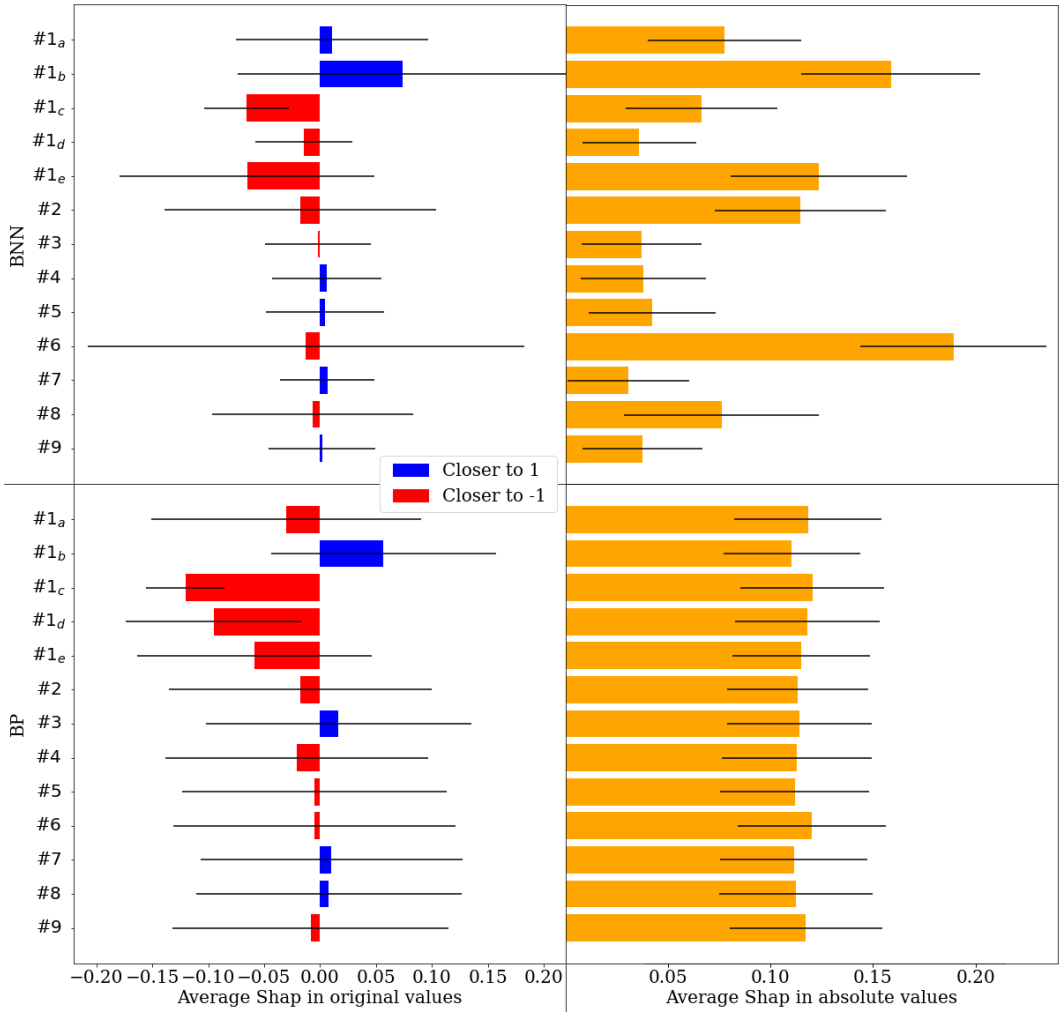

FIGURE 7.1: Average Shap-score for each feature in 100 different entities from *California Housing Prices*, based on the original Shaps (left) and their absolute values (right), for both the BNN (above) and the BP (below).

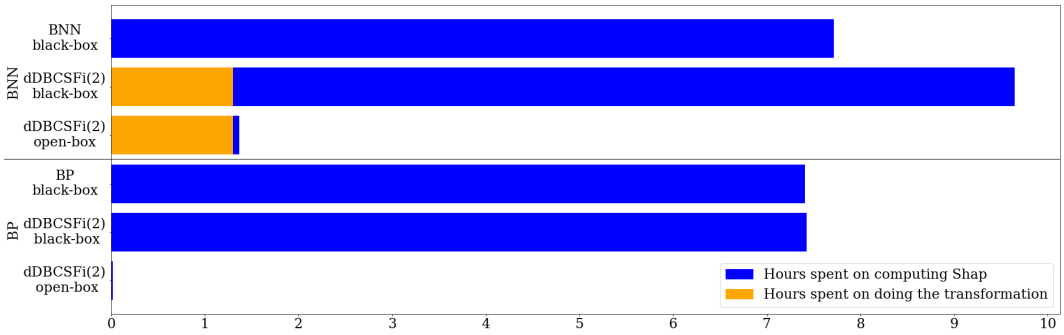

FIGURE 7.2: Hours needed to make the initial transformation to dDBCSFi(2) (orange bar) and calculate Shap over 100 entities (blue bar), both for the BNN (first three bars) and for the BP (last three bars).

calculated on the absolute values of the Shaps recorded for each model. In this way, it is not only possible to know the usual category with which each feature is associated, but also the average relevance that it has on the final result.

