Verification of Binarized Neural Networks via Inter-neuron Factoring
(Short Paper)

Chih-Hong Cheng(✉), Georg Nührenberg, Chung-Hao Huang, and Harald Ruess

fortiss - Landesforschungsinstitut des Freistaats Bayern, Munich, Germany
{cheng,nuehrenberg,huang,ruess}@fortiss.org

Abstract. Binarized Neural Networks (BNN) have recently been proposed as an energy-efficient alternative to more traditional learning networks. Here we study the problem of formally verifying BNNs by reducing it to a corresponding hardware verification problem. The main step in this reduction is based on factoring computations among neurons within a hidden layer of the BNN in order to make the BNN verification problem more scalable in practice. The main contributions of this paper include results on the NP-hardness and hardness of PTAS approximability of this essential optimization and factoring step, and we design polynomial-time search heuristics for generating approximate factoring solutions. With these techniques we are able to scale the verification problem to moderately-sized BNNs for embedded devices with thousands of neurons and inputs.

1 Introduction

Neural networks are used for perception and scene understanding [12,16,20] and also for control and decision making [4,9,14,23] in autonomous systems. Implementations of artificial neural networks, however, are very power-intensive due to complex floating point arithmetics. Binarized Neural Networks (BNNs), which are based on bit-level arithmetic, have therefore recently been proposed [6,11] as an attractive alternative to more traditional neural networks for resource-constrained embedded applications (e.g. based on FPGAs [1]). BNNs also demonstrate satisfactory performance on a number of standard benchmark datasets in image recognition including MNIST, CIFAR-10 and SVHN [6].

Here we study the verification problem for BNNs. Given a trained BNN and a specification of its intended input-output behavior we develop verification procedures for establishing that the given BNN indeed meets its intended specification for all possible inputs. For solving the verification problem of BNNs, we build on well-known methods from the hardware verification domain (Sect. 4). However, even with efficient neuron-to-circuit encoding we were not able to verify BNNs with thousands of inputs and hidden nodes as encountered in some of our embedded systems case studies.

R. Piskac and P. Rümmer (Eds.): VSTTE 2018, LNCS 11294, pp. 279–290, 2018.
https://doi.org/10.1007/978-3-030-03592-1_16

Table 1. An example of computing the output of a BNN neuron, using bipolar domain (up) and using 0/1 boolean variables (down).

index j	0 (bias node)	1	2	3	4
$x_j^{(l-1)}$	+1 (constant)	+1	−1	+1	+1
$w_{ji}^{(l)}$	−1 (bias)	+1	−1	−1	+1
$x_j^{(l-1)} w_{ji}^{(l)}$	−1	+1	+1	−1	+1
$\mathsf{im}_i^{(l)}$	+1				
$\mathsf{x}_i^{(l)}$	+1, as $\mathsf{im}_i^{(l)} \geq 0$				
index j	0 (bias node)	1	2	3	4
$x_j^{(l-1)}$	1	1	0	1	1
$w_{ji}^{(l)}$	0 (bias)	1	0	0	1
$x_j^{(l-1)} \overline{\oplus} w_{ji}^{(l)}$	0	1	1	0	1
# of 1's in $x_j^{(l-1)} \overline{\oplus} w_{ji}^{(l)}$	3				
$\mathsf{x}_i^{(l)}$	1, as $(3 \geq \lceil \frac{5}{2} \rceil)$				

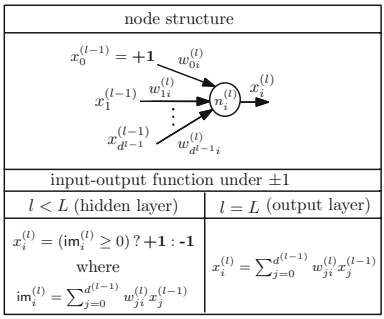

Fig. 1. Computation inside a neuron of a BNN, under bipolar domain ±1.

It turns out that one critical ingredient for efficient BNN verification is to factor computations among neurons in the same layer, which is possible due to the binary weights of inter-neuron connections in BNNs. Notice, however, that these factorings techniques are not directly applicable to floating-point based neural networks [5,7,10,15,19]. The key theorem regarding the hardness of finding optimal factoring as well as the hardness of inapproximability (Sect. 4.2) leads to the design of polynomial time search heuristics for generating factorings. These factorings substantially increase the scalability of formal verification via SAT solving (Sect. 5) to moderately-sized BNNs for embedded applications with thousands of neurons and inputs.

