Efficient Computation of Shap Explanation Scores for Neural Network Classifiers via Knowledge Compilation

Leopoldo Bertossi[1(✉)] and Jorge E. León[2]

[1] SKEMA Business School, Montreal, Canada
leopoldo.bertossi@skema.edu
[2] Universidad Adolfo Ibáñez (UAI), Santiago, Chile
jorgleon@alumnos.uai.cl

Abstract. The use of Shap scores has become widespread in Explainable AI. However, their computation is in general intractable, in particular when done with a black-box classifier, such as neural network. Recent research has unveiled classes of open-box Boolean Circuit classifiers for which Shap can be computed efficiently. We show how to transform binary neural networks into those circuits for efficient Shap computation. We use logic-based knowledge compilation techniques. The performance gain is huge, as we show in the light of our experiments.

1 Introduction

In recent years, there has been a growing demand for methods to explain and interpret the results from machine learning (ML) models. Explanations come in different forms, and can be obtained through different approaches. A common one assigns *attribution scores* to the features values associated to an input that goes through an ML-based model, to *quantify* their relevance for the obtained outcome. We concentrate on *local* scores, i.e. associated to a particular input, as opposed to a global score that indicated the overall relevance of a feature. We also concentrate on explanations for binary classification models that assign labels 0 or 1 to inputs.

A popular local score is Shap [18], which is based on the Shapley value that was introduced in coalition game theory and practice [29,31]. Shap scores can be computed with a black-box or an open-box model [30]. With the former, we do not know or use its internal components, but only its input/output relation. This is the most common approach. In the latter case, we can have access to its internal structure and components, and we can use them for score computation. 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 Shap computation is provably hard, actually $\#P$-hard for several kinds of binary classification models, independently from whether the internal components of the model are used when computing

L. Bertossi—Member of the Millennium Institute for Foundational Research on Data (IMFD, Chile).

S. Gaggl et al. (Eds.): JELIA 2023, LNAI 14281, pp. 49–64, 2023.
https://doi.org/10.1007/978-3-031-43619-2_4

Shap [1,2,4]. However, there are classes of classifiers for which, using the model components and structure, the complexity of Shap computation can be brought down to polynomial time [2,19,37].

A polynomial time algorithm for Shap computation with *deterministic and decomposable Boolean circuits* (dDBCs) was presented in [2]. From this result, the tractability of Shap computation can be obtained for a variety of Boolean circuit-based classifiers and classifiers that can be represented as (or compiled into) them. In particular, this holds for *Ordered Binary Decision Diagrams* (OBDDs) [8], decision trees, and other established classification models that can be compiled into (or treated as) OBDDs [11,23,33]. This applies, in particular, to *Sentential Decision Diagrams* (SDDs) [14] that form a convenient *knowledge compilation* target language [12,36].

In this work, we show 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.

More precisely, we concentrate on explicitly developing the compilation-based approach to the computation of Shap for *binary (or binarized) neural networks* (BNNs) [17,23,27,35]. For this, a BNN is transformed into a dDBC using techniques from *knowledge compilation* [12], an area that investigates the transformation of (usually) propositional theories into an equivalent one with a canonical syntactic form that has some good computational properties, e.g. tractable model counting. The compilation may incur in a relatively high computational cost [12,13], but it may still be worth the effort when a particular property is checked often, as is the case of explanations for the same BNN.

More specifically, we describe 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 dDBC. 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 and the application to Shap computation are new. We show how Shap is computed on the resulting circuit via the efficient algorithm in [2]. 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.

We also make experimental comparisons of computation times between this open-box and circuit-based Shap computation, and that based directly on the BNN treated as a black-box, i.e. using only its input/output relation. For our experiments, we consider real estate as an application domain, where house prices depend on certain features, which we appropriately binarize[1]. The problem consists in classifying property blocks, represented as entity records of thirteen feature values, as *high-value* or *low-value*, a binary classification problem for which a BNN is used.

To the best of our knowledge, our work is the first at using knowledge compilation techniques for efficiently computing Shap scores, and the first at reporting experiments with the polynomial time algorithms for Shap computation on binary circuits. We confirm that Shap computation via the dDBC vastly outperforms the direct Shap

[1] California Housing Prices dataset: https://www.kaggle.com/datasets/camnugent/california-housing-prices.

computation on the BNN. It is also the case that the scores obtained are fully aligned, as expected since the dDBC represents the BNN. The same probability distribution associated to the Shapley value is used with all the models.

Compilation of BNNs into OBDDs was done in [11,33] for other purposes, not for Shap computation or any other kind of attribution score. In this work we concentrate only on explanations based on Shap scores. There are several other explanations mechanisms for ML-based classification and decision systems in general, and also specific for neural networks. See [16] and [28] for surveys.

This paper is structured as follows. Section 2 contains background on Shap and Boolean circuits (BCs). Section 3 shows in detail, by means of a running example, the kind of compilation of BNNs into dDBCs we use for the experiments. Section 4 presents the experimental setup, and the results of our experiments with Shap computation. In Sect. 5 we draw some conclusions.

2 Preliminaries

In coalition game theory and its applications, the Shapley value is a established measure of the contribution of a player to a shared wealth that is modeled as a game function. Given a set of players S, and a game function $G : \mathcal{P}(S) \to \mathbb{R}$, mapping subsets of players to real numbers, the Shapley value for a player $p \in S$ quantifies its contribution to G. It emerges as the only measure that enjoys certain desired properties [29]. In order to apply the Shapley value, one has to define an appropriate game function.

