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 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

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, Barceló, Bertossi, & Monet, 2021). This transformation follows the next path:

$$\underset{\text{(a)}}{\text{BNN} \longmapsto} \text{ CNF formula } \underset{\text{(b)}}{\longmapsto} \text{ SDD } \underset{\text{(c)}}{\longmapsto} \text{ dDBCSFi(2)} \tag{1.1}$$

This is not the only way to get a dDBCSFi(2). For example, (Shi, Shih, Darwiche, & Choi, 2020) converts BNNs to OBDDs, which can also be converted to dDBCSFi(2)s. Some of the steps in (1.1) 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 1.1). Then the modified version will be presented, which does not use auxiliary variables (Section 1.2). This will be followed by the remaining steps to get from a CNF formula to a dDBCSFi(2) (Section 1.3). An example illustrating the entire process will also be shown to ensure its understanding (Section 1.4). Finally, the section will end with a small analysis on the efficiency of the conversion method (Section 1.5).

1.1 BNN to CNF Formula: Auxiliary Variables

This method can be seen in (Shih, Darwiche, & Choi, 2019; Narodytska, Kasiviswanathan, Ryzhyk, Sagiv, & Walsh, 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:

$$\phi_{\text{hidden layer}}(x) := \begin{cases} 1 & , \quad x \geq 0 \\ -1 & , \quad x < 0 \end{cases} \qquad \phi_{\text{output layer}}(x) := \begin{cases} 1 & , \quad x \geq 0 \\ 0 & , \quad x < 0 \end{cases} \tag{1.2}$$

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:

$$
d := \left\lceil \frac{-b + \sum_{j=1}^{\ell_{z-1}} w_j}{2} \right\rceil + \sum_{j=1}^{\ell_{z-1}} \frac{-w_j + 1}{2} \tag{1.3}
$$

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:

$$
SQ(\bar{w}, \bar{i}, d) := \begin{cases}
r_{(\ell_{z-1},d)} & , \ d < 1 \\
\begin{array}{l}
(w_1 \cdot i_1 \Leftrightarrow r_{(1,1)}) \wedge \\
\bigwedge_{j=2}^{d} (-r_{(1,j)}) \wedge \\
\bigwedge_{l=2}^{\ell_{z-1}} ((r_{(l,1)} \Leftrightarrow (w_l \cdot i_l \vee r_{(l-1,1)})) \wedge \\
\qquad \bigwedge_{j=2}^{d} (r_{(l,j)} \Leftrightarrow ((w_l \cdot i_l \ \wedge \ r_{(l-1,j-1)}) \vee r_{(l-1,j)})))
\end{array} & , \begin{array}{l} |\bar{i}| \geq d \\ d \geq 1 \end{array} \\
-r_{(\ell_{z-1},d)} & , \begin{array}{l} |\bar{i}| < d \\ d \geq 1 \end{array}
\end{cases} \tag{1.4}
$$

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:

$$\hat{d}_{kl} := \left\lceil \left\lceil \frac{b_l - b_k + \sum_{j=1}^{\ell_m} (w_{k_j} - w_{l_j})}{2} \right\rceil / 2 \right\rceil + \left| \left\{ w \in \frac{\bar{w}_k - \bar{w}_l}{2} \mid w = -1 \right\} \right| \tag{1.5}$$

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:

$$\bar{\omega}_{kl} := \left\{ w \in \frac{\bar{w}_k - \bar{w}_l}{2} \mid w \neq 0 \right\} \qquad \bar{\imath}_{kl} := \left\{ i \in \frac{\bar{i} \odot (\bar{w}_k - \bar{w}_l)}{2} \mid i \neq 0 \right\} \tag{1.6}$$

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: $\bar{r}_k = \langle r_{(|\bar{\omega} k 1|, \hat{d}_{k1})}, \ldots, r_{(|\bar{\omega}_{kn}|, \hat{d}_{kn})} \rangle$. 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).

