Verifying Binarized Neural Networks by Angluin-Style Learning

Andy Shih, Adnan Darwiche, and Arthur Choi

Computer Science Department
University of California, Los Angeles
{andyshih,darwiche,aychoi}@cs.ucla.edu

Abstract. We consider the problem of verifying the behavior of binarized neural networks on some input region. We propose an Angluin-style learning algorithm to compile a neural network on a given region into an Ordered Binary Decision Diagram (OBDD), using a SAT solver as an equivalence oracle. The OBDD allows us to efficiently answer a range of verification queries, including counting, computing the probability of counterexamples, and identifying common characteristics of counterexamples. We also present experimental results on verifying binarized neural networks that recognize images of handwritten digits.

Keywords: Verification · Neural Networks · Decision Diagrams

1 Introduction

Neural networks are used for a wide array of tasks, including speech recognition, image classification, and language translation. They also power safety-critical applications, such as autonomous driving, where humans need to understand and formally verify the behavior of underlying neural networks. While recent advancements have improved the performance and scale of neural networks, there are not enough methods for providing formal guarantees about their behavior. In addition, the intricate structure of a neural network makes it impractical to reason about their behavior manually. This has sparked a recent line of research that aims to automatically verify neural network properties [6, 20, 26, 28, 35, 36].

We propose in this paper an approach for verifying the properties of neural networks, which is based on knowledge compilation [5, 11, 12, 29]. Our approach applies to the class of neural networks with discrete inputs and output, but we will highlight the special case of Binarized Neural Networks (BNNs) [14], which have binary weights and activations at runtime, leading to space and computational efficiencies. BNNs have been shown to achieve comparable performance on some standard datasets, compared to more traditional networks using floating-point precision [14].

One particular property of BNNs that has been studied is robustness [23]. Users of a BNN can pinpoint a particular input instance $\mathbf{x}$ and ask for guarantees on the behavior of the BNN for other inputs in the neighborhood of $\mathbf{x}$, which we call an *input region*, denoted by $S_{\mathbf{x}}$. This has practical applications, e.g.,

for image classification, where users expect an image of, say, a dog to remain classified as a dog if only a few pixels are modified. Since the number of ways to tweak an image is exponential in the number of modified pixels, it is impractical to perform the verification by enumeration.

A method was recently proposed for detecting counterexamples in any input region $S_{\mathbf{x}}$ encode-able as a CNF [26]. Our proposed approach pushes this direction further by harnessing techniques from knowledge compilation, allowing one to also *reason* about counterexamples. For example, we can efficiently count the counterexamples in $S_{\mathbf{x}}$, compute their probability, enumerate a subset of them, and identify their common characteristics. Another useful query supported by our approach, the *prime-implicant query,* returns a subset of inputs that, if fixed, will guarantee that the neural network output will stick even if we vary the unfixed inputs [16, 31].

Using the example of image classification, our new techniques allow us to perform reasoning on all images that are some pixels away from some target image, say, of a dog. Whereas previous methods only tell us that it is possible to classify another image in the neighborhood of the dog image as a cat, our new method can determine how many neighborhood images are classified as cats and identify key characteristics that are shared among all such images. Moreover, the prime-implicant query identifies a minimal set of pixels in the dog image that guarantees a correct classification even if we modify some of the unfixed pixels.

To reason about BNNs, we compile them into a tractable representation and then apply verification queries to the compiled representation. The compilation is done once per input region and, if successful, allows one to efficiently answer a range of queries that are otherwise NP-hard [30].

We now give an overview of our compilation algorithm. We compile BNNs into Ordered Binary Decision Diagrams (OBDDs), which are decision graphs that are tractable for many queries and transformations due to an enforced variable ordering [4, 12, 24, 33]. 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 as an OBDD, we leverage an Angluin-style algorithm for learning the OBDD representation of B_S using standard membership and equivalence queries [1, 25]. 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 $D(\mathbf{x}) = B(\mathbf{x}) : \forall \mathbf{x} \in S$, a notion related to the `Constrain` operator on OBDDs [24]. We then verify properties of BNN B by performing efficient verification queries on OBDD D.

