Anytime Compilation of Binary Neurons to OBDDs

Aidan Boyce
Kennesaw State University
Marietta, GA, USA
aboyce10@students.kennesaw.edu

Arthur Choi
Kennesaw State University
Marietta, GA, USA
achoi13@kennesaw.edu

***Abstract*—An artificial neuron with binary inputs and a binary output corresponds to a Boolean function. Hence, to explain and verify the behavior of such a neuron (and by extension, a neural network), it suffices to explain and verify its Boolean function. There has been recent interest in representing the Boolean function of such a neuron as an Ordered Binary Decision Diagram (OBDD), which facilitates such analyses. In this paper, we propose an anytime algorithm for compiling a binary neuron into an OBDD, based on a recently proposed compiler that decomposes a binary neuron's Boolean function into its prime implicants, represented as a decision tree. We augment this compiler so that it outputs an OBDD instead. Our augmented compiler is also anytime, as it produces intermediate OBDDs that represent inner- and outer-bounds of the original neuron, which tighten as compilation progresses. Theoretically, decision graphs of binary neurons are exponentially more succinct than their decision trees. Empirically, compilation to decision graphs can scale to neurons with over a thousand features, compared to dozens of features using other compilers. We highlight the utility of our approach via a case study in eXplainable AI.**

***Index Terms*—explainable artificial intelligence (XAI), knowledge compilation, decision diagrams**

I. Introduction

Formal approaches to eXplainable Artificial Intelligence (XAI) seek to provide mathematical guarantees on the behavior of machine learning models [1]–[4]. For example, consider a neural network with binary inputs and a binary output. The input/output behavior of such a network can be faithfully represented as a Boolean function, independent of any numerical weights or training data that the network could have. If one is able to extract the Boolean function of a neural network, then tools and techniques from logical AI (including SAT solving and model counting) could be used to explain and formally verify the behavior of the neural network.

We take a *knowledge compilation* approach to formal XAI [3]–[5]. Knowledge compilation is a sub-field of AI that studies *tractable* Boolean circuits, and the trade-offs between their succinctness (size of the circuit) and tractability (availability of polytime queries and transformations) [6], [7]. Our goal is to compile a classifier into a tractable circuit representation that facilitates efficient explanation and formal verification.

Recently, there has been growing interest in compiling linear classifiers (including binary neurons), into Ordered Binary Decision Diagrams (OBDDs) [8]–[13]. Chan & Darwiche provided a thorough analysis of the problem, and proposed an $O(n2^{\frac{n}{2}})$ algorithm for compiling a linear classifier into an OBDD [8]. Chubarian & Turan generalized this algorithm to compile (tree-augmented) naive Bayes classifiers into OBDDs [11]. In general, compiling a linear classifier into an OBDD is an NP-hard problem [9]. However, if a binary neuron has integer weights, it can be compiled into an OBDD in pseudo-polynomial time [10]. Shi et al. leveraged this algorithm to compile a neural network of binary neurons into an OBDD, which was used to analyze the robustness of the network to adversarial perturbations [10]. More recently, a boosting approach was proposed to compile linear classifiers to OBDDs [12]. Kennedy et al. proposed to learn binary neurons that were guaranteed to have compact OBDDs, directly from data [13].

In this paper, we propose an *anytime* algorithm for compiling binary neurons into OBDDs. It is based on a recently proposed algorithm that explored the decision tree of a binary neuron [14], and searched for the prime implicants of its Boolean function [15]. This search provided inner- and outer-bounds on the Boolean function, which became tighter with each prime implicant that was enumerated. We first provide a simplified view of [14] and propose a more balanced heuristic that improves the efficiency of search. We then view this search from the lens of knowledge compilation [16], and show how to augment the search so that it compresses the decision tree into a decision graph (i.e., an OBDD). We also show that this search is anytime: it starts with (vacuous) inner- and outer-bounds on the neuron's Boolean function, which become tighter the more we explore the search space. Theoretically, we show that the decision graphs of binary neurons are exponentially more compact than their decision trees. Empirically, we can compile binary neurons with more features, by an order-of-magnitude or more, into decision graphs compared to decision trees. Finally, we provide a case study in explaining the behavior of a classifier.

This paper is organized as follows. In Section II, we review binary neurons, viewing them as linear threshold tests. In Section III, we review how to compile a binary neuron into a decision tree. In Section IV, we show how to compile a binary neuron into a decision graph. In Section V, we provide a case study in XAI, and we conclude in Section VI.