Now, consider a fixed entity $\mathbf{e} = \langle F_1(\mathbf{e}), \ldots, F_N(\mathbf{e})\rangle$ subject to classification. It has values $F_i(\mathbf{e})$ for features in $\mathcal{F} = \{F_1, \ldots, F_N\}$. These values are 0 or 1 for binary features. In [18,19], the Shapley value is applied with $\mathcal{F}$ as the set of players, and with the game function $\mathcal{G}_\mathbf{e}(s) := \mathbb{E}(L(\mathbf{e}') \mid \mathbf{e}'_s = \mathbf{e}_s)$, giving rise to the Shap score. Here, $s \subseteq \mathcal{F}$, and $\mathbf{e}_s$ is the projection (or restriction) of $\mathbf{e}$ on (to) the s. The label function L of the classifier assigns values 0 or 1. The $\mathbf{e}'$ inside the expected value is an entity whose values coincide with those of $\mathbf{e}$ for the features in s. For feature $F \in \mathcal{F}$:

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

The expected value assumes an underlying probability distribution on the entity population. Shap quantifies the contribution of feature value $F(\mathbf{e})$ to the outcome label.

In order to compute Shap, we only need function L, and none of the internal components of the classifier. Given that all possible subsets of features appear in its definition, Shap is bound to be hard to compute. Actually, for some classifiers, its computation may become $\#P$-hard [2]. However, in [2], it is shown that Shap can be computed in polynomial time for every *deterministic and decomposable Boolean circuit* (dDBC) used as a classifier. The circuit's internal structure is used in the computation.

Figure 1 shows a Boolean circuit that can be used as a binary classifier, with binary features x_1, x_2, x_3, whose values are input at the bottom nodes, and then propagated upwards through the Boolean gates. The binary label is read off from the top node. This circuit is *deterministic* in that, for every $\vee$-gate, at most one of its inputs is 1 when

the output is 1. It is *decomposable* in that, for every $\wedge$-gate, the inputs do not share features. The dDBC in the Figure is also *smooth*, in that sub-circuits that feed a same $\vee$-gate share the same features. It has a *fan-in* at most two, in that every $\wedge$-gate and $\vee$-gate have at most two inputs. We denote this subclass of dDBCs with dDBCSFi(2).

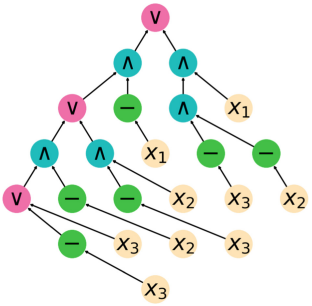

Fig. 1. A dDBC.

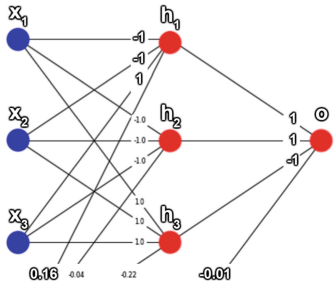

Fig. 2. A BNN.

More specifically, in [2] it is established that Shap can be computed in polynomial time for dDBCSFi(2)-classifiers, assuming that the underlying probability distribution is the uniform, P^{u}, or the product distribution, $P^{\times}$. They are as follows for binary features: $P^{\mathsf{u}}(\mathbf{e}) := \frac{1}{2^N}$ and $P^{\times}(\mathbf{e}) := \Pi_{i=1}^{N} p_i(F_i(\mathbf{e}))$, where $p_i(v)$ is the probability of value $v \in \{0, 1\}$ for feature F_i.

3 Compiling BNNs into dDBCs

In order to compute Shap with a BNN, we convert the latter into a dDBC, on which Shap scores will be computed with the polynomial time algorithm in [2]. The transformation goes along the the following path that we describe in this section:

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

A BNN can be converted into a CNF formula [23,34], which, in its turn, can be converted into an SDD [14,25]. It is also known that SDDs can be compiled into a formula in d-DNNF (deterministic and decomposable negated normal form) [12], which forms a subclass of dDBCs. More precisely, the resulting dDBC in (2) is finally compiled in polynomial time into a dDBCSFi(2).

Some of the steps in (2) may not be polynomial-time transformations, which we will discuss in more technical terms later in this section. 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[2].

[2] The path in (2) is not the only way to obtain a dDBC. For example, [33] describe a conversion of BNNs into OBDDs, which can also be used to obtain dDBCs. However, the asymptotic time complexity is basically the same.

We will describe, explain and illustrate the conversion path (2) by means of a running example with a simple BNN, which is not the BNN used for our experiments. For them, we used a BNN with one hidden layer with 13 gates.

Example 1. The BNN in Fig. 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* $\phi_g(\bar{i})$ of the form:

$$sp(\bar{w}_g \bullet \bar{i} + b_g) := \begin{cases} 1 \text{ if } \bar{w}_g \bullet \bar{i} + b_g \geq 0, \\ -1 \text{ otherwise and } g \text{ is hidden}, \\ 0 \text{ otherwise and } g \text{ is output}, \end{cases} \tag{3}$$

which is parameterized by a vector of binary weights $\bar{w}_g$ and a real-valued constant bias b_g[3]. The $\bullet$ is the inner vector product. For technical, non-essential reasons, for a hidden gate, g, we use 1 and -1, instead of 1 and 0, in $\bar{w}_g$ and outputs. Similarly, $\bar{x} \in \{-1, 1\}^3$. Furthermore, we assume we have a single output gate, for which the activation function does return 1 or 0, for *true* or *false*, respectively.