Compared to the two main compilation paradigms of *bottom-up* and *top-down* compilation,[1] the Angluin-style learning algorithm is more similar to top-down

[1] Bottom-up compilation constructs constants and literals of a knowledge base and then composes them together using the *Apply* operation [11]. Top-down compilation recursively conditions the knowledge base and then combines the recursive compilations to obtain the final compilation [13, 27].

approaches, in that it never creates unnecessary nodes [11]. The main feature that distinguishes the Angluin-style learning algorithm from top-down approaches is the support of incremental and anytime compilation. The Angluin-style learning algorithm can slowly increase the region of interest, so that 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 later if we decide the initial region is too small.

This paper is structured as follows. Section 2 provides an introduction to BNNs and OBDDs. Section 3 describes the encodings of BNNs and OBDDs into CNF. Section 4 goes over the Angluin-style learning algorithm, which is used by our compilation algorithm in Section 5. We report experiments on the efficiency of our compilation algorithm in Section 6, followed by a case study in Section 7. We finally discuss related work in Section 8 and conclude in Section 9.

2 Background

In this section, we describe Binarized Neural Networks and Ordered Binary Decision Diagrams in more detail.

2.1 Binarized Neural Networks

A Binarized Neural Network is a feed-forward neural network where the weights and activations are binarized using $\{-1, 1\}$. A BNN 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 $\mathbf{a}$ (weights) and b (bias). Given an input $\mathbf{x}$, this layer returns $\langle \mathbf{a}, \mathbf{x} \rangle + b$.
- 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. [26]. For convenience we consider a BNN with outputs 0 or 1.

2.2 Ordered Binary Decision Diagrams

An *Ordered Binary Decision Diagram* (OBDD) is a *tractable* representation of a Boolean function over variables $\mathbf{X} = X_1, \ldots, X_n$ [4, 24, 33]. An OBDD is a rooted, directed acyclic graph with two sinks called the 1-sink and 0-sink. Every node (except the sinks) in the OBDD is labeled with a variable X_i and has two labeled outgoing edges: the 1-edge and the 0-edge. The labeling of the OBDD nodes respects some global ordering of the variables $\mathbf{X}$: if there is an edge from

(a) BNN with two outputs $\{0, 1\}$. The output with higher activation is the classification.

(b) OBDD with sinks $\{0, 1\}$.

Fig. 1: A BNN and its corresponding OBDD on four inputs. The two representations compute the same function.

a node labeled X_i to a node labeled X_j, then X_i must come before X_j in the global ordering. To evaluate the OBDD on an instance $\mathbf{x}$, start at the root node of the OBDD. Let x_i be the value of variable X_i of the current node. Repeatedly follow the x_i-edge of the current node, until a sink node is reached. Reaching the 1-sink means $\mathbf{x}$ is evaluated to 1 and reaching the 0-sink means $\mathbf{x}$ is evaluated to 0 by the OBDD. Hence, an OBDD can be viewed as representing a function $f(\mathbf{X})$ that maps instances $\mathbf{x}$ into $\{0, 1\}$.

Consider the BNN in Figure 1a, which classifies a movie as a box-office success or not. It has four binary inputs: A (*Adapted Screenplay*), G (*Great Cinematography*), F (*Famous Cast*), and M (*Marketing*). The parameters of the BNN are not shown, but it computes the truth table as shown in Table 1. The OBDD in Figure 1b also computes the truth table in Table 1, so we can verify properties of the BNN by performing verification on the OBDD. We can examine, for example, a movie that is an adapted screenplay, has great cinematography, a famous cast, heavy marketing, and is classified as being a box office success. This movie corresponds to input $\{A = 1, G = 1, F = 1, M = 1\}$ and a classification of 1. Using the OBDD in Figure 1b we can deduce, in time linear in the size of the OBDD, that the movie could have had poor cinematography and low marketing, and would still be classified as being a box office success. In fact, the partial input $\{A = 1, F = 1\}$ completely determines that the movie will be classified as being successful, regardless of how the remaining input is set. This is an example of the many types of efficient verification queries that can be done on an OBDD [31].

3 CNF Encodings

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

Table 1: The Boolean function on the 16 possible inputs computed by the BNN and OBDD in Figure 1.