II. Technical Background

A *binary* neuron is a neuron with binary (0/1) inputs, and a binary output. If a neuron has n binary inputs $X_1, \ldots, X_n$ and a step activation function, then it also has a binary output. It is thus a binary neuron that represents a function of the form:

$$f(X_1, \ldots, X_n) = \sigma(w_1 X_1 + w_2 X_2 + \cdots + w_n X_n + b).$$

The w_i are the neuron's weights, b is the bias, and the step function $\sigma(z)$ outputs 1 if $z \geq 0$, and 0 otherwise. For example

$$f(X_1, X_2, X_3) = \sigma(2.1 \cdot X_1 + 0.9 \cdot X_2 - 1.9 \cdot X_3 - 0.5) \quad (1)$$

is a binary neuron, where we call f the Boolean function of the neuron, and call $X_1, \ldots, X_n$ the variables of f. We can evaluate this function on all $2^3 = 8$ inputs, and obtain a truth table of the neuron's Boolean function:

X_1	X_2	X_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0

X_1	X_2	X_3	f
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Following [14], we view a binary neuron as a threshold test; cf. [8], [15]. A *threshold test* is a function f of the form:

$$f(X_1, \ldots, X_n) = \begin{cases} 1 & \text{if } w_1 \cdot X_1 + \cdots + w_n \cdot X_n \geq T \\ 0 & \text{otherwise} \end{cases}$$

Weights w_i are the same as a neuron's weights, and T is a threshold, equal to the negated bias $-b$. Given a setting of the inputs, if the function f outputs 1, then we say that the threshold test passes. Otherwise, we say the threshold test fails.

The neuron of Equation 1 has the equivalent threshold test:

$$2.1 \cdot X_1 + 0.9 \cdot X_2 - 1.9 \cdot X_3 \geq 0.5. \quad (2)$$

We say a threshold test f *always passes* iff the left-hand side is always greater than or equal to the threshold, no matter how we set the inputs. We say a threshold test f *always fails* iff the left-hand side is always less than the threshold. We call a threshold test *trivial* if it always passes or always fails.

Consider the following threshold tests:

$$-1.9 \cdot X_3 \geq 0.5 \qquad -1.9 \cdot X_3 \geq -2.5. \quad (3)$$

The threshold test on the left is found by setting both X_1 and X_2 to 0, in the threshold test of Equation 2. This simpler threshold test always fails, no matter how we set X_3. The threshold test on the right is found by setting both X_1 and X_2 to 1, and then subtracting 3 from both sides. The resulting threshold test always passes, no matter how we set input X_3.

Note that the left-hand side is minimized when we set all variables with positive weights to 0, and those with negative weights to 1. It is maximized if we set all variables with positive weights to 1, and those with negative weights to 0. Thus, the left-hand side of a threshold test has a range $[L, U]$ where lower bound L is the sum of all negative weights, and upper bound U is the sum of all positive weights.

Given a threshold test with threshold T and range $[L, U]$, then it always passes iff $T \leq L \leq U$, and it always fails iff $L \leq U < T$. The threshold test of Equation 2 has a range $[-1.9, 3]$ and a threshold 0.5 and is not trivial. The threshold test of Equation 3 (left) has a range $[-1.9, 0]$ and a threshold 0.5 and always fails. The threshold test of Equation 3 (right) has a range $[-1.9, 0]$ and a threshold -2.5 and always passes.

III. Bounding the Behavior of a Neuron

Our goal is to obtain a faithful representation of a binary neuron's input/output behavior, satisfying the following properties. First, it is more compact than a truth table. Second, it facilitates the explanation and formal verification of a binary neuron's behavior. Third, it should provide bounds on a neuron's behavior, if a complete and faithful representation is infeasible. Previously, Borowski & Choi showed that the input/output behavior of a threshold test can be represented using a decision tree, satisfying all three properties [14].

Definition 1. *A (pruned) decision tree for a threshold test f is a rooted tree, where each node n is annotated with a threshold test g, and the root note is annotated with f. Nodes n and their threshold tests g are defined recursively, starting from the root.*

- *If g is not trivial, node n is an internal node annotated with decision variable X. Node n has two children, a low child $n_{\bar{x}}$ and a high child n_x, with the respective threshold tests $g_{\bar{x}}$ and g_x that are obtained from g by setting variable X to 0 and 1, respectively.*
- *If g is trivial, node n is a leaf node (any potential children are pruned), and is annotated as passing or failing, based on whether g is always passing or always failing.*

Consider, as an example, the following threshold test:

$$1 \cdot X_1 + 2 \cdot X_2 - 2 \cdot X_2 + 4 \cdot X_4 - 4 \cdot X_5 \geq 1.$$

Figure 1 depicts the decision tree of this threshold test, where each node's decision variable is the last variable of the node's threshold test. The threshold test above appears at the root node, with decision variable X_5 and two children: a low child found by setting X_5 to 0 (following the dashed line), and a high child found by setting X_5 to 1 (following the solid line). When we set X_5 to 0, we obtain a simpler threshold test after dropping the term $-4 \cdot X_5$. When we set X_5 to 1, we add 4 to both sides, and obtain a simpler threshold test with a threshold of 5. We continue setting variables to values until the threshold test becomes trivial, and the test's output has been determined. A threshold test that always passes (outputs 1) or always fails (outputs 0) is highlighted in green or red, respectively.

Like a truth table, a decision tree encodes the input/output behavior of a threshold test. Given an input to a threshold test, we start at the root of the decision tree. We look up the decision variable, and the value that the input sets to it: if it is 0, follow the low branch, and if it is 1, follow the high branch. We repeat at the next node, until we reach a leaf node, which corresponds to a trivial threshold test, and thus the remaining inputs (if any) do not matter. If the leaf is always passing, we output 1; otherwise, it is always failing, and we output 0.

