Annales Mathematicae et Informaticae
53 (2021) pp. 183–200
DOI: https://doi.org/10.33039/ami.2021.03.007
URL: https://ami.uni-eszterhazy.hu

Portfolio solver for verifying Binarized Neural Networks

Gergely Kovásznai[a], Krisztián Gajdár[b], Nina Narodytska[c]

[a]Eszterházy Károly University, Eger, Hungary
kovasznai.gergely@uni-eszterhazy.hu

[b]Ericsson Hungary, Budapest, Hungary

[c]VMware Research, Palo Alto, USA

Submitted: November 21, 2020
Accepted: March 17, 2021
Published online: May 18, 2021

Abstract

Although deep learning is a very successful AI technology, many concerns have been raised about to what extent the decisions making process of deep neural networks can be trusted. Verifying of properties of neural networks such as adversarial robustness and network equivalence sheds light on the trustiness of such systems. We focus on an important family of deep neural networks, the Binarized Neural Networks (BNNs) that are useful in resource-constrained environments, like embedded devices. We introduce our portfolio solver that is able to encode BNN properties for SAT, SMT, and MIP solvers and run them in parallel, in a portfolio setting. In the paper we propose all the corresponding encodings of different types of BNN layers as well as BNN properties into SAT, SMT, cardinality constrains, and pseudo-Boolean constraints. Our experimental results demonstrate that our solver is capable of verifying adversarial robustness of medium-sized BNNs in reasonable time and seems to scale for larger BNNs. We also report on experiments on network equivalence with promising results.

Keywords: Artificial intelligence, neural network, adversarial robustness, formal method, verification, SAT, SMT, MIP

1. Introduction

Deep learning is a very successful AI technology that makes impact in a variety of practical applications ranging from vision to speech recognition and natural language [17]. However, many concerns have been raised about the decision-making process behind deep learning technology, in particular, deep neural networks. For instance, can we trust decisions that neural networks make [14, 18, 32]? One way to address this problem is to define properties that we expect the network to satisfy. Verifying whether the network satisfies these properties sheds light on the properties of the function that it represents [7, 23, 31, 34, 37].

One important family of deep neural networks is the class of *Binarized Neural Networks (BNNs)* [20]. These networks have a number of useful features that are useful in resource-constrained environments, like embedded devices or mobile phones [25, 28]. Firstly, these networks are memory efficient, as their parameters are primarily binary. Secondly, they are computationally efficient as all activations are binary, which enables the use of specialized algorithms for fast binary matrix multiplication. Moreover, BNNs allow a compact representation in Boolean logic [7, 31].

There exist approaches that formulate the verification of neural networks to Satisfiability Modulo Theories (SMT) [13, 19, 23], while others do the same to Mixed-Integer Programming (MIP) [11, 15, 36]. In some sense, this work can be considered to be the continuation of that in [7, 31], which translate all the MIP constraints to SAT.

The goal of this work to attack the problem of verifying important properties of BNNs by applying several kinds of approaches and solvers, such as *SAT*, *SMT* and *MIP* solvers. We introduce our solver that is able to encode BNN properties for those solvers and run them in parallel, in a *portfolio setting*. We focus on the important properties of neural networks *adversarial robustness* and *network equivalence*.

In this paper we introduce how to use our solver and report on experiments on verifying both robustness and equivalence. Experimental results show that our solver is capable of verifying those properties of medium-sized BNNs in reasonable runtime, especially when the solvers MiniCARD + Z3 are run in parallel.

2. Preliminaries

A *literal* is a Boolean variable x or its negation $\neg x$. A *clause* is a disjunction of literals. A Boolean formula is in *Conjunctive Normal Form (CNF)*, if it is a conjunction of clauses. We say that a Boolean formula, typically in CNF, is *satisfiable*, if there exists a truth assignment to the Boolean variables of the formula such that the formula evaluates to 1 (true). Otherwise, it is said to be *unsatisfiable (UNSAT)*. The *Boolean Satisfiability (SAT)* problem is the problem of determining if a Boolean formula is satisfiable.

Satisfiability Modulo Theories (SMT) is the decision problem of checking satis-

fiability of a Boolean formula with respect to some background theory. Common theories include the theory of integers, reals, fixed-size bit-vectors, etc. The logics that one could use might differ from each other in the linearity or non-linearity of arithmetic and the presence or absence of quantifiers. In this paper, we use the theory of integers combined with linear arithmetic and without quantifiers – denoted as QF_LIA in the SMT-LIB standard [5].