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. 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 (2), represents the BNN as a CNF formula, i.e. as a conjunction of disjunctions of *literals*, i.e. atomic formulas or their negations.

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.

3.1 Representing BNNs as Formulas in CNF

Our conversion of the BNN into a CNF formula is inspired by a technique introduced in [23], in their case, to verify properties of BNNs. In our case, the NN is fully binarized in that inputs, parameters (other than bias), and outputs are always binary, whereas they may have real values as parameters and outputs. Accordingly, they have to binarize values along the transformation process. They also start producing logical constraints that are later transformed into CNF formulas. Furthermore, [23] introduces auxiliary variables during and at the end of the transformation. With them, in our case, such a BC could not be used for Shap computation. Furthermore, the elimination of auxiliary variables, say via *variable forgetting* [26], could harm the determinism of the final circuit. In the following we describe a transformation that avoids introducing auxiliary variables[4]. However, before describing the method in general, we give an example, to convey the main ideas and intuitions.

[3] We could also used binarized *sigmoid* and *softmax* functions.

[4] At this point is where using $1, -1$ in the BNN instead of $1, 0$ becomes useful.

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 having a larger number on the LHS of the comparison in (3). 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 the LHS of the comparison in (3) 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 [23]:

$$d = \left\lceil (-b + \sum_{j=1}^{|\bar{i}|} w_j)/2 \right\rceil + \#\ of\ negative\ weights\ in\ \bar{w}. \tag{4}$$

For h_1, with 2 negative weights: $d(h_1) = \lceil (-0.16 + (-1 - 1 + 1))/2 \rceil + 2 = 2$. 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:

$$h_1 \longleftrightarrow (-x_1 \wedge -x_2) \vee (-x_1 \wedge x_3) \vee (-x_2 \wedge x_3). \tag{5}$$

This DNF formula is directly obtained -and just to convey the intuition- from considering all possible convenient pairs (which is already better that trying all cases of three variables at a time). However, the general iterative method presented later in this subsection, is more expedite and compact than simply listing all possible cases; and still uses the number of convenient inputs. Using this general algorithm, we obtain, instead of (5), this equivalent formula defining h_1:

$$h_1 \longleftrightarrow (x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1). \tag{6}$$

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

$$\begin{aligned} h_2 &\longleftrightarrow (-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1), \\ h_3 &\longleftrightarrow (x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1), \\ o &\longleftrightarrow (-h_3 \wedge (h_2 \vee h_1)) \vee (h_2 \wedge h_1). \end{aligned} \tag{7}$$

Replacing the definitions of h_1, h_2, h_3 into (7), we finally obtain:

$$\begin{aligned} o \longleftrightarrow\ & (-[(x_3 \wedge (x_2 \vee x_1)) \vee (x_2 \wedge x_1)] \wedge ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)] \\ & \vee [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)])) \vee ([(-x_3 \wedge (-x_2 \vee -x_1)) \vee \\ & (-x_2 \wedge -x_1)] \wedge\ [(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)]). \end{aligned} \tag{8}$$

The final part of step (a) in path (2), requires transforming this formula into CNF. In this example, it can be taken straightforwardly into CNF. For our experiments, we implemented and used the general algorithm presented right after this example. It guarantees that the generated CNF formula does not grow unnecessarily by eliminating some redundancies along the process. The resulting CNF formula is, in its turn, simplified into a shorter and simpler new CNF formula by means of the *Confer* SAT solver [20]. For this example, the simplified CNF formula is as follows:

$$o \longleftrightarrow (-x_1 \vee -x_2) \wedge (-x_1 \vee -x_3) \wedge (-x_2 \vee -x_3). \tag{9}$$

Having a CNF formula will be convenient for the next steps along path (2). □

In more general terms, consider a BNN with L layers, numbered with $Z \in [L] := \{1, \ldots, L\}$. W.l.o.g., we may assume all layers have M neurons (a.k.a. gates), except for the last layer that has a single, output neuron. We also assume that every neuron receives an input from every neuron at the preceding layer. Accordingly, each neuron at the first layer receives the same binary input $\bar{i}_1 = \langle x_1, \ldots, x_N \rangle$ containing the values for the propositional input variables for the BNN. Every neuron at a layer Z to the right receives the same binary input $\bar{i}_Z = \langle i_1, \ldots, i_M \rangle$ formed by the output values from the M neurons at layer $Z - 1$. Variables $x_1, \ldots, x_N$ are the only variables that will appear in the final CNF representing the BNN[5].

To convert the BNN into a representing CNF, we iteratively convert every neuron into a CNF, layerwise and from input to output (left to right). The CNFs representing neurons at a given layer Z are used to build all the CNFs representing the neurons at layer $Z + 1$.

Now, for each neuron g, at a layer Z, the representing CNF, φ^g, is constructed using a matrix-like structure M^g with dimension $M \times d_g$, where M is the number of inputs to g (and N for the first layer), and d_g is computed as in (4), i.e. the number of inputs to conveniently instantiate to get output 1. Formula φ^g represents g's activation function $sp(\bar{w}_g \bullet \bar{i} + b_g)$. The entries c_{ij} of M^g contain terms of the form $w_k \cdot i_k$, which are not interpreted as numbers, but as propositions, namely i_k if $w_k = 1$, and $\neg i_k$ if $w_k = -1$ (we recall that i_k is the k-th binary input to g, and w_k is the associated weight).