	A	G	F	M	$f(\mathbf{x})$		A	G	F	M	$f(\mathbf{x})$
1	-	-	-	-	-	9	+	-	-	-	-
2	-	-	-	+	-	10	+	-	-	+	-
3	-	-	+	-	-	11	+	-	+	-	+
4	-	-	+	+	+	12	+	-	+	+	+
5	-	+	-	-	-	13	+	+	-	-	-
6	-	+	-	+	-	14	+	+	-	+	-
7	-	+	+	-	+	15	+	+	+	-	+
8	-	+	+	+	+	16	+	+	+	+	+

3.1 BNN to CNF

We use the conversion given by Narodytska et al. [26]. 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 $\mathbf{a}$ (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 output function $h(\mathbf{x})$ of a neuron on an input instance $\mathbf{x}$ [26].

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

Since the weights $\mathbf{a}$ and input $\mathbf{x}$ 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 [26]. 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.

3.2 OBDD to CNF

We convert an OBDD into a CNF using the well-known Tseitin transformation [32], 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:

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

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 $\mathbf{X}$ be the CNF variables corresponding to the BNN inputs and O be the variable corresponding to its output. Then $\alpha \wedge \mathbf{x} \wedge O$ will be satisfiable iff the BNN outputs 1 under input $\mathbf{x}$. Similarly, $\alpha \wedge \mathbf{x} \wedge \neg O$ will be satisfiable iff the BNN outputs 0 under input $\mathbf{x}$. Now let $\mathbf{X}$ be the CNF variables corresponding to the OBDD variables and R be the variable we introduced for the OBDD root. Then $\beta \wedge \mathbf{x} \wedge R$ will be satisfiable iff the OBDD outputs 1 under input $\mathbf{x}$ and $\beta \wedge \mathbf{x} \wedge \neg R$ will be satisfiable iff the OBDD outputs 0 under input $\mathbf{x}$.

When the BNN and the OBDD share the same inputs $\mathbf{x}$, we can check for their *inequivalence* with the formula $\phi = \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R)$ [26]. Then, ϕ is satisfiable iff there is some instantiation of $\mathbf{x}$ such that $(O \wedge \neg R) \vee (\neg O \wedge R)$ (i.e. BNN and OBDD disagree).

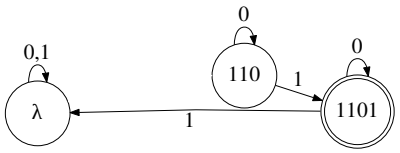
4 Angluin-Style Exact Learning of Finite Automaton

In this section we describe Angluin's algorithm for learning Deterministic Finite Automata (DFA) [1]. The DFA learning algorithm has an adaptation for learning OBDDs [25], 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 [33]) is also a DFA (but a DFA is not necessarily an OBDD).

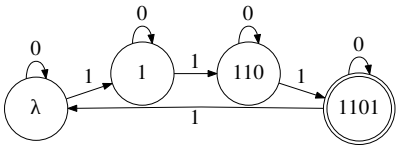
We roughly summarize the exposition on the topic of learning DFAs from the textbook by Kearns and Vazirani [21]. The learning algorithm falls under the category of *active* learning where the algorithm can learn through experimentation, as opposed to *passive* learning where the algorithm has no control 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 $\mathbf{x}$ and the oracle returns the value of $f(\mathbf{x})$.
- 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 $\mathbf{x}$ for which $h(\mathbf{x}) \neq f(\mathbf{x})$.

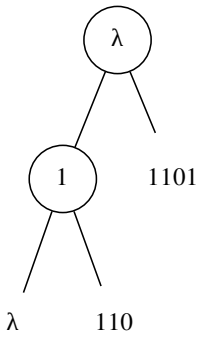
The main idea of the algorithm is as follows. Let S be the set of states of a minimal DFA we want to learn. Recall that each state represents a distinct equivalence class of input strings. At all times we keep a hypothesis DFA whose states $S^\star$ represent a partition of S. We iteratively refine the partition by splitting



(a) Hypothesis DFA h



(b) Updated DFA



(c) Binary classification tree of h

Fig. 2: Learning the finite automaton for the 3 mod 4 counter. Using the counterexample `1101`, we modify the hypothesis DFA into the updated DFA.