1.2 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 (1.3). We can see this process as filling an matrix $M_{|\tilde{i}|\times d}$ 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:

$$M := \begin{bmatrix} w_1 \cdot i_1 & \mathit{false} & \mathit{false} & \dots & \mathit{false} \\ w_2 \cdot i_2 \vee c_{1,1} & w_2 \cdot i_2 \wedge c_{1,1} & \mathit{false} & \dots & \mathit{false} \\ w_3 \cdot i_3 \vee c_{2,1} & (w_3 \cdot i_3 \wedge c_{2,1}) \vee c_{2,2} & w_3 \cdot i_3 \wedge c_{2,2} & \dots & \mathit{false} \\ \dots & \dots & \dots & \dots & \dots \\ w_{|\bar{i}|} \cdot i_{|\bar{i}|} \vee c_{|\bar{i}|-1,1} & (w_{|\bar{i}|} \cdot i_{|\bar{i}|} \wedge c_{|\bar{i}|-1,1}) \vee c_{|\bar{i}|-1,2} & (w_{|\bar{i}|} \cdot i_{|\bar{i}|} \wedge c_{|\bar{i}|-1,2}) \vee c_{|\bar{i}|-1,3} & \dots & (w_{|\bar{i}|} \cdot i_{|\bar{i}|} \wedge c_{|\bar{i}|-1,d-1}) \vee c_{|\bar{i}|-1,d} \end{bmatrix} \tag{1.7}$$

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 1.1, 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 1.1 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 1.5 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 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.

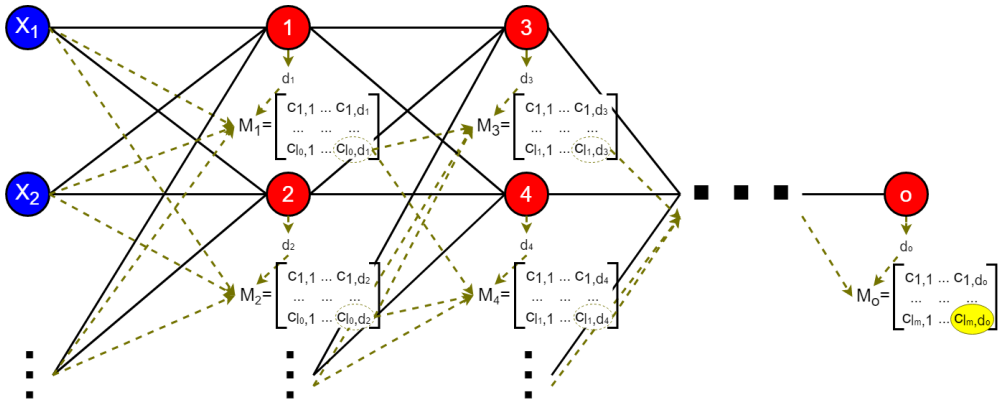

FIGURE 1.1: Conversion from a BNN to a CNF formula. The inputs for the first layer are the respective variables (blue nodes). Each neuron (red nodes) g has a d_g and a M_g, of which the component $c_{|\bar{i}|,d_g}$ is the final encoding of g. These final encodings are given as inputs to the M_gs of 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.

1.3 CNF Formula to dDBCSFi(2)

As noted in (Darwiche, 2011; Oztok & Darwiche, 2014), a CNF formula can be converted into a *Sentential Decision Diagram* (SDD) (Darwiche, 2011; 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 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

[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, 2011) and commented in (Choi & Darwiche, 2013).

[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).

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, 2011).

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, 2011), 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, 2011), so it seemed best to stick with the latter option.

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 (1.1).

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).

[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 1 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*).

```
1:  function FIX_NODE(dDBC_node)
2:      if dDBC_node is a disjunction then
3:          new_circuit = false
4:          for each subcircuit in dDBC_node do
5:              fixed_subcircuit = FIX_NODE(subcircuit)
6:              if fixed_subcircuit is a true value or is equivalent to −new_circuit then
7:                  return true
8:              else if fixed_subcircuit is not a false value then
9:                  for each variable v in new_circuit and not in fixed_subcircuit do
10:                     fixed_subcircuit = conjunction(fixed_subcircuit, disjunction(v, −v))
11:                 for each variable v in fixed_subcircuit and not in new_circuit do
12:                     new_circuit = conjunction(new_circuit, disjunction(v, −v))
13:                 new_circuit = disjunction(new_circuit, fixed_subcircuit)
14:         return new_circuit
15:     else if dDBC_node is a conjunction then
16:         new_circuit = true
17:         for each subcircuit in dDBC_node do
18:             fixed_subcircuit = FIX_NODE(subcircuit)
19:             if fixed_subcircuit is a false value or is equivalent to −new_circuit then
20:                 return false
21:             else if fixed_subcircuit is not a true value then
22:                 new_circuit = conjunction(new_circuit, fixed_subcircuit)
23:         return new_circuit
24:     else if dDBC_node is a negation then
25:         return negation(FIX_NODE(negation(dDBC_node)))
26:     else                                   ▷ (dDBC_node is a literal or a truth value)
27:         return dDBC_node
28: dDBCSFi(2) = FIX_NODE(output_node)
```

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, Barceló, Bertossi, & Monet, 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 1 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 (1.1), 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.