A *Boolean cardinality constraint* is defined as an expression $\sum_{i=1}^{n} l_i \circ_{\text{rel}} c$, where $l_1, \ldots, l_n$ are literals, $\circ_{\text{rel}} \in \{\geq, \leq, =\}$, and $c \in \mathbb{N}$ is a constant where $0 \leq c \leq n$.

A *pseudo-Boolean constraint* can be considered as a "weighted" Boolean cardinality constraint, and can be defined as an expression $\sum_{i=1}^{n} w_i l_i \ \circ_{\text{rel}} c$, where $w_i \in \mathbb{N}$, $w_i > 0$.

We assume the reader is familiar with the notion and elementary properties of feedforward neural networks. We consider a feedforward neural network to compute a function F where $F(\boldsymbol{x})$ represents the output of F on the input $\boldsymbol{x}$. Let $\ell(\boldsymbol{x})$ denote the ground truth label of $\boldsymbol{x}$. Our tool can analyze two properties of neural networks: adversarial robustness and network equivalence. We call a neural network robust on a given input if small perturbations to the input do not lead to misclassification, as defined as follows, where $\boldsymbol{\tau}$ represents the perturbation and $\epsilon \in \mathbb{N}$ the upper bound for the p-norm of $\boldsymbol{\tau}$.

Definition 2.1 (Adversarial robustness)**.** A feedforward neural network F is (ϵ, p)-robust for an input $\boldsymbol{x}$ if $\neg\exists \boldsymbol{\tau}$, $\|\boldsymbol{\tau}\|_p \leq \epsilon$ such that $F(\boldsymbol{x} + \boldsymbol{\tau}) \neq \ell(\boldsymbol{x})$.

The case of $p = \infty$, which bounds the maximum perturbation applied to each entry in $\boldsymbol{x}$, is especially interesting and has been considered frequently in literature.

Similar to robustness, the equivalence of neural networks is also a property that many would like to verify. We consider two neural networks equivalent if they generate the same output on all inputs, as defined as follows, where $\mathcal{X}$ denotes the input domain.

Definition 2.2 (Network equivalence)**.** Two feedforward neural networks F_1 and F_2 are equivalent if $\forall \boldsymbol{x} \in \mathcal{X}\ F_1(\boldsymbol{x}) = F_2(\boldsymbol{x})$.

3. Encoding of Binarized Neural Networks

A *Binarized Neural Network (BNN)* is a feedforward network where weights and activations are predominantly binary [20]. It is convenient to describe the structure of a BNN in terms of composition of blocks of layers rather than individual layers. Each block consists of a collection of linear and non-linear transformations. Blocks are assembled sequentially to form a BNN.

Internal block. Each internal block (denoted as Block) in a BNN performs a collection of transformations over a binary input vector and outputs a binary vector. While the input and output of a Block are binary vectors, the internal layers of Block can produce real-valued intermediate outputs. A common construction

of an internal BLOCK (taken from [20]) is composed of three main operations:[1] a linear transformation (LIN), batch normalization (BN), and binarization (BIN). Table 1 presents the formal definition of these transformations. Figure 1 shows two BLOCKs connected sequentially.

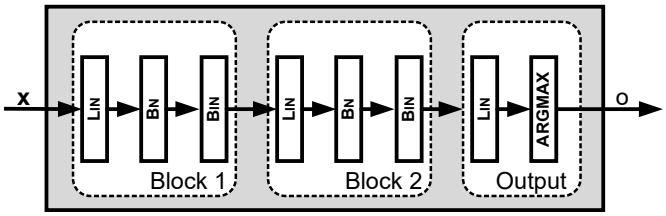


Figure 1. A schematic view of a binarized neural network. The internal blocks also have an additional HARDTANH layer during the training.

Table 1. Structure of internal and outputs blocks which stacked together form a binarized neural network. In the training phase, there might be an additional HARDTANH layer after batch normalization. A_k and b_k are parameters of the LIN layer, whereas $\alpha_{k_i}, \gamma_{k_i}, \mu_{k_i}, \sigma_{k_i}$ are parameters of the BN layer. The μ's and σ's correspond to mean and standard deviation computed in the training phase. The BIN layer is parameter free.