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 2a and its binary classification tree in Figure 2c. 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 2b, which computes f correctly.

The automaton learning algorithm was adapted into an OBDD learning algorithm by Nakamura [25]. 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.

Algorithm 1 `CompileBNN`$(B, \mathbf{X}, S)$

input: A Binarized Neural Network B with input variables $\mathbf{X}$, 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`$(B, \mathbf{X})$
2: $D \leftarrow$ initial hypothesis OBDD
3: $\beta, R \leftarrow$ `OBDDToCNF`$(D, \mathbf{X})$
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 **s do**
6: $\quad \mathbf{x} \leftarrow$ projection of **s** on $\mathbf{X}$
7: $\quad D \leftarrow$ `UpdateHypothesis`$(D, \mathbf{x})$
8: $\quad \beta, R \leftarrow$ `OBDDToCNF`$(D, \mathbf{X})$
9: $\quad \phi \leftarrow \alpha \wedge \beta \wedge (O \vee R) \wedge (\neg O \vee \neg R) \wedge S$
10: **return** D

5 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. $D(\mathbf{x}) = B(\mathbf{x}) : \forall \mathbf{x} \in S$). We require region S to be encoded as a CNF.

Algorithm 1 implements our proposal. The subroutines `BNNToCNF` and `OBDDToCNF` perform the encodings described in Section 3. We encode the BNN B as a CNF α with output variable O. Then, we start the OBDD learning algorithm as described in Section 4 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 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 $\mathbf{x}$. The subroutine `UpdateHypothesis` then edits our hypothesis OBDD using counterexample $\mathbf{x}$. 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 2 we propose the construction of an input region that captures all instances in the neighborhood of some instance $\mathbf{x}$ on n variables. More specifically, Algorithm 2 takes in an instance $\mathbf{x}$, a radius r, and outputs a CNF S on variables $X_1, \ldots, X_n$. An instance $\mathbf{x}^\star$ is a satisfying assignment for S iff the Hamming distance between $\mathbf{x}$ and $\mathbf{x}^\star$ is no greater than r. This becomes a cardinality constraint, which can be encoded in many ways [2]. 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

Algorithm 2 `r-RadiusDomain`$(\mathbf{x}, r)$

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

output: A CNF that encodes all instances $\mathbf{x}^\star$ such that $h(\mathbf{x}, \mathbf{x}^\star) \leq r$, 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`$(d_{n,r}, \mathbf{X})$

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 1, 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 features. The time and space complexity of Algorithm 2 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 1 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.

6 Experiments

In this section we present experiments on two types of neural networks:

- binarized neural networks (BNNs) [14], as described in Section 2. In particular, we assumed a fully-connected multi-layer feedforward architecture;
- convolutional neural networks (CNN), where we simply used step activations instead of the more commonly used ReLU activations [7].[2] In such a network, if the network inputs are binary, then the inputs and outputs of all neurons are binary (note that we do not use max-pooling in our experiments). Such a network corresponds to a Boolean circuit, although in general it will not

[2] 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.

be tractable. However, we can encode it as a CNF using the Tseitin transformation, and use the same algorithm described in Section 5 to learn its (tractable) OBDD.

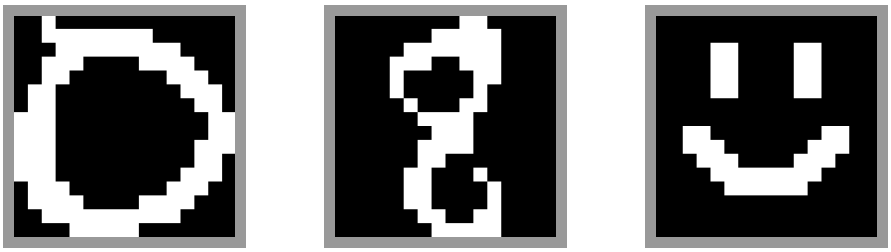
We considered the USPS digits dataset, and binarized the inputs to get 16×16 black and white images [15]. We then trained our neural networks to distinguish between digit '0' images (false-class) and digit '8' images (true-class). We also tested on other pairs of digits, which gave similar results.