1.4 A Complete Example of the Conversion

Some of the steps in (1.1) 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 (1.1) by means of a running example with a simple BNN.

Example 1 The BNN in Figure 1.2 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* (1.2). 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.

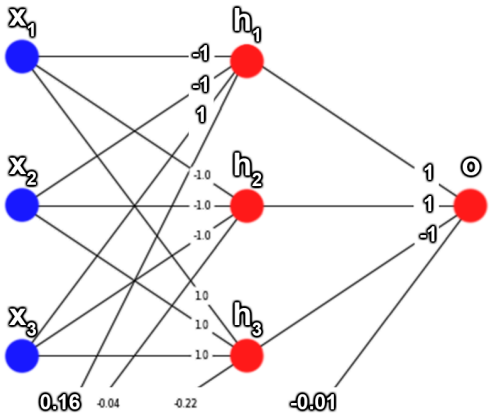

FIGURE 1.2: A BNN.

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 (1.1), 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 (1.3). 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:

$$h_1 \longleftrightarrow (-x_1 \wedge -x_2) \vee (-x_1 \wedge x_3) \vee (-x_2 \wedge x_3). \tag{1.8}$$

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 1.2). Using this algorithm, we obtain an equivalent formula defining h_1:

$$h_1 \longleftrightarrow (x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1). \tag{1.9}$$

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

$$\begin{aligned} h_2 &\longleftrightarrow (-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1), \\ h_3 &\longleftrightarrow (x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1), \\ o &\longleftrightarrow (-h_3 \wedge (h_2 \vee h_1)) \vee (h_2 \wedge h_1). \end{aligned} \tag{1.10}$$

Replacing the definitions of h_1, h_2, h_3 into (1.10), we finally obtain:

$$\begin{aligned} o \longleftrightarrow\ & (-[(x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1)] \wedge \\ & ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)] \vee \\ & [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)])) \vee \\ & ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)] \wedge \\ & [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)]). \end{aligned} \tag{1.11}$$

The final part of step (a) in path (1.1), 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:

$$o \longleftrightarrow (-x_1 \vee -x_2) \wedge (-x_1 \vee -x_3) \wedge (-x_2 \vee -x_3). \tag{1.12}$$

Having a CNF formula will be convenient for the next conversion steps along path (1.1). □

Following with step (b) along path (1.1), the resulting CNF formula is transformed into an SDD.

Example 3 (example 2 cont.) Figure 1.3(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 1.3.) As previously said in Section 1.3, 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

[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 1.2).

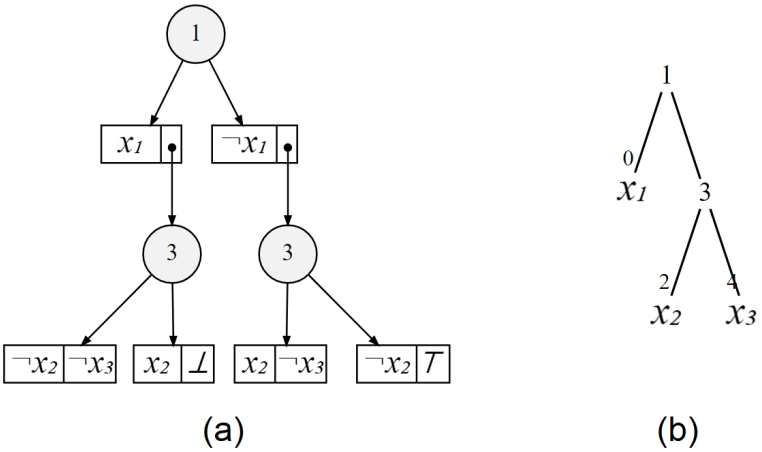

FIGURE 1.3: An SDD (a) and a vtree (b).