Assuming a uniform distribution across the entities, these results tell us that the three most impactful features in the BNN are $\#1_b$, $\#1_e$, and #6. Oddly enough, when considering the meaning of the feature, it appears that #6 (the most impactful feature and the only one based on a number among the top three) tends to lower the home values. This may be a sign of a deficiency in the BNN, supported by the relatively low accuracy seen in Table 6.2. However, since the averages of the original values in the BP fall short of how binary one might expect them to be, this is probably due in part to not choosing a sufficiently large and diverse sample of entities. This is just a demonstration of how someone might use this information to explain and tune their model(s).

It is important to say that the coincidence in the values of both methods (i.e. black-box and open-box) is mainly due to the rare assumption that all entities have the same probability of occurring ($\frac{1}{2^{13}}$). As we distance from it, we can expect a greater difference in values. This could be amended by changing the probability of each feature, considering a product distribution for the entities, as long as the features have statistical independence. This last condition is hard to find in real world data, so it would be convenient to find another alternative.

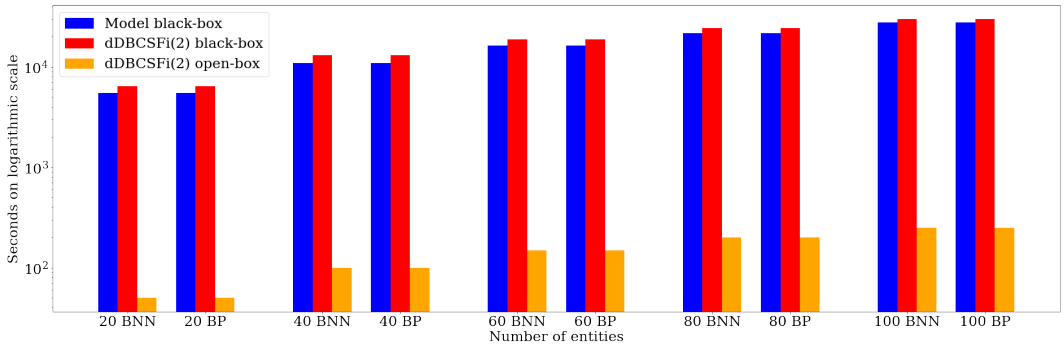

FIGURE 7.3: Seconds needed to compute Shap over 20, 40, 60, 80 and 100 entities; using the original model as a black-box (blue bar), the dDBCSFi(2) as a black-box (red bar) and the dDBCSFi(2) as an open-box (orange bar). This is for both models (i.e. the BNN and the BP). Note that the vertical axis uses a logarithmic scale.

Finally, the times necessary to carry out the transformations and calculations are presented in Figure 7.2. It should be remembered that the transformation to generate a dDBCSFi(2) is performed only once and, after this, the circuit can be reused as many times as desired (and it is in this reuse where the greatest gain of this method is seen).

Complementarily, included in Figure 7.3 is a plot with the times it takes to calculate 20, 40, 60, 80 and 100 entities, with each method and model. Remember that the black-box refers to the direct method and the open-box to the efficient one and notice that these times are represented in *logarithmic scale* (if it was not the case, we would clearly see a linear growth in all bars, but most bars of the open-box cases could not be seen). For example, with the BNN, the original model took 7.7 hrs to compute all the Shap-scores for 100 entities, whereas its dDBCSFi(2), treated as an open-box, just took 4.2 min. Those times do not show the one-time computation for the transformation of the models into the dDBCSFi(2)s. If the latter was added, each red and orange bar of the BNN would have an increase of 1.3 hrs, while the ones from the BP would have an increase of just 5.2389 s. For reference, even considering this extra one-time computation, with the open-box approach on the dDBCSFi(2) of the BNN we can still compute all of the Shap-scores for 100 entities in less time than with the original model with just 20 entities.