- We trained a BNN which achieved 94% accuracy using the training algorithm from [9]. We down-sampled the inputs to 8×8 images to get 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 pre-process auxiliary variables, we compressed the CNF to $3,438$ variables and $23,254$ clauses [19]. The original and compressed CNFs are equivalent after existentially quantifying out all variables except for the inputs/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 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 [19].

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 full network (i.e., over the space of all images, and 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 3a), and compiled the neighborhood around it using Algorithms 1 and 2. The variable order we used for the OBDD is the natural row-by-row left-to-right ordering of the pixels in the images. We used the `riss` SAT solver for our experiments [19]. Table 2 (BNN) and Table 3 (CNN) shows the compilation results for increasing values of r. We did the same for an instance that is classified as digit '8' (Figure 3b). We also compiled around the neighborhood of an image that is neither a '0' nor an '8' (a smile, Figure 3c). 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



(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.

Fig. 3: 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).

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, i.e., number of nodes, of the OBDD (sizes are given in Tables 2 & 3). As the OBDD grows, the membership queries become a bottleneck as well since the number of membership queries is quadratic on N.

7 Case Study

In this section we perform verification queries on the convolutional neural networks (CNNs) that we trained and compiled in Section 6. First, we counted the number of counterexamples. Second, we performed prime-implicant queries (PI queries for short), which give a subset of pixels that render the remaining pixels irrelevant for the classification [31], up to the region under consideration.[3]

Consider the instance visualized in Figure 3a, classified as a '0' digit. For $r = 3$ in Table 3, 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 3a.

For $r = 6$, we get a reduced OBDD of size $1,469$, indicating the existence of counterexamples. We first consider the number of assignments satisfying this

[3] Ignatiev et al. [16] also subsequently proposed to use (prime) implicants to explain the decisions made by neural networks. While they computed implicants directly, we learned the OBDD of a neural network. Having an OBDD not only facilitates the computation of prime implicants, but it also allows model counting to be performed efficiently [12], which provides more powerful tools for analysis, as we shall show.

Table 2: 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

OBDD (i.e., 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 (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 3a. The result is shown in Figure 4a. This PI query tells us about the behavior of our CNN classifier, in the space of images around Figure 3a. 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.'

Table 3: 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

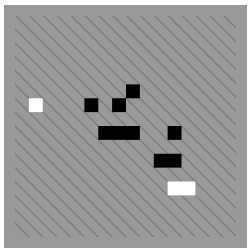
r	input space	OBDD size	compile time (s)
1	257	0 ($\top$)	< 1
2	32,897	0 ($\top$)	< 1
3	2,796,417	0 ($\top$)	< 1
4	177,589,057	64	18
5	8,987,138,113	573	250
6	377,519,940,289	3,345	3,486

smile

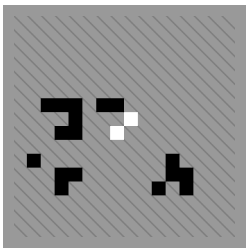
r	input space	OBDD size	compile time (s)
1	257	0 ($\bot$)	< 1
2	32,897	8	< 1
3	2,796,417	93	7
4	177,589,057	622	138
5	8,987,138,113	3,269	1,661

We can ask the same queries for the instance visualized in Figure 3b and classified as digit '8.' For $r = 3$ in Table 3 (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 classified incorrectly as the digit '0.' The PI query identified the minimal set of pixels in Figure 4b which guaranteed a correct classification regardless of how the remaining pixels are set (within a radius of 6 of Figure 3b).

For the "smile" image in Figure 3c, 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 3). 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). 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.



(a) 12 out of 256 pixels fixed from Figure 3a



(b) 19 out of 256 pixels fixed from Figure 3b

Fig. 4: Prime implicant results for $r = 6$ for the images in Figure 3a and 3b. The grey striped region represents 'don't care' pixels. If we fix the black/white pixels in Figure 4a, any completing image within a radius of 6 from Figure 3a must be classified as '0'. If we fix the black/white pixels in Figure 4b, any completing image within a radius of 6 from Figure 3b must be classified as '8'.