Starting at the root of the decision tree, say we are given $(X_1 = 1, X_2 = 1, X_3 = 0, X_4 = 1, X_5 = 0)$. We first follow the dashed line, as X_5 is set to 0. We then follow the solid line as X_4 is set to 1. The resulting threshold test is always passing, so the original threshold test (at the root) outputs 1.

Note that there is no need to represent the decision tree of a trivial threshold test, since its outcome has already been

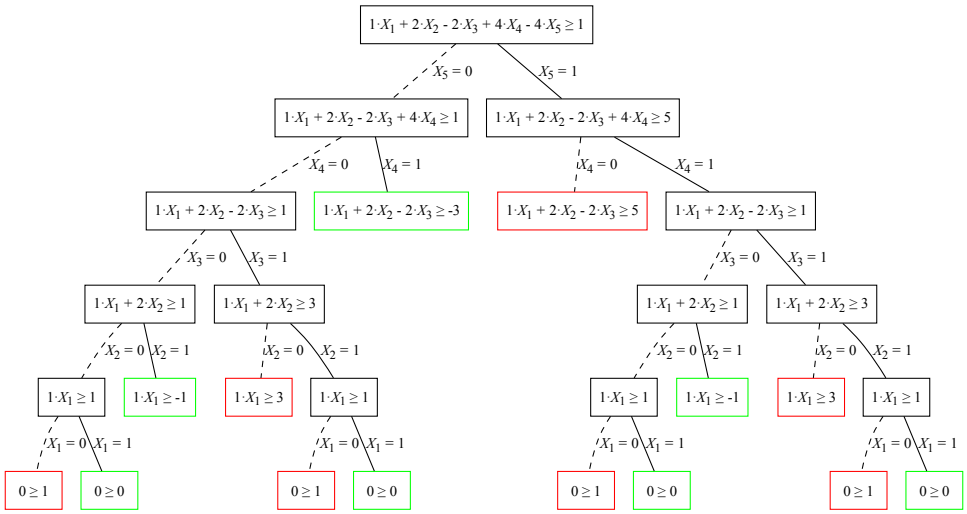


Fig. 1. The decision tree of a threshold test. The decision variable at each node is the last variable of the corresponding threshold test.

determined. Such a (pruned) decision tree can be *exponentially* more compact than a truth table. The threshold test $X_1 + \cdots + X_n \geq 0$ has a truth table with 2^n rows, but its decision tree has a single node, since the threshold test is always passing.

A. Explaining the Behavior of a Binary Neuron

Since the behavior of a decision tree is (relatively) easy to explain, we can explain the behavior of a less transparent classifier (like a neural network), by representing it as a decision tree; cf. [17]. Typically, the "reason" why a decision tree labels an input a value, is based on the path that the decision tree takes to reach that label. Thus, to explain why a threshold test passes or fails, we take the *decision path* that its decision tree takes on a given input. Consider the input $(X_1 = 1, X_2 = 1, X_3 = 0, X_4 = 1, X_5 = 0)$ in the decision tree of Figure 1. The decision path $(X_4 = 1, X_5 = 0)$ ends at a leaf node whose threshold test is always passing. Thus, the reason why the threshold test passes is because it always passes given $(X_4 = 1, X_5 = 0)$. Such an explanation found a subset of the inputs that is *sufficient* for the resulting decision.

This type of explanation of a classifier's behavior, is called a prime implicant (PI) explanation, an abductive explanation, or a sufficient reason [5], [18]–[20]. Such explanations are also closely related to Anchor explanations [21]. For example, the reason why a classifier labels an image as a dog, according to a PI-explanation, is based on finding a (small) sub-image that is sufficient for the classifier to label the image as a dog. For example, the presence of a dog's head and front legs may be sufficient for a classifier to label an image as a dog.

More formally, a PI-explanation is a prime implicant of a classifier's Boolean function. The Boolean function of a classifier's decision tree is the disjunction of all its decision paths (to a given label). Each decision path is the term (conjunction of literals) of the decisions made on that path. For example, the decision tree of Figure 1 has the following Boolean function, based on decision paths to passing leaves: $f(X_1, X_2, X_3, X_4, X_5) = (x_4 \wedge \bar{x}_5) \vee (x_2 \wedge \bar{x}_3 \wedge x_4 \wedge x_5) \vee (x_2 \wedge \bar{x}_3 \wedge \bar{x}_4 \wedge \bar{x}_5) \vee (x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5) \vee (x_1 \wedge \bar{x}_2 \wedge \bar{x}_3 \wedge x_4 \wedge x_5) \vee (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4 \wedge \bar{x}_5) \vee (x_1 \wedge \bar{x}_2 \wedge \bar{x}_3 \wedge \bar{x}_4 \wedge \bar{x}_5)$. This function is also in Disjunctive Normal Form (DNF). Each decision path corresponds to a term in the DNF. Each term is also an *implicant* of the Boolean function, where an implicant is a (partial) term that can always be extended to a satisfying assignment of f (over all variables). That is, an implicant α entails the Boolean function f, denoted $\alpha \models f$. An implicant α is called a *prime implicant* if no sub-term of α is also an implicant. The term $(x_4 \wedge \bar{x}_5)$ is a prime implicant of f. The term $(x_2 \wedge \bar{x}_3 \wedge x_4 \wedge x_5)$ is an implicant that is not prime, since literal x_5 can be dropped and $(x_2 \wedge \bar{x}_3 \wedge x_4)$ is an implicant.

