UNIVERSITY OF CALIFORNIA

Los Angeles

Explaining Classifiers

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Boyun Shih

2019

ABSTRACT OF THE THESIS

Explaining Classifiers

by

Boyun Shih
Master of Science in Computer Science
University of California, Los Angeles, 2019
Professor Adnan Youssef Darwiche, Chair

We study the task of explaining machine learning classifiers. We explore a symbolic approach to this task, by first *compiling* the decision function of a classifier into a tractable decision diagram, and then *explaining* its behavior using exact reasoning techniques on the tractable form. On the compilation front, we propose new algorithms for encoding the decision functions of Bayesian Network Classifiers and Binarized Neural Network Classifiers into tractable decision diagrams. On the explanation front, we examine techniques for generating a variety of instance-based and classifier-based explanations on tractable decision diagrams. Finally, we evaluate our approach on real-world and synthetic classifiers. Using our algorithms, we can efficiently produce exact explanations that deepen our understanding of these classifiers.

The thesis of Boyun Shih is approved.

Eliezer M Gafni

Alexandr Sherstov

Adnan Youssef Darwiche, Committee Chair

University of California, Los Angeles

2019

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Adnan and Arthur for introducing me to the wonderful world of research. Thank you for being patient with me and providing me with a safe environment to explore my ideas and grow as a researcher.

I am grateful to have shared the lab space with the Automated Reasoning and the StarAI group. You guys never fail to keep up a lively and fun atmosphere in the lab. I will especially miss our many dining hall treks up the Hill.

I would also like to thank my family for being supportive of my goals, and always asking me to explain my research to you. Even when I'm unsuccessful, it pushes me to understand my own work more deeply.

Finally, I am lucky to have Cindy, with whom everyday is a little more colorful.

CHAPTER 1

Introduction

Recent progress in artificial intelligence and the increased deployment of AI systems have led to the need for *explaining* the decisions made by such systems, particularly classifiers [RSG16, EDF17, LL17, RSG18]. It is now recognized that opacity, or lack of explainability, is one of "the biggest obstacle[s] to widespread adoption of artificial intelligence" [CN17]. For example, one may want to explain *why* a classifier turned down a loan application, rejected an applicant for an academic program, or recommended surgery for a patient. Answering such *why?* questions is important for gaining a user's trust in the classification decision and for government regulations, such as the EU General Data Protection Regulation [GF17].

In this thesis, we propose a *symbolic* approach to explaining classifiers, which is based on the following observation [CD03]. Consider a classifier that labels instances either positively or negatively based on a number of binary feature variables. The classifier specifies a function that maps features into a 0/1 decision (1 for a positive instance). We call this function the classifier's *decision function*, which unambiguously describes the classifier's behavior, regardless of its implementation. Our goal is then to obtain a symbolic and tractable representation of this decision function, to enable efficient reasoning about its behavior, including generating explanations for its decisions. We refer to the constructing of the symbolic and tractable representation of a classifier's decision function as *compiling* the classifier.

Choosing the target representation of the decision function involves a trade-off between compilation cost and tractability of explanations. On one extreme, we can maintain the classifier's representation, incurring no compilation cost but possibly leaving many explanation tasks expensive. Another choice may be to compile the classifier into Conjunctive Normal Form (CNF) and answer explanation tasks using a SAT solver [NKR18]. In this thesis we

focus on compiling classifiers into Ordered Binary Decision Diagrams (OBDDs) [Bry86], which require a possibly costly compilation phase but enable us to answer many interesting explanation tasks in linear/quadratic time on the size of the decision diagram.

In particular, we study Naive Bayes, Bayesian Network, and Binarized Neural Network classifiers. We review an existing technique for compiling Naive Bayes classifiers into OBDDs [CD03], and introduce new algorithms for compiling Bayesian Network and Binarized Neural Network classifiers into OBDDs. We then convert the OBDDs into Sentential Decision Diagrams (SDDs) [Dar11], which can be much more succinct yet still maintain the tractability properties required for our explanation tasks.

Once we have the classifiers represented as tractable decision diagrams, we explore which explanation tasks are efficient on the decision diagrams. The explanation tasks we consider come in two flavors: instance-based or classifier-based. Instance-based explanations reason about the classifier's behavior on one specific instance, and asks questions such as "What is the minimum perturbation required on this instance to flip its classification decision?" or "What is the minimal subset of features on this instance that can guarantee its classification decision, regardless of how the remaining features are flipped?" Classifier-based explanations reason about the classifier's global behavior, and asks questions such as "Is the classifier monotonic?" Reasoning about these questions gives us insight into the behavior of the decision diagrams, which is exactly the behavior of the original classifiers.

We next provide an overview for the remaining chapters of this thesis. In Chapter 2, we define our notation and review material on tractable decision diagrams and machine learning classifiers. In Chapter 3, we describe an existing algorithm for compiling Naive Bayes classifiers and present novel algorithms for compiling Bayesian Network and Binarized Neural Network classifiers into OBDDs. In Chapter 4, we formally define our explanation tasks of interest and provide efficient algorithms for answering these tasks on our decision diagrams. Finally, in Chapter 5, we present experimental results and case studies, and conclude in Chapter 6.

CHAPTER 2

Background

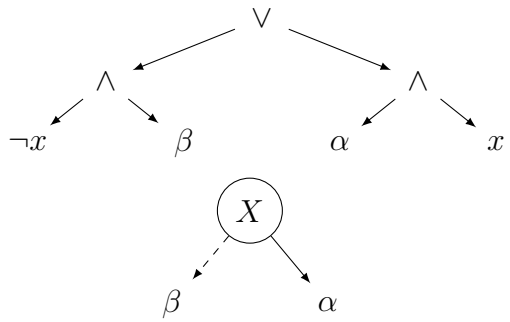
We introduce notations and definitions, and review the relevant technical preliminaries.

Capital letters X denote variables and lower-case letters x denote their values, also called a *literal*. Bold capital letters $\boldsymbol{X}$ denote sets of variables and bold lower-case letters $\boldsymbol{x}$ denote their instantiations, also called an *instance*. When referring to machine learning classifiers, *feature variables* are the inputs and *class variable* is the output of the classifier.

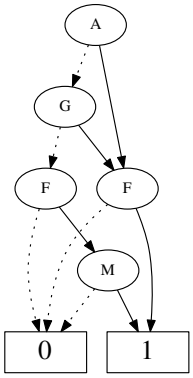
We begin with knowledge compilation, which examines languages for representing Boolean functions [CD97, DM02, SK96]. The languages that have been studied are generally subsets of Negation Normal Form (NNF). A NNF is a rooted, directed acyclic graph where each internal node is labelled with $\wedge$ or $\vee$ and each leaf node is labelled with $\top$, $\bot$, or a literal. Note that negations ($\neg$) can only appear with a literal, and not as an internal node.

A well-known subset of NNF is Conjunctive Normal Form (CNF). A CNF is a conjunction of clauses, where each clause is a disjunction of literals. This is also the subset of NNFs with height at most 2, where the children of each $\vee$-node are leaves and the root is a $\wedge$-node.

CNF is considered a *representation* language, since it is suitable for human specification and interpretation. But, it is not a *target compilation* language since there is no known efficient algorithm for important queries/transformations, such as model counting or negation. These queries/transformations are necessary for generating explanations and reasoning about a Boolean function. As such, we next examine two target compilation languages that do efficiently support many queries/transformations.



(a) Circuit (top) and diagram (bottom) notation of a decision node.



(b) Ordered Binary Decision Diagram

Figure 2.1: An OBDD represented using the diagram notation.

2.1 Tractable Decision Diagrams

The main properties that will give us tractability on many queries/transformations is *decision* and *ordering*. A decision node is recursively defined as either $\top$, $\bot$, or a $\vee$-node with the form $(X \wedge \alpha) \vee (\neg X \wedge \beta)$, where α and β are decision nodes and X is a variable. In the last case, we say that the decision node is labelled by X. Binary Decision Diagram (BDD) is the subset of NNF where the root is a decision node.

2.1.1 Ordered Binary Decision Diagrams

An *Ordered Binary Decision Diagram* (OBDD) is a BDD that respects the *ordering* property [Bry86]. This means that the decision nodes of the OBDD respect some global ordering of the variables: if a decision node labelled by X_i is a parent of a decision node labelled by X_j, then X_i must come before X_j in the global ordering. Figure 2.1b shows the decision diagram notation of an OBDD with the variable ordering A, G, F, M, and sinks 0/1 denoting $\bot/\top$. To evaluate the OBDD on an instance $\boldsymbol{x}$, start at the root and repeatedly follow the solid edge if the variable of the current node is set to 1 by $\boldsymbol{x}$, and follow the dashed edge otherwise. The sink node that is reached determines the evaluation of $\boldsymbol{x}$.

2.1.2 Sentential Decision Diagrams

A *Sentential Decision Diagram* (SDD) is a variation of the OBDD where the decision nodes decompose on sets of variables instead of single variables [Dar11]. A decision node labeled by variable X in an OBDD decomposes a function f into $(\neg X \wedge h_0) \vee (X \wedge h_1)$. On the other hand, a decision node in an SDD is labeled by a splitting of the variables into two sets $\boldsymbol{X}$ and $\boldsymbol{Y}$, and decomposes a function f into $(g_0 \wedge h_0) \vee \ldots \vee (g_n \wedge h_n)$, where $g_i : \boldsymbol{X} \rightarrow \{\bot, \top\}$, $h_i : \boldsymbol{Y} \rightarrow \{\bot, \top\}$, and $g_1, \ldots, g_n$ are exhaustive and mutually exclusive. SDDs are a superset of OBDDs, since OBDDs can be seen as the special case when $\boldsymbol{X}$ (in the $\boldsymbol{X}, \boldsymbol{Y}$ split of an SDD node) is a set with a single variable. For a more detailed treatment of SDDs, see [Dar11].

SDDs support almost all of the tractable operations that are offered by OBDDs [Dar11], and can be more succinct than OBDDs [Bov16]. There is also a comprehensive SDD software package with all of the necessary operations for running our explanation techniques [CD18].

2.2 Machine Learning Classifiers

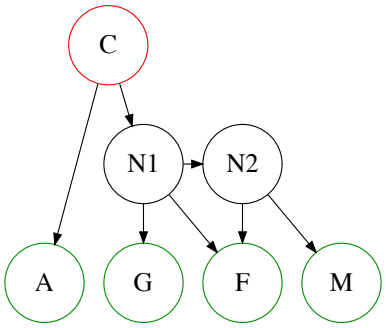
We will now describe some commonly used machine learning classifiers. We consider versions of these classifiers that have binary inputs and outputs, so we can aim to compile their decision functions into tractable decision diagrams for reasoning and generating explanations.

2.2.1 Bayesian Network Classifiers

A Bayesian Network is a directed acyclic graph (DAG) along with conditional probability tables (CPTs) [Dar09]. In the DAG, nodes specify variables and edges specify conditional dependencies. A CPT specifies the distribution on a node for each state of its parents in the DAG. Together, the DAG and CPTs generate a probability distribution $Pr(.)$ over the variables of the Bayesian Network.

The *Bayesian Network classifiers* we consider are Bayesian Networks with a single binary class variable C, n binary feature variables $\boldsymbol{X} = \{X_1, \ldots, X_n\}$, and a classification threshold t [FGG97]. The class C is a root in the network and the features $\boldsymbol{X}$ are leaves. A Bayesian

Network classifier classifies an instance $\boldsymbol{x}$ as 1 if $Pr(c \mid \boldsymbol{x}) \geq t$ and 0 otherwise, where $Pr(.)$ is the probability distribution specified by the underlying Bayesian Network. An example of a Bayesian Network classifier is shown in Figure 2.2, whose decision function matches the OBDD in Figure 2.1b. We provide the truth table in Table 2.1.



(a) Directed acyclic graph

Threshold	0.5
$C = 1$	0.5

C	0	1
$A = 1$	0.6	0.7

C	0	1
$N1 = 1$	0.3	0.9

$N1$	0	1
$N2 = 1$	0.3	0.7

$N1$	0	1
$G = 1$	0.5	0.6

$N2$	0	1
$M = 1$	0.2	0.9

$N1, N2$	00	01	10	11
$F = 1$	0.1	0.8	0.3	0.9

(b) Conditional probability tables and threshold

Figure 2.2: The DAG and CPTs of a Bayesian Network classifier.

Table 2.1: The function on the 16 possible inputs computed by the OBDD in Figure 2.1b. The Bayesian Network classifier in Figure 2.2 also computes the same function.