Each M^g is iteratively constructed in a row-wise manner starting from the top, and then column-wise from left to right, as follows: (in it, the c_{ik} are entries already created in the same matrix)

$$M^g = \begin{bmatrix} w_1 \cdot i_1 & false & false & \ldots & false \\ w_2 \cdot i_2 \vee c_{11} & w_2 \cdot i_2 \wedge c_{11} & false & \ldots & false \\ w_3 \cdot i_3 \vee c_{21} & (w_3 \cdot i_3 \wedge c_{21}) \vee c_{22} & w_3 \cdot i_3 \wedge c_{22} & \ldots & false \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ w_M \cdot i_M \vee c_{(M-1)1} & (w_M \cdot i_M \wedge c_{(M-1)1}) \vee c_{(M-1)2} & (w_M \cdot i_M \wedge c_{(M-1)2}) \vee c_{(M-1)3} & \ldots & \underline{(w_M \cdot i_M \wedge c_{(M-1)(d_g-1)}) \vee c_{(M-1)d_g}} \end{bmatrix} \quad (10)$$

The k-th row represents the first $k \in [M]$ inputs considered for the encodings, and each column, the threshold $t \in [d_g]$ to surpass, meaning that at least t inputs should be instantiated conveniently. For every component $c_{k,t}$with $k < t$, the threshold cannot be reached, which makes every component in the upper-right triangle $false$.

The propositional formula of interest, namely the one that represents neuron g and will be passed over as an "input" to the next layer to the right, is the bottom-right most, c_{Md_g} (underlined). Notice that it is not necessarily a CNF; nor does the construction of M^g requires using CNFs. It is also clear that, as we construct the components of matrix M^g, they become formulas that keep growing in size. Accordingly, before passing over this formula, it is converted into a CNF φ^g that has also been simplified by means of a SAT solver (this, at least experimentally, considerably reduces the size of the CNF).

[5] We say "a CNF" meaning "a formula in CNF". Similarly in plural.

The vector $\langle \varphi^{g_1}, \ldots, \varphi^{g_M} \rangle$ becomes the input for the construction of the matrices $M^{g'}$, for neurons g' in layer $Z+1$. Reducing the sizes of these formulas is important because the construction of $M^{g'}$ will make the the formula sizes grow further.

Example 3. (Example 2 cont.) Let us encode neuron h_1 using the matrix-based construction. Since $d_{h_1} = 2$, and it has 3 inputs, matrix M^{h_1} will have dimension 3×2. Here, $\bar{w}_{h_1} = \langle -1, -1, 1 \rangle$ and $\bar{i}_{h_1} = \langle x_1, x_2, x_3 \rangle$. Accordingly, M_{h_1} has the following structure:

$$\begin{bmatrix} w_1 \cdot i_1 & \mathit{false} \\ w_2 \cdot i_2 \vee c_{11} & w_2 \cdot i_2 \wedge c_{11} \\ w_3 \cdot i_3 \vee c_{21} & \underline{(w_3 \cdot i_3 \wedge c_{21}) \vee c_{22}} \end{bmatrix}$$

Replacing in its components the corresponding values, we obtain:

$$\begin{bmatrix} -x_1 & \mathit{false} \\ -x_2 \vee -x_1 & -x_2 \wedge -x_1 \\ x_3 \vee -x_2 \vee -x_1 & \underline{(x_3 \wedge (-x_2 \vee -x_1)) \vee (-x_2 \wedge -x_1)} \end{bmatrix}$$

The highlighted formula coincides with that in (6). □

In our implementation, and this is a matter of choice and convenience, it turns out that each component of M^g is transformed right away into a simplified CNF before being used to build the new components. This is not essential, in that we could, in principle, use (simplified) propositional formulas of any format all long the process, but making sure that the final formula representing the whole BNN is in CNF. Notice that all the M^g matrices for a same layer $Z \in L$ can be generated in parallel and without interaction. Their encodings do not influence each other. With this construction, no auxiliary propositional variables other that those for the initial inputs are created.

Departing from [23], our use of the M^g arrays helps us directly build (and work with) CNF formulas without auxiliary variables all along the computation. The final CNF formula, which then contains only the input variables for the BNN, is eventually transformed into a dDBC. The use of a SAT solver for simplification of formulas is less of a problem in [23] due to the use of auxiliary variables. Clearly, our simplification steps make us incur in an extra computational cost. However, it helps us mitigate the exponential growth of the CNFs generated during the transformation of the BNN into the representing CNF.

Overall, and in the worst case that no formula simplifications are achieved, having still used the SAT solver, the time complexity of building the final CNF is exponential in the initial input. This is due to the growth of the formulas along the process. The number of operations in which they are involved in the matrices construction is quadratic.

3.2 Building an SDD Along the Way

Following with step (b) along path (2), the resulting CNF formula is transformed into a *Sentential Decision Diagram* (SDD) [14,36], which, as a particular kind of *decision diagram* [6], is a directed acyclic graph. So as the popular OBDDs [8], that SDDs generalize, they can be used to represent general Boolean formulas, in particular, propositional formulas (but without necessarily being *per se* propositional formulas).