2 Related Work

There has been a flurry of recent results on formal verification of neural networks (e.g. [5,7,10,15,19]). These approaches usually target the formal verification of floating-point arithmetic neural networks (FPA-NNs). Huang et al. propose an (incomplete) search-based technique based on *satisfiability modulo theories* (SMT) solvers [8]. For FPA-NNs with ReLU activation functions, Katz et al. propose a modification of the Simplex algorithm which prefers fixing of binary variables [10]. This verification approach has been demonstrated on the verification of a collision avoidance system for UAVs. In our own previous work on neural network verification we establish maximum resilience bounds for FPA-NNs based on reductions to *mixed-integer linear programming* (MILP) problems [5]. The feasibility of this approach has been demonstrated, for example, by verifying a motion predictor in a highway overtaking scenario. The work of Ehlers [7] is based on sound abstractions, and approximates non-linear behavior in the activation functions. Scalability is the overarching challenge for these formal approaches to the verification of FPA-NNs. Case studies and experiments

reported in the literature are usually restricted to the verification of FPA-NNs with a couple of hundred neurons.

Around the time (Oct 9th, 2017) we first release of our work regarding formal verification of BNNs, Narodytska et al have also worked on the same problem [17]. Their work focuses on efficient encoding within a single neuron, while we focus on computational savings among neurons within the same layer. One can view our result and their results being complementary.

3 Preliminaries

Let $\mathbb{B}$ be the set of *bipolar binaries* ± 1, where $+1$ is interpreted as "true" and -1 as "false". A *Binarized Neural Network* (BNN) [6,11] consists of a sequence of layers labeled from $l = 0, 1, \ldots, L$, where 0 is the index of the *input layer*, L is the *output layer*, and all other layers are so-called *hidden layers*. Superscripts $^{(l)}$ are used to index layer l-specific variables. Elements of both inputs and outputs vectors of a BNN are of bipolar domain $\mathbb{B}$.

Layers l are comprised of *nodes* $n_i^{(l)}$ (so-called neurons), for $i = 0, 1, \ldots, d^{(l)}$, where $d^{(l)}$ is the *dimension* of the layer l. By convention, $n_0^{(l)}$ is a *bias node* and has constant bipolar output $+1$. Nodes $n_j^{(l-1)}$ of layer $l-1$ can be connected with nodes $n_i^{(l)}$ in layer l by a directed edge of *weight* $w_{ji}^{(l)} \in \mathbb{B}$. A layer is fully connected if every node (apart from the bias node) in the layer is connected to all neurons in the previous layer. Let $\boldsymbol{w}_i^{(l)}$ denote the array of all weights associated with neuron $n_i^{(l)}$. Notice that we consider all weights in a network to have fixed bipolar values.

Given an input to the network, computations are applied successively from neurons in layer 1 to L for generating outputs. Figure 1 illustrates the computations of a neuron in bipolar domain. Overall, the activation function is applied to the intermediately computed weighted sum. It outputs $+1$ if the weighted sum is greater or equal to 0; otherwise, output -1. For the output layer, the activation function is omitted. For $l = 1, \ldots, L$ let $x_i^{(l)}$ denote the output value of node $n_i^{(l)}$ and $\boldsymbol{x}^{(l)} \in \mathbb{B}^{|d^{(l)}|+1}$ denotes the array of all outputs from layer l, including the constant bias node; $\boldsymbol{x}^{(0)}$ refers to the input layer.

For a given BNN and a relation ϕ_{risk} specifying the undesired property between the bipolar input and output domains of the given BNN, the *BNN safety verification problem* asks if there exists an input $\boldsymbol{a}$ to the BNN such that the risk property $\phi_{risk}(\boldsymbol{a}, \boldsymbol{b})$ holds, where $\boldsymbol{b}$ is the output of the BNN for input $\boldsymbol{a}$.

It turns out that safety verification of BNN is no simpler than safety verification of floating point neural networks with ReLU activation function [10]. Nevertheless, compared to floating point neural networks, the simplicity of binarized weights allows an efficient translation into SAT problems, as can be seen in later sections.

Theorem 1. *The problem of BNN safety verification is NP-complete.*

Proof. Given a BNN and a relation ϕ_{risk} specifying the undesired property between the bipolar input and output domains of the given BNN, the *BNN safety verification problem* asks if there exists an input $\boldsymbol{a}$ to the BNN such that the risk property $\phi_{risk}(\boldsymbol{a}, \boldsymbol{b})$ holds, where $\boldsymbol{b}$ is the output of the BNN for input $\boldsymbol{a}$.