	$AGFM$	$f(\boldsymbol{x})$		$AGFM$	$f(\boldsymbol{x})$		$AGFM$	$f(\boldsymbol{x})$		$AGFM$	$f(\boldsymbol{x})$
0	0000	0	4	0100	0	8	1000	0	12	1100	0
1	0001	0	5	0101	0	9	1001	0	13	1101	0
2	0010	0	6	0110	1	10	1010	1	14	1110	1
3	0011	1	7	0111	1	11	1011	1	15	1111	1

2.2.2 Naive Bayes Classifiers

A *Naive Bayes classifier* is a special type of a Bayesian Network classifier where there is an edge from the node of the class variable to the nodes of each feature variable, and there are no other edges or nodes.

2.2.3 Binarized Neural Network Classifiers

A *Binarized Neural Network classifier* is a feed-forward neural network where the weights and activations are binarized using $\{-1, 1\}$ [HCS16]. A Binarized Neural Network classifier is composed of internal blocks and one output block. Internal blocks consist of three layers: a linear transformation (LIN), batch normalization (BN), and binarization (BIN).

- The LIN layer has parameters $\boldsymbol{a}$ (weights) and b (bias). Given an input $\boldsymbol{x}$, this layer returns $\langle \boldsymbol{a}, \boldsymbol{x} \rangle + b$.
- The BN layer has parameters μ (mean), σ (standard deviation), α (weight), and γ (bias). Given an input y, this layer returns $\alpha(\frac{y-\mu}{\sigma}) + \gamma$.
- The BIN layer returns the sign (1 or -1) of its input.

The output block consists of a LIN layer and an ARGMAX layer. The ARGMAX layer picks the output class with the highest activation. More details regarding these blocks and layers and their exact definitions is given by Narodytska et al. [NKR18]. For convenience we consider a Binarized Neural Network classifier with outputs 0/1 instead of $-1/1$.

Both the Bayesian Network and Binarized Neural Network classifiers we consider have underlying decision functions that map binary inputs into a binary output. As such, we aim to compile their decision functions into tractable decision diagrams for efficient generation of explanations.

CHAPTER 3

Compilation Algorithms

In this section we explore techniques for compiling machine learning classifiers into tractable decision diagrams. The goal is to construct an OBDD that completely captures the decision function, or the input/output behavior, of the classifier.

We begin by reviewing an algorithm for compiling Naive Bayes classifiers from [CD03]. Then, we will introduce methods for compiling Bayesian Network and Binarized Neural Network classifiers, based on work published in [SCD19, SDC19].

3.1 Compiling Naive Bayes Classifiers

[CD03] proposed an algorithm for compiling a Naive Bayes classifier into an OBDD, while guaranteeing an upper bound on the time of compilation and the size of the resulting OBDD. In particular, for a classifier with n feature variables, the compiled OBDD has a number of nodes that is bounded by $O(2^{n/2})$ and can be obtained in time $O(n2^{n/2})$. Experimentally, the time and space costs can still be quite low, depending on the classifier's parameters and variable order used for the OBDD.

The algorithm is based on the following insights. Let $\boldsymbol{X}$ be all feature variables. Classifying an instance $\boldsymbol{x}$ is based on the test $Pr(c|\boldsymbol{x}) \geq t$, which is equivalent to the following test [CD03].

$$\frac{Pr(c)Pr(\boldsymbol{x}|c)}{Pr(\neg c)Pr(\boldsymbol{x}|\neg c)} \geq \frac{t}{1-t} \tag{3.1}$$

Since all the feature variables are independent in a Naive Bayes classifier, we can partition $\boldsymbol{X}$ into two sets $\boldsymbol{U}$ and $\boldsymbol{V}$ in any way and rewrite Equation 3.1. For convenience we will let

$s(\boldsymbol{u}) = \frac{Pr(\boldsymbol{u}|c)}{Pr(\boldsymbol{u}|\neg c)}$ and $s(\boldsymbol{v}) = \frac{Pr(\boldsymbol{v}|c)}{Pr(\boldsymbol{v}|\neg c)}$.

$$\frac{Pr(c)}{Pr(\neg c)} \frac{Pr(\boldsymbol{u}|c)}{Pr(\boldsymbol{u}|\neg c)} \frac{Pr(\boldsymbol{v}|c)}{Pr(\boldsymbol{v}|\neg c)} \geq \frac{T}{1-T} \tag{3.2}$$

$$\frac{Pr(c)}{Pr(\neg c)} s(\boldsymbol{u}) s(\boldsymbol{v}) \geq \frac{t}{1-t} \tag{3.3}$$

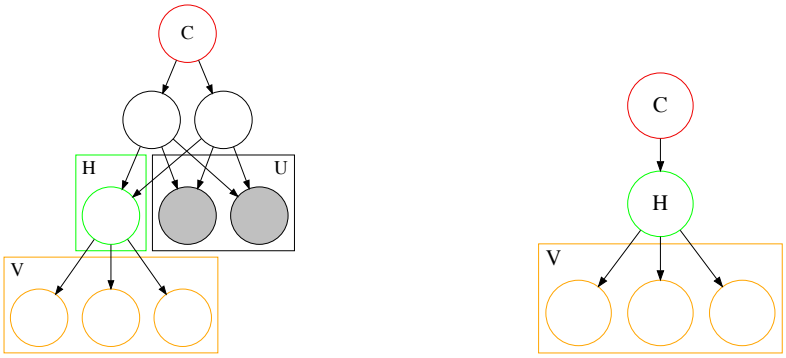
This formulation will help the efficient construction of OBDDs. In particular, at level k in our OBDD, we have 2^k possible partial instantiations, leading to 2^k values in $\{s(\boldsymbol{u}) : \boldsymbol{u} \in \{0,1\}^{|\boldsymbol{U}|}\}$. Moreover, there are only $n-k$ remaining feature variables, leading two 2^{n-k} values in $\{s(\boldsymbol{v}) : \boldsymbol{v} \in \{0,1\}^{|\boldsymbol{V}|}\}$. As such, we can bound the number of distinct subfunctions at level k by $O(\min(2^n, 2^{n-k}))$, which gives us the same Sieling and Wegener bound on the number of OBDD nodes [Weg00]. To finish, the total number of nodes in the OBDD has a bound of $2\sum_{k=1}^{n/2} c2^k = O(2^{n/2})$ [CD03].

3.2 Compiling Bayesian Network Classifiers

We now present the algorithm for compiling Bayesian Network classifiers, which is more involved since the feature variables of the classifier are not necessarily independent. As such, we cannot partition the feature variables $\boldsymbol{X}$ in any way, since those in $\boldsymbol{U}$ may interact with those in $\boldsymbol{V}$ in the Bayesian Network.

Our compilation algorithm is based on recursively decomposing into smaller classifiers and identifying those that are equivalent to avoid the compilation of a classifier if an equivalent one has already been compiled. We first describe the method of decomposing into smaller classifiers, which can be described at a high level as follows. Given a Bayesian Network classifier B, let $\boldsymbol{U}$ and $\boldsymbol{V}$ be a partition of its features variables $\boldsymbol{X}$ and let H be a variable not in $\boldsymbol{X}$. Suppose now that we are only interested in classifying instances $\boldsymbol{x}$ that set feature variables $\boldsymbol{U}$ to some state $\boldsymbol{u}$. When certain conditions hold, we can perform the classification using a *smaller* classifier, which is obtained from classifier B as follows:

– Node H is disconnected from its parents and C, the class node, is added as the new parent of H.



(a) Classifier B. (b) Subclassifier $B_{\boldsymbol{u}}^{\boldsymbol{H}}$.

Figure 3.1: Variable H splits feature variables into $(\boldsymbol{U}, \boldsymbol{V})$. When given an instantiation $\boldsymbol{u}$ on the feature variables $\boldsymbol{U}$, we can construct a subclassifier that has the same decision function as the original classifier over feature variables $\boldsymbol{V}$.

- A CPT is assigned to H based on inference on B.
- A prior is assigned to C based on inference on B.
- Leaves are repeatedly removed from classifier B as long as they are not in $C \cup H \cup \boldsymbol{V}$.

The resulting classifier is called a *subclassifier*. Figure 3.1 depicts an example of the structural changes needed to construct a subclassifier. We let F_B denote the decision function of the original classifier B and $F_{B_{\boldsymbol{u}}^H}$ denote the decision function of the subclassifier $B_{\boldsymbol{u}}^H$. The main insight regarding these subclassifiers is that for any $\boldsymbol{u}$ and its corresponding subclassifier, $F_{B_{\boldsymbol{u}}^H}(\boldsymbol{v}) = F_B(\boldsymbol{uv})$ for any $\boldsymbol{v}$. This key property will be used in the algorithm to reduce the compilation of a classifier into the compilation of subclassifiers.

We will now spell out the above result on subclassifiers. We first state the conditions under which a subclassifier can be constructed.

Definition 1 *Let $(\boldsymbol{U}, \boldsymbol{V})$ be a partition of the feature variables $\boldsymbol{X}$ in a Bayesian Network classifier B, and let H be a variable outside $\boldsymbol{X}$. We say that H **splits** feature variables $\boldsymbol{X}$ into $(\boldsymbol{U}, \boldsymbol{V})$ if H d-separates feature variables $\boldsymbol{V}$ from C and $\boldsymbol{U}$.*

Recall that $\boldsymbol{Z}$ d-separates $\boldsymbol{X}$ from $\boldsymbol{Y}$ if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent given $\boldsymbol{Z}$ [Dar09]. We are now ready to define subclassifiers.

Definition 2 *Let B be a Bayesian Network classifier, H be a variable that splits feature variables into $(\boldsymbol{U}, \boldsymbol{V})$, and $\boldsymbol{u}$ be an instantiation of feature variables $\boldsymbol{U}$. The* ***subclassifier*** *for H and $\boldsymbol{u}$, denoted $B^H_{\boldsymbol{u}}$, is obtained from classifier B as follows:*

1. *Disconnect node H from its parents.*
2. *Make H a child of class variable C, and set its Conditional Probability Table (CPT) to $P(H|C\boldsymbol{u})$.*
3. *Set the CPT of C to $P(C|\boldsymbol{u})$.*
4. *Repeatedly remove every leaf node from B that is not in $C \cup H \cup \boldsymbol{V}$.*

Constructing a subclassifier requires some computational work on the original classifier B. First, we need to identify a variable H that satisfies the condition of Definition 1. This can be done in polynomial time as it only involves reasoning about d-separation [Dar09]. Second, we need to determine the CPTs of H and C, which require the computation of posteriors on the H and C, given the state $\boldsymbol{u}$ of feature variables $\boldsymbol{U}$. This requires exact inference on the classifier B. We will later provide a bound on the number of inference calls made by our compilation algorithm for this purpose. The next theorem formalizes the property of subclassifiers.

Theorem 1 *Let B be a Bayesian Network classifier and let H be a variable that splits feature variables into $(\boldsymbol{U}, \boldsymbol{V})$. For a subclassifier $B^H_{\boldsymbol{u}}$, we have $F_B(\boldsymbol{uv}) = F_{B^H_{\boldsymbol{u}}}(\boldsymbol{v})$ for all instantiations $\boldsymbol{v}$ of feature variables $\boldsymbol{V}$.*

According to this theorem, the classification of an instance $\boldsymbol{uv}$ by classifier B will match the classification of $\boldsymbol{v}$ by subclassifier $B^H_{\boldsymbol{u}}$. As we shall see, when our compilation algorithm fixes the state of feature variables $\boldsymbol{U}$ to $\boldsymbol{u}$, it will construct and recursively compile the subclassifier $B^H_{\boldsymbol{u}}$.

We will now introduce the second key result that will form the basis of our algorithm for compiling a Bayesian Network classifier into an OBDD. This result provides a method for detecting when two "similar" Bayesian Network classifiers induce the same decision function.

In this section we assume, without loss of generality, that the classifier has a class node C which has a single child H. This assumption is satisfied by all subclassifiers and can easily be satisfied for any classifier by adding a dummy class node with the original class node as its single child. First we define when two classifiers are considered similar.

Definition 3 *Let B be a Bayesian Network classifier with a class node C which has a single child node H. A second Bayesian Network classifier is* ***similar*** *to B if it has the same structure as B and differs only in the CPTs of C and H.*

Let $\boldsymbol{X}$ be the feature variables of two similar classifiers B and B'. Note that $P(\boldsymbol{X}|H)$ is the same across the two classifiers, and H d-separates C from $\boldsymbol{X}$ by our earlier assumption. Thus, we can rewrite $P(C|\boldsymbol{X})$ as follows:

$$\begin{aligned} P(C|\boldsymbol{x}) &= \sum_h P(C|h)P(h|\boldsymbol{x}) \\ &= \sum_h P(C,h)P(\boldsymbol{x}|h)/P(\boldsymbol{x}) \end{aligned} \tag{3.4}$$

So far, these results have not assumed that the class variable C and the variable H are binary. For the rest of this section, we will assume that nodes C and H are binary, and the classification threshold is t. In this setting, we have an efficient way of detecting when two similar classifiers share the same decision function, in time linear in the number of feature variables $\boldsymbol{X}$. We present the details next.