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 1.3(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 1.3.) 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 (1.12), 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:

$$[x_1 \wedge ((-x_2 \wedge -x_3) \vee (x_2 \wedge \bot))] \vee [-x_1 \wedge ((x_2 \wedge -x_3) \vee (-x_2 \wedge \top))] \tag{1.13}$$

Which is logically equivalent to the formula ψ that represents the BNN. Accordingly, the BNN is represented by the SDD in Figure 1.3(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, 2011; 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 (1.12) 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, 2011). 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 **??**) 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 1.3.

Example 4 (example 3 cont.) By interpreting decision nodes and elements as disjunctions and conjunctions, resp., the SDD in Figure 1.3(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 (1.13) and applying Algorithm 1 on it. We now proceed to show the steps of the final transformation for the formula (1.13), 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: $(x_1\ \&\ (\boldsymbol{-x_2 \wedge -x_3})) \mid (\_)$
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: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((x_2 \wedge -x_3) \mid (\boldsymbol{-x_2}\ \boldsymbol{\&}\ (\_))))$
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: $(x_1 \wedge (-x_2 \wedge -x_3)) \mid (-x_1\ \&\ ((x_2 \wedge -x_3) \mid \boldsymbol{(-x_2 \wedge (x_3 \vee -x_3))}))$
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:

$$[x_1 \wedge (-x_2 \wedge -x_3)] \vee [-x_1 \wedge ((x_2 \wedge -x_3) \vee [-x_2 \wedge (x_3 \vee -x_3)])] \tag{1.14}$$

This formula (1.14) is equivalent to the dDBCSFi(2) shown in Figure **??**. □

Figure 1.4 summarizes this example with the central diagrams.

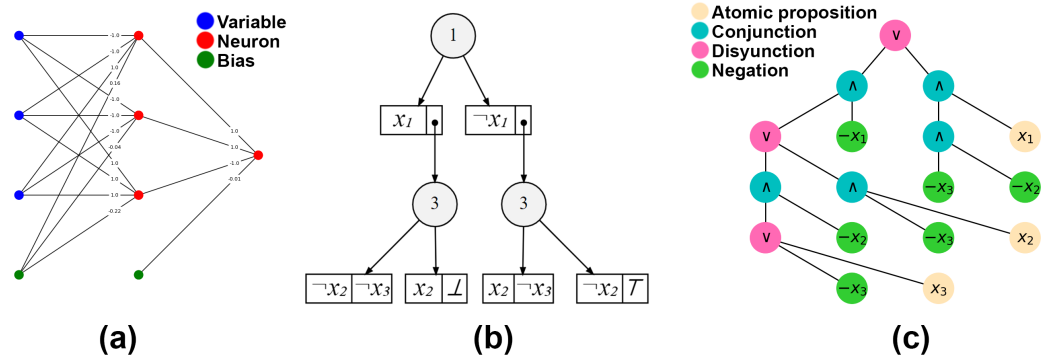

FIGURE 1.4: Real example of the conversion of a BNN (a) to an SDD (b) and to a dDBCSFi(2) (c).

An additional example, for a binary perceptron, is presented in Figure 1.5. As stated at the end of Section 1.3, 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 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.

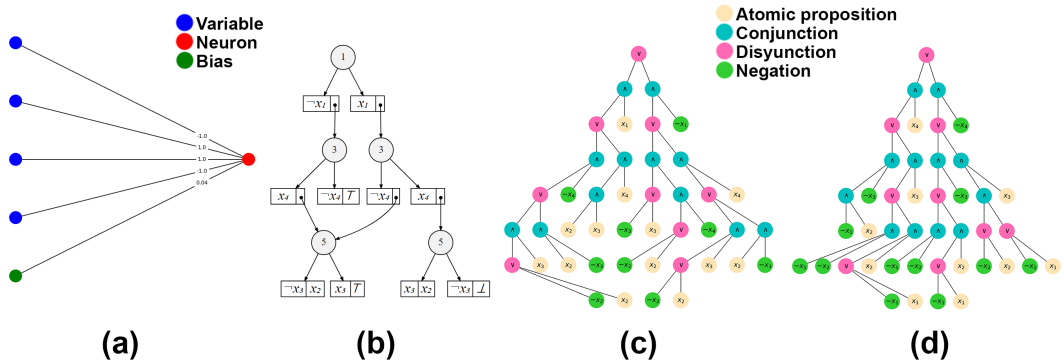

FIGURE 1.5: Real example of the conversion of a binary perceptron (a) to an SDD (b) and to a dDBCSFi(2) (c). (d) shows the dDBCSFi(2) obtained directly from the perceptron, by adjusting the algorithm.