Thus, a PI-explanation (or sufficient reason) for the decision of a classifier, is a subset of its input that is also a (short) prime implicant of the classifier's Boolean function [5], [18]–[20].

B. Inner and Outer Bounds on the Behavior of a Neuron

A *prime cover* is a decomposition of a function f into prime implicants [22]. Previously, [14] compiled a binary neuron into a (pruned) decision tree, where a prime cover is found by dropping irrelevant literals from the terms of its decision paths. The decision tree of Figure 1 has the prime cover: $f(X_1, X_2, X_3, X_4, X_5) = (x_4 \wedge \bar{x}_5) \vee (x_2 \wedge \bar{x}_3 \wedge x_4) \vee (x_2 \wedge \bar{x}_3 \wedge \bar{x}_5) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge \bar{x}_3 \wedge x_4) \vee (x_1 \wedge x_2 \wedge \bar{x}_5) \vee (x_1 \wedge \bar{x}_3 \wedge \bar{x}_5)$. Note that each prime implicant summarizes a subset of assignments that satisfy f, with shorter prime implicants summarizing a larger subset. Similarly, each prime implicant of the negation $\neg f$ summarizes a subset of assignments that cannot satisfy f. Hence, by enumerating prime implicants of f and $\neg f$, we can cover the total input/output behavior of f.

Say that P and Q are sets of terms representing prime covers of a Boolean function f and its negation $\neg f$. Subsets $A \subseteq P$ and $B \subseteq Q$ yield inner- and outer- bounds on f as follows:[1]

$$\bigvee_{\alpha \in A \subseteq P} \alpha \quad \models \quad f \quad \models \quad \bigwedge_{\beta \in B \subseteq Q} \neg \beta. \tag{4}$$

[1]A function f entails g, denoted $f \models g$, iff every assignment satisfying f also satisfies g, i.e., $w \models f$ implies $w \models g$ for all assignments w.

Algorithm 1 `compile_decision_tree`(f)

input: A threshold test f, with sorted inputs

output: A decision tree for threshold test f

main:

```
make new node r for f
initialize priority queue Q with (r, f)
while Q is not empty do
    pop node and threshold test (n, g) from Q
    if g is not trivial then
        let g_lo/g_hi be result of setting last input of g to 0/1
        make new nodes n_lo and n_hi for g_lo and g_hi
        make n_lo and n_hi the low and high children of n
        push (n_lo, g_lo) and (n_hi, g_hi) into Q
return r
```

We have a trivial inner-bound (no satisfying assignments are known) and a trivial outer-bound (no satisfying assignments are excluded) when the subsets A and B are empty. Larger subsets A and B provide tighter bounds on f, which become tight when A and B correspond to prime covers of f and $\neg f$.

[15] showed how to enumerate the prime implicants of a linear classifier's Boolean function, with only a polynomial time delay between each prime implicant. This includes classifiers like our binary neurons and threshold tests. [14] showed how best-first search can enumerate the shortest prime implicants of a threshold test first, which leads to tighter inner- and outer-bounds, which we review and improve on next.

C. From Binary Neurons to Decision Trees via Search

We view the decision tree of a binary neuron as a search space. The leaf nodes of the (pruned) decision tree, correspond to prime implicants of the neuron's Boolean function, and its negation. Thus, more efficient navigation of this search space translates to tighter inner- and outer-bounds. [14] proposed to explore this search space using best-first search (i.e., A^* search), using a heuristic that expanded nodes first that were closest to becoming trivially passing. That is, the search enumerated the shortest prime implicants first. Empirically, [14] showed that best-first search enumerates prime implicants more efficiently than a depth-first search based on [15].

Algorithm 1 summarizes the best-first search of [14]. Given a threshold test f as input, we first create a root node r from f, and initialize a priority queue with the pair (r, f). While the priority queue is not empty, we pop the next node/test pair (n, g) from the priority queue. If test g is not trivial, then we find two simpler threshold tests by setting the last input of g to 1 and 0, making their nodes the high and low children of n. We push the two node/test pairs back into the priority queue. If test g is trivial, we do not update the priority queue.

Our first contributions simplify and improve the best-first search of [14]. First, in Algorithm 1, we assume that the weights $w_1, \ldots, w_n$ of the input threshold test are sorted, in increasing order, by the absolute value of their weight. This simplifies variable selection during search, as the next variable to set is now the one having the largest weight.[2] Second, and more significantly, we propose a more balanced heuristic, that expands nodes first based on how close they are to becoming trivial, either always passing or always failing. Hence, we seek to tighten both the inner- and outer-bounds at the same time. In contrast, [14] proposed a simpler approach that tightened the inner-bound but potentially ignored the outer-bound. In fact, in their experiments, they performed a second search to tighten the outer-bound, which is less efficient.