Structure of k^{th} internal block, $\text{BLOCK}_k : \{-1,1\}^{n_k} \to \{-1,1\}^{n_{k+1}}$ on $\boldsymbol{x}_k \in \{-1,1\}^{n_k}$	
LIN	$\boldsymbol{y} = A_k \boldsymbol{x}_k + \boldsymbol{b}_k$, where $A_k \in \{-1,1\}^{n_{k+1} \times n_k}$ and $\boldsymbol{b}_k, \boldsymbol{y} \in \mathbb{R}^{n_{k+1}}$
BN	$z_i = \alpha_{k_i} \left(\frac{y_i - \mu_{k_i}}{\sigma_{k_i}} \right) + \gamma_{k_i}$, where $\boldsymbol{\alpha}_k, \boldsymbol{\gamma}_k, \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k, \boldsymbol{z} \in \mathbb{R}^{n_{k+1}}$. Assume $\sigma_{k_i} > 0$.
BIN	$\boldsymbol{x}_{k+1} = \text{sign}(\boldsymbol{z})$ where $\boldsymbol{x}_{k+1} \in \{-1,1\}^{n_{k+1}}$
Structure of output block, $\text{O} : \{-1,1\}^{n_m} \to [1,s]$ on input $\boldsymbol{x}_m \in \{-1,1\}^{n_m}$	
LIN	$\boldsymbol{w} = A_m \boldsymbol{x}_m + \boldsymbol{b}_m$, where $A_m \in \{-1,1\}^{s \times n_m}$ and $\boldsymbol{b}_m, \boldsymbol{w} \in \mathbb{R}^s$
ARGMAX	$o = \text{argmax}(\boldsymbol{w})$, where $o \in [1,s]$

Output block. The output block (denoted as O) produces the classification decision for a given binary input vector. It consists of two layers (see Table 1). The first layer applies a linear (affine) transformation that maps its input to a vector of integers, one for each output label class. This is followed by an ARGMAX layer, which outputs the index of the largest entry in this vector as the predicted label.

[1] In the training phase, there is an additional HARDTANH layer after batch normalization layer that is omitted in the inference phase [20].

Network of blocks. BNN is a deep feedforward network formed by assembling a sequence of internal blocks and an output block. Suppose we have $m-1$ internal blocks, $\textsc{Block}_m, \ldots, \textsc{Block}_{m-1}$ that are placed consecutively, so the output of a block is the input to the next block in the list. Let n_k denote the number of input values to $\textsc{Block}_k$. Let $\boldsymbol{x}_k \in \{-1,1\}^{n_k}$ be the input to $\textsc{Block}_k$ and $\boldsymbol{x}_{k+1} \in \{-1,1\}^{n_{k+1}}$ be its output. The input of the first block is the input of the network. We assume that the input of the network is a vector of integers, which holds for the image classification task if images are in the standard RGB format. Note that these integers can be encoded with binary values $\{-1,1\}$ using a standard encoding. It is also an option to add an additional BnBin block before $\textsc{Block}_1$ to binarize the input images (see Sections 3.3 and 6.1). Therefore, we keep notations uniform for all layers by assuming that inputs are all binary. The output of the last layer, $\boldsymbol{x}_m \in \{-1,1\}^{n_m}$, is passed to the output block O to obtain one of the s labels.

Definition 3.1 (Binarized Neural Network)**.** A binarized neural network $\mathrm{BNN} : \{-1,1\}^{n_1} \to [1,\ldots,s]$ is a feedforward network that is composed of m blocks, $\textsc{Block}_1, \ldots, \textsc{Block}_{m-1}, \mathrm{O}$. Formally, given an input $\boldsymbol{x}$,

$$\mathrm{BNN}(\boldsymbol{x}) = \mathrm{O}(\textsc{Block}_{m-1}(\ldots \textsc{Block}_1(\boldsymbol{x})\ldots)).$$

In the following sections, we show how to encode an entire BNN structure into Boolean constraints, including cardinality constraints.