Setting $a_h = P(c, H = h) - tP(H = h)$, we can rewrite the classification as a linear inequality.

$$\begin{aligned} \sum_h P(c,h)P(\boldsymbol{x}|h) &\geq tP(\boldsymbol{x}) \\ \sum_h (P(c,h) - tP(h))P(\boldsymbol{x}|h) &\geq 0 \\ \sum_h a_h P(\boldsymbol{x}|h) &\geq 0 \end{aligned} \tag{3.5}$$

For two similar classifiers, the values a_h vary but $P(\boldsymbol{x}|h)$ is the same. To detect if two similar classifiers share the same decision function, we just need to verify that the two sets

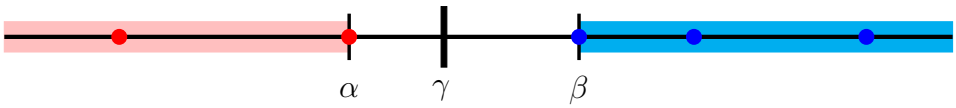


Figure 3.2: Visualization of the equivalence interval of a classifier B. The red dots represent instances classified as 1 and the blue dots represent instances classified as 0. A classifier that is similar to B shares the same decision function as B if its coefficient γ falls within the equivalence interval of B, depicted by the white region between α and β.

of values a_h classify all instances $\boldsymbol{x}$ the same way. To do so, we define the sign, margin, and coefficient of such classifiers.

Definition 4 *Let B be a non-trivial*[1] *Bayesian Network classifier with a threshold t and a binary class node C which has a single binary child node H. Let σ denote the* ***sign*** *of the classifier, which is defined to be 1 if $P(c|H=1) \geq t$ and 0 otherwise. The* ***margin*** *α, β and* ***coefficient*** *γ of B are defined as follows:*

$$\begin{aligned}
\alpha &= \max_{\boldsymbol{x}:F_B(\boldsymbol{x})=1} P(\boldsymbol{x}|H=1-\sigma)/P(\boldsymbol{x}|H=\sigma) \\
\beta &= \min_{\boldsymbol{x}:F_B(\boldsymbol{x})=0} P(\boldsymbol{x}|H=1-\sigma)/P(\boldsymbol{x}|H=\sigma) \\
\gamma &= -1 \cdot \frac{P(c, H=\sigma) - tP(H=\sigma)}{P(c, H=1-\sigma) - tP(H=1-\sigma)}
\end{aligned}$$

That is, α is the largest value of $P(\boldsymbol{x}|H=1-\sigma)/P(\boldsymbol{x}|H=\sigma)$ attained by any instance classified as 1, and β is the smallest such value attained by any instance classified as 0 (see Figure 3.2). The values α, β, and γ come from a rearrangement of Equation 3.5 for the case of a binary H variable. The notion of a margin was actually identified by [CD03] in connection to Naive Bayes classifiers, and turns out to apply to general Bayesian Network classifiers.

The next result was proven only for Naive Bayes classifiers in [CD03]. We generalize this to Bayesian Network classifiers.

[1] *A non-trivial classifier with a binary class node classifies at least one instance as 1 and at least one instance as 0.*

Theorem 2 *Let B be a non-trivial Bayesian Network classifier with a binary class node C and a single binary child node H. Let B' be a second classifier that is similar to B and has the same sign as B. Let t be their threshold, (α, β) be the margin of classifier B, and γ be the coefficient of B'. The two classifiers have the same decision function, $F_B = F_{B'}$, iff γ belongs to the interval $[\alpha, \beta)$. This is called the equivalence interval of classifier B.*

The above theorem enables us to perform binary search over equivalence intervals to identify equivalent subclassifiers: ones that lead to the same decision function and hence the same compilation. This technique avoids the compilation of a subclassifier if an equivalent one has already been compiled.

From Bayesian Network Classifiers to OBDDs

We now present the full algorithm for compiling a Bayesian Network classifier B into an OBDD. We first identify a binary variable H that splits the feature variables into $(\boldsymbol{U}, \boldsymbol{V})$. We then start enumerating over the values of feature variables in $\boldsymbol{U}$ as if we are building a decision tree, in a depth-first manner. Each leaf of this decision tree corresponds to a distinct instantiation $\boldsymbol{u}$ and a subclassifier $B^H_{\boldsymbol{u}}$ with its equivalence interval. These subclassifiers are similar to one another, since they differ only in the CPTs of class C and variable H. Our algorithm will then compile these subclassifiers recursively using the same technique, except that it will avoid compiling a subclassifier if it already compiled an equivalent one—as determined by Theorem 2.

The efficiency of this algorithm depends on the choice of variable H and the corresponding feature variable decomposition $(\boldsymbol{U}, \boldsymbol{V})$, as we want the size of $\boldsymbol{U}$ to be small. We identify such feature variable decompositions in a preprocessing step. That is, after first decomposing feature variables into $(\boldsymbol{U}, \boldsymbol{V})$ using an appropriate H, we follow by decomposing $\boldsymbol{V}$ recursively. This leads us to the notion of a *block ordering* of feature variables.

Definition 5 *Given a Bayesian Network classifier, a* ***block ordering*** *of its feature variables $\boldsymbol{X}$ is a sequence $\pi = (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m)$ such that $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m$ is a partition of feature variables*

$\boldsymbol{X}$, and for each $0 < k < m$, there exists a binary variable H that splits feature variables $\boldsymbol{X}$ into $(\boldsymbol{X}_1 \cup \ldots \cup \boldsymbol{X}_k, \boldsymbol{X}_{k+1} \cup \ldots \cup \boldsymbol{X}_m)$.

Each element $\boldsymbol{X}_i$ is called a *block* of the block ordering π. We will assume that the feature variables in a block are ordered (arbitrarily). As such, we will refer to feature variables by their position in the block ordering π.

We will later discuss a heuristic for obtaining a block order, which we used in our experiments. But for now, we will discuss Algorithms 1 and 2. Algorithm 1 is passed a Bayesian Network classifier B and a block ordering π of feature variables. It creates the sinks of the OBDD and calls Algorithm 2.

Algorithm 2 implements the proposal we discussed earlier. It maintains a cache that stores tuples of the form (D, I, σ, k), where D is an OBDD node, I is an equivalence interval, σ is a boolean, and k is an integer. Such a cache entry means that OBDD D is the result of compiling a subclassifier $B_{\boldsymbol{u}}$ that has equivalence interval I and sign σ. It also means that the last feature variable in block $\boldsymbol{U}$ is at position $k-1$ in the block ordering π. The cache is fetched based on a coefficient γ, a sign σ and a level k. That is, it returns OBDD D if $\gamma \in I$ for the same σ and same k.

Algorithm 2 makes use of four auxiliary functions. First, `get-subclassifier`$(B, \boldsymbol{u}, \pi, k)$ constructs a subclassifier and requires a constant number of calls to an exact inference algorithm to get the coefficients of the subclassifier. `get-sink`(B) takes in a subclassifier with no more feature variables, and returns either the `0-sink` or the `1-sink` based on a simple check. `equivalence-interval`(D) computes the equivalence interval of the classifier leading to OBDD D. This is done in constant time using the equivalence intervals for the children of D [CD03]. Finally, `get-OBDD-node`(S) returns an OBDD node, which is defined by the set S that specifies the node's children and the labels of edges pointing to these children.

Algorithm 1 `compile-classifier`(B, π)

input: Bayesian Network classifier B and block ordering π of feature variables

output: OBDD for the decision function of classifier B

main:

1: `0-sink` $\leftarrow$ terminal OBDD node labeled with 0
2: `1-sink` $\leftarrow$ terminal OBDD node labeled with 1
3: $D \leftarrow$ `compile-subclassifier`$(B, \{\}, \pi, 0)$
4: **return** reduced form of D

Algorithm 2 `compile-subclassifier`$(B, \boldsymbol{u}, \pi, k)$

input: Bayesian Network classifier B, instantiation $\boldsymbol{u}$ of some feature variables, block ordering π of feature variables, integer k

output: OBDD for the decision function of classifier B

main:

1: **if** $\boldsymbol{u}$ is an instantiation of a block in ordering π **then**
2: $\quad B \leftarrow$ `get-subclassifier`$(B, \boldsymbol{u}, \pi, k)$
3: $\quad$ **if** B has no feature variables **then**
4: $\quad\quad$ **return** `get-sink`(B)
5: $\quad \gamma, \sigma \leftarrow$ coefficient and sign of B
6: $\quad D \leftarrow$ `find-in-cache`(γ, σ, k)
7: $\quad$ **if** $D = null$ **then**
8: $\quad\quad D \leftarrow$ `compile-subclassifier`$(B, \{\}, \pi, k)$
9: $\quad\quad I \leftarrow$ `equivalence-interval`(D)
10: $\quad\quad$ `store-in-cache`(D, I, σ, k)
11: $\quad$ **return** D
12: $X \leftarrow$ feature variable at position k in ordering π
13: $S \leftarrow \{\}$
14: **for** each state x of feature variable X **do**
15: $\quad C \leftarrow$ `compile-subclassifier`$(N, \boldsymbol{u} \cup x, \pi, k+1)$
16: $\quad$ add (C, x) to set S
17: **return** `get-OBDD-node`(S)

Algorithm 3 implements a simple, greedy heuristic for obtaining a block ordering of feature variables. Its running time is $O(n^4)$, where n is the number of feature variables, which was sufficient for our experiments.

Algorithm 3 `block-order`$(B, \boldsymbol{X})$

input: A Bayesian Network classifier B with feature variables $\boldsymbol{X}$

output: A block ordering π of the feature variables $\boldsymbol{X}$

main:

1: $H, \boldsymbol{U}, \boldsymbol{V} \leftarrow$ class variable of B, $\boldsymbol{X}, \emptyset$
2: **for** each variable H' that splits feature variables $\boldsymbol{X}$ into $(\boldsymbol{U}', \boldsymbol{V}')$ **do**
3: **if** $|\boldsymbol{U}'| \leq |\boldsymbol{U}|$ **then**
4: $H, \boldsymbol{U}, \boldsymbol{V} \leftarrow H', \boldsymbol{U}', \boldsymbol{V}'$
5: $\boldsymbol{u} \leftarrow$ some instantiation of $\boldsymbol{U}$
6: **return** $\boldsymbol{U}$, `block-order`$(B^H_{\boldsymbol{u}}, \boldsymbol{V})$

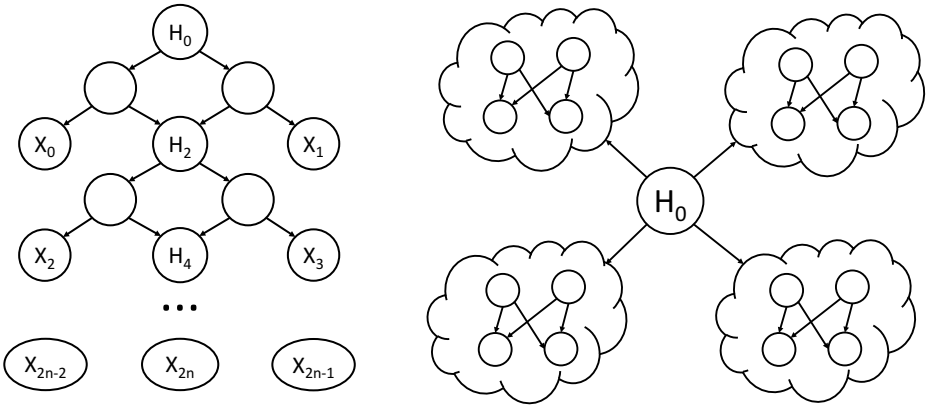
We close this section by providing time and size bounds on our compilation algorithm. We later show that for certain classes of Bayesian Networks, these bounds can be as tight as the bounds provided by [CD03] for compiling Naive Bayes classifiers into OBDDs.

Definition 6 *Let $\pi = (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m)$ be a block ordering of the feature variables in a Bayesian Network classifier B. Let p_i denote the number of feature variables in block i, and let $s(i, j) = \sum_{k=i}^{j} p_k$. The **width** w_π of this order and the **compilation width** w_B of classifier B are defined as:*

$$\begin{aligned} w_\pi &= \max_{i \in \{1,\ldots,m\}} \big[p_i \cdot \min(s(1, i-1), s(i, m))\big] \\ w_B &= \min_{\pi} w_\pi. \end{aligned}$$

We now have the following bounds on Algorithm 1.