Complementarily, it must be said that the generated dDBCSFi(2) based on the BNN has 18,671 nodes, while that of the BP only has 4,571. Also, the BNN can predict labels of multiple entities at the same time, while the circuit must go one by one, which explains the time differences between using the black-box method on the original model vs. the generated dDBCSFi(2).

It is easy to see that not only does the time spent doing the transformation grows, as the network becomes more complex (appreciable by the fact that for the BNN it takes 1.3 h, while for the BP only 6 s, despite having the same number of features), but also the size of the dDBCSFi(2). The latter evidently changes the time it takes to traverse the circuit and, consequently, to perform operations with it. This reaffirms the need to find size optimizations for these circuits.

Although the implementation itself could probably be improved, it is confirmed that the efficient algorithm reduces the time needed. Even though the conversion process takes longer as the number of features, neurons, and layers grow, it is clearly a faster alternative for a BNN that is simple enough.

Chapter 8

Conclusions

The validity and usefulness of the algorithm described in (Arenas et al., 2021) has been empirically demonstrated. There are many situations where it might end up being more advisable than the direct method. Specifically, with a low enough number of variables and a relatively simple BNN. However, there is still work to be done to really minimize the time it takes to compute Shap-scores and generate robust guidelines for its application.

It is important to stress once again that the efforts invested in transforming the BNN into a dDBCSFi(2) are only done once, with the most costly step being the compilation of the BNN into a CNF formula. By comparison, subsequent transformations to an SDD and dDBCSFi(2) take much less time. For example, for the BNN of the experiments, generating the CNF formula took 1.3 h, while the rest of the compilation to dDBCSFi(2) only took 0.8276 s. Despite the intrinsic complexity involved, there should be plenty of room for improvement in the algorithmic and implementation aspects of the compilation. The same applies to the implementation of the efficient algorithm for Shap computation. Furthermore, the circuit can be used for other purposes, such as *verification* of general properties of the classifier (Narodytska et al., 2018; Darwiche & Hirth, 2020).

Regardless of the algorithmic and implementation issues for computing Shap, an important research problem has to do with bringing *domain knowledge* or *domain semantics* to definitions and computations of attribution scores, in order to obtain more meaningful and interpretable results. This additional knowledge could come, for example, in declarative terms, expressed as *logical constraints*. They could be used to appropriately modify the algorithm or the underlying distribution (Bertossi, 2021). In addition, it is likely that domain knowledge can be more easily included in a calculation of score when performed over a logical classifier, such as a dDBCSFi(2), rather than on a BNN.

Finally, among the future related challenges to be addressed, the following can be identified:

- Explore the consequences of using different probability distributions for the efficient algorithm. We use only the uniform distribution, but the results should be just as good for the product distribution (see (Bertossi et al., 2020) for a discussion of its empirical version and related issues).
- Answer if there is an efficient way to convert any traditional neural network into a BNN or not. To the best of our knowledge, this question has not been officially resolved for the large family of neural networks nor considered a priority issue, as seen in (Qin et al., 2020; Yuan & Agaian, 2021; Simons & Lee, 2019). If so, what compromises would be involved and what kind of results linked to Shap would the dDBCSFi(2) of such BNN produce?

- Find out how to further minimize the size of the generated dDBCSFi(2) or the time it takes to compile it, while keeping its properties.
- Corroborate, in ideal implementations, how the computation time of Shap for a BNN compares between the original model and the respective dDBCSFi(2), including the time needed for the conversion.
- Determine more accurately how the size of the circuits scales with the complexity of the BNN and confirm if there really is a point where using the direct method on the BNN is faster than the efficient one on the dDBCSFi(2).
- Reexamine the efficient algorithm to find out if it is feasible to improve it further. Possibly, incorporating domain knowledge.
- Empirically compare the speed of generating SDDs based on formulas in CNF vs. in DNF, in order to check if it makes sense to use the latter in other conversion paths from BNN to dDBCSFi(2).

Acknowledgments

To carry out these experiments, some researchers were contacted in order to understand some concepts of their work and/or ask for help with some code. Therefore, I want to leave a special thanks to:

- Arthur Choi, for his *PySDD* repository and all his kind advice on how to use it, while also answering fundamental questions.
- Andy Shih, for sharing some preliminary code for the conversion of BNNs into CNF, and explaining many related issues.
- Norbert Manthey, for his *Riss tool collection* repository, as well as answering questions about its use and the simplification of CNF formulas.
- Maximilian Schleich, for sharing some preliminary code for the black-box computation of Shap.
- Adnan Darwiche, for sharing useful insights.

Part of this work was funded by ANID - Millennium Science Initiative Program - Code ICN17002; and "Centro Nacional de Inteligencia Artificial" CENIA, FB210017 (Financiamiento Basal para Centros Científicos y Tecnológicos de Excelencia de ANID), Chile. Both CENIA and SKEMA Canada funded my visit to Montreal.