3.1. Encoding of internal blocks

Each internal block is encoded separately as proposed in [7, 31]. Here we follow the encoding by Narodystka *et al.* Let $\boldsymbol{x} \in \{-1,1\}^{n_k}$ denote the input to the k^{th} block, $\boldsymbol{o} \in \{-1,1\}^{n_{k+1}}$ the output. Since the block consists of three layers, they are encoded separately as follows:

Lin. The first layer applies a linear transformation to the input vector $\boldsymbol{x}$. Let $\boldsymbol{a}_i$ denote the i^{th} row of the matrix A_k and b_i the i^{th} element of the vector $\boldsymbol{b}_k$. We get the constraints

$$y_i = \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle + b_i, \quad \text{for all } i \in [1, n_{k+1}].$$

Bn. The second layer applies batch normalization to the output $\boldsymbol{y}$ of the previous layer. Let $\alpha_i, \gamma_i, \mu_i, \sigma_i$ denote the i^{th} element of the vectors $\boldsymbol{\alpha}_k, \boldsymbol{\gamma}_k, \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k$, respectively. Assume $\alpha_i \neq 0$. We get the constraints

$$z_i = \alpha_i \frac{y_i - \mu_i}{\sigma_i} + \gamma_i, \quad \text{for all } i \in [1, n_{k+1}].$$

Bin. The third layer applies binarization to the output $\boldsymbol{z}$ of the previous layer, by implementing the sign function as follows:

$$o_i = \begin{cases} 1, & \text{if } z_i \geq 0, \\ -1, & \text{if } z_i < 0, \end{cases} \quad \text{for all } i \in [1, n_{k+1}].$$

The entire block can then be expressed as the constraints

$$o_i = \begin{cases} 1, & \text{if } \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle \circ_{\text{rel}} C_i, \\ -1, & \text{otherwise,} \end{cases} \quad \text{for all } i \in [1, n_{k+1}], \tag{3.1}$$

where

$$C_i = -\frac{\sigma_i}{\alpha_i}\gamma_i + \mu_i - b_i$$
$$\circ_{\text{rel}} = \begin{cases} \geq, & \text{if } \alpha_i > 0, \\ \leq, & \text{if } \alpha_i < 0. \end{cases}$$

Let us recall that the input variables x_j and the output variables o_i take the values -1 and 1. We need to replace them with the Boolean variables $x_j^{(\text{b})}, o_i^{(\text{b})} \in \{0, 1\}$ in order to further translate the constraints in (3.1) to the Boolean constraints

$$\sum_{j=1}^{n_k} l_{ij} \circ_{\text{rel}} D_i \;\Leftrightarrow\; o_i^{(\text{b})}, \quad \text{for all } i \in [1, n_{k+1}],$$

where

$$l_{ij} = \begin{cases} x_j^{(b)}, & \text{if } j \in \boldsymbol{a}_i^+, \\ \neg x_j^{(b)}, & \text{if } j \in \boldsymbol{a}_i^-, \end{cases}$$
$$D_i = \begin{cases} \lceil C_i' \rceil + |\boldsymbol{a}_i^-|, & \text{if } \alpha_i > 0, \\ \lfloor C_i' \rfloor + |\boldsymbol{a}_i^-|, & \text{if } \alpha_i < 0, \end{cases}$$
$$C_i' = \Big(C_i + \sum_j a_{ij}\Big)/2,$$
$$\boldsymbol{a}_i^+ = \{j \mid a_{ij} > 0\},$$
$$\boldsymbol{a}_i^- = \{j \mid a_{ij} < 0\}.$$

For further details on the derivation, see [31].

3.2. Encoding of the output block

The output block consists of a Lin layer followed by an ArgMax layer. To encode ArgMax, we need to encode an ordering relation over the outputs of the linear layer, and therefore we introduce the Boolean variables $d_{ii'}^{(\text{b})}$ such that

$$\langle \boldsymbol{a}_i, \boldsymbol{x} \rangle + b_i \geq \langle \boldsymbol{a}_{i'}, \boldsymbol{x} \rangle + b_{i'} \;\Leftrightarrow\; d_{ii'}^{(\text{b})}, \quad \text{for all } i, i' \in [1, s].$$