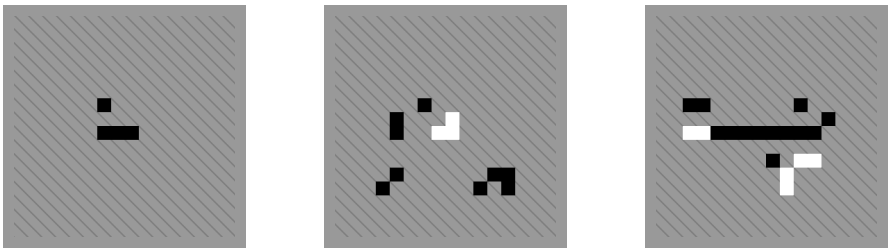
8 Related Work

The success of neural networks has led to the recent line of work on understanding and verifying their behaviors [6, 20, 26, 28]. These works use, for example, solvers for NP-complete problems such as Mixed-Integer Linear Programming (MILP), satisfiability (SAT), or satisfiability modulo theory (SMT). These systems seek to verify a particular property of a neural network, or otherwise provide a counter-example. We push this line of work further by allowing one to reason about the distribution or the characteristics of counterexamples, which is enabled by learning the OBDD of a given neural network. These richer queries allow us to better understand the neural network behavior beyond detecting the presence of counterexamples.

Choi et al. [7] also consider the compilation of neural networks into a tractable representation, and in particular, into a Sentential Decision Diagram (SDD) [8, 10].[4] They focus on a different class of neural networks and take the approach of reducing a neural network to a Boolean circuit, and then compiling the circuit into a tractable one using classical knowledge compilation techniques. While this approach allows a larger set of verification queries, it does not allow for local or incremental compilation, so it may be less scalable, other things being equal.

Finally, there is also recent work on learning finite state automata from recurrent neural networks (RNNs) [22, 34]. Weiss et al. [34] also use an Angluin-style approach for learning the finite state automaton of an RNN. More specifically, their approach is based on learning the finite state automaton of an iteratively-refined abstraction of an RNN's state space, and hence the final automaton learned is not necessarily equivalent to the original RNN. Koul et al. [22] trains an RNN and then quantizes the state space using an autoencoder. The result

[4] Note that SDDs are known to be exponentially more succinct than OBDDs [3].



(a) 4 out of 256 pixels fixed from Figure 3a

(b) 12 out of 256 pixels fixed from Figure 3b

(c) 19 out of 256 pixels fixed from Figure 3c

Fig. 5: Prime implicant results for $r = 5$ for the images shown in Figure 3. The grey striped region represents 'don't care' pixels. If we fix the black/white pixels in Figure 5a, any completing image within a radius of 5 from Figure 3a must be classified as '0'. If we fix the black/white pixels in Figure 5b, any completing image within a radius of 5 from Figure 3b must be classified as '8'. If we fix the black/white pixels in Figure 5c, any completing image within a radius of 5 from Figure 3c must be classified as '8'.

is a quantized network, whose corresponding state machine can be readily extracted. Angluin-style approaches, including ours, can be viewed as instances of *program synthesis*, where a program (a finite state automaton) is learned from a specification (a neural network). For more on formal synthesis, which lies at the increasingly important intersection of the fields of formal verification and machine learning, see, e.g., [17, 18].

9 Conclusion

We presented new techniques for verifying the behavior of a binarized neural network on some input region. We outlined an algorithm for compiling a BNN into an OBDD on any input region that can be encoded efficiently as a CNF. Our algorithm combines existing methods for CNF encodings with an Angluin-style algorithm for learning OBDDs. The compiled OBDD gives us access to a range of efficient verification queries and allows us to reason about counterexamples, such as computing their probability and identifying their common characteristics. In domains such as image classification, our approach can let users pinpoint a specific input image I, and then reason about images that are some pixels away from I but classified differently from I. We showed some experiments on a digits classifier, performing verification queries and scaling to 256 inputs.