Finally, I want to thank my thesis supervisor, Leopoldo Bertossi, whose generous help was indispensable throughout this research. L. Bertossi is a Professor Emeritus at Carleton University, Ottawa, Canada; and a Senior Fellow of the Universidad Adolfo Ibáñez (UAI), Chile.

References

Arenas, M., Barceló, P., Bertossi, L., & Monet, M. (2021). The Tractability of SHAP-Score-Based Explanations for Classification over Deterministic and Decomposable Boolean Circuits. In *Proceedings of the 35th aaai conference on artificial intelligence* (p. 6670-6678). pages 2, 4, 6, 7, 8, 9, 10, 31

Arenas, M., Barceló, P., Bertossi, L., & Monet, M. (2023). On the Complexity of SHAP-Score-Based Explanations: Tractability via Knowledge Compilation and Non-Approximability Results. *Journal of Machine Learning Research*, *24*(63), 1-58. pages 2, 17, 21

Bertossi, L. (2021). Declarative Approaches to Counterfactual Explanations for Classification. *Theory and Practice of Logic Programming*, 1–35. (https://doi.org/10.1017/S1471068421000582. Posted as Corr ArXiv Paper 2011.07423) pages 31

Bertossi, L. (2022). Score-Based Explanations in Data Management and Machine Learning: An Answer-Set Programming Approach to Counterfactual Analysis. In M. Simkus & I. Varzinczak (Eds.), *Reasoning web. declarative artificial intelligence, lecture notes in computer science 13100* (p. 145-184). Springer. pages 1, 5

Bertossi, L., Li, J., Schleich, M., Suciu, D., & Vagena, Z. (2020). Causality-Based Explanation of Classification Outcomes. In *Proceedings of the 4th international workshop on "data management for end-to-end machine learning" (deem) at acm sigmod/pods* (p. 1-10). (Posted as Corr ArXiv Paper 2003.06868) pages 1, 2, 5, 31

Bollig, B., & Buttkus, M. (2019). On the Relative Succinctness of Sentential Decision Diagrams. *Theory of Computing Systems*, *63*(6), 1250–1277. pages 15, 16, 20

Bova, S. (2016). SDDs Are Exponentially More Succinct than OBDDs. In *Proceedings of the 30th aaai conference on artificial intelligence* (p. 929-935). pages 7, 16, 20

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, *C-35*(8), 677-691. pages 2, 15

Chockler, H., & Halpern, J. (2004). Responsibility and Blame: A Structural-Model Approach. *Journal of Artificial Intelligence Research*, *22*, 93-115. pages 1, 5

Choi, A., & Darwiche, A. (2013). Dynamic Minimization of Sentential Decision Diagrams. In *Proceedings of the 27th aaai conference on artificial intelligence* (p. 187-194). pages 15, 16, 21

Choi, A., & Darwiche, A. (2018). SDD Advanced-User Manual Version 2.0 [Computer software manual]. pages 15, 16, 21

Darwiche, A. (2011a). On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics*, *11*(1-2), 11-34. pages 2

Darwiche, A. (2011b). SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the 22th international joint conference on artificial intelligence (ijcai-11)* (p. 819-826). pages 2, 15, 16, 21, 24

Darwiche, A., & Hirth, A. (2020). On The Reasons Behind Decisions. In *Proceedings of the 24th european conference on artificial intelligence* (p. 712-720). pages 2, 31

Darwiche, A., & Marquis, P. (2002). A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, *17*(1), 229–264. pages 2

Flum, J., & Grohe, M. (2006). *Parameterized Complexity Theory*. Springer. pages 21, 24

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., & Pedreschi, D. (2018). A Survey of Methods for Explaining Black Box Models. *ACM Computing Surveys*, *51*(5), 1–42. pages 2

Halpern, J. Y., & Pearl, J. (2005). Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Philosophy of Science*, *56*(4), 843-887.

pages 1, 5

Lundberg, S., Erion, G., Chen, H., DeGrave, A., Prutkin, J., Nair, B., ... Lee, S.-I. (2020). From Local Explanations to Global Understanding with Explainable AI for Trees. *Nature Machine Intelligence*, *2*(1), 56-67. (ArXiv Paper 1905.04610) pages 2, 5

Lundberg, S. M., & Lee, S.-I. (2017). A Unified Approach to Interpreting Model Predictions. In *Proceedings of the 31st international conference on neural information processing systems* (p. 4768–4777). (ArXiv Paper 1705.07874) pages 1, 5