These constraints can be further translated into Boolean constraints, as proposed by Narodystka *et al.* in [31] and supplemented by us as follows:

$$\sum_{j=1}^{n_m} l_{ii'j} \geq E_{ii'} \;\Leftrightarrow\; d^{(\mathrm{b})}_{ii'}, \quad \text{for all } i, i' \in [1, s], \;\; i \neq i',$$

where

$$l_{ii'j} = \begin{cases} x^{(\mathrm{b})}_j, & \text{if } j \in \boldsymbol{a}^+_{ii'}, \\ \neg x^{(\mathrm{b})}_j, & \text{if } j \in \boldsymbol{a}^-_{ii'}, \\ 0, & \text{otherwise,} \end{cases}$$

$$E_{ii'} = \left\lceil \left(b_{i'} - b_i + \sum_j a_{ij} - \sum_j a_{i'j} \right) / 4 \right\rceil + |\boldsymbol{a}^-_{ii'}|,$$

$$\boldsymbol{a}^+_{ii'} = \{j \mid a_{ij} > 0 \wedge a_{i'j} < 0\},$$

$$\boldsymbol{a}^-_{ii'} = \{j \mid a_{ij} < 0 \wedge a_{i'j} > 0\}.$$

In the case of $i = i'$, $d^{(\mathrm{b})}_{ii'}$ must obviously be assigned to 1.

Finally, to encode ArgMax, we have to pick the row in the matrix $(d_{ii'})$ which contains only 1s, as it can be encoded by the Boolean constraint

$$\sum_{i'} d^{(\mathrm{b})}_{ii'} = s \;\Leftrightarrow\; o^{(\mathrm{b})}_i, \quad \text{for all } i \in [1, s].$$

3.3. Encoding of the input binarization block

In our paper, and also in [31], experiments on checking adversarial robustness under the L_∞ norm are run on grayscale input images that are binarized by an additional BnBin block before Block_1. We now propose how this BnBin block can be encoded to Boolean constraints.

Let $\boldsymbol{\alpha}_0, \boldsymbol{\gamma}_0, \boldsymbol{\mu}_0, \boldsymbol{\sigma}_0$ denote the parameters of the Bn layer. Since adversarial robustness is about to be checked, the input $\boldsymbol{x} \in \mathbb{N}^{n_1}$ consists of constants, while the perturbation $\boldsymbol{\tau} \in [-\epsilon, \epsilon]^{n_1}$ consists of integer variables and the output $\boldsymbol{o}^{(\mathrm{b})} \in \{0,1\}^{n_1}$ consists of Boolean variables. The BnBin block can be encoded by the constraints

$$\alpha_i \frac{x_i + \tau_i - \mu_i}{\sigma_i} + \gamma_i \geq 0 \;\Leftrightarrow\; o^{(\mathrm{b})}_i, \quad \text{for all } i \in [1, n_1], \tag{3.2}$$

where $\alpha_i, \gamma_i, \mu_i, \sigma_i$ denote the i^{th} element of the vectors $\boldsymbol{\alpha}_0, \boldsymbol{\gamma}_0, \boldsymbol{\mu}_0, \boldsymbol{\sigma}_0$, respectively.

The constraints in (3.2) further translate to

$$x_i + \tau_i - \mu_i + \frac{\sigma_i \gamma_i}{\alpha_i} \;\circ_{\mathrm{rel}}\; 0 \;\Leftrightarrow\; o^{(\mathrm{b})}_i, \tag{3.3}$$

where

$$\circ_{\mathrm{rel}} = \begin{cases} \geq, & \text{if } \alpha_i > 0, \\ \leq, & \text{if } \alpha_i < 0. \end{cases}$$

Then (3.3) translates to

$$\tau_i \circ_{\text{rel}} B_i \;\Leftrightarrow\; o_i^{(\text{b})}, \tag{3.4}$$

where

$$B_i = \begin{cases} \lceil B_i' \rceil\,, & \text{if } \alpha_i > 0, \\ \lfloor B_i' \rfloor\,, & \text{if } \alpha_i < 0, \end{cases}$$
$$B_i' = \mu_i - x_i - \frac{\sigma_i \gamma_i}{\alpha_i}.$$

Since τ_i is in the given range $[-\epsilon, \epsilon]$, we can represent it as a *bit-vector* of a given bit-width. In order to apply *unsigned* bit-vector arithmetic, we translate the domain of τ_i into $[0, 2\epsilon]$. Thus, we can represent τ_i as a bit-vector variable of bit-width $w = \lceil \log_2(2\epsilon + 1) \rceil$ and apply unsigned bit-vector arithmetic to (3.4) as follows:

$$\tau_i^{[w]} \circ_{\text{rel}}^{\text{u}} (B_i + \epsilon)^{[w]} \;\Leftrightarrow\; o_i^{(\text{b})}, \tag{3.5}$$

where $\circ_{\text{rel}}^{\text{u}}$ denotes the corresponding unsigned bit-vector relational operator `bvuge` or `bvule`, respectively, and the bound $B_i + \epsilon$ is represented as a bit-vector constant of bit-width w. For the syntax and semantics of common bit-vector operators, see [24].