(NP) Given an input, compute the output and check if $\phi_{risk}(\boldsymbol{a}, \boldsymbol{b})$ holds can easily be done in time linear to the size of BNN and size of the property formula.

(NP-hardness) The NP-hardness proof is via a reduction from 3SAT to BNN safety verification. Consider variables $x_1, \ldots, x_m$, clauses $c_1, \ldots, c_d$ where for each clause c_j, it has three literals $l_{j_1}, l_{j_2}, l_{j_3}$. We build a single layer BNN with inputs to be $x_0 = +1$ (constant for bias), $x_1, \ldots, x_m, x_{m+1}$ (from CNF variables), connected to d neurons.

For neuron n_j^1, its weights and connection to previous layers is decided by clause c_j.

- If l_{j_1} is a positive literal x_i, then in BNN create a link from x_i to neuron n_j^1 with weight -1. If l_{j_1} is a negative literal x_i, then in BNN create a link from x_i to neuron n_j^1 with weight $+1$. Proceed analogously for l_{j_2} and l_{j_3}.
- Add an edge from x_{m+1} to n_j^1 with weight -1.
- Add an edge with weight -1 from x_0 to n_j^1 as bias term.

For example, consider the CNF having variables $x_1, \ldots, x_6$, then the translation of the clause $(x_3 \vee \neg x_5 \vee x_6)$ will create in BNN the weighted sum computation $(-x_3 + x_5 - x_6) - x_7 - 1$.

Assume that x_7 is constant $+1$, then if there exists any assignment to make the clause $(x_3 \vee \neg x_5 \vee x_6)$ true, then by interpreting the true assignment in CNF to be $+1$ in the BNN input and false assignment in CNF to be -1 in the BNN input, the weighted sum is at most -1, i.e., the output of the neuron is -1. Only when $x_3 = \mathsf{false}$, $x_5 = \mathsf{true}$ and $x_6 = \mathsf{false}$ (i.e., the assignment makes the clause false), then the weighed sum is $+1$, thereby setting output of the neuron to be $+1$.

Following the above exemplary observation, it is easy to derive that 3SAT formula is satisfiable *iff* in the generated BNN, there exists an input such that the risk property $\phi_{risk} := (x_{m+1} = +1 \rightarrow (\bigwedge_{i=1}^{n} x_i^{(1)} = -1))$ holds. It is done by interpreting the 3SAT variable assignment $x_i := \mathsf{true}$ in CNF to be assignment $+1$ for input x_i in the BNN, while interpreting $x_i := \mathsf{false}$ in 3SAT to be -1 for input x_i in the BNN. □

4 Verification of BNNs via Hardware Verification

The BNN verification problem is encoded by means of a *combinational miter* [3], which is a hardware circuit with only one Boolean output and the output should always be $\mathsf{0}$. The main step of this encoding is to replace the bipolar domain operation in the definition of BNNs with corresponding operations in the $\mathsf{0/1}$ Boolean domain.

We recall the encoding of the update function of an individual neuron of a BNN in bipolar domain (Eq. 1) by means of operations in the 0/1 Boolean domain [6,11]: (1) perform a bitwise XNOR ($\overline{\oplus}$) operation, (2) count the number of 1s, and (3) check if the sum is greater than or equal to the half of the number of inputs being connected. Table 1 illustrates the concept by providing the detailed computation for a neuron connected to five predecessor nodes. Therefore, the update function of a BNN neuron (in the fully connected layer) in the Boolean domain is as follows.

$$x_i^{(l)} = \mathsf{geq}_{\left\lceil \frac{|d^{(l-1)}|+1}{2} \right\rceil}(\mathsf{count1}(\boldsymbol{w}_i^{(l)} \,\overline{\oplus}\, \boldsymbol{x}^{(l-1)})), \tag{1}$$

where count1 simply counts the number of 1s in an array of Boolean variables, and $\mathsf{geq}_{\left\lceil \frac{|d^{(l-1)}|+1}{2} \right\rceil}(x)$ is 1 if $x \geq \left\lceil \frac{|d^{(l-1)}|+1}{2} \right\rceil$, and 0 otherwise. Notice that the value $\left\lceil \frac{|d^{(l-1)}|+1}{2} \right\rceil$ is constant for a given BNN. Here we omit details, but specifications in the bipolar domain can also be easily re-encoded in the Boolean domain.