Manthey, N. (2017). *Riss tool collection.* `https://github.com/nmanthey/riss-solver`. pages 13, 19

Meert, W., & Choi, A. (2018). *Python Wrapper Package to Interactively use Sentential Decision Diagrams (SDD).* `https://github.com/wannesm/PySDD`. pages 15, 21

Nakamura, K., Denzumi, S., & Nishino, M. (2020). Variable Shift SDD: A More Succinct Sentential Decision Diagram. In *Proceedings of the 18th international symposium on experimental algorithms (sea 2020), leibniz international proceedings in informatics 160* (p. 22:1-22:13). pages 16, 20

Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., & Walsh, T. (2018). Verifying Properties of Binarized Deep Neural Networks. In *Proceedings of the 32nd aaai conference on artificial intelligence* (p. 6615–6624). pages 2, 4, 6, 10, 11, 14, 18, 31

Nugent, C. (2018). *California Housing Prices.* `https://www.kaggle.com/datasets/camnugent/california-housing-prices`. pages 2, 9, 25

Oztok, U., & Darwiche, A. (2014). On Compiling CNF into Decision-DNNF. In *Proceedings of the 20th international conference on principles and practice of constraint programming, lecture notes in computer science 8656* (p. 42-57). pages 15, 21

Oztok, U., & Darwiche, A. (2017). On Compiling DNNFs without Determinism. *ArXiv*, *1709.07092*. pages 13

Pace, R. K., & Barry, R. (1997). Sparse Spatial Autoregressions. *Statistics & Probability Letters*, *33*(3), 291-297. pages 25

Pipatsrisawat, T., & Darwiche, A. (2008). New Compilation Languages Based on Structured Decomposability. In *Proceedings of the 23rd aaai conference on artificial intelligence* (p. 517-522). pages 16

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., & Sebe, N. (2020). Binary Neural Networks: A Survey. *Pattern Recognition*, *105*, 107281. pages 31

Ras, G., Xie, N., van Gerven, M., & Doran, D. (2022). Explainable Deep Learning: A Field Guide for the Uninitiated. *Journal of Artificial Intelligence Research*, *73*, 329-396. pages 2

Roth, A. E. e. (1988). *The Shapley Value: Essays in Honor of Lloyd S. Shapley*. Cambridge University Press. pages 1, 5

Rudin, C. (2019). Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead. *Nature Machine Intelligence*, *1*, 206-215. (ArXiv Paper 1811.10154) pages 1

Shapley, L. S. (1953). A Value for n-Person Games. In *Contributions to the theory of games (am-28)* (Vol. 2, p. 307-318). pages 1

Shi, W., Shih, A., Darwiche, A., & Choi, A. (2020). On Tractable Representations of Binary Neural Networks. In *Proceedings of the 17th international conference on principles of knowledge representation and reasoning* (p. 882-892). (ArXiv Paper 2004.02082) pages 2, 4, 7, 10

Shih, A., Darwiche, A., & Choi, A. (2019). Verifying Binarized Neural Networks by Angluin-Style Learning. In *Proceedings of the theory and applications of satisfiability testing - sat 2019, lecture notes in computer science 11628* (p. 354-370). pages 4, 6, 10

Simons, T., & Lee, D.-J. (2019). A Review of Binarized Neural Networks. *Electronics*, *8*(6), 661. pages 31

Sinz, C. (2005). Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In P. van Beek (Ed.), *Proceedings of the principles and practice of constraint programming - cp 2005* (p. 827-831). pages 6, 12

Van den Broeck, G., & Darwiche, A. (2015). On the Role of Canonicity in Knowledge Compilation. In *Proceedings of the 29th aaai conference on artificial intelligence* (p. 1641-1648). pages 15, 16, 20

Van den Broeck, G., Lykov, A., Schleich, M., & Suciu, D. (2021). On the Tractability of SHAP Explanations. In *Proceedings of the 35th aaai conference on artificial intelligence* (p. 6505-6513). pages 2, 7

Yuan, C., & Agaian, S. S. (2021). A Comprehensive Review of Binary Neural Network. *ArXiv*, *2110.06804*. pages 31

Appendix A

Extra Material from the Experiments

As anticipated in Chapter 6, in Figures A.1, A.2 and A.3 we can see the graphical representations of the generated dDBCSFi(2)s, obtained for the experiments. As also said in Section 5.3, Figure A.3 uses an alternative method (that goes directly from BP to dDBCSFi(2)) that may be promising if further thought is put into it, but right now ends up being much more inefficient. Ignoring the mere aesthetic aspect of these Boolean trees, we suspect that future research may benefit in some degree by having them for reference.