1.5 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 1.1, 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 (1.3) 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 $t \in \mathbb{N}$):

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\ 1 & 2 & 1 & 1 & 1 & 1 & 1 & \\ 1 & 4 & 3 & 1 & 1 & 1 & 1 & \\ 1 & 8 & 15 & 4 & 1 & 1 & 1 & \ldots \\ 1 & 16 & 135 & 64 & 5 & 1 & 1 & \\ 1 & 32 & 2{,}295 & 8{,}704 & 325 & 6 & 1 & \\ 1 & 64 & 75{,}735 & 19{,}984{,}384 & 2{,}829{,}125 & 1{,}956 & 7 & \\ & & & \ldots & & & & \end{bmatrix} \tag{1.15}$$

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 $2^{|\bar{x}|}$ 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.

[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.

The previous is just considering the size growth, but, as said in Section 1.2, 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, 2011), 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.

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 1

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 9, 13

Bollig, B., & Buttkus, M. (2019). On the Relative Succinctness of Sentential Decision Diagrams. *Theory of Computing Systems*, *63*(6), 1250–1277. pages 7, 8, 12

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, 8, 11, 12

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, *C-35*(8), 677-691. pages 7

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 7, 8, 12

Choi, A., & Darwiche, A. (2018). SDD Advanced-User Manual Version 2.0 [Computer software manual]. pages 7, 8, 12

Darwiche, A. (2011). 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 7, 8, 12, 13, 16

Flum, J., & Grohe, M. (2006). *Parameterized Complexity Theory*. Springer. pages 13, 16

Manthey, N. (2017). *Riss tool collection.* `https://github.com/nmanthey/riss-solver`. pages 4, 11

Meert, W., & Choi, A. (2018). *Python Wrapper Package to Interactively use Sentential Decision Diagrams (SDD).* `https://github.com/wannesm/PySDD`. pages 7, 12

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 7, 11, 12

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 1, 2, 6, 10

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 7, 12

Oztok, U., & Darwiche, A. (2017). On Compiling DNNFs without Determinism. *ArXiv*, *1709.07092*. pages 4

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 7

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 1

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 1

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 3

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 7, 8, 11

Appendix A

Extra Material from the Experiments

As anticipated in Chapter **??**, 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 1.3, 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.