4.1 From BNN to Hardware Verification

We are now ready for stating the basic decision procedure for solving BNN verification problems. This procedure first constructs a combinational miter for a BNN verification problem, followed by an encoding of the combinational miter into a corresponding propositional SAT problem. Here we rely on standard transformation techniques as implemented in logic synthesis tools such as ABC [3] or Yosys [24] for constructing SAT problems from miters. The decision procedure takes as input a BNN network description, an input-output specification ϕ_{risk} and can be summarized by the following workflow:

1. Transform all neurons of the given BNN into neuron-modules. All neuron-modules have identical structure, but only differ based on the associated weights and biases of the corresponding neurons.
2. Create a BNN-module by wiring the neuron-modules realizing the topological structure of the given BNN.
3. Create a property-module for the property ϕ_{risk}. Connect the inputs of this module with all the inputs and all the outputs of the BNN-module. The output of this module is true if the property is satisfied and false otherwise.
4. The combination of the BNN-module and the property-module is the miter.
5. Transform the miter into a propositional SAT formula.
6. Solve the SAT formula. If it is unsatisfiable then the BNN is safe w.r.t. ϕ_{risk}; if it is satisfiable then the BNN exhibits the risky behavior being specified in ϕ_{risk}.

4.2 Counting Optimization

The goal of the counting optimization is to speed up SAT-solving times by reusing redundant counting units in the circuit and, thus, reducing redundancies

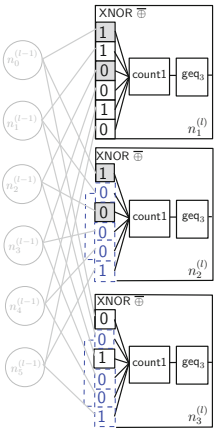

Fig. 2. One possible factoring to avoid redundant counting.

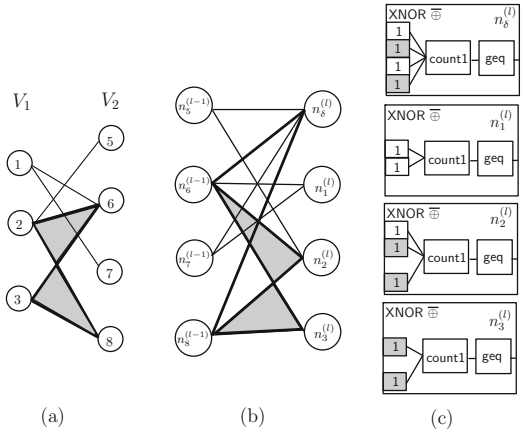

Fig. 3. From bipartite graph (a) to BNN where all weights are with value 1 (b), to optimal factoring (c).

in the SAT formula. This method involves the identification and factoring of redundant counting units, illustrated in Fig. 2, which highlights one possible factoring. The main idea is to exploit similarities among the weight vectors of neurons in the same layer, because the counting over a portion of the weight vector has the same result for all neurons that share it. The circuit size is reduced by using the factored counting unit in multiple neuron-modules. We define a factoring as follows:

Definition 1 (factoring and saving). *Consider the l-th layer of a BNN where $l > 0$. A* factoring $f = (I, J)$ *is a pair of two sets, where $I \subseteq \{1, \ldots, d^{(l)}\}$, $J \subseteq \{1, \ldots, d^{(l-1)}\}$, such that $|I| > 1$, and for all $i_1, i_2 \in I$, for all $j \in J$, we have $w^{(l)}_{ji_1} = w^{(l)}_{ji_2}$. Given a factoring $f = (I, J)$, define its* saving $\mathsf{sav}(f)$ *be $(|I| - 1) \cdot |J|$.*

Definition 2 (non-overlapping factorings). *Two factorings $f_1 = (I_1, J_1)$ and $f_2 = (I_2, J_2)$ are non-overlapping when the following condition folds: if $(i_1, j_1) \in f_1$ and $(i_2, j_2) \in f_2$, then either $i_1 \neq i_2$ or $j_1 \neq j_2$. In other words, weights associated with f_1 and f_2 do not overlap.*

Definition 3 (k-factoring optimization problem). *The k-factoring optimization problem searches for a set F of size k factorings $\{f_1, \ldots, f_k\}$, such that any two factorings are non-overlapping, and the total saving $\mathsf{sav}(f_1) + \cdots + \mathsf{sav}(f_k)$ is maximum.*

For the example in Fig. 2, there are two non-overlapping factorings $f_1 = (\{1, 2\}, \{0, 2\})$ and $f_2 = (\{2, 3\}, \{1, 3, 4, 5\})$. $\{f_1, f_2\}$ is also an optimal solution for the 2-factoring optimization problem, with the total saving being $(2 - 1) \cdot 2 + (2 - 1) \cdot 4 = 6$. Even finding one factoring f_1 which has the overall maximum

saving $\mathsf{sav}(f_1)$, is computationally hard. This NP-hardness result is established by a reduction from the NP-complete problem of finding maximum edge biclique in bipartite graphs [18].