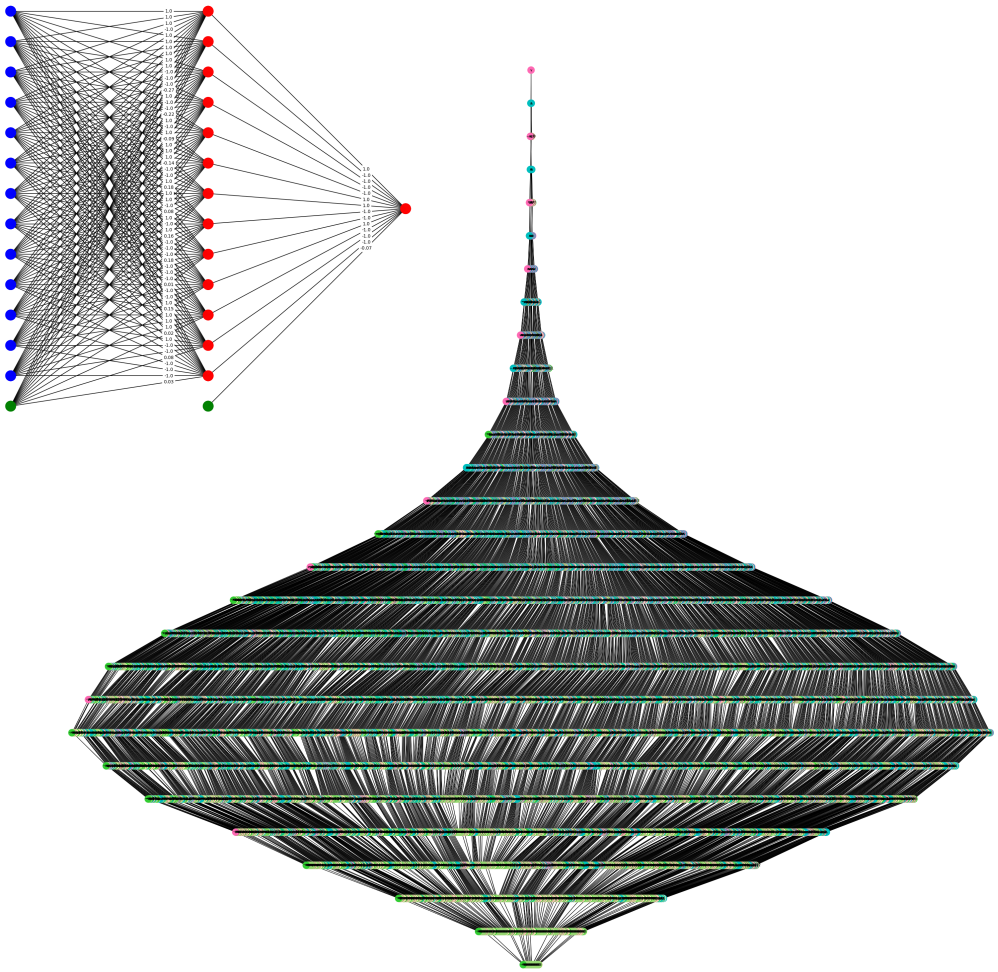

FIGURE A.1: dDBCSFi(2), as a Boolean tree, of the BNN (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 18,671 nodes.

We take the chance to mention that, as an additional guarantee, the equivalence of each dDBCSFi(2) with its respective model was corroborated by our program for all of the 8,192 possible combinations of variables (i.e. all possible entities). Needless to say, the matches were exact.