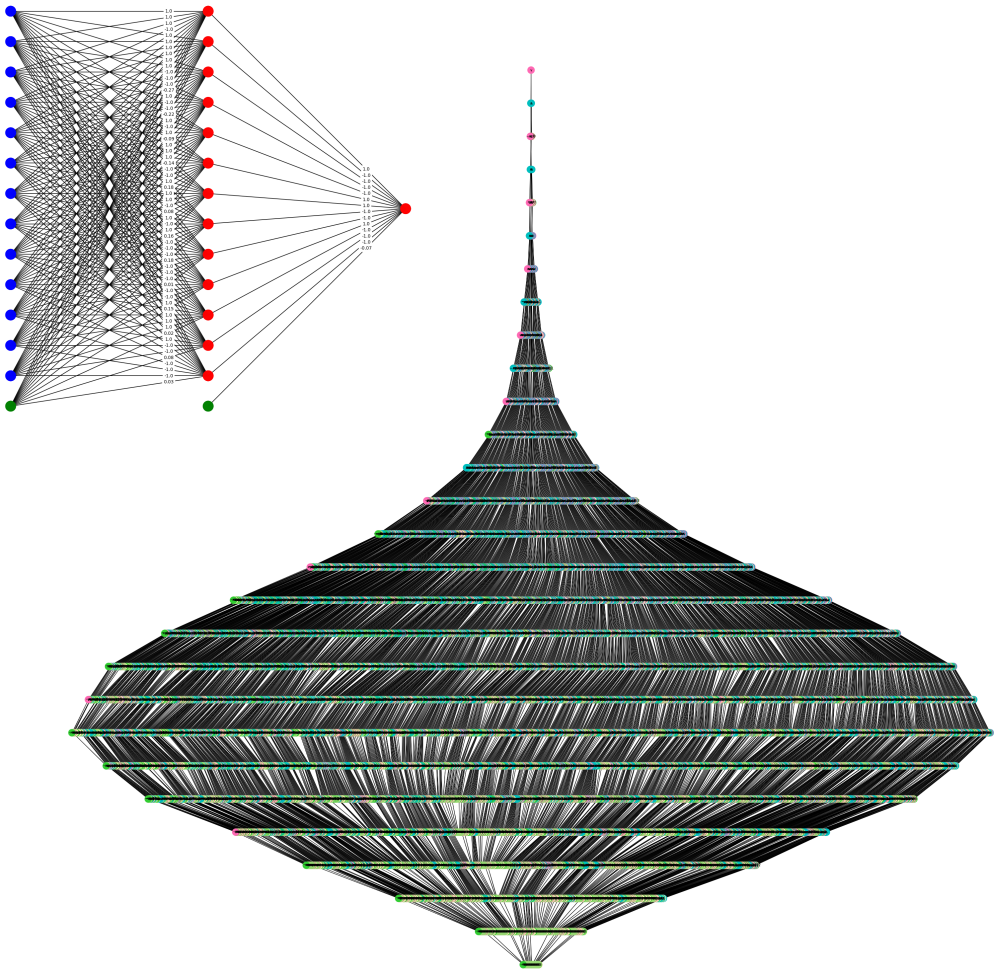

FIGURE A.1: dDBCSFi(2), as a Boolean tree, of the BNN (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 18,671 nodes.

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.