Theorem 3 *The number of nodes in the OBDD returned by Algorithm 1 is $O(2^{w_\pi})$, where w_π is the width of order π. Moreover, the running time of the algorithm is $O(PT + w_\pi 2^{w_\pi})$, where P is the sum of the state space sizes of blocks in π and T is the time of an inference call on the classifier.*



(a) Ladder classifier (variables X_i are feature variables)

(b) Cluster classifier (clouds contain arbitrary structure)

Figure 3.3: Examples of classifier families with improved compilation time.

Consider the family of ladder classifiers depicted in Figure 3.3a, which has $n = 2m + 1$ feature variables. We can use the sequence of nodes $H_2, H_4, \ldots, H_{2m}$ to decompose feature variables, leading to the block ordering

$$[X_0, X_1], [X_2, X_3], \ldots, [X_{2m-2}, X_{2m-1}, X_{2m}],$$

which has width $n/2$. The size of the OBDD is $O(2^{n/2})$ and the running time is $O(nT+n2^{n/2})$.

The family of cluster classifiers in Figure 3.3b has similar bounds. Assume that we have n feature variables and k clusters, with each cluster having n/k feature variables. We can repeatedly use the node H_0 to split feature variables into k blocks of size n/k, leading to a block order width $n/2$, and a largest block size n/k. The size of the OBDD is $O(2^{n/2})$ and the running time is $O(k2^{n/k}T + n2^{n/2})$.

What is interesting about these bounds is that they match the ones for compiling Naive Bayes classifiers to OBDDs [CD03]—an NP-hard problem as shown by [SCD18b]. In practice, however, the time and space costs of Algorithm 1 can be quite low as we show in the experiments in Chapter 5.

3.3 Compiling Neural Network Classifiers

In this section, we explore the compilation of Binarized Neural Network classifiers (BNNs) into OBDDs. We have the same motivations as before, except this time we may be interested in only compiling the decision function over an *input space*, rather than over all possible inputs. For example, users of a BNN can pinpoint a particular input instance $\boldsymbol{x}$ and ask for guarantees on the behavior of the BNN for other inputs in the neighborhood of $\boldsymbol{x}$. This has practical applications for image classification, where users expect an image of a dog to remain classified as a dog if only a few pixels are modified.

Let B be a BNN, and let B_S represent the function of B on S, an input region of interest. To obtain B_S in a tractable form, we propose an Angluin-style algorithm for learning the OBDD representation of B_S [Ang87]. Our algorithm leverages an existing technique for learning an OBDD using standard membership and equivalence queries [Nak05]. First, we construct a hypothesis OBDD and then iteratively call equivalence queries, adding OBDD nodes until its output agrees with B_S. To answer equivalence queries efficiently, we encode the BNN and the hypothesis OBDD into a CNF, and require that the region S can be encoded as a CNF as well. When the algorithm terminates, it returns an OBDD D such that $D(\boldsymbol{x}) = B(\boldsymbol{x}) : \forall \boldsymbol{x} \in S$, a notion related to the `Constrain` operator on OBDDs [MT98]. We then verify properties of BNN B by performing efficient verification queries on OBDD D.

Our algorithm can also be used as an incremental and anytime compilation algorithm, by slowly increasing the region of interest. The compiled OBDD of a smaller region can be used as the hypothesis OBDD for the compilation task of a larger region, without starting over. We can essentially save our progress, and build on it at a later time if we decide the initial region is too small.

We next provide the encoding of BNNs and OBDDs into CNF, which will serve an important role in our main compilation algorithm.

BNN to CNF

We use the conversion given by Narodytska et al. [NKR18]. An internal block of a BNN consists of three layers: a linear transformation (LIN), batch normalization (BN), and binarization (BIN). The LIN layer has parameters $\boldsymbol{a}$ (weights) and b (bias). The BN layer has parameters μ (mean), σ (std), α (weight), and γ (bias). Put together, the three layers of an internal block can be translated to the following function $h(\boldsymbol{x})$ on an input instance $\boldsymbol{x}$ [NKR18].

$$h(\boldsymbol{x}) = 1 \iff \langle \boldsymbol{a}, \boldsymbol{x} \rangle \geq -\frac{\sigma}{\alpha}\gamma + \mu - b$$

Since the weights $\boldsymbol{a}$ and input $\boldsymbol{x}$ are binarized as $\{-1, 1\}$, the above computation reduces to a cardinality constraint of the form $\sum_{i=1}^{m} l_i \geq C$, where $l_i \in \{0, 1\}$ and $C \in \mathbb{R}$. This cardinality constraint can be encoded as a CNF.

The output block has a LIN layer followed by an ARGMAX layer, which can be encoded using a similar technique. First, we encode a cardinality constraint for all pairs of classes, which tells us the class that has a higher activation function in the pairing. Then, we use a final set of cardinality constraints to determine the class that was the winner in all of its pairings [NKR18]. Since we focus on Neural Networks with binary output classes in this paper, a single CNF variable is enough to represent the output of the BNN.

The space complexity of this conversion is $O(NC^2)$, where N is the number of neurons in the BNN and C is the constant from the above cardinality constraint.

OBDD to CNF

We convert an OBDD into a CNF using the well-known Tseitin transformation [Tse68], which converts a Boolean circuit into a CNF. Consider an OBDD node labelled by variable X. If the two children of this node compute Boolean functions C_0, C_1, then the OBDD node computes the Boolean function $R = (C_0 \wedge \neg X) \vee (C_1 \wedge X)$. We can then represent the Boolean function of this node by the following five clauses:

$$\neg R \vee C_0 \vee X \qquad\qquad R \vee \neg C_0 \vee X$$
$$\neg R \vee C_1 \vee \neg X \qquad\qquad R \vee \neg C_1 \vee \neg X$$
$$\neg R \vee C_0 \vee C_1$$

Applying this conversion to all OBDD nodes leads to a CNF representation of the Boolean function computed by the OBDD. The number of CNF clauses produced by this conversion is $5N$, where N is the number of OBDD nodes.

The above encodings allow us to convert a BNN into a CNF α and an OBDD into a CNF β. Let $\boldsymbol{X}$ be the CNF variables corresponding to the BNN inputs and O be the variable corresponding to its output. Then $\alpha \wedge \boldsymbol{x} \wedge O$ will be satisfiable iff the BNN outputs 1 under input $\boldsymbol{x}$. Similarly, $\alpha \wedge \boldsymbol{x} \wedge \neg O$ will be satisfiable iff the BNN outputs 0 under input $\boldsymbol{x}$. Now let $\boldsymbol{X}$ be the CNF variables corresponding to the OBDD variables and R be the variable we introduced for the OBDD root. Then $\beta \wedge \boldsymbol{x} \wedge R$ will be satisfiable iff the OBDD outputs 1 under input $\boldsymbol{x}$ and $\beta \wedge \boldsymbol{x} \wedge \neg R$ will be satisfiable iff the OBDD outputs 0 under input $\boldsymbol{x}$.

When the BNN and the OBDD share the same inputs $\boldsymbol{x}$, we can check for their inequivalence with the formula $\phi = \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R)$ [NKR18]. Then, ϕ is satisfiable iff there is some instantiation of $\boldsymbol{x}$ such that $(O \wedge \neg R) \vee (\neg O \wedge R)$ (i.e. BNN and OBDD disagree).

Angluin-Style Exact Learning of Finite Automaton

In this section we describe Angluin's algorithm for learning Deterministic Finite Automata (DFA) [Ang87]. The DFA learning algorithm has an adaptation for learning OBDDs [Nak05], which serves as the backbone for our neural network compilation algorithm. DFAs and OBDDs are initimately related: a Complete OBDD (an OBDD that does not skip variables [Weg00]) is also a DFA (but a DFA is not necessarily an OBDD).

We roughly summarize the exposition on the topic of learning DFAs from the textbook by Kearns and Vazirani [KV94]. The learning algorithm falls under the category of *active* learning where the algorithm can learn through experimentation, as opposed to *passive*

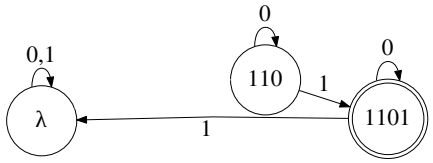
learning where the algorithm has no control over the sample of examples. To learn the DFA for a function f, the learning process requires access to oracles for two types of queries:

- Membership Queries: The learning process selects an instance $\boldsymbol{x}$ and the oracle returns the value of $f(\boldsymbol{x})$.
- Equivalence Queries: The learner submits a hypothesis automaton h. The oracle tells the learner if h computes the correct function (i.e. $h = f$), otherwise the oracle returns a counterexample $\boldsymbol{x}$ for which $h(\boldsymbol{x}) \neq f(\boldsymbol{x})$.

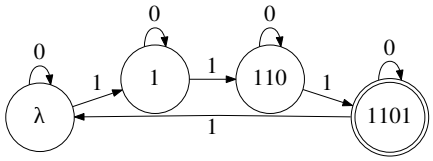
The main idea of the algorithm is as follows. Let S be the set of states of a minimal DFA we want to learn. Recall that each state represents a distinct equivalence class of input strings. At all times we keep a hypothesis DFA whose states $S^\star$ represent a partition of S. We iteratively refine the partition by splitting some partition element of $S^\star$ into two, so that $|S^\star|$ increases. When $|S^\star| = |S|$, each element in the partition contains exactly one equivalence class from S, so our hypothesis DFA computes the target DFA.

Initially, we start with a one-node hypothesis DFA with just one state, which partitions all the states in S into one group. As long as our DFA is incorrect, we will receive counterexamples from the equivalence query. Given a counterexample e, we can simulate e on our hypothesis DFA and identify the first state $s^\star$ for which its following step in the simulation is provably incorrect. This can be done efficiently by maintaining a binary classification tree, the details of which we omit. We then refine the partition by splitting $s^\star$ into two nodes. This process repeats until we have learned all the states of S, at which point the equivalence query gives no more counterexamples and our algorithm terminates.

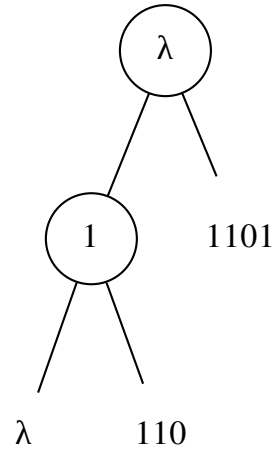
Suppose we wish to learn a DFA on binary inputs for the 3 mod 4 counter f, and we currently have the hypothesis DFA h in Figure 3.4a and its binary classification tree in Figure 3.4c. Since $h(1101) = 0 \neq f(1101)$, we get the string `1101` as a counterexample. Using the binary classification tree along with membership queries, the algorithm identifies the state λ in h as faulty, and splits it into two. This generates the updated DFA in Figure 3.4b, which computes f correctly.



(a) Hypothesis DFA h



(b) Updated DFA



(c) Binary classification tree of h

Figure 3.4: Learning the finite automaton for the 3 mod 4 counter. Using the counterexample `1101`, we modify the hypothesis DFA into the updated DFA.

The automaton learning algorithm was adapted into an OBDD learning algorithm by Nakamura [Nak05]. This variation requires n equivalence queries and $6n^2 + n\log(m)$ membership queries, where n is the number of nodes in the final OBDD and m is the number of variables in the OBDD.

BNN Compilation Algorithm

We now describe our main contribution: a compilation algorithm from a BNN to an OBDD. Given a BNN B on n binary inputs and one binary output, we wish to obtain an OBDD D that computes the function of B on a region S (i.e. $D(\boldsymbol{x}) = B(\boldsymbol{x}) : \forall \boldsymbol{x} \in S$). We require region S to be encoded as a CNF.