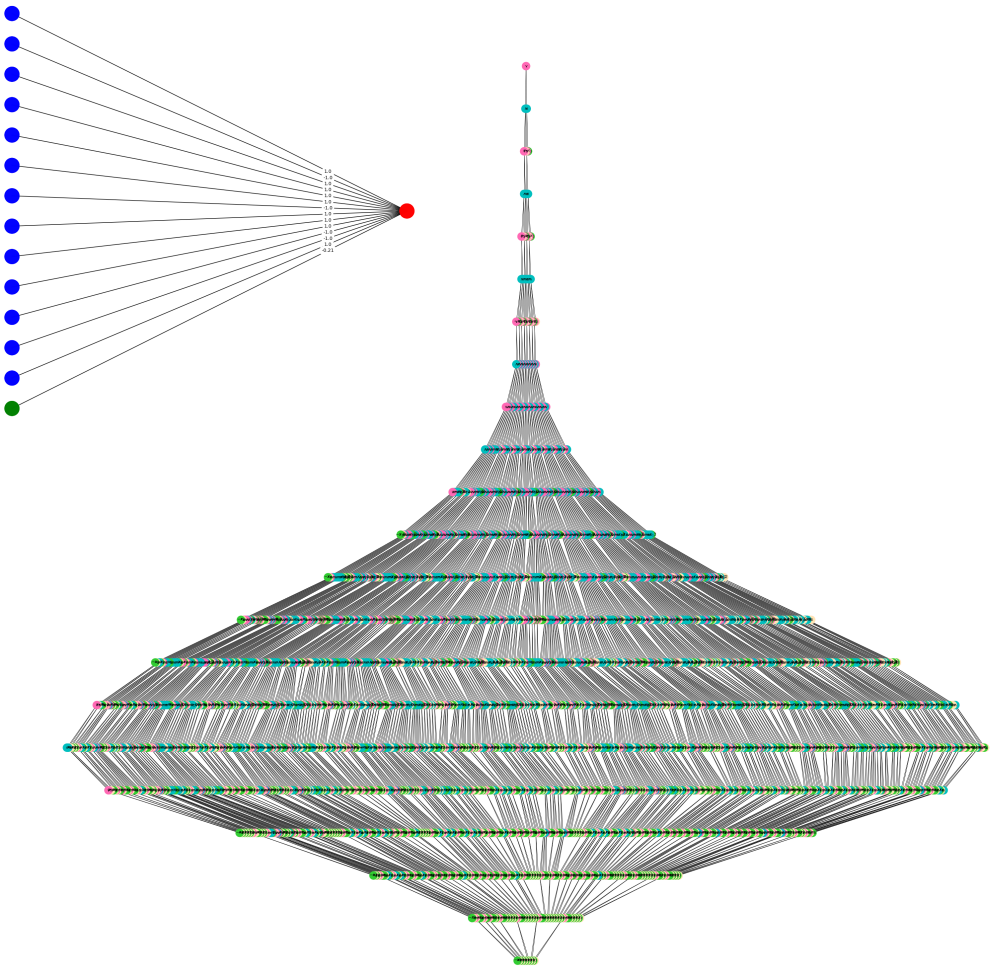

FIGURE A.2: dDBCSFi(2), as a Boolean tree, of the BP (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 4,571 nodes.