Figure 3(a) depicts the tighter inner- and outer-bounds of our new heuristic, by evaluating the corresponding lower- and upper-bounds on the model count of a threshold test [14]; see also Section V and Footnote 9. We consider the threshold test:

$$(2^0 \cdot X_0 - 2^0 \cdot X_1) + \cdots + (2^7 \cdot X_{14} - 2^7 \cdot X_{15}) \geq 0$$

Best-first search progresses as we move left-to-right on the x-axis, where we enumerate more prime implicants, and evaluate the resulting model counts on the y-axis. We see that the lower-bound of [14] (dotted blue line) is initially tight, but the upper-bound (dotted red line) is loose and essentially ignored until the end of the search. In contrast, our lower- and upper-bounds (solid blue and red lines) are both tightened early in the search.

IV. From Decision Trees to Decision Graphs

Consider the decision tree of Figure 1. Starting at the root, the path $(X_5 = 0, X_4 = 0)$ leads to the threshold test: $1 \cdot X_1 + 2 \cdot X_2 - 2 \cdot X_3 \geq 1$. The path $(X_5 = 1, X_4 = 1)$ also arrives at the same threshold test. Hence, these two nodes represent two copies of the same decision tree. If we continue at either node, the two paths $(X_3 = 0, X_2 = 0)$ and $(X_3 = 1, X_2 = 1)$ arrive at the same threshold test: $1 \cdot X_1 \geq 1$. This time, there are four nodes with four copies of the same decision tree. By merging equivalent nodes, we can obtain a much more compact *decision graph* representation of a threshold test.

Definition 2. *A decision graph for a threshold test is a rooted graph, obtained from a decision tree of the same threshold test, by merging nodes with equivalent threshold tests.*

In Figure 2, we merged all nodes for the same threshold test into a single node, i.e., we redirected all their parents to point to the same node. After this simplification, the decision tree of 27 nodes became a decision graph of only 13 nodes.

Note that to obtain the decision graph of a threshold test, one does not need to first obtain the corresponding decision tree. We will soon show how to directly compile a threshold test into a decision graph. This algorithm relies on checking if two threshold tests are equivalent, which also determines how much a decision graph can compress a decision tree.

We say that two threshold tests are equivalent if and only if they represent the same Boolean function. If two threshold tests have the same weights and the same threshold, then obviously, they are equivalent. Unfortunately, it is NP-hard in

[2] As observed by [15], this is also the fastest way to make a threshold test trivial, which corresponds to finding the shortest prime implicant.

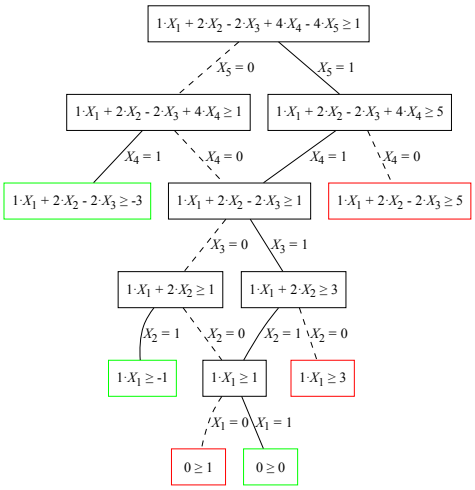


Fig. 2. Enumerating the behavior of a threshold test using a decision graph.

general to decide if two threshold tests are equivalent.[3] However, we propose to use a simple but incomplete equivalence test that can still find many (but not all) pairs of equivalent threshold tests, based on the analyses of [8], [10].

In particular, if we assume there is a fixed ordering of the variables in a decision graph (as we did in Section III), then two threshold tests at the same depth of the decision graph will have the same weights on the left-hand side.[4] Hence, if they also have the same threshold, then they must be equivalent. Thus, we have a sufficient, but not necessary, condition for equivalence that we can test in constant time. It is also possible to quantize the weights and thresholds, in a way that tries to preserve the behavior of the classifier, but encourages the occurrence of equal thresholds, which in turn, leads to a more compressed decision graph (with fewer nodes) [10], [13].[5]

A. Anytime Compilation via Search

We next propose an anytime algorithm to compile a given binary neuron (or threshold test) into a decision graph, and more specifically, into an Ordered Binary Decision Diagram (OBDD).[6] We extend the best-first search algorithm of [14] and Section III, for compiling a threshold test into a decision tree. In particular, as we search the space of threshold tests, by detecting and caching equivalent nodes, the search will trace the structure of a decision graph [8], [16]. Moreover, any partial search corresponds to a decision graph representing a bound on the binary neuron's Boolean function.