Theorem 2 (Hardness of factoring optimization). *The k-factoring optimization problem, even when $k = 1$, is NP-hard.*

Proof. The proof proceeds by a polynomial reduction from the problem of finding maximum edge biclique in bipartite graphs(MEB) [18][1]. Given a bipartite graph G, this reduction is defined as follows.

1. For $v_{1\alpha}$, the α-th element of V_1, create a neuron $n_\alpha^{(l)}$.
2. Create an additional neuron $n_\delta^{(l)}$
3. For $v_{2\beta}$, the β-th element of V_2, create a neuron $n_\beta^{(l-1)}$.
   - Create weight $w_{\beta\delta}^{(l)} = 1$.
   - If $(v_{1\alpha}, v_{2\beta}) \in E$, then create $w_{\beta\alpha}^{(l)} = 1$.

This construction can clearly be performed in polynomial time. Figure 3 illustrates the construction process. It is not difficult to observe that G has a maximum edge size κ biclique $\{A; B\}$ iff the neural network at layer l has a factoring (I, J) whose saving equals $(|I| - 1) \cdot |J| = \kappa$. The gray area in Fig. 3-a shows the structure of maximum edge biclique $\{\{2, 3\}; \{6, 8\}\}$. For Fig. 3-c, the saving is $(|\{n_\delta^{(l)}, n_2^{(l)}, n_3^{(l)}\}| - 1) \cdot 2 = 4$, which is the same as the edge size of the biclique. □

Furthermore, even having an approximation algorithm for the k-factoring optimization problem is hard - there is no polynomial time approximation scheme (PTAS), unless NP-complete problems can be solved in randomized subexponential time. The proof follows an intuition that building a PTAS for 1-factoring can be used to build a PTAS for finding maximum complete bipartite subgraph which also has known inapproximability results [2].

Theorem 3. *Let $\epsilon > 0$ be an arbitrarily small constant. If there is a PTAS for the k-factoring optimization problem, even when $k = 1$, then there is a (probabilistic) algorithm that decides whether a given SAT instance of size n is satisfiable in time 2^{n^ϵ}.*

Proof. We will prove the Theorem by showing that a PTAS for the k-factoring optimization problem can be used to manufacture a PTAS for MEB. Then the result follows from the inapproximability of MEB assuming the exponential time hypothesis [2].

[1] Let $G = (V_1, V_2, E)$ be a bipartite graph with vertex set $V_1 \uplus V_2$ and edge set E connecting vertices in V_1 to vertices in V_2. A pair of two disjoint subsets $A \subset V_1$ and $B \subset V_2$ is called a *biclique* if $(a, b) \in E$ for all $a \in A$ and $b \in B$. Thus, the edges $\{(a, b)\}$ form a complete bipartite subgraph of G. A biclique $\{A; B\}$ clearly has $|A| \cdot |B|$ edges.

Assume that $\mathcal{A}$ is a ρ-approximation algorithm [2] for the k-factoring optimization problem. We formulate the following algorithm $\mathcal{B}$:
Input: MEB instance M (a bipartite graph $G = (V, E)$)
Output: a biclique in G

1. perform reduction of proof of Theorem 2 to obtain k-factoring instance $F :=$ reduce(M)
2. factoring $(I, J) := \mathcal{A}(F)$
3. return $(I \setminus \{n_\delta^{(l)}\}, J)$

Remark: step 3 is a small abuse of notation. It should return the original vertices corresponding to these neurons.

Now we prove that $\mathcal{B}$ is a ρ-approximation algorithm for MEB: Note that by our reduction two corresponding MEB and k-factoring instances M and F have the same optimal value, i.e., $\textsc{Opt}(M) = \textsc{Opt}(F)$.