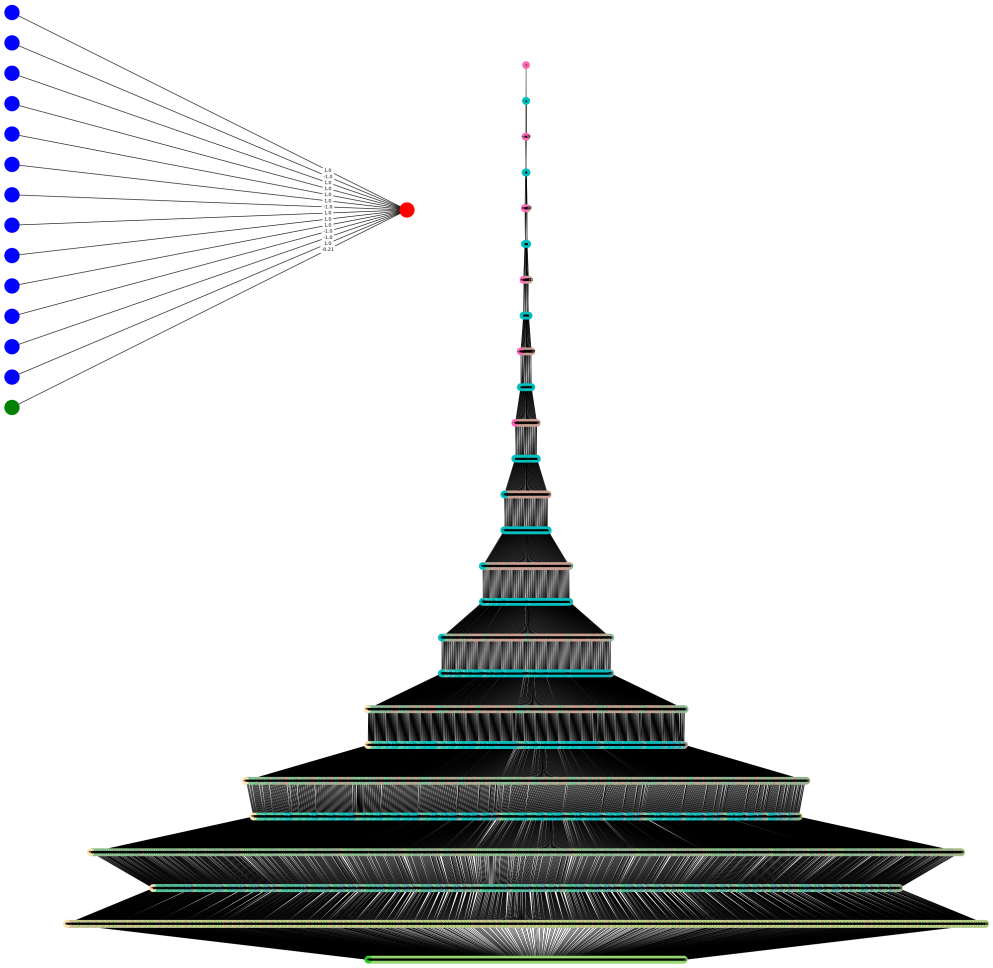

FIGURE A.3: Alternative dDBCSFi(2) (obtained directly, omitting the CNF formula and the SDD), as a Boolean tree, from the BP (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 15,439 nodes.

Appendix B

About the GitHub Repository

As stated at the end of Chapter 6, all the code is available at:

`https://github.com/Jorvan758/dDBCSFi2`

This repository contains:

1. The *California Housing Prices* dataset, as the `housingc.csv` file.
2. The weights and biases of the models used in our experiments (both of the BNN and the BP), as the files `BNN_weights.h5` and `BP_weights.h5`.

   These can be loaded onto a network with the respective architecture of Chapter 6, via the `load_weights()` function.
3. We also give the CNF formulas obtained, in four files: (a) One with the original formula for the BNN (`BNN_CNFf.cnf`); (b) Another with the simplified version of said formula (`BNN_CNFf_simplified.cnf`); as well as (c) One with the original formula of BP (`BP_CNFf.cnf`); and (d) Another with its simplified version (`BP_CNFf_simplified.cnf`). All in the *DIMACS CNF* format.
4. All code is available in the *Jupyter Notebook* `dDBCSFi(2).ipynb`.

   With our implementation, the SDD and dDBCSFi(2) are not, by default, saved as a files, as they are only kept in memory (from which the Shap-scores are calculated). In any case, their compilations are extremely fast, making use of the CNF files we provided. Therefore, we omit their inclusion.

   The code is written in *Python* and is designed to run in *Google Colab* (to run it on another machine, some adjustments will probably be necessary). It is also meant to be executed in descending order (if no changes are made, it will give the same results as we got).

   The code is divided into three sections **Common preparations**, **BNN Experiments** and **BP Experiments**, which we will describe below.

Common Preparations: It contains eight subsections:

- **Random seeds**: Defines a function that initializes all relevant random seeds with the specific values one chooses. The default values are the ones that we used.
- **Installations**: Installs the *PySDD* and *Larq* libraries, and the SAT solver *Riss*. *Tensorflow* is already installed by default.
- **Tensorflow**: Loads the library to train the models and defines a function to plot them.
- **CNF formula**: Contains everything related to converting BNNs into CNF formulas, with the method described in Section 5.2.
- **SDD**: Loads the *PySDD* library to convert CNF formulas into SDDs.
- **dDBCSFi(2)**: Defines the *Python* class for the circuits that we use, including methods for compiling them from SDDs, plotting them, checking for equivalence with the original model, counting nodes, predicting labels for entities, and, of course, calculating Shap efficiently.
- **SHAP**: Defines functions to compute multiple Shaps as a black-box for both BNNs and dDBCSFi(2)s, and as an open-box for dDBCSFi(2)s.
- **Preprocessing of the dataset**: Binarizes the *California Housing Prices* dataset, as described in Chapter 6. Additionally, it automatically downloads it from the repository.

BNN Experiments & **BP Experiments**: The structure of both sections is identical, so the three subsections that make up each one will only be described once:

- **Model training and testing**: Trains the respective model, as described in Chapter 6, ensuring that the weights and activation functions are binarized. It also provides performance information on the test data subset.
- **Conversion of the model to a dDBCSFi(2)**: Follows the path described in Chapter 5, converting the model to a CNF formula, then to an SDD, and finally to a dDBCSFi(2). All execution times involved are also recorded. Additionally, it corroborates the equivalence with the original model, and plots both the latter and the dDBCSFi(2).
- **SHAP calculation**: Computes Shap for 100 different entities, present in the training data subset, recording the time to compute 20, 40, 60, 80, and 100 of them. This is done with all three methods, i.e. with the original model as a black-box, with the dDBCSFi(2) as a black-box, and with the dDBCSFi(2) as an open-box. All execution times and their averages, mentioned in Chapter 7, are also printed.