Algorithm 4 implements our proposal. The subroutines `BNNToCNF` and `OBDDToCNF` perform the encodings described earlier. We encode the BNN B as a CNF α with output variable O. Then, we start the OBDD learning algorithm as described in Section 3.3 to learn the reduced OBDD representation of B. The learning algorithm creates a hypothesis OBDD D, which we encode as a CNF β with variable R representing the OBDD output. We

Algorithm 4 `CompileBNN`$(B, \boldsymbol{X}, S)$

input: A Binarized Neural Network B with input variables $\boldsymbol{X}$, and a CNF S encoding an input region

output: An OBDD D computing the function of B on S

main:

1: $\alpha, O \leftarrow$ `BNNToCNF`$(B, \boldsymbol{X})$
2: $D \leftarrow$ initial hypothesis OBDD
3: $\beta, R \leftarrow$ `OBDDToCNF`$(D, \boldsymbol{X})$
4: $\phi \leftarrow \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R) \wedge S$
5: **while** ϕ has a satisfying assignment $\boldsymbol{s}$ **do**
6: $\quad \boldsymbol{x} \leftarrow$ projection of $\boldsymbol{s}$ on $\boldsymbol{X}$
7: $\quad D \leftarrow$ `UpdateHypothesis`$(D, \boldsymbol{x})$
8: $\quad \beta, R \leftarrow$ `OBDDToCNF`$(D, \boldsymbol{X})$
9: $\quad \phi \leftarrow \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R) \wedge S$
10: **return** D

set ϕ on Line 4 such that ϕ has a satisfying assignment iff the current hypothesis OBDD D does not compute the same function as BNN B on region S. While ϕ is satisfiable, we take the satisfying assignment and keep only the variables corresponding to the BNN/OBDD inputs as our counterexample $\boldsymbol{x}$. The subroutine `UpdateHypothesis` then edits our hypothesis OBDD using counterexample $\boldsymbol{x}$. Once we have an unsatisfiable ϕ, we return the OBDD D with the guarantee that it computes the same function as BNN B on S. Note that there are no guarantees on the output of OBDD D on instances outside S. The number of iterations of the `while` loop is N, where N is the number of nodes in the final output D.

In Algorithm 5 we propose the construction of an input region that captures all instances in the neighborhood of some instance $\boldsymbol{x}$ on n variables. More specifically, Algorithm 5 takes in an instance $\boldsymbol{x}$, a radius r, and outputs a CNF S on variables $X_1, \ldots, X_n$. An instance $\boldsymbol{x}^\star$ is a satisfying assignment for S iff the Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{x}^\star$ is no greater than r. This becomes a cardinality constraint, which can be encoded in many ways [BB03].

Algorithm 5 `r-RadiusDomain`$(\boldsymbol{x}, r)$

input: An input $\boldsymbol{x} = x_1, \ldots, x_n$ and a radius $r \leq n$

output: A CNF that encodes all instances $\boldsymbol{x}^\star$ such that $h(\boldsymbol{x}, \boldsymbol{x}^\star) \leq r$, where h measures the Hamming distance

main:

1: $d \leftarrow$ a 2D array with dimensions $[0, n] \times [0, r]$
2: **for** $j \leftarrow 0$ to r **do**
3: $\quad d_{0,j} \leftarrow \top$
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad$ **for** $j \leftarrow 0$ to r **do**
6: $\quad\quad h \leftarrow d_{i-1,j}$
7: $\quad\quad l \leftarrow d_{i-1,j-1}$ **if** $j > 0$ **else** $\bot$
8: $\quad\quad d_{i,j} \leftarrow$ OBDD node: label X_i, x_i-child h, $\neg x_i$-child l
9: **return** `OBDDToCNF`$(d_{n,r}, \boldsymbol{X})$

For ease of exposition, we use an OBDD for the constraint and then convert it to CNF. In the algorithm, node $d_{i,j}$ stores the state with $n - i$ variables processed and a current Hamming distance of $r - j$. On Line 8, the child edge of $d_{i,j}$ that agrees with x_i points to $d_{i-1,j}$. The other child edge points to $d_{i-1,j-1}$ if $j > 0$, otherwise it points to $\bot$. By using S as an input for Algorithm 4, we can compile an OBDD that exactly computes the function of a BNN for all instances close to some instance of interest, measured by the number of differing variables. The time and space complexity of Algorithm 5 is $O(nr)$.

To extend our algorithm into an anytime compilation algorithm, we start with a small region of interest and increase its size over time. The compiled OBDD D will compute the same function as B on this small region. To compile the OBDD for a larger region, we can feed in D as the initial hypothesis OBDD in Algorithm 4 on Line 2, without the need to build D from scratch. Then, we can use the updated OBDD to verify the properties of B on the enlarged region. We can continue to enlarge this region until it becomes $\{0, 1\}^n$, at which point $S = \top$ and the compiled OBDD computes the same function as B everywhere.

3.A Proofs

Proof of Theorem 1 We want to show that $F_B(\boldsymbol{uv}) = F_{B^{\boldsymbol{H}}_{\boldsymbol{u}}}(\boldsymbol{v})$. We let $P(.)$ denote the probability distribution of the original classifier B and let $P'(.)$ denote the probability distribution of the subclassifier $B^{\boldsymbol{H}}_{\boldsymbol{u}}$. First, we work out the following equalities:

$$\begin{aligned} P(\boldsymbol{h}|\boldsymbol{u}) &= \sum_c P(\boldsymbol{h}|c\boldsymbol{u})P(c|\boldsymbol{u}) \\ &= \sum_c P'(h|c)P'(c) = P'(h) \end{aligned}$$

$$\begin{aligned} P(\boldsymbol{v}|\boldsymbol{u}) &= \sum_{\boldsymbol{h}} P(\boldsymbol{v}|\boldsymbol{hu})P(h|\boldsymbol{u}) \\ &= \sum_{\boldsymbol{h}} P(\boldsymbol{v}|\boldsymbol{h})P(\boldsymbol{h}|\boldsymbol{u}) \\ &= \sum_h P'(\boldsymbol{v}|h)P'(h) = P'(\boldsymbol{v}) \end{aligned}$$

This gives us the following list of identities:

$$P(c|\boldsymbol{u}) = P'(c) \qquad P(\boldsymbol{h}|c\boldsymbol{u}) = P'(h|c)$$
$$P(\boldsymbol{h}|\boldsymbol{u}) = P'(h) \qquad P(\boldsymbol{v}|\boldsymbol{u}) = P'(\boldsymbol{v})$$

Next, we will show the main property we are after: for any c and $\boldsymbol{v}$, $P(c|\boldsymbol{uv}) = P'(c|\boldsymbol{v})$.

$$\begin{aligned} P(c|\boldsymbol{uv}) &= \sum_{\boldsymbol{h}} P(c|\boldsymbol{huv})P(\boldsymbol{h}|\boldsymbol{uv}) \\ &= \sum_{\boldsymbol{h}} P(c|\boldsymbol{hu})P(\boldsymbol{h}|\boldsymbol{vu}) \\ &= \sum_{\boldsymbol{h}} \frac{P(\boldsymbol{h}|c\boldsymbol{u})P(c|\boldsymbol{u})}{P(\boldsymbol{h}|\boldsymbol{u})} \frac{P(\boldsymbol{v}|\boldsymbol{hu})P(\boldsymbol{h}|\boldsymbol{u})}{P(\boldsymbol{v}|\boldsymbol{u})} \\ &= \sum_h \frac{P'(h|c)P'(c)}{P'(h)} \frac{P'(\boldsymbol{v}|h)P'(h)}{P'(\boldsymbol{v})} \\ &= \sum_h P'(c|h)P'(h|\boldsymbol{v}) = P'(c|\boldsymbol{v}) \end{aligned}$$

Since the threshold is the same for the classifier B and the subclassifier $B^{\boldsymbol{H}}_{\boldsymbol{u}}$, it follows that $F_B(\boldsymbol{uv}) = F_{B^{\boldsymbol{H}}_{\boldsymbol{u}}}(\boldsymbol{v})$ for all $\boldsymbol{v}$. □

Proof of Theorem 2 We let $P(.)$ denote the probability distribution of the classifier B and let $P'(.)$ denote the probability distribution of the classifier B'. Using Equation 3.5 we can rewrite the classification decision of classifier B as $\sum_h a_h P(\boldsymbol{x}|h) \geq 0$, where $a_h = P(c, H = h) - tP(H = h)$. Since h is binary, we can expand the summation.

$$a_0 P(\boldsymbol{x}|H = 0) + a_1 P(\boldsymbol{x}|H = 1) \geq 0$$

Since the classifier B is nontrivial, we know that $a_0/a_1 < 0$. Suppose that the sign σ of B is 1, and thus $a_1 > 0$. Rearranging, we get:

$$-1 \cdot \frac{a_1}{a_0} \geq \frac{P(\boldsymbol{x}|H = 0)}{P(\boldsymbol{x}|H = 1)} \tag{3.6}$$

Now suppose $F_B(\boldsymbol{x}) = 1$ for some $\boldsymbol{x}$. Recall that α is the maximum value of $\frac{P(\boldsymbol{x}|H=0)}{P(\boldsymbol{x}|H=1)}$ attained by any instance classified as 1. Let $a'_h = P'(c, H = h) - tP'(H = h)$.

$$-1 \cdot \frac{a'_1}{a'_0} = \gamma \geq \alpha \geq \frac{P(\boldsymbol{x}|H = 0)}{P(\boldsymbol{x}|H = 1)}$$

So we have that $F'_B(\boldsymbol{x}) = 1$. The proof is analogous for instances classified as 0, as well as for classifiers with sign 0, thus $F_B(\boldsymbol{x}) = F'_B(\boldsymbol{x})$ for all $\boldsymbol{x}$. □

Proof of Theorem 3 Let $\pi = (X_1, \ldots, X_m)$ be the block ordering of the feature variables, and let $s(i, j) = \sum_{k=i}^{j} p_k$, where p_k denotes the number of feature variables in block k. Let t_k be the number of OBDD nodes in levels $s(1, k-1)$ to $s(1, k) - 1$, so $\sum_k t_k$ is the total number of nodes in the OBDD.

We will bound the number of nodes in the OBDD by bounding the number of nodes in each block. The number of OBDD nodes on level $s(1, k-1)$ is bounded from above by $2^{s(1,k-1)}$ (by decision trees) and also by $2 \cdot 2^{s(k,m)}$ (by equivalence intervals). The bound of $2 \cdot 2^{s(k,m)}$ from equivalence intervals is due to the following observation. For the subclassifiers stored in cache of sign 1 and level $s(1, k-1)$, their classification decision on an instance $\boldsymbol{x}$ can be written as in Equation 3.6. Since the RHS of the inequality of Equation 3.6 is the same among all subclassifiers of sign 1 and level $s(1, k-1)$, and there are $2^{s(k,m)}$ distinct instances for such subclassifiers, there are at most $2^{s(k,m)}$ equivalence classes of subclassifiers of sign

1 and level $s(1, k-1)$. The analysis for subclassifiers of sign 0 and level $s(1, k-1)$ is the same, so we have at most $2 \cdot 2^{s(k,m)}$ equivalence classes for subclassifiers on level $s(1, k-1)$.

From level $s(1, k-1)$ to level $s(1, k)-1$, the algorithm constructs the OBDD in a decision tree manner. Therefore, we have that t_k is bounded by $p_k \cdot \min(2^{s(1,k-1)}, 2 \cdot 2^{s(k,m)})$. So, $t_j = O(2^{w_\pi})$, where $j = \text{argmax}_k(t_k)$. To finish, observe that both sequences $t_{j+1}, t_{j+2}, \ldots$ and $t_{j-1}, t_{j-2}, \ldots$ on either side of j decay exponentially fast, so we have that the total number of nodes is $\sum_k t_k = O(t_j) = O(2^{w_\pi})$.

Next we will bound the time complexity of the algorithm. We start by showing that the number of exact inference calls is $P = p_1 + \ldots + p_m$. This number is much smaller than the number of subclassifiers constructed, which is $O(2^{w_\pi})$, because we can share the results of inference calls across different subclassifier constructions.

We want to show that for multiple classifiers that are similar, the construction of their subclassifiers can reuse the same inference calls. For a set of similar classifiers, let H' be the child of class node C and let H be the new splitting node used to construct the subclassifiers. Note that to construct a subclassifier, we need the values $P(h|\boldsymbol{u}c)$ and $P(c|\boldsymbol{u})$.

$$P(h|\boldsymbol{u}c) = \sum_{h'} P(h|\boldsymbol{u}ch')P(h'|c)$$

The terms $P(h|\boldsymbol{u}ch')$ are actually the same across similar classifiers, since similar classifiers only differ in the CPTs of C and H' and those variables are fixed in these terms. As well, the terms $P(h'|c)$ do not require any inference at all, since these are just the CPTs encoded in the network. A similar analysis shows that inference calls can also be shared when calculating the value of $P(c|\boldsymbol{u})$. Therefore, the total number of inference calls for the i-th block is $O(p_i)$. Finally, computing equivalence intervals in the algorithm can be done without any inference calls using the equivalence intervals of subclassifiers. So, the total number of inference calls is $O(P)$.