In step 3 the algorithm returns $(I \setminus \{n_\delta^{(l)}\}, J)$. This is valid since we can assume w.l.o.g. that I returned by $\mathcal{A}$ contains $n_\delta^{(l)}$. This neuron is connected to all neurons from the previous layer by construction, so it can be added to any factoring. The following relation holds for the number of edges in the biclique returned by $\mathcal{B}$:

$$\begin{aligned} \|I \setminus \{n_\delta^{(l)}\}\| \cdot \|J\| &= (\|I\| - 1) \cdot \|J\| && \text{(2a)} \\ &\geq \rho \cdot \textsc{Opt}(F) && \text{(2b)} \\ &= \rho \cdot \textsc{Opt}(M) && \text{(2c)} \end{aligned}$$

The inequality in step (2b) holds by the assumption that $\mathcal{A}$ is a ρ-approximation algorithm for k-factoring and (2c) follows from the construction of our reduction. Equations (2) and the result of [2] imply Theorem 2. □

As finding an optimal factoring is computationally hard, we present a *polynomial time heuristic algorithm* (Algorithm 1) that finds factoring possibilities among neurons in layer l. The `main` function searches for an unused pair of neuron i and input j (line 3 and 5), considers a certain set of factorings determined by the subroutine `getFactoring` (line 6) where weight $w_{ji}^{(l)}$ is guaranteed to be used (as input parameter i, j), picks the factoring with greatest `sav`() (line 7) and then adds the factoring greedily and updates the set `used` (line 8).

The subroutine `getFactoring`() (lines 10–14) computes a factoring (I, J) guaranteeing that weight $w_{ji}^{(l)}$ is used. It starts by creating a set $\mathbb{I}$, where each element $I_{j'} \in \mathbb{I}$ is a set containing the indices of neurons whose j'-th weight matches the j'-th weight in neuron i (the condition $(w_{j'i'}^{(l)} = w_{j'i}^{(l)})$ in line 11). In the example in Fig. 4a, the computation generates Fig. 4b where $I_3 = \{1, 2, 3\}$ as $w_{31}^{(l)} = w_{32}^{(l)} = w_{33}^{(l)} = 0$. The intersection performed on line 12 guarantees that the set $I_{j'}$ is always a subset of I_j – as weight w_{ji} should be included, I_j already defines the maximum set of neurons where factoring can happen. E.g., I_3 changes from $\{1, 2, 3\}$ to $\{1, 2\}$ in Fig. 4c.

Algorithm 1. Finding factoring possibilities for BNN.

Data: BNN network description (cf Sect. 3)
Result: Set F of factorings, where any two factorings of F are non-overlapping.

1 **function** `main()`:
2 **let** used $:= \emptyset$ and $F := \emptyset$;
3 **foreach** neuron $n_i^{(l)}$ **do**
4 **let** f_i^{opt} := empty factoring;
5 **foreach** weight $w_{ji}^{(l)}$ where $(i,j) \notin$ used **do**
6 $f_{ij} =$ `getFactoring`$(i, j,$ used$)$;
7 **if** sav$(f_{ij}) >$ sav(f_i^{opt}) **then** $f_i^{opt} := f_{ij}$;
8 used $:=$ used $\cup \{(i,j) \mid (i,j) \in f_i^{opt}\}$; $F := F \cup \{f_i^{opt}\}$;
9 **return** F;
10 **function** `getFactoring`$(i,$ $j,$ used$)$:
11 **build** $\mathbb{I} := \{I_0, ..., I_{d^{(l-1)}}\}$ where $I_{j'} :=$
$\{i' \in \{0, ..., d^{(l)}\} \mid w_{j'i'}^{(l)} = w_{j'i}^{(l)} \wedge (i', j') \notin$ used$\}$;
12 **foreach** $I_m \in \mathbb{I}$ **do** $I_m := I_m \bigcap I_j$;
13 **build** $\mathbb{J} := \{J_0, \ldots, J_{j'}, \ldots, J_{d^{(l-1)}}\}$ where $J_{j'} :=$
$\{j'' \in \{0, ..., d^{(l-1)}\} \mid I_{j'} \subseteq I_{j''}\}$;
14 **return** $(I, J) := (I_{j^*}, J_{j^*})$ where $I_{j^*} \in \mathbb{I}, J_{j^*} \in \mathbb{J}$, and
$(|I_{j^*}| - 1) \cdot |J_{j^*}| = \max_{j' \in \{0,\ldots,d^{(l-1)}\}} (|I'_j| - 1) \cdot |J'_j|$;

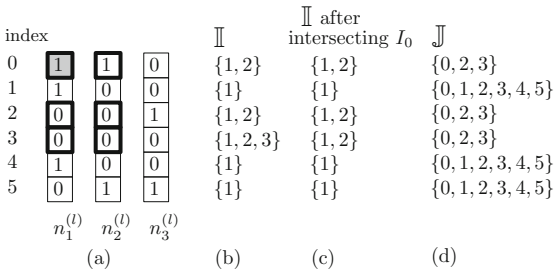

Fig. 4. Executing `getFactoring`$(1, 0, \emptyset)$, meaning that we consider a factoring which includes the top-left corner of (a). The returned factoring is highlighted in thick lines.