Example 4. (Example 2 cont.) Figure 3(a) shows an SDD, $\mathcal{S}$, representing our CNF formula on the RHS of (9). An SDD has different kinds of nodes. Those represented with encircled numbers are *decision nodes* [36], 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. The *sub* can also be a pointer, •, with an edge to a decision node. $[\ell_1|\ell_2]$ represents two conditions that have to be satisfied simultaneously (in essence, a conjunction). An element without • is a *terminal*. (See [7,22] for a precise definition of SDD.)

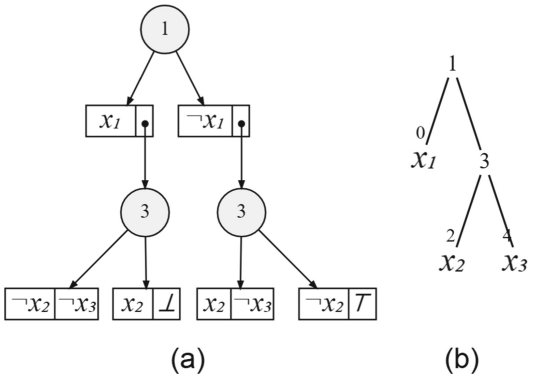

Fig. 3. An SDD (a) and a vtree (b).

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 $F_{\mathcal{S}}(0,1,1) = 0$. □

In SDDs, the orders of occurrence of variables in the diagram must be compliant with a so-called *vtree* (for "variable tree")[6]. The connection between a vtree and an SDD refers to the compatibility between the partitions $[prime|sub]$ and the tree structure (see Example 5 below). 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. SDDs can easily be combined via propositional operations, resulting in a new SDD [14].

A vtree for a set of variables $\mathcal{V}$ is binary tree that is full, i.e. every node has 0 or 2 children, and ordered, i.e. the children of a node are totally ordered, and there is a bijection between the set of leaves and $\mathcal{V}$ [7].

Example 5. (Example 4 cont.) Figure 3(b) shows a vtree, $\mathcal{T}$, for $\mathcal{V} = \{x_1, x_2, x_3\}$. Its leaves, $0, 2, 4$, show their associated variables in $\mathcal{V}$. The SDD $\mathcal{S}$ in Fig. 3(a) is

[6] Extending OBDDs, whose vtrees make variables in a path always appear in the same order. This generalization makes SDDs much more succinct than OBDDs [6,7,36].

compatible with $\mathcal{T}$. Intuitively, the variables at $\mathcal{S}$'s terminals, when they go upwards through decision nodes $\textcircled{n}$, also go upwards through the corresponding nodes n in $\mathcal{T}$. (See [6,7,22] for a precise, recursive definition.)

The SDD S can be straightforwardly represented as a propositional formula by interpreting decision points as disjunctions, and elements as conjunctions, obtaining $[x_1 \wedge ((-x_2 \wedge -x_3) \vee (x_2 \wedge \bot))] \vee [-x_1 \wedge ((x_2 \wedge -x_3) \vee (-x_2 \wedge \top))]$, which is logically equivalent to the formula on the RHS of (9) that represents our BNN. □

For the running example and experiments, we used the *PySDD* system [21]: Given a CNF formula ψ, it computes an associated vtree and a compliant SDD, both optimized in size [9,10]. This compilation step, the theoretically most expensive along path (2), takes exponential space and time only in $TW(\psi)$, the *tree-width* of the *primal graph* $\mathcal{G}$ associated to ψ [14,25]. $\mathcal{G}$ 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. This is a positive *fixed-parameter tractability* result [15], in that $TW(\psi)$ is in general smaller than $|\psi|$. For example, the graph $\mathcal{G}$ for the formula ψ on the RHS of (9) 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 $TW(\psi) = 2$. Overall, this step in the transformation process has a time complexity that, in the worst case, is exponential in the size of the tree-width of the input CNF.

3.3 The Final dDBC

Our final dDBC is obtained from the resulting SDD: An SDD corresponds to a d-DNNF Boolean circuit, for which decomposability and determinism hold, and has only variables as inputs to negation gates [14]. And d-DNNFs are also dDBCs. Accordingly, this step of the whole transformation is basically for free, or better, linear in the size of the SDD if we locally convert decision nodes into disjunctions, and elements into conjunctions (see Example 5).

The algorithm in [1] for efficient Shap computation needs the dDBC to be a dDBCSFi(2). To obtain the latter, we use the transformation Algorithm 1 below, which is based on [1, sec. 3.1.2]. In a bottom-up fashion, fan-in 2 is achieved by rewriting every $\wedge$-gate (resp., and $\vee$-gate) of fan-in $m > 2$ with a chain of $m - 1$ $\wedge$-gates (resp., $\vee$-gates) of fan-in 2. After that, to enforce smoothness, for every disjunction gate (now with a fan-in 2) of subcircuits C_1 and C_2, find the set of variables in C_1, but not in C_2 (denoted V_{1-2}), along with those in C_2, but not in C_1 (denoted V_{2-1}). For every variable $v \in V_{2-1}$, redefine C_1 as $C_1 \wedge (v \vee -v)$. Similarly, for every variable $v \in V_{1-2}$, redefine C_2 as $C_2 \wedge (v \vee -v)$. For example, for $(x_1 \wedge x_2 \wedge x_3) \vee (x_2 \wedge -x_3)$, becomes $((x_1 \wedge x_2) \wedge x_3) \vee ((x_2 \wedge -x_3) \wedge (x_1 \vee -x_1))$. This algorithm takes quadratic time in the size of the dDBC, which is its number of edges [1, sec. 3.1.2], [32].