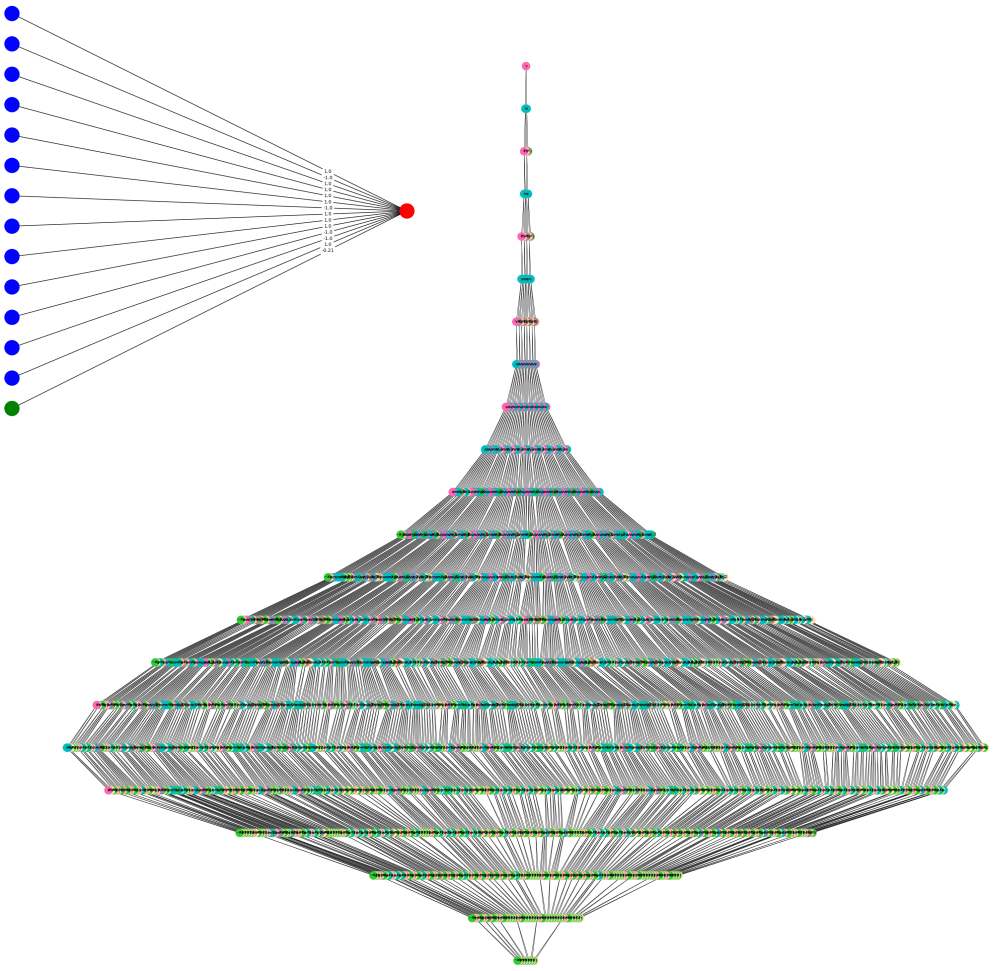

FIGURE A.2: dDBCSFi(2), as a Boolean tree, of the BP (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 4,571 nodes.

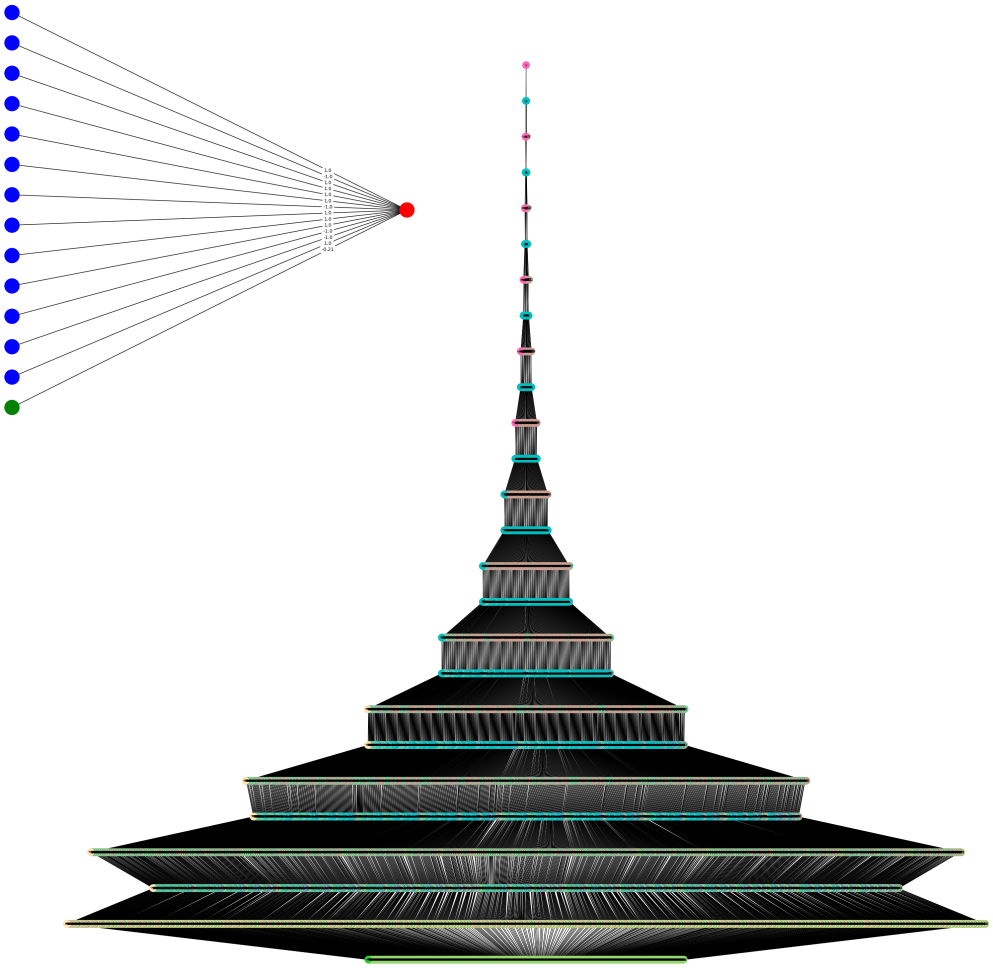

FIGURE A.3: Alternative dDBCSFi(2) (obtained directly, omitting the CNF formula and the SDD), as a Boolean tree, from the BP (visible at top left) trained with *California Housing Prices*, after the preprocessing that generated 13 variables. It has 15,439 nodes.

Appendix B

About the GitHub Repository

As stated at the end of Chapter **??**, 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 **??**, 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 1.2.
- **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 **??**. 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 **??**, 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 1, 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 **??**, are also printed.