The algorithm then builds a set $\mathbb{J}$ of all the candidates for J. Each element $J_{j'}$ contains all the inputs j'' that would benefit from $I_{j'}$ being the final result I. Based on the observation mentioned above, $J_{j'}$ can be built through superset computation between elements of $\mathbb{I}$ (line 13, Fig. 4d). After we build $\mathbb{I}$ and $\mathbb{J}$, finally line 14 finds a pair of (I_{j^*}, J_{j^*}) where $I_{j^*} \in \mathbb{I}$, $J_{j^*} \in \mathbb{J}$ with the maximum saving $(|I_j^*|-1)\cdot|J_j^*|$. The maximum saving as produced in Fig. 4 equals $(|\{1,2\}|-1)\cdot|\{0,2,3\}| = 3$.

There are only polynomial operations in this algorithm such as nested for loops, superset checking and intersection which makes the heuristic algorithm polynomial. When one encounters a huge number of neurons and long weight

vectors, we further partition neurons and weights into smaller regions as input to Algorithm 1. By doing so, we find factoring possibilities for each weight segment of a neuron and the algorithm can be executed in parallel.

5 Evaluation and Outlook

We have created a verification tool, which first reads a BNN description based on the Intel Nervana Neon framework[2], generates a combinational miter in Verilog and calls `Yosys` [24] and `ABC` [3] for generating a CNF formula. No further optimization commands (e.g., `refactor`) are executed inside `ABC` to create smaller CNFs. Finally, `Cryptominisat5` [21] is used for solving SAT queries. The experiments are conducted in a Ubuntu 16.04 Google Cloud VM equipped with 18 cores and 250 GB RAM, with `Cryptominisat5` running with 16 threads. We use two different datasets, namely the MNIST dataset for digit recognition [13] and the German traffic sign dataset [22]. We binarize the gray scale data to ± 1 before actual training. For the traffic sign dataset, every pixel is quantized to 3 Boolean variables.

Table 2 summarizes the result of verification in terms of SAT solving time, with a timeout set to 90 min. The properties that we use here are characteristics of a BNN given by numerical constraints over outputs, such as "simultaneously classify an image as a priority road sign and as a stop sign with high confidence" (which clearly demonstrates a risk behavior). It turns out that factoring techniques are essential to enable better scalability, as it halves the verification times in most cases and enables us to solve some instances where the plain approach ran out of memory or timed out. However, we also observe that solvers like `Cryptominisat5` might get trapped in some very hard-to-prove properties. Regarding the instance in Table 2 where the result is unknown, we suspect that

Table 2. Verification results for each instance and comparing the execution times of the plain hardware verification approach and the optimized version using counting optimizations.

ID	# inputs	# neurons hidden layer	Properties being investigated	SAT/UNSAT	SAT solving time (normal)	SAT solving time (factored)
MNIST 1	784	3×100	$\text{out}_1 \geq 18 \wedge \text{out}_2 \geq 18$ ($\geq 18\%$)	SAT	2m 16.336 s	0m 53.545 s
MNIST 1	784	3×100	$\text{out}_1 \geq 30 \wedge \text{out}_2 \geq 30$ ($\geq 30\%$)	SAT	2m 20.318 s	0m 56.538 s
MNIST 1	784	3×100	$\text{out}_1 \geq 60 \wedge \text{out}_2 \geq 60$ ($\geq 60\%$)	SAT	timeout	10m 50.157 s
MNIST 1	784	3×100	$\text{out}_1 \geq 90 \wedge \text{out}_2 \geq 90$ ($\geq 90\%$)	UNSAT	2m 4.746 s	1m 0.419 s
Traffic 2	2352	3×500	$\text{out}_1 \geq 90 \wedge \text{out}_2 \geq 90$ ($\geq 18\%$)	SAT	10m 27.960 s	4m 9.363 s
Traffic 2	2352	3×500	$\text{out}_1 \geq 150 \wedge \text{out}_2 \geq 150$ ($\geq 30\%$)	SAT	10m 46.648 s	4m 51.507 s
Traffic 2	2352	3×500	$\text{out}_1 \geq 200 \wedge \text{out}_2 \geq 200$ ($\geq 40\%$)	SAT	10m 48.422 s	4m 19.296 s
Traffic 2	2352	3×500	$\text{out}_1 \geq 300 \wedge \text{out}_2 \geq 300$ ($\geq 60\%$)	unknown	timeout	timeout
Traffic 2	2352	3×500	$\text{out}_1 \geq 475 \wedge \text{out}_2 \geq 475$ ($\geq 95\%$)	UNSAT	31m 24.842 s	41m 9.407 s
Traffic 3	2352	3×1000	$\text{out}_1 \geq 120 \wedge \text{out}_2 \geq 120$ ($\geq 12\%$)	SAT	out-of-memory	9m 40.77 s
Traffic 3	2352	3×1000	$\text{out}_1 \geq 180 \wedge \text{out}_2 \geq 180$ ($\geq 18\%$)	SAT	out-of-memory	9m 43.70 s
Traffic 3	2352	3×1000	$\text{out}_1 \geq 300 \wedge \text{out}_2 \geq 300$ ($\geq 30\%$)	SAT	out-of-memory	9m 28.40 s
Traffic 3	2352	3×1000	$\text{out}_1 \geq 400 \wedge \text{out}_2 \geq 400$ ($\geq 40\%$)	SAT	out-of-memory	9m 34.95 s