Example 6. (Example 4 cont.) By interpreting decision points and elements as disjunctions and conjunctions, resp., the SDD in Fig. 3(a) can be easily converted into d-DNNF circuit. Only variables are affected by negations. Due to the children of node ③, that do not have the same variables, the resulting dBBC is not smooth (but it has fan-in 2). Algorithm 1 transforms it into the dDBCSFi(2) in Fig. 1. □

Algorithm 1: From dDBC to dDBCSFi(2)

Input : Original $dDBC$ (starting from root node).
Output: A $dDBCSFi(2)$ equivalent to the given $dDBC$.

```
function FIX_NODE(dDBC_node)
    if dDBC_node is a disjunction then
        c_new = false
        for each subcircuit sc in dDBC_node
            sc_fixed = FIX_NODE(sc)
            if sc_fixed is a true value or is equal to ¬c_new then
                return true
            else if sc_fixed is not a false value then
                for each variable v in c_new and not in sc_fixed
                    sc_fixed = sc_fixed ∧ (v ∨ ¬v)
                for each variable v in sc_fixed and not in c_new
                    c_new = c_new ∧ (v ∨ ¬v)
                c_new = c_new ∨ sc_fixed
        return c_new
    else if dDBC_node is a conjunction then
        c_new = true
        for each subcircuit sc in dDBC_node
            sc_fixed = FIX_NODE(sc)
            if sc_fixed is a false value or is equal to ¬c_new then
                return false
            else if sc_fixed is not a true value then
                c_new = c_new ∧ sc_fixed
        return c_new
    else if dDBC_node is a negation then
        return ¬FIX_NODE(¬dDBC_node)
    else
        return dDBC_node
dDBCSFi(2) = FIX_NODE(root_node)
```

4 Shap Computation: Experiments

The "California Housing Prices" dataset was used for our experiments (it can be downloaded from Kaggle [24]). It consists of 20,640 observations for 10 features with information on the block groups of houses in California, from the 1990 Census. Table 1 lists and describes the features, and the way they are binarized, actually by considering if the value is above the average or not[7] to better The categorical feature #1 is one-hot encoded, giving rise to 5 binary features: $\#1_a$, ..., $\#1_e$. Accordingly, we end up with 13 binary input features, plus the binary output feature, #10, representing whether the median price at each block is high or low, i.e. above or below the average of the original #10. We used the "Tensorflow" and "Larq" Python libraries to train a BNN with one

[7] Binarization could be achieved in other ways, depending on the feature, for better interaction with the feature independence assumption.

hidden layer, with as many neurons as predictors, i.e. 13, and one neuron for the output. For the hidden neurons, the activation functions are step function, as in (3).

Table 1. Features of the "California Housing Prices" dataset.

Id #	Feature Name	Description	Original Values	Binarization
#1	*ocean_proximity*	A label of the location of the house w.r.t sea/ocean	Labels *1h_ocean*, *inland*, *island*, *near_bay* and *near_ocean*	Five features (one representing each label), for which 1 means a match with the value of *ocean_proximity*, and -1 otherwise
#2	*households*	The total number of households (a group of people residing within a home unit) for a block	Integer numbers from 1 to 6,082	1 (above average of the feature) or -1 (below average)
#3	*housing_median_age*	The median age of a house within a block (lower numbers means newer buildings)	Integer numbers from 1 to 52	1 (above average of the feature) or -1 (below average)
#4	*latitude*	The angular measure of how far north a block is (the higher value, the farther north)	Real numbers from 32.54 to 41.95	1 (above average of the feature) or -1 (below average)
#5	*longitude*	The angular measure of how far west a block is (the higher value, the farther west)	Real numbers from -124.35 to -114.31	1 (above average of the feature) or -1 (below average)
#6	*median_income*	The median income for households 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 average)
#7	*population*	The total number of people residing within a block	Integer numbers from 3 to 35,682	1 (above average of the feature) or -1 (below average)
#8	*total_bedrooms*	The total number of bedrooms within a block	Integer numbers from 1 to 6,445	1 (above average of the feature) or -1 (below average)
#9	*total_rooms*	The total number of rooms within a block	Integer numbers from 2 to 39,320	1 (above average of the feature) or -1 (below average)
#10	*median_house_value*	The 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 average)

According to the transformation path (2), the constructed BNN was first represented as a CNF formula with 2,391 clauses. It has a tree-width of 12, which makes sense having a middle layer of 13 gates, each with all features as inputs. The CNF was transformed, via the SDD conversion, into a dDBCSFi(2), $\mathcal{C}$, which ended up having 18,671 nodes (without counting the negations affecting only input gates). Both transformations were programmed in Python. For the intermediate simplification of the CNF, the *Riss* SAT solver was used [20]. The initial transformation into CNF took 1.3 hrs. This is the *practically* most expensive step, as was explained at the end of Sect. 3.1. The conversion of the simplified CNF into the dDBCSFi(2) took 0.8276 s.