As for the number of computations of the algorithm, observe that the most expensive operation is finding and storing equivalent subclassifiers in cache, which requires binary search on $O(2^{w_\pi})$ intervals. This gives us $O(w_\pi 2^{w_\pi})$ computations and a time complexity of $O(PT + w_\pi 2^{w_\pi})$. □

CHAPTER 4

Explanation Techniques

In Chapter 3, we examined compilation algorithms for converting the decision function of machine learning classifiers into OBDDs. In this section, we will leverage the tractability of OBDDs to develop efficient methods for reasoning and generating explanations for these decision functions. OBDDs can handle a wide range of queries and transformations in polynomial time, which will serve as building blocks for our explanation techniques. The work in this chapter is published in [SCD18a, SCD18b]

We describe these explanation techniques with respect to a decision function, rather than a classifier, so the set of all variables in the explanations is the set of all feature variables in the original classifier. We first look at *instance-based techniques*, which analyze a decision function with respect to a particular instance. Then, we will contrast this with *classifier-based techniques*, which examine a decision function more generally, taking into account every possible input instance. We will then use these explanation techniques to analyze the behavior of machine learning classifiers in Chapter 5.

4.1 Instance-Based Techniques

4.1.1 Prime Implicant

The first class of explanations we consider is *prime-implicant*, or PI for short. These explanations answer the following question: what is the smallest subset of variables that renders the remaining variables irrelevant to the current decision? In other words, which subset of variables—when fixed—would allow us to arbitrarily toggle the values of other variables, while maintaining the classifier's decision?

Let $\boldsymbol{y}$ and $\boldsymbol{z}$ be instantiations of some variables and call them *partial instances.* We will write $\boldsymbol{y} \supseteq \boldsymbol{z}$ to mean that $\boldsymbol{y}$ extends $\boldsymbol{z}$. That is, it includes the variables in $\boldsymbol{z}$ but may set some additional variables.

Definition 7 (PI-Explanation) *Let f be a decision function and $\boldsymbol{x}$ be an instance. A PI-explanation of f on $\boldsymbol{x}$ is a partial instance $\boldsymbol{y}$ such that*

(a) $\boldsymbol{y} \subseteq \boldsymbol{x}$,

(b) $f(\boldsymbol{x}) = f(\boldsymbol{x}^\star)$ for every $\boldsymbol{x}^\star \supseteq \boldsymbol{y}$, and

(c) no other partial instance $\boldsymbol{z} \subset \boldsymbol{y}$ satisfies (a) and (b).

So, fixing the partial instantiation of $\boldsymbol{y}$ guarantees the classification decision of $f(\boldsymbol{x})$ regardless of how the remaining variables are set. A PI-explanation on an instance for a decision function represented as an OBDD can be computed using Algorithm 6. In fact, Algorithm 6 returns a set of PI-explanations, encoded as an Ordered (Non-Binary) Decision Diagram with three types of edges: negative literal, positive literal, and don't-care (the literal does not appear in the PI-explanation). In the output decision diagram, the edges on a path from the root to the 1-sink will give us a PI-explanation.

4.1.2 Robustness

The second class of explanations we consider is *robustness.* Robustness explanations answer the following question: what is the smallest perturbation on an instance required to change the function decision? As an example, in the domain of educational assessment, we may want to know whether a passing student was only a few test questions away from failing a test, or whether they would have still passed it, even if they had gotten many more questions wrong. As another example, in the context of image classification and self-driving cars, we may want to know whether or not an image is only a few pixels away from being classified as a stop sign. Given a decision function f, we may define the robustness of an instance $\boldsymbol{x}$ as the number of variables that we need to flip, before we change the function decision.

Algorithm 6 `pi-inst`$(f, \pi, \boldsymbol{x})$

input: OBDD f, variable ordering π, and instance $\boldsymbol{x}$

output: Ordered Decision Diagram g for PI-explanations of f on $\boldsymbol{x}$

main:

1: **if** π is empty **return** f
2: remove first variable X from order π
3: $g_* \leftarrow$ `pi-inst`$(f_{\bar{x}} \wedge f_x, \pi, \boldsymbol{x})$
4: **if** $\boldsymbol{x}$ sets X to $\bar{x}$ **then**
5: $\quad g_{\bar{x}} \leftarrow$ `pi-inst`$(f_{\bar{x}}, \pi, \boldsymbol{x})$, $\quad g_x \leftarrow \bot$
6: **else**
7: $\quad g_{\bar{x}} \leftarrow \bot$, $\quad g_x \leftarrow$ `pi-inst`$(f_x, \pi, \boldsymbol{x})$
8: $g_{\bar{x}} \leftarrow g_{\bar{x}} \wedge \neg g_*$, $\quad g_x \leftarrow g_x \wedge \neg g_*$
9: **return** Ordered Decision Diagram with branches $g_{\bar{x}}, g_x, g_*$

Definition 8 *Given a non-trivial decision function f and an instance $\boldsymbol{x}$, we define the robustness of the decision as:*

$$\mathsf{robustness}_f(\boldsymbol{x}) = \min_{\boldsymbol{x}': f(\boldsymbol{x}') \neq f(\boldsymbol{x})} d(\boldsymbol{x}', \boldsymbol{x})$$

where $d(\boldsymbol{x}', \boldsymbol{x})$ denotes the Hamming distance between $\boldsymbol{x}'$ and $\boldsymbol{x}$, i.e., the number of variables X where $\boldsymbol{x}$ and $\boldsymbol{x}'$ differ.

The following observation implies an efficient algorithm for computing robustness. Consider a decision function $f(Y, \boldsymbol{X})$. The robustness of a positive instance $y, \boldsymbol{x}$ satisfies:

$$\mathsf{robustness}_f(y, \boldsymbol{x}) = \min\{\mathsf{robustness}_{f|y}(\boldsymbol{x}), 1 + \mathsf{robustness}_{f|\bar{y}}(\boldsymbol{x})\}$$

where $\mathsf{robustness}_f(\boldsymbol{x}) = 0$ if $f = \bot$ (false) and $\mathsf{robustness}_f(\boldsymbol{x}) = \infty$ if $f = \top$ (true). So, it takes time linear in the size of an OBDD to compute the robustness of a given instance (by caching intermediate results, each node of an OBDD is evaluated at most once).

4.1.3 Minimum Cardinality

The next class of explanations we consider is *minimum-cardinality*, or MC for short. To motivate these explanations, consider a classifier that has diagnosed a patient with some disease based on some observed test results, some of which were positive and others negative. Some of the positive test results may not be necessary for the classifier's decision: the decision would remain intact if these test results were negative.

A MC explanation then tells us which of the positive test results are the culprits for the classifier's decision. In other words, we identify a minimal subset of the positive test results that is sufficient for the current decision.

Consider two instances $\boldsymbol{x}^\star$ and $\boldsymbol{x}$. We write $\boldsymbol{x}^\star \subseteq^1 \boldsymbol{x}$ to mean the variables set to 1 in $\boldsymbol{x}^\star$ are a subset of those set to 1 in $\boldsymbol{x}$. We define $\boldsymbol{x}^\star \subseteq^0 \boldsymbol{x}$ analogously. Moreover, we write $\boldsymbol{x} \leq^1 \boldsymbol{x}^\star$ to mean: the count of 1-variables in $\boldsymbol{x}$ is no greater than their count in $\boldsymbol{x}^\star$. We define $\boldsymbol{x} \leq^0 \boldsymbol{x}^\star$ analogously.

Definition 9 (MC-Explanation) *Let $f(\boldsymbol{X})$ be a given decision function. An MC-explanation of a positive instance $\boldsymbol{x}$ is another positive instance $\boldsymbol{x}^\star$ such that $\boldsymbol{x}^\star \subseteq^1 \boldsymbol{x}$ and there is no other positive instance $\boldsymbol{x}' \subseteq^1 \boldsymbol{x}$ where $\boldsymbol{x}' <^1 \boldsymbol{x}^\star$. An MC-explanation of a negative instance $\boldsymbol{x}$ is another negative instance $\boldsymbol{x}^\star$ such that $\boldsymbol{x}^\star \subseteq^0 \boldsymbol{x}$ and there is no other negative instance $\boldsymbol{x}' \subseteq^0 \boldsymbol{x}$ where $\boldsymbol{x}' <^0 \boldsymbol{x}^\star$.*

Cardinality minimization was explored in [Dar01] for Decomposable Negation Normal Form, which is a superset of OBDD. The minimization procedure, which can be done in a linear pass, applies directly to OBDDs.

4.2 Classifier-Based Techniques

The next explanation techniques examine the decision function as a whole, without specifying a particular instance.

4.2.1 Monotonicity

We consider the property of *monotonicity* on a decision function. Intuitively, a monotone function satisfies the following. A positive instance remains positive if we flip some of its variables from 0 to 1. Moreover, a negative instance remains negative if we flip some of its variables from 1 to 0. In certain domains, one expects or desires a classifier learned from data to be monotone [GBF04]. For example, in the context of educational assessment, we expect a student to be assessed positively if their correct answers are a superset of those of another student who has been assessed positively.

More formally, consider two instances $\boldsymbol{x}^\star$ and $\boldsymbol{x}$. As before, we write $\boldsymbol{x}^\star \subseteq^1 \boldsymbol{x}$ to mean the variables set to 1 in $\boldsymbol{x}^\star$ is a subset of those set to 1 in $\boldsymbol{x}$. Monotone classifiers are then characterized by the following property of their decision functions.

Definition 10 *A decision function $f(\boldsymbol{X})$ is monotone if*

$$\boldsymbol{x}^\star \subseteq^1 \boldsymbol{x} \quad \textit{only if} \quad f(\boldsymbol{x}^\star) \leq f(\boldsymbol{x}).$$

So, if the positive variables in instance $\boldsymbol{x}$ contain those in instance $\boldsymbol{x}^\star$, then instance $\boldsymbol{x}$ must be positive if instance $\boldsymbol{x}^\star$ is positive.

For a decision function represented as an OBDD, monotonicity can be decided in time quadratic in the OBDD size [HI02]. A simpler but less efficient approach is based on the following observation: a decision function f is monotone iff $f|\bar{x} \models f|x$ for all variables X. Given an OBDD f, we can perform conditioning ($f|\bar{x}$, $f|x$) and test entailment ($f|\bar{x} \models f|x$) in time polynomial in the size of input OBDDs.

We also consider unateness, a mild generalization of monotonicity, which can also be verified efficiently given a decision function represented by an OBDD.

Definition 11 *A decision function $f(\boldsymbol{X})$ is unate if for all X*

$$f|\bar{x} \models f|x \quad \textit{or} \quad f|x \models f|\bar{x}.$$

Any monotone decision function is also unate. Intuitively, in a unate function, for each variable X there exists a polarity x or $\bar{x}$ where flipping variable X to that polarity will

cause a positive instance to remain positive. In other words, it behaves like a monotone function when we interpret the appropriate polarity of each variable as if it were positive. As with monotone functions, we can efficiently test if a given decision function is unate given an OBDD representation of that function. Moreover, if a decision function is unate, then certain explanations become more efficient to compute, such as prime-implicant explanations, as shown by [SCD18b].

4.2.2 Irrelevant Variables

Another important explanation technique is identifying irrelevant variables. This particularly comes up with Bayesian networks that have multiple class variables, as only a subset of the variables may turn out to be relevant to a particular class variable. In this case, we would like to detect and then drop these irrelevant variables, to obtain a simpler and more computationally efficient classifier.

Definition 12 *A decision function $f(\boldsymbol{X})$ essentially depends on variable X if $f|x \neq f|\bar{x}$.*

If the decision function does not essentially depend on variable X, we say that variable X is irrelevant to the decision. In an OBDD, conditioning on a variable takes time that is linear in the size of the OBDD. Moreover, equivalence testing can be done in constant time. Hence, determining if a variable is irrelevant can be done efficiently. In fact, for a reduced OBDD, it suffices to scan the OBDD to see if any node is labeled by variable X.

CHAPTER 5

Experiments and Case Studies

In this chapter we report on experiments for the compilation algorithms presented in Chapter 3, and we run the explanation queries on the compiled tractable decision diagrams using the techniques from Chapter 4. First, we compile Bayesian Network classifiers from literature into OBDDs, such as ones that diagnose printer failures or assess educational outcomes. Then, we train Binarized Neural Network classifiers on the USPS digit dataset and compile them into OBDDs. These OBDDs are then converted into SDDs using the SDD software package [CD18]. The SDD representation can be much smaller than the OBDD representation, and preserves tractability of our explanation techniques. Lastly, we implement our explanation algorithms on top of the SDD software package, and examine the behavior of the machine learning classifiers.

5.1 Bayesian Network Classifier Experiments

Table 5.3b summarizes compilation experiments we ran on three Bayesian Network classifiers using leaf nodes as feature variables. For each network we included a number of classifiers, each corresponding to one root of the network, using a threshold of $\frac{1}{2}$. Table 5.2 provides similar results on two other networks, except in this case we sampled some of the leaf nodes to include as feature variables (the networks were too large to fully compile). Inference calls were performed using the SamIAm library [Dar].