Algorithm 2 `compile_decision_graph`(f)

input: An integer threshold test f, with sorted inputs
output: A decision graph for threshold test f
main:

initialize `cache`
set depth d to 0 and parent node p to $n_\star$
initialize priority queue Q with (p, d, f)
while Q is not empty **do**
  pop parent node, depth and test (p, d, g) from Q
  let T be the threshold of test g
  if key (d, T) is in `cache` **then**
    $n \leftarrow$ `cache`$[(d, T)]$
  else if g is not trivial **then**
    let n be new node for g
    `cache`$[(d, T)] \leftarrow n$
    let $g_{\text{lo}}/g_{\text{hi}}$ be result of setting last input of g to $0/1$
    push $(n, d+1, g_{\text{lo}})$ and $(n, d+1, g_{\text{hi}})$ into Q
  else if g is trivial **then**
    let n be new node for g
  set n as child of p
set depth d to 0 and T be threshold of test f
return `cache`$[(d, T)]$

Typically, algorithms for constructing OBDDs use a "unique table" to detect and eliminate redundant or superfluous nodes [23]. Compilers that convert Boolean formulas to OBDDs, also use a "formula cache" to detect and re-use the results of repeated sub-problems [16], [25], [26]. We thus propose to cache every threshold test found during search, using the equivalence test we just proposed. That is, during search, if we detect that a node that we visit already exists in the cache, we simply point the parent to the node that we already generated. Algorithm 2 provides the pseudo-code for the modified search.

Algorithm 2 is also an anytime algorithm, in that any intermediate result is a valid decision graph that provides an inner- and outer-bound on the original threshold test's Boolean function. Say that any new node created by Algorithm 2 has, by default, children that point to an always failing node (for the inner-bound) or an always passing node (for the outer-bound). Initially, we have a single always failing node (representing the function false or $\bot$) or a single always passing node (representing the function true or $\top$), This corresponds to vacuous inner- and outer-bounds: $\bot \models f \models \top$. As the best-first search enumerates more always-passing nodes, we tighten the inner-bound. As we enumerate more always-failing nodes, we tighten the outer-bound. Once we have enumerated all nodes, then the inner- and outer-bounds both become tight.

B. On the Relative Sizes of Decision Graphs and Trees

Decision graphs are well-known to be exponentially more succinct than decision trees [24]. For example, to represent

[3]Number partitioning is an NP-complete problem, that can be reduced to testing if an integer threshold test ever meets its threshold exactly [5]. One can test if a threshold test ever meets its threshold exactly, by testing if it is equivalent to another threshold test where the threshold is increased by one.

[4]If we set a variable X to 0 or 1 in a threshold test, the term mentioning X on the left-hand side becomes a constant, which we move to the right, updating the threshold. At depth d of the decision tree, we set the last d inputs, and the sum $w_1 \cdot X_1 + \cdots + w_{n-d} \cdot X_{n-d}$ always remains on the left-hand side. Hence, tests at depth d differ only by their threshold [8], [10].

[5]We call a threshold test *integer* if it has integer weights and thresholds. A (floating-point) threshold test can be made integer, by multiplying both sides by a constant γ and then truncating. A larger γ better preserves the behavior of the classifier, but a smaller γ leads to more equal thresholds.

[6]In the knowledge compilation literature [6], a decision graph is often called a Binary Decision Diagram (BDD) [23], [24]. If a BDD has a fixed variable ordering, it is called an OBDD. Note that our decision graphs are not reduced OBDDs, but they can be easily reduced in a post-processing step. First, the leaf nodes of always passing (or failing) threshold tests should be merged into a single 1-sink (or 0-sink). Further, we use a sufficient, but not necessary, test of threshold test equivalence, hence we miss merging some nodes; cf. [8].

the parity function, decision trees and graphs require $\Omega(2^n)$ nodes and $O(n)$ nodes, respectively. This does not imply that the decision graphs of threshold tests are exponentially more succinct than their decision trees, as a threshold test (or binary neuron) cannot represent a parity function [27]. However, the following theorem confirms an exponential separation.

Theorem 1. *There is a family of threshold tests whose decision graphs and decision trees have sizes that are quadratic and exponential, respectively, in the number of inputs.*

Proof. Say we have n inputs $X_1, \ldots, X_n$ where all weights w_i are 1, and a threshold T. This threshold test corresponds to a simple threshold gate. Let $c_g(n, T)$ and $c_t(n, T)$ be the number of internal nodes (representing non-trivial threshold tests) in the decision graph and tree, respectively. We have $c_g(n, T) = T \cdot (n - T + 1)$ and $c_t(n, T) = \binom{n+1}{T} - 1$. When $T = \frac{n}{2}$ then $c_g(n, T) = \Theta(n^2)$ and $c_t(n, T) = \Theta(2^n)$, which provides an exponential separation in size.

First, consider $c_g(n, T)$. Let t denote the threshold of an internal node's test. We have $0 < t \leq n$, otherwise the threshold test is trivial. A threshold t appears from depth $d = T - t$ (all variables were set to 1) to depth $n - t$ (all remaining variables must be set to 1). For each of the T possible thresholds, there are $(T-t)-(n-t)+1 = T-n+1$ different depths it appears in. Hence, there are $T \cdot (n-T+1)$ internal nodes representing threshold tests that are not trivial.