[2] https://github.com/NervanaSystems/neon/tree/master/examples/binary.

the simultaneous confidence value of 60% for the two classes $\texttt{out}_1$ and $\texttt{out}_2$, is close to the value where the property flips from satisfiable to unsatisfiable. This makes SAT solving on such cases extremely difficult for solvers as the instances are close to the "border" between SAT and UNSAT instances.

In the future, we plan to directly synthesize propositional clauses without the support of third party tools such as `Yosys` in order to avoid extraneous transformations and repetitive work in the synthesis workflow. Similar optimizations of the current verification tool chain should result in substantial performance improvements.

Acknowledgments. We thank Dr. Ljubo Mercep from Mentor Graphics for indicating to us some recent results on quantized neural networks, Dr. Alan Mishchenko from UC Berkeley for his kind suggestions and support regarding `ABC`, and Hugo A. Andrade from Xilinx for exchanging the view of BNN.

References

1. Umuroglu, Y., et al.: FINN: a framework for fast, scalable binarized neural network arXiv preprint arXiv:1612.07119 (2017)
2. Ambühl, C., Mastrolilli, M., Svensson, O.: Inapproximability results for maximum edge biclique, minimum linear arrangement, and sparsest cut. SIAM J. Comput. **40**(2), 567–596 (2011)
3. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
4. Chen, C., Seff, A., Kornhauser, A., Xiao, J.: Deepdriving: learning affordance for direct perception in autonomous driving. In: ICCV, pp. 2722–2730 (2015)
5. Cheng, C.-H., Nührenberg, G., Ruess, H.: Maximum resilience of artificial neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 251–268. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_18
6. 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. arXiv preprint arXiv:1602.02830 (2016)
7. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
8. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
9. Huval, B., et al. An empirical evaluation of deep learning on highway driving. arXiv preprint arXiv:1504.01716 (2015)
10. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
11. Kim, M., Smaragdis, P.: Bitwise neural networks. arXiv preprint arXiv:1601.06071 (2016)

12. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS, pp. 1097–1105 (2012)
13. LeCun, Y.: The MNIST database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/
14. Lenz, D., Diehl, F., Le, M.T., Knoll, A.: Deep neural networks for Markovian interactive scene prediction in highway scenarios. In: Intelligent Vehicles Symposium IV. IEEE (2017)
15. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks. arXiv preprint arXiv:1706.07351 (2017)
16. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: CPVR, pp. 3431–3440. IEEE (2015)
17. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. arXiv preprint arXiv:1709.06662 (2014)
18. Peeters, R.: The maximum edge biclique problem is NP-complete. Discret. Appl. Math. **131**(3), 651–654 (2003)
19. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 243–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_24
20. Sermanet, P., Eigen, D., Zhang, X. , Mathieu, M., Fergus, R., LeCun, Y.: Overfeat: integrated recognition, localization and detection using convolutional networks. arXiv preprint arXiv:1312.6229 (2013)
21. Soos, M.: The Cryptominisat 5 set of solvers at sat competition 2016. In: Sat Competition 2016, p. 28 (2016)
22. Stallkamp, J., Schlipsing, M., Salmen, J., Igel, C.: The German traffic sign recognition benchmark: a multi-class classification competition. In: IEEE International Joint Conference on Neural Networks, pp. 1453–1460 (2011)
23. Sun, L., Peng, C., Zhan, W., Tomizuka, M.: A fast integrated planning and control framework for autonomous driving via imitation learning (2017). arXiv preprint arXiv:1707.02515
24. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)