The sizes of the resulting OBDDs are quite small. For example, the size of the OBDD of the Andes classifier with root ValueKnownEq(VKE) is less than 1% of the state space size given by the block order width. The limiting factor for the compilation algorithm is the

Table 5.1: win95pts has 76 nodes, 16 feature variables and width 9. Andes has 223 nodes, 24 feature variables and width 18. cpcs54 has 54 nodes, 13 feature variables and width 14. Width refers to the network tree-width, approximated by the minfill heuristic.

network	class	block order width	largest / # blocks	OBDD size	compile time (s)
win95pts	GRDS	15	15 / 2	498	21
win95pts	PO	15	15 / 2	291	21
win95pts	PAT	13	13 / 4	636	5
win95pts	PDrvr	14	14 / 3	352	11
win95pts	PMem	13	13 / 4	890	5
win95pts	POn	14	14 / 3	31	11
win95pts	PPpr	14	14 / 3	31	10
Andes	TK	23	18 / 7	47	11,708
Andes	VKE	19	19 / 6	2,107	27,495
Andes	CNBG	19	19 / 6	2,893	24,374
Andes	MDA	19	19 / 6	5,454	26,614
cpcs54	x3	12	12 / 2	25	69
cpcs54	x4	12	12 / 2	92	69
cpcs54	x9	11	11 / 3	13	35

compilation time, which depends on the treewidth of the network and scales exponentially with respect to the largest block. The treewidth affects the time of each inference call, and the largest block bounds the number of inference calls made by the compilation algorithm. For example, the two emdec6g experiments in Table 5.2 with 27 and 30 feature variables differ only by 2 in block order width. But, since they differ by 11 in the largest block, we notice a large jump in the compilation time for these two experiments (two orders of magnitude). On the other hand, the experiment with 30 and 33 feature variables also differ by 2 in block order width. Since their largest blocks differ only by 2, the compilation times are comparable (factor of 2). The OBDD size and compilation time are also significantly affected by the threshold of the classifier. A heavily biased threshold can lead to a very small OBDD and a short compilation time, while a balanced threshold generally leads to larger OBDDs.

Table 5.2: Network tcc4e has 98 nodes and width 10. Network emdec6g has 168 nodes and width 7. We use t27 as the class node for tcc4e, and x29 as the class node for emdec6g.

network	# feature variables	block order width	largest / # blocks	OBDD size	compile time (s)
tcc4e	21	15	11 / 7	167	4
tcc4e	26	19	14 / 8	930	11
tcc4e	30	30	20 / 6	3,057	1,873
tcc4e	37	37	25 / 8	10,442	39,705
tcc4e	38	38	26 / 8	22,508	91,332
emdec6g	24	24	7 / 13	115	6
emdec6g	27	27	10 / 10	122	11
emdec6g	30	29	21 / 7	4,154	2,487
emdec6g	33	31	22 / 8	3,855	5,308

5.2 Bayesian Network Classifier Case Study

We illustrate the utility of our compilation algorithm by showing how the resulting OBDDs can be used to explain and verify a given classifier. We consider two networks from the literature: win95pts and Andes; see Table 5.3b. We treat each network as a set of classifiers, taking each root node as a class variable. We treat each leaf node as a feature variable, and use a threshold of $\frac{1}{2}$.

We compile an OBDD for each classifier and then explain their decisions using two types of explanations shown in Chapter 4: minimum cardinality (MC) explanations and prime implicant (PI) explanations [SCD18b].

The win95pts network is used to diagnose why a printing job has failed [BH96]. It has 76 binary variables, 16 of which are leaves which we take as the feature variables of the classifier. One of its root nodes PtrOffline (PO) represents a failure mode (the printer is offline), and has two states: Online (0) and Offline (1). An instance is classified positively if the the probability of being Offline is $\geq \frac{1}{2}$. We first consider a positively classified instance (indicating a printing failure) that sets 7 of 16 feature variables as 1. The unique MC-explanation for this decision consists of a *single* feature variable set to 1: the printer icon

is grayed out (PrtIcon is GrayedOut (1)). That is, observing this one symptom positively is sufficient for a positive decision (printer is offline), even if all of the other 6 feature variables were observed as 0 instead of 1. A technician using such a classifier for decision support can troubleshoot a printer failure using this symptom, as it is the most pertinent among all positively observed feature variables.

Consider the shortest PI-explanation of this positive instance, which consists of three feature variables: the printer icon is grayed out (PrtIcon is GrayedOut), the problem is repeatable (NotRepeat is No), and the graphics are not distorted (GraphicsDistorted is No). With just these three observations, the classifier will always decide that the printer is offline (PtrOffline is Offline), no matter how the other feature variables are observed. Such a guarantee can help users trust the classifier, especially if its behavior matches the users' intuition. Users can even enter their own partial observation of interest (say, PrtIcon is Normal and GraphicsDistorted is Yes) and check if the classifier is guaranteed to behave according to their expectations (say, decide that PtrOffline is Online) regardless of how the remaining feature variables are set.

Next, we consider the Andes network, which models students' problem-solving skills in physics [GCV98]. We consider the class node TryKinematics (TK), which has two states: false (0) and true (1). This class predicts whether a student has developed problem-solving skills in kinematics, and assesses the student positively if the probability of true is $\geq \frac{1}{2}$. This classifier has 24 binary feature variables. First, we verify whether the classifier is monotonic or not: it is indeed monotonic. Next, we consider a positively classified instance that observed 5 of these feature variables as 1. The MC-explanation tells us that 3 of these 5 feature variables are responsible for the decision: TryKinematicsForAccel, TryKinematicsForDuration, and TryKinematicsForDisplacement. That is, we can flip the other two feature variables to 0 and still maintain a positive classification. We can also efficiently test whether the classification of this instance is robust, given an OBDD of the classifier's decision function [SCD18a]. In our example, it only takes a single feature variable to be flipped (from 1 to 0) to flip the decision to negative.

5.3 Binarized Neural Network Experiments

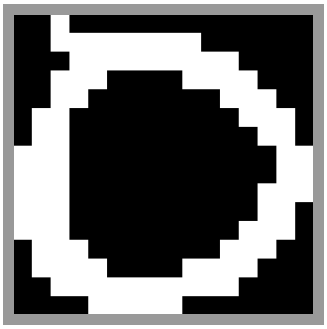
We present similar compilation experiments on Binarized Neural Networks. We consider the fully connected version as well as the sparser binary Convolutional Neural Network, based on the setup from [CSS19].

- Binarized Neural Networks (BNNs) [HCS16], as described in Chapter 2. In particular, we assumed a fully-connected multi-layer feedforward architecture;
- Convolutional Neural Networks (CNN) with binary inputs and outputs, where we used step activations instead of the more commonly used ReLU activations [CSS19].[1] Such a network corresponds to a Boolean circuit, although in general it will not be tractable. However, we can encode it as a CNF using the Tseitin transformation, and use the same algorithm described in Section 3.3 to learn its (tractable) OBDD.

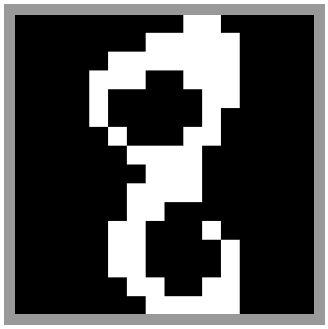
We considered the USPS digits dataset, and binarized the inputs to get 16×16 black and white images [Hul94]. We then trained our neural networks to distinguish between digit '0' images (false-class) and digit '8' images (true-class).

- We trained a BNN which achieved 94% accuracy using the training algorithm from [CHS16]. We first down-sampled the inputs to 8×8 images. The BNN thus had 64 input nodes; we further used 5 hidden nodes, and 2 output nodes. The network was encoded into a CNF with $10,664$ variables and $41,553$ clauses. Using `riss-coprocessor` to preprocess auxiliary variables, we compressed the CNF to $3,438$ variables and $23,254$ clauses [KKM15]. The original and compressed CNFs are equivalent after existentially quantifying out all variables except for the inputs and output, which is enough for the correctness of our algorithm.
- We trained a CNN which achieved 97% accuracy, using TensorFlow. The network used the original 16×16 images, and thus had 256 input nodes. We created two convolution

[1] We first train the network using sigmoid activations, and then at test time we replace the sigmoid activations with step activations, while keeping the learned weights.



(a) A digit 0 that is classified as '0'.



(b) A digit 8 that is classified as '8'.

(c) A smile which is classified as '8' by the BNN and '0' by the CNN.

Figure 5.1: Three 16×16 images: digit 0, digit 8, and a smile. For each image we compile around its r-neighborhood (the used 8×8 images are not shown).

layers, each with stride size 2. We first swept a 3×3 filter on the original 16×16 image (resulting in a 7×7 grid), followed by a second 2×2 filter (resulting in a 3×3 grid). These outputs were the inputs of a fully-connected layer with a single output. We encoded this network into a CNF with $10,547$ variables and $31,682$ clauses and using `riss-coprocessor`, we pre-processed the auxiliary variables to get a compressed CNF with $1,473$ variables and $11,638$ clauses [KKM15].

Experiments were done using a single Intel Xeon CPU E5-2670 processor. We used a time limit of one hour for each compilation. In general, we find that the fully-connected architecture of the BNN was more challenging to compile (hence, the reason for down-sampling the input images). In fact, when we trained a CNN on the 8×8 inputs, we were able to compile the network over the space of all images, not just for a fixed region around a given image.

For the BNN and CNN that we trained, we identified instances classified as digit '0' (Figure 5.1a), and compiled the neighborhood around it using Algorithms 4 and 5. We used the `riss` SAT solver for our experiments [KKM15]. Table 5.3a (BNN) and Table 5.3b (CNN) shows the compilation results for increasing values of r. We did the same for an instance that is classified as digit '8' (Figure 5.1b). We also compiled around the neighborhood of an image that is neither a '0' nor an '8' (a smile, Figure 5.1c). For experiments with small

input spaces, we manually verified the correctness of the OBDD through enumeration.

We make a few observations. For both the BNN and CNN, compiling larger regions around the smile was more challenging than compiling the regions around a digit. This is perhaps because there is less structure around an image that the network was not trained with. Next, while we scaled to a larger radius r using the BNN, the space of images was still much larger for the smaller radius that we compiled with the CNN, since the input images were much bigger (16×16 for the CNN versus 8×8 for the BNN).

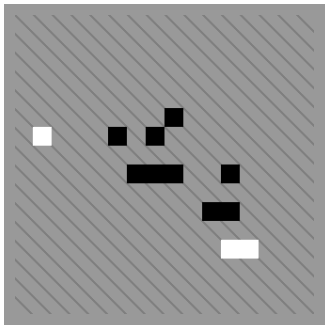
The bottleneck in our experiments is the average time for a SAT query, which is done once for each of the N equivalence queries, where N is the size of the OBDD (sizes are given in Tables 5.3a & 5.3b). As the OBDD grows, the membership queries become a bottleneck as well since the number of membership queries is quadratic on N.

5.4 Binarized Neural Network Case Study

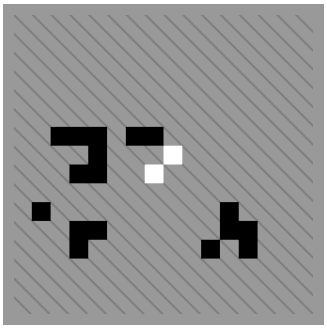
In this section we perform verification queries on the CNNs that we trained and compiled. Here, we first counted the number of counterexamples. Second, we performed prime-implicant queries, which give us a subset of pixels that render the remaining pixels irrelevant for the neural network classification [SCD18b], up to the region under consideration.

Consider the instance visualized in Figure 5.1a, classified as a '0' digit. For $r = 3$ in Table 5.3b, the reduced OBDD is just the constant false ($\bot$). This means that there were no counterexamples in this region, and that flipping any $r = 3$ pixels in our image will still produce another image classified as digit '0' (the false class). Recall that an image has 256 pixels in our example, so this classification holds for all of the $2,796,417$ possible inputs within a radius of 3 around our image in Figure 5.1a.