With the resulting BC, we computed Shap, for each input entity, in three ways:

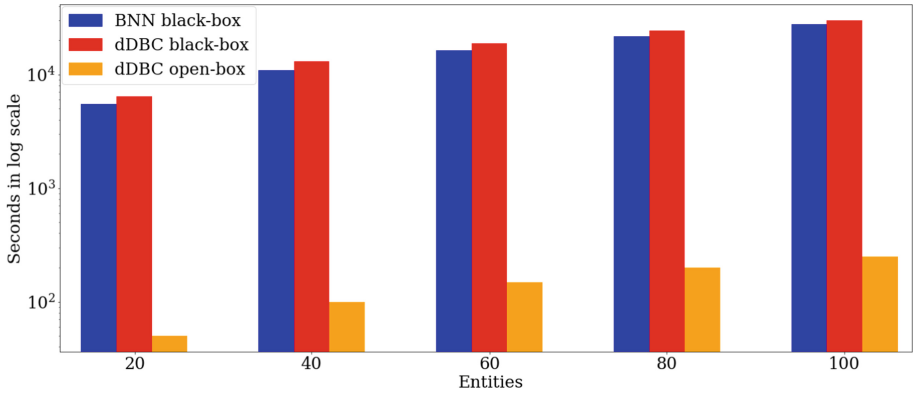

Fig. 4. Seconds taken to compute all Shap scores on 20, 40, 60, 80 and 100 input entities; using the BNN as a black-box (blue bar), the dDBC as a black-box (red bar), and the dDBC as an open-box (orange bar). Notice the logarithmic scale on the vertical axis. (Color figure online)

(a) Directly on the BNN as a black-box model, using formula (1) and its input/output relation for multiple calls;
(b) Similarly, using the circuit $\mathcal{C}$ as a black-box model; and
(c) Using the efficient algorithm in [1, page 18] treating circuit $\mathcal{C}$ as an open-box model.

These three computations of Shap scores were performed for sets of 20, 40, 60, 80, and 100 input entities, for *all* 13 features, and *all* input entities in the set. In all cases, using the uniform distribution over population of size 2^{13}. Since the dDBC faithfully represents the BNN, we obtained exactly the same Shap scores under the modes of computation (a)–(c) above. The *total* computation times were compared. The results are shown in Fig. 4. Notice that these times are represented *in logarithmic scale*. For example, with the BNN, the computation time of all Shap scores for 100 input entities was 7.7 hrs, whereas with the open-box dDBC it was 4.2 min. We observe a huge gain in performance with the use of the efficient algorithm on the open-box dDBC. Those times do not show the one-time computation for the transformation of the BNN into the dDBC. If the latter was added, each red and orange bar would have an increase of 1.3 hrs. For reference, even considering this extra one-time computation, with the open-box approach on the dDBC we can still compute all of the Shap scores for 100 input entities in less time than with the BNN with just 20 input entities[8].

For the cases (a) and (b) above, i.e. computations with black-box models, the classification labels were first computed for all input entities in the population $\mathcal{E}$. Accordingly, when computing the Shap scores for a particular input entity $\mathbf{e}$, the labels for all the other entities related to it via a subset of features S as specified by the game function

[8] The experiments were run on *Google Colab* (with an NVIDIA Tesla T4 enabled). Algorithm 1 was programmed in Python. The complete code for *Google Colab* can be found at: https://github.com/Jorvan758/dDBCSFi2.

were already precomputed. This allows to compute formula (1) much more efficiently[9]. The specialized algorithm for (c) does not require this precomputation. The difference in time between the BNN and the black-box dDBC, cases (a) and (b), is due the fact that BNNs allow some batch processing for the label precomputation; with the dDBC it has to be done one by one.

5 Conclusions

We have showed in detail the practical use of logic-based knowledge compilation techniques in a real application scenario. Furthermore, we have applied them to the new and important problem of efficiently computing attribution scores for explainable ML. We have demonstrated the huge computational gain, by comparing Shap computation with a BNN classifier treated as an open-box vs. treating it as a black-box. The performance gain in Shap computation with the circuit exceeds by far both the compilation time and the Shap computation time for the BNN as a black-box classifier.

We emphasize that the effort invested in transforming the BNN into a dDBC is something we incur once. The resulting circuit can be used to obtain Shap scores multiple times, and for multiple input entities. Furthermore, the circuit can be used for other purposes, such as *verification* of general properties of the classifier [11,23], and answering explanation queries about a classifier [3]. Despite the intrinsic complexity involved, there is much room for improving the algorithmic and implementation aspects of the BNN compilation. The same applies to the implementation of the efficient Shap computation algorithm.

We computed Shap scores using the uniform distribution on the entity population. There are a few issues to discuss in this regard. First, it is computationally costly to use it with a large number of features. One could use instead the *empirical distribution* associated to the dataset, as in [4] for black-box Shap computation. This would require appropriately modifying the applied algorithm, which is left for future work. Secondly, and more generally, the uniform distribution does not capture possible dependencies among features. The algorithm is still efficient with the *product distribution*, which also suffers from imposing feature independence (see [4] for a discussion of its empirical version and related issues). It would be interesting to explore to what extent other distributions could be used in combination with our efficient algorithm.

Independently from the algorithmic and implementation aspects of Shap computation, an important research problem is that of bringing *domain knowledge* or *domain semantics* into attribution scores and their computations, 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 [5]. It is likely that domain knowledge can be more easily be brought into a score computation when it is done on a BC classifier.

In this work we have considered only binary NNs. It remains to be investigated to what extent our methods can be suitably modified for dealing with non-binary NNs.

[9] As done in [4], but with only the entity sample.