Next, consider $c_t(n, T)$. Each path to a node in the decision graph is a path to a unique node in the decision tree. There are $\binom{d}{s}$ paths to each decision graph node at depth d, where $s = T - t$ is the number of inputs set to 1. Thus, $c_g(n, T) =$

$$\sum_{t=1}^{T} \sum_{d=T-t}^{n-t} \binom{d}{T-t} = \sum_{s=0}^{T-1} \sum_{d=s}^{n-T+s} \binom{d}{s} = \sum_{s=0}^{T-1} \binom{n-T+s+1}{s+1}$$
$$= \sum_{r=1}^{T} \binom{n-T+r}{r} = \sum_{r=0}^{T} \binom{n-T+r}{r} - 1 = \binom{n+1}{T} - 1$$

since $\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$ and $\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}$. □

Figure 3(b) empirically confirms the scalability of compiling decision graphs. As in the above proof, all $w_i = 1$ and $T = \frac{n}{2}$. We vary the number of variables n on the x-axis, and report the compilation time on the y-axis. Compilation of decision trees surpass 60s after $n = 26$, while compilation of decision graphs do not surpass 60s until n is greater than 1,000. Hence, decision graph compilation scales to problems that are larger by an order-of-magnitude or more (the x-axis is in log-scale).

V. Case Study

We now highlight how compiling binary neurons to OBDDs can provide insights about a classifier's behavior. Consider the MNIST dataset of 55,000 grayscale images of handwritten digits, which we binarized to B&W. Each image has $28 \times 28 = 784$ pixels, and is labeled with a digit from 0 to 9. We consider one-vs.-one classification on pairs of digits. We used `scikit-learn` to train binary neurons.[7]

In Figure 3(c), we compare the algorithm of [14] for compiling decision trees, including our improvements from Section III, and our compilation algorithm for decision graphs, from Section IV.[8] We first consider the pair of digits $(0, 8)$, and evaluate the quality of the inner- and outer-bounds from the anytime compiler, based on the corresponding lower- and upper-bounds on the binary neuron's model count.[9] We evaluate the quality of the bounds on the y-axis, as we are given more time on the x-axis. Upper-bounds are red, lower-bounds are blue, and the true model count is plotted as a purple, dashed line. The bounds for the decision tree (dotted line), tighten slowly and did not converge after 60s. The bounds for our decision graph (solid line), tighten quickly and obtain the exact model count after only 2.16s.

We compared both algorithms on all $\binom{10}{2} = 45$ digit pairs. Given a 60s time limit, decision tree compilation completed in only 22 out of 45 pairs, with an average running time of 11.6s for completed runs. Decision graph compilation completed for all 45 pairs, with an average running time of 2.7s.

Finally, we highlight how decision graphs provide insight into a classifier's behavior; see Figure 4. Using the same 0-vs.-8 classifier, we classified an image of a 0 and an image of an 8. We first ask "Why did the image of a 0 get classified as a 0?" Using the decision graph, we found a PI-explanation [5], [18]–[20], which is a small set of pixels that are *sufficient* for a classifier's decision. Consider Figure 4(a): a white pixel to the left, and a few black pixels in the middle are enough for the classifier to believe the image will be a 0. The reverse holds true in Figure 4(b), for the image of an 8. This indicates that the classifier learned a simple heuristic rule to distinguish a 0 from an 8, which was enough to obtain a training set accuracy of 97.75%. We also ask: "Why didn't this image of a 0 get classified as an 8?" Using the decision graph, we found a counterfactual explanation [19], [28], [29], which is a small set of pixels that, if flipped, would result in a reversal of the classifier's decision. In Figure 4(c), the highlighted pixels (in white) correspond to a few black pixels in the middle of the image that, if flipped to white, would cause the classifier to label the image as an 8. Analogously for Figure 4(d). These counterfactual explanations reinforce the idea that the classifier is using a simple heuristic rule to label these images.

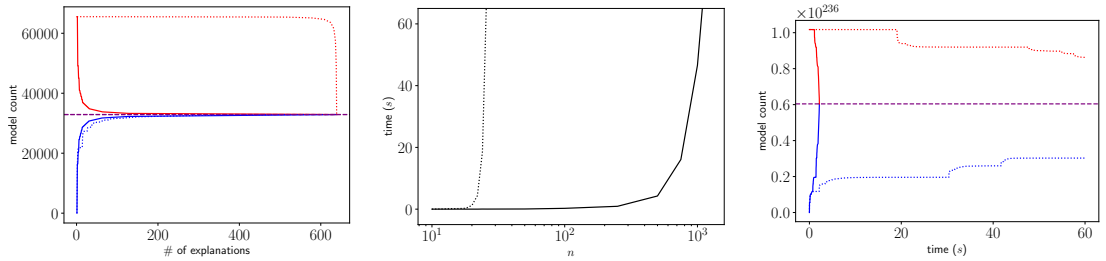
VI. Conclusion

In this paper, we proposed an anytime algorithm for compiling binary neurons into decision graphs, i.e., OBDDs. We augmented a recently proposed algorithm for compiling binary neurons into decision trees, which in turn was based on searching for prime implicants of a binary neuron's Boolean

[7] As in [14], we trained a logistic regression classifier, with L_1 penalty, the `liblinear` solver, and increased the regularization strength $C = 0.003$. To obtain a binary neuron, we replaced the sigmoid with a step activation.

[8] Source at `https://github.com/aboy4321/neuron-compiler`

[9] The model count of a binary neuron is the number of inputs that have a 1-output. It is NP-hard to compute this model count [5]. See [14] for details.