The constraints in (3.5) are not even needed to add in certain cases:

- if $B_i \leq -\epsilon$, then assign $o_i^{(\text{b})}$ to 1 if $\alpha_i > 0$, and to 0 if $\alpha_i < 0$;
- if $B_i > \epsilon$, then assign $o_i^{(\text{b})}$ to 0 if $\alpha_i > 0$, and to 1 $\alpha_i < 0$.

Some further constraints are worth to add to restrict the domain of τ_i:

$$\begin{aligned} &{\tau_i}^{[w]} \geq^{\text{u}} 0^{[w]} \\ &{\tau_i}^{[w]} \leq^{\text{u}} (2\epsilon)^{[w]} \\ &{\tau_i}^{[w]} \geq^{\text{u}} (\epsilon - x_i)^{[w]}, \text{ if } x_i < \epsilon \\ &{\tau_i}^{[w]} \leq^{\text{u}} (\epsilon + \max_x - x_i)^{[w]}, \text{ if } x_i > \max_x - \epsilon \end{aligned} \tag{3.6}$$

where $\max_x$ is the highest possible value for the input values in $\boldsymbol{x}$.[2]

In our tool, all the bit-vector constraints in (3.5) and (3.6) are bit-blasted into CNF.

3.4. Encoding of BNN properties

In this paper, we focus on the properties defined in Section 2, namely adversarial robustness and network equivalence.

[2]In our experiments, the input represents pixels of grayscale images, therefore $\max_x = 255$.

3.4.1. Adversarial robustness

We assume that the BNN consists of an input binarization block, internal blocks and an output block. Let $\mathrm{BNN}\big(\boldsymbol{x}+\boldsymbol{\tau}, \boldsymbol{o}^{(\mathrm{b})}\big)$ denote the encoding of the whole BNN over the perturbated input $\boldsymbol{x}+\boldsymbol{\tau}$ and the output $\boldsymbol{o}^{(\mathrm{b})}$. Note that $\boldsymbol{x} \in \mathbb{N}^{n_1}$ is an input from the the training or test set, therefore its ground truth label $\ell(\boldsymbol{x})$ is given. On the other hand, the perturbation $\boldsymbol{\tau} \in [-\epsilon, \epsilon]^{n_1}$ consists of integer variables. The output $\boldsymbol{o}^{(\mathrm{b})} \in \{0,1\}^s$ consists of Boolean variables. Basically, we are looking for a satisfying assignment for the perturbation variables $\boldsymbol{\tau}$ such that the BNN outputs a label different from $\ell(\boldsymbol{x})$. Thus, checking adversarial robustness translates into checking the satisfiability of the following constraint:

$$\mathrm{BNN}\big(\boldsymbol{x}+\boldsymbol{\tau}, \boldsymbol{o}^{(\mathrm{b})}\big) \ \wedge \neg o^{(\mathrm{b})}_{\ell(\boldsymbol{x})}.$$

3.4.2. Network equivalence

We want to check if two BNNs classify binarized inputs completely the same. Therefore we assume that those BNNs do not have BnBin blocks, or if they do, then they apply the same BnBin block. Therefore, let $\mathrm{BNN}_1\big(\boldsymbol{x}^{(\mathrm{b})}, o^{(\mathrm{b})}_1\big)$ and $\mathrm{BNN}_2\big(\boldsymbol{x}^{(\mathrm{b})}, o^{(\mathrm{b})}_2\big)$ denote the encoding of the internal blocks and the output block of the two BNNs, respectively, over the same binary input $\boldsymbol{x}^{(\mathrm{b})}$. Checking the equivalence of those BNNs translates into checking the satisfiability of the following constraint:

$$\mathrm{BNN}