Acknowledgments. Special thanks to Arthur Choi, Andy Shih, Norbert Manthey, Maximilian Schleich and Adnan Darwiche, for their valuable help. Work was funded by ANID - Millennium Science Initiative Program - Code ICN17002; CENIA, FB210017 (Financiamiento Basal para Centros Científicos y Tecnológicos de Excelencia de ANID), Chile; SKEMA Business School, and NSERC-DG 2023-04650. L. Bertossi is a Professor Emeritus at Carleton University, Canada.

References

1. Arenas, M., Barceló, P., Bertossi, L., Monet, M.: On the complexity of SHAP-score-based explanations: tractability via knowledge compilation and non-approximability results. J. Mach. Learn. Res. **24**(63), 1–58 (2023). Extended version of [2]
2. Arenas, M., Barceló, P., Bertossi, L., Monet, M.: 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, pp. 6670–6678 (2021)
3. Audemard, G., Koriche, F., Marquis, P.: On tractable XAI queries based on compiled representations. In: Proceedings KR 2020, pp. 838–849 (2020)
4. Bertossi, L., Li, J., Schleich, M., Suciu, D., Vagena, Z.: 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, pp. 1–10 (2020). Posted as Corr arXiv Paper arXiv:2003.06868
5. Bertossi, L.: Declarative approaches to counterfactual explanations for classification. Theory Pract. Logic Program. **23**(3), 559–593 (2023)
6. Bollig, B., Buttkus, M.: On the relative succinctness of sentential decision diagrams. Theory Comput. Syst. **63**(6), 1250–1277 (2019)
7. Bova, S.: SDDs are exponentially more succinct than OBDDs. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence, pp. 929–935 (2016)
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
9. Choi, A., Darwiche, A.: Dynamic minimization of sentential decision diagrams. In: Proceedings of the 27th AAAI Conference on Artificial Intelligence, pp. 187–194 (2013)
10. Choi, A., Darwiche, A.: SDD Advanced-User Manual Version 2.0. Automated Reasoning Group, UCLA (2018)
11. Darwiche, A., Hirth, A.: On the reasons behind decisions. In: Proceedings of the 24th European Conference on Artificial Intelligence, pp. 712–720 (2020)
12. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. **17**(1), 229–264 (2002)
13. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. Appl. Non-Classical Logics **11**(1–2), 11–34 (2011)
14. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: Proceedings of the 22th International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 819–826 (2011)
15. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-29953-X
16. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. **51**(5), 1–42 (2018)
17. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Proceedings of the NIPS 2016, pp. 4107–4115 (2016)
18. Lundberg, S.M., Lee, S.-I.: A unified approach to interpreting model predictions. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 4768–4777 (2017). arXiv Paper arXiv:1705.07874

19. Lundberg, S., et al.: From local explanations to global understanding with explainable AI for trees. Nat. Mach. Intell. **2**(1), 56–67 (2020). arXiv Paper arXiv:1905.04610
20. Manthey, N.: RISS tool collection (2017). https://github.com/nmanthey/riss-solver
21. Meert, W., Choi, A.: Python Wrapper Package to Interactively Use Sentential Decision Diagrams (SDD) (2018). https://github.com/wannesm/PySDD
22. Nakamura, K., Denzumi, S., Nishino, M.: 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, vol. 160, pp. 22:1–22:13 (2020)
23. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence, pp. 6615–6624 (2018)
24. Nugent, C.: California Housing Prices (2018). https://www.kaggle.com/datasets/camnugent/california-housing-prices
25. Oztok, U., Darwiche, A.: On compiling CNF into decision-DNNF. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 42–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_7
26. Oztok, U., Darwiche, A.: On compiling DNNFs without determinism (2017). arXiv:1709.07092
27. Qin, H., Gong, R., Liu, X., Bai, X., Song, J., Sebe, N.: Binary neural networks: a survey. Pattern Recogn. **105**, 107281 (2020)
28. Ras, G., Xie, N., van Gerven, M., Doran, D.: Explainable deep learning: a field guide for the uninitiated. J. Artif. Intell. Res. **73**, 329–396 (2022)
29. Roth, A.: The Shapley Value: Essays in Honor of Lloyd S. Shapley. Cambridge University Press (1988)
30. Rudin, C.: Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nat. Mach. Intell. **1**, 206–215 (2019). arXiv Paper arXiv:1811.10154
31. Shapley, L.S.: A value for n-person games. In: Contributions to the Theory of Games (AM-28), vol. 2, pp. 307–318 (1953)
32. Shih, A., Van den Broeck, G., Beame, P., Amarilli, A.: Smoothing structured decomposable circuits. In: Proceedings of the NeurIPS (2019)
33. Shi, W., Shih, A., Darwiche, A., Choi, A.: On tractable representations of binary neural networks. In: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, pp. 882–892 (2020)
34. Shih, A., Darwiche, A., Choi, A.: Verifying binarized neural networks by Angluin-Style learning. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 354–370. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_25
35. Simons, T., Lee, D.-J.: A review of binarized neural networks. Electronics **8**(6), 661 (2019)
36. Van den Broeck, G., Darwiche, A.: On the role of canonicity in knowledge compilation. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence, pp. 1641–1648 (2015)
37. Van den Broeck, G., Lykov, A., Schleich, M., Suciu, D.: On the tractability of SHAP explanations. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence, pp. 6505–6513 (2021)