(a) (b) (c)

Fig. 3. (a) Tightness of lower- and upper-bounds (blue and red) based on our new heuristic (solid) versus [14] (dotted). (b) Compilation time (y-axis) in the decision tree and graph (dashed and solid), with increasing number of variables (x-axis). (c) Tightness of lower- and upper-bounds (blue and red) based on compilation to decision tree and graph (dotted and solid). In sub-figures (a) & (c), the model count is plotted as a purple and dashed horizontal line.

(a) (b) (c) (d)

Fig. 4. PI-explanations, (a) & (b), and counterfactual explanations, (c) & (d).

function. We first simplified the search, and proposed an improved heuristic. We then proposed to augment this search with a cache, which compresses the decision tree into a decision graph representing inner- and outer-bounds on the behavior of a neuron, that tighten the more we search. We proved that the decision graphs of binary neurons can be exponentially more compact than their decision trees, and empirically demonstrated improved scalability in practice. We highlighted the utility of our approach to XAI, via a case study.

Acknowledgment

The authors were supported by the Office of Undergraduate Research at Kennesaw State University.

References

[1] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification (CAV)*, 2017, pp. 97–117.

[2] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," *CoRR*, vol. abs/1805.09938, 2018.

[3] A. Shih, A. Choi, and A. Darwiche, "Formal verification of Bayesian network classifiers," in *9th International Conference on Probabilistic Graphical Models (PGM)*, 2018.

[4] G. Audemard, F. Koriche, and P. Marquis, "On tractable XAI queries based on compiled representations," in *17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2020.

[5] A. Shih, A. Choi, and A. Darwiche, "A symbolic approach to explaining Bayesian network classifiers," in *27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[6] A. Darwiche and P. Marquis, "A knowledge compilation map," *Journal of Artificial Intelligence Research (JAIR)*, vol. 17, pp. 229–264, 2002.

[7] A. Darwiche, "Tractable knowledge representation formalisms," in *Tractability: Practical Approaches to Hard Problems*, 2014.

[8] H. Chan and A. Darwiche, "Reasoning about Bayesian network classifiers," in *19th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2003, pp. 107–115.

[9] A. Shih, A. Darwiche, and A. Choi, "Verifying binarized neural networks by Angluin-style learning," in *22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2019.

[10] W. Shi, A. Shih, A. Darwiche, and A. Choi, "On tractable representations of binary neural networks," in *17th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2020.

[11] K. Chubarian and G. Turan, "Interpretability of Bayesian network classifiers: OBDD approximation and polynomial threshold functions," in *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2020.

[12] Y. Tang, K. Hatano, and E. Takimoto, "Boosting-based construction of BDDs for linear threshold functions and its application to verification of neural networks," in *26th International Conference on Discovery Science (DS)*, 2023, pp. 477–491.

[13] L. Kennedy, I. Kindo, and A. Choi, "On training neurons with bounded compilations," in *20th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2023.

[14] R. Borowski and A. Choi, "On bounding the behavior of neurons," *International Journal on AI Tools (IJAIT)*, vol. 33, no. 3, 2024.

[15] J. Marques-Silva, T. Gerspacher, M. C. Cooper, A. Ignatiev, and N. Narodytska, "Explaining naive Bayes and other linear classifiers with polynomial time and delay," in *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020.

[16] J. Huang and A. Darwiche, "DPLL with a trace: From SAT to knowledge compilation," in *19th International Joint Conference on Artificial Intelligence (IJCAI)*, 2005, pp. 156–162.

[17] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?": Explaining the predictions of any classifier," in *Knowledge Discovery and Data Mining (KDD)*, 2016.

[18] A. Ignatiev, N. Narodytska, and J. Marques-Silva, "Abduction-based explanations for machine learning models," in *Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*, 2019, pp. 1511–1519.

[19] ——, "On relating explanations and adversarial examples," in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019, pp. 15 857–15 867.

[20] A. Darwiche and A. Hirth, "On the reasons behind decisions," in *24th European Conference on Artificial Intelligence (ECAI)*, 2020.

[21] M. T. Ribeiro, S. Singh, and C. Guestrin, "Anchors: High-precision model-agnostic explanations," in *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[22] O. Coudert, J. C. Madre, H. Fraisse, and H. Touati, "Implicit prime cover computation: An overview," in *4th SASIMI Workshop*, 1993.

[23] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*. Springer, 1998.

[24] I. Wegener, *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.

[25] S. M. Majercik and M. L. Littman, "Using caching to solve larger probabilistic planning problems," in *15th National Conference on Artificial Intelligence (AAAI)*, 1998, pp. 954–959.

[26] A. Darwiche, "New advances in compiling CNF into decomposable negation normal form," in *16th Eureopean Conference on Artificial Intelligence (ECAI)*, 2004, pp. 328–332.

[27] M. Minsky and S. Papert, *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[28] J. Pearl, M. Glymour, and N. P. Jewell, *Causal inference in statistics: A primer*. Wiley, 2016.

[29] S. Wachter, B. D. Mittelstadt, and C. Russell, "Counterfactual explanations without opening the black box: Automated decisions and the GDPR," *CoRR*, vol. abs/1711.00399, 2017.