For $r = 6$, we get a reduced OBDD of size $1,469$. We first consider the number of counterexamples, which can be done in time linear in the size of the OBDD. In particular, we found that $20,413,779$ out of the $377,519,940,289$ images (0.005%) were classified incorrectly as the digit '8.' Hence, not only can we detect if a given instance is sensitive to perturbations



(a) 12 out of 256 pixels fixed from Figure 5.1a



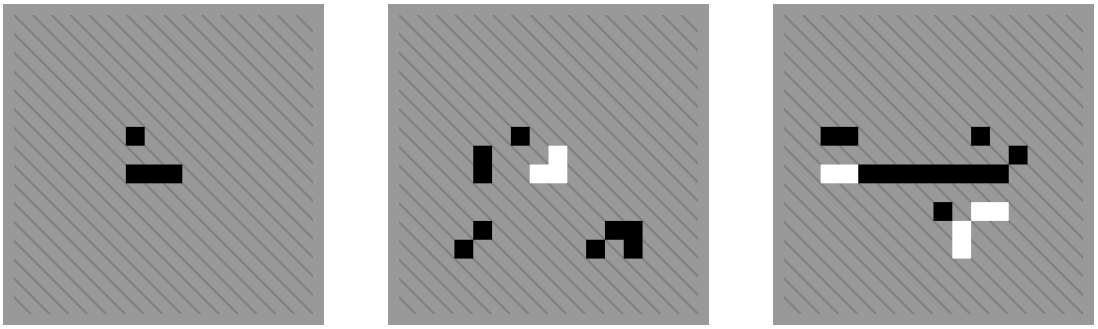
(b) 19 out of 256 pixels fixed from Figure 5.1b

Figure 5.2: Prime implicant results for $r = 6$ for the images in Figure 5.1a and 5.1b. The grey striped region represents 'don't care' pixels. If we fix the black/white pixels in Figure 5.2a, any completing image within a radius of 6 from Figure 5.1a must be classified as '0'. If we fix the black/white pixels in Figure 5.2b, any completing image within a radius of 6 from Figure 5.1b must be classified as '8'.

(flips of the pixels), we can also *quantify* how robust it is by counting *how many* ways the instance can be flipped. This is in contrast to approaches to neural network verification based on solving NP-complete problems, such as those relying (just) on SAT-solvers, where counting is in general out of scope (counting is a #P-complete problem).

Next, using the PI query, we identified a minimal set of pixels that guaranteed a correct classification, regardless of how the other pixels are set, within a radius of 6 of Figure 5.1a. The result is shown in Figure 5.2a. This PI query tells us about the behavior of our CNN classifier, in the space of images around Figure 5.1a. In particular, it suffices to have these particular white pixels near the border of the image, and these black pixels in the center of the image, for the classifier to fix its decision that the image is of a digit '0.'

We can ask the same queries for the instance visualized in Figure 5.1b and classified as digit '8.' For $r = 3$ in Table 5.3b (middle), the OBDD is just the constant true ($\top$), which means that flipping any 3 pixels of our instance will still produce another image classified correctly as digit '8' (the true class). For $r = 6$, we get an OBDD of size $3,345$. Using this OBDD, we found that $181,664,350$ out of the $377,519,940,289$ images (0.05%) are



(a) 4 out of 256 pixels fixed from Figure 5.1a

(b) 12 out of 256 pixels fixed from Figure 5.1b

(c) 19 out of 256 pixels fixed from Figure 5.1c

Figure 5.3: Prime implicant results for $r = 5$ for the images shown in Figure 5.1. The grey striped region represents 'don't care' pixels. If we fix the black/white pixels in Figure 5.3a, any completing image within a radius of 5 from Figure 5.1a must be classified as '0'. If we fix the black/white pixels in Figure 5.3b, any completing image within a radius of 5 from Figure 5.1b must be classified as '8'. If we fix the black/white pixels in Figure 5.3c, any completing image within a radius of 5 from Figure 5.1c must be classified as '8'.

classified incorrectly as the digit '0.' The PI query identified the minimal set of pixels in Figure 5.2b which guaranteed a correct classification regardless of how the remaining pixels are set (within a radius of 6 of Figure 5.1b).

For the "smile" image in Figure 5.1c, the compiled OBDD for the $(r = 5)$-neighborhood is larger than the corresponding OBDDs of the first two images (see each $r = 5$ row in Table 5.3b). As well, for $r = 5$, the PI query for the "smile" requires 19 out of the 256 pixels to be fixed in order to guarantee a classification, while the PI queries for the digit '0' and digit '8' only require 4 and 12 pixels respectively (Figure 5.3). This suggests that the behavior of the BNN is less structured in the region around the image of the "smile", possibly because it is unclear how the image should be classified.

Table 5.3: Compilation experiments

(a) Compilation of a BNN on 64 variables around the r-neighborhood of an image of a digit 0, digit 8, and a smile.

digit 0

r	input space	OBDD size	compile time (s)
1	65	0 ($\bot$)	< 1
2	2,081	0 ($\bot$)	< 1
3	43,745	0 ($\bot$)	< 1
4	679,121	0 ($\bot$)	< 1
5	8,303,633	0 ($\bot$)	2
6	83,278,001	509	403
7	704,494,193	2,202	2,166

digit 8

r	input space	OBDD size	compile time (s)
1	65	0 ($\top$)	< 1
2	2,081	0 ($\top$)	< 1
3	43,745	0 ($\top$)	< 1
4	679,121	0 ($\top$)	2
5	8,303,633	243	111
6	83,278,001	765	584
7	704,494,193	2,431	3,168

smile

r	input space	OBDD size	compile time (s)
1	65	0 ($\top$)	< 1
2	2,081	258	31
3	43,745	1,437	420
4	679,121	6,048	3,336

(b) Compilation of a CNN on 256 variables around the r-neighborhood of an image of a digit 0, a digit 8, and a smile.

digit 0

r	input space	OBDD size	compile time (s)
1	257	0 ($\bot$)	< 1
2	32,897	0 ($\bot$)	< 1
3	2,796,417	0 ($\bot$)	< 1
4	177,589,057	12	2
5	8,987,138,113	220	29
6	377,519,940,289	1,469	450

digit 8

r	input space	OBDD size	compile time (s)
1	257	0 ($\top$)	< 1
2	32,897	0 ($\top$)	< 1
3	2,796,417	0 ($\top$)	< 1
4	177,589,057	64	18
5	8,987,138,113	573	250
6	377,519,940,289	3,345	3,486

smile

r	input space	OBDD size	compile time (s)
1	257	0 ($\bot$)	< 1
2	32,897	8	< 1
3	2,796,417	93	7
4	177,589,057	622	138
5	8,987,138,113	3,269	1,661

CHAPTER 6

Conclusion

The increased adoption of machine learning classifiers has sparked a need for explaining their behavior. Safety-critical applications, as well as legal regulations, may even require explaining the decision of machine learning classifiers before deploying them. This has led to a growing line of research in developing techniques for generating good explanations.

In this thesis we contribute to this line of research by proposing an approach to explaining machine learning classifiers based on knowledge compilation. First, we capture the input/output behavior of binary classifiers as tractable decision diagrams. To do so, we provide algorithms for compiling Bayesian Network and Binarized Neural Network classifiers into Ordered Binary Decision Diagrams. Then, we efficiently generate explanations for the tractable decision diagrams, which in turn give explanations for the behavior of the original classifiers. We outline specific explanation queries, such as prime-implicant and robustness explanations, and describe how to compute them from a tractable decision diagram. Finally, we demonstrate these techniques on classifiers for printer diagnosis, educational assessment, and digit images. By leveraging the strengths of knowledge compilation, we are able to efficiently generate insightful and exact explanations on machine learning classifiers.

REFERENCES

[Ang87] Dana Angluin. "Learning regular sets from queries and counterexamples." *Information and computation*, **75**(2):87–106, 1987.

[BB03] Olivier Bailleux and Yacine Boufkhad. "Efficient CNF encoding of boolean cardinality constraints." In *International conference on principles and practice of constraint programming*, pp. 108–122. Springer, 2003.

[BH96] John S Breese and David Heckerman. "Decision-theoretic troubleshooting: A framework for repair and experiment." In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*, pp. 124–132. Morgan Kaufmann Publishers Inc., 1996.

[Bov16] Simone Bova. "SDDs Are Exponentially More Succinct than OBDDs." In *AAAI*, pp. 929–935, 2016.

[Bry86] R. E. Bryant. "Graph-based algorithms for Boolean function manipulation." *IEEE Transactions on Computers*, **C-35**:677–691, 1986.

[CD97] Marco Cadoli and Francesco M. Donini. "A Survey on Knowledge Compilation." *AI Commun.*, **10**(3-4):137–150, 1997.

[CD03] Hei Chan and Adnan Darwiche. "Reasoning About Bayesian Network Classifiers." In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 107–115, 2003.

[CD18] Arthur Choi and Adnan Darwiche. "SDD Advanced-User Manual Version 2.0.", 2018. http://reasoning.cs.ucla.edu/sdd/.

[CHS16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1.", 2016.

[CN17] Sara Castellanos and Steven Norton. "Inside DARPA's Push to Make Artificial Intelligence Explain Itself.", August 2017. The Wall Street Journal.

[CSS19] Arthur Choi, Weijia Shi, Andy Shih, and Adnan Darwiche. "Compiling Neural Networks into Tractable Boolean Circuits." In *AAAI Spring Symposium on Verification of Neural Networks (VNN19)*, 2019.

[Dar] Adnan Darwiche. "SamIam." http://reasoning.cs.ucla.edu/samiam/.

[Dar01] Adnan Darwiche. "Decomposable Negation Normal Form." *Journal of the ACM*, **48**(4):608–647, 2001.

[Dar09] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[Dar11] Adnan Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases." In *Proceedings of IJCAI*, pp. 819–826, 2011.

[DM02] Adnan Darwiche and Pierre Marquis. "A knowledge compilation map." *JAIR*, **17**:229–264, 2002.

[EDF17] Ethan R. Elenberg, Alexandros G. Dimakis, Moran Feldman, and Amin Karbasi. "Streaming Weak Submodularity: Interpreting Neural Networks on the Fly." In *Advances in Neural Information Processing Systems 30 (NIPS)*, pp. 4047–4057, 2017.

[FGG97] Nir Friedman, Dan Geiger, and Moisés Goldszmidt. "Bayesian Network Classifiers." *Machine Learning*, **29**(2-3):131–163, 1997.

[GBF04] Linda C. van der Gaag, Hans L. Bodlaender, and A. J. Feelders. "Monotonicity in Bayesian Networks." In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence (UAI)*, pp. 569–576, 2004.

[GCV98] Abigail S Gertner, Cristina Conati, and Kurt VanLehn. "Procedural help in Andes: Generating hints using a Bayesian network student model." *AAAI/IAAI*, **1998**:106–11, 1998.

[GF17] Bryce Goodman and Seth Flaxman. "European Union regulations on algorithmic decision-making and a right to explanation." *AI Magazine*, **38**(3):50–57, 2017.

[HCS16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks." In *Advances in Neural Information Processing Systems (NIPS)*, pp. 4107–4115, 2016.

[HI02] Takashi Horiyama and Toshihide Ibaraki. "Ordered binary decision diagrams as knowledge-bases." *Artificial Intelligence (AIJ)*, **136**(2):189–213, 2002.

[Hul94] Jonathan J. Hull. "A database for handwritten text recognition research." *IEEE Transactions on pattern analysis and machine intelligence*, **16**(5):550–554, 1994.

[KKM15] Lucas Kahlert, Franziska Krüger, Norbert Manthey, and Aaron Stephan. "Riss solver framework v5. 05.", 2015.

[KV94] Michael Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[LL17] Scott M. Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30 (NIPS)*, pp. 4768–4777, 2017.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications*. Springer, 1998.

[Nak05] Atsuyoshi Nakamura. "An efficient query learning algorithm for ordered binary decision diagrams." *Information and Computation*, **201**(2):178–198, 2005.

[NKR18] Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying Properties of Binarized Deep Neural Networks." In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier." In *Knowledge Discovery and Data Mining (KDD)*, 2016.

[RSG18] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Anchors: High-Precision Model-Agnostic Explanations." In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[SCD18a] Andy Shih, Arthur Choi, and Adnan Darwiche. "Formal Verification of Bayesian Network Classifiers." In *Proceedings of the 9th International Conference on Probabilistic Graphical Models (PGM)*, 2018.

[SCD18b] Andy Shih, Arthur Choi, and Adnan Darwiche. "A Symbolic Approach to Explaining Bayesian Network Classifiers." In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[SCD19] Andy Shih, Arthur Choi, and Adnan Darwiche. "Compiling Bayesian Networks into Decision Graphs." In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[SDC19] Andy Shih, Adnan Darwiche, and Arthur Choi. "Verifying Binarized Neural Networks by Angluin-Style Learning." In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2019.

[SK96] Bart Selman and Henry A. Kautz. "Knowledge Compilation and Theory Approximation." *J. ACM*, **43**(2):193–224, 1996.

[Tse68] Grigori Tseitin. "On the complexity of derivation in propositional calculus." *Studies in constructive mathematics and mathematical logic*, pp. 115–125, 1968.

[Weg00] Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.