Acknowledgments

This work has been partially supported by NSF grant #IIS-1514253, ONR grant #N00014-18-1-2561 and DARPA XAI grant #N66001-17-2-4032.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and computation **75**(2), 87–106 (1987)
2. Bailleux, O., Boufkhad, Y.: Efficient cnf encoding of boolean cardinality constraints. In: International conference on principles and practice of constraint programming. pp. 108–122. Springer (2003)
3. Bova, S.: SDDs are exponentially more succinct than OBDDs. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 929–935 (2016)
4. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**, 677–691 (1986)
5. Cadoli, M., Donini, F.M.: A survey on knowledge compilation. AI Commun. **10**(3-4), 137–150 (1997)
6. Cheng, C., Nührenberg, G., Huang, C., Ruess, H.: Verification of binarized neural networks via inter-neuron factoring (short paper). In: Proceedings of the 10th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE). pp. 279–290 (2018)
7. Choi, A., Shi, W., Shih, A., Darwiche, A.: Compiling neural networks into tractable Boolean circuits. In: AAAI Spring Symposium on Verification of Neural Networks (VNN19) (2019)
8. Choi, A., Xue, Y., Darwiche, A.: Same-decision probability: A confidence measure for threshold-based decisions. International Journal of Approximate Reasoning (IJAR) **53**(9), 1415–1428 (2012)
9. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1 (2016)
10. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: In Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI). pp. 819–826 (2011)
11. Darwiche, A.: Tractable knowledge representation formalisms. In: Tractability: Practical Approaches to Hard Problems, pp. 141–172. Cambridge University Press (2014)
12. Darwiche, A., Marquis, P.: A knowledge compilation map. JAIR **17**, 229–264 (2002)
13. Huang, J., Darwiche, A.: The language of search. Journal of Artificial Intelligence Research **29**, 191–219 (2007)
14. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Advances in Neural Information Processing Systems (NIPS). pp. 4107–4115 (2016)
15. Hull, J.J.: A database for handwritten text recognition research. IEEE Transactions on pattern analysis and machine intelligence **16**(5), 550–554 (1994)
16. Ignatiev, A., Narodytska, N., Marques-Silva, J.: Abduction-based explanations for machine learning models. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI) (2019)
17. Jha, S., Raman, V., Pinto, A., Sahai, T., Francis, M.: On learning sparse Boolean formulae for explaining AI decisions. In: Proceedings of the 9th NASA Formal Methods Symposium (NFM). pp. 99–114 (2017)
18. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Informatica **54**(7), 693–726 (2017)
19. Kahlert, L., Krüger, F., Manthey, N., Stephan, A.: Riss solver framework v5. 05 (2015)

20. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)
21. Kearns, M., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press (1994)
22. Koul, A., Fern, A., Greydanus, S.: Learning finite state representations of recurrent policy networks. In: Proceedings of the Seventh International Conference on Learning Representations (ICLR) (2019)
23. Leofante, F., Narodytska, N., Pulina, L., Tacchella, A.: Automated verification of neural networks: Advances, challenges and perspectives. CoRR **abs/1805.09938** (2018), `http://arxiv.org/abs/1805.09938`
24. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design: OBDD — Foundations and Applications. Springer (1998)
25. Nakamura, A.: An efficient query learning algorithm for ordered binary decision diagrams. Information and Computation **201**(2), 178–198 (2005)
26. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI) (2018)
27. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI). pp. 3141–3148 (2015)
28. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: International Conference on Computer Aided Verification (2010)
29. Selman, B., Kautz, H.A.: Knowledge compilation and theory approximation. J. ACM **43**(2), 193–224 (1996)
30. Shih, A., Choi, A., Darwiche, A.: Formal verification of Bayesian network classifiers. In: Proceedings of the 9th International Conference on Probabilistic Graphical Models (PGM) (2018)
31. Shih, A., Choi, A., Darwiche, A.: A symbolic approach to explaining Bayesian network classifiers. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI) (2018)
32. Tseitin, G.: On the complexity of derivation in propositional calculus. Studies in constructive mathematics and mathematical logic pp. 115–125 (1968)
33. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)
34. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: Proceedings of the 35th International Conference on Machine Learning (ICML). pp. 5244–5253 (2018)
35. Weng, T.W., Zhang, H., Chen, H., Song, Z., Hsieh, C.J., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: Proceedings of the Thirty-Fifth International Conference on Machine Learning (ICML) (2018)
36. Zhang, H., Zhang, P., Hsieh, C.J.: Recurjac : An efficient recursive algorithm for bounding jacobian matrix of general neural networks and its applications. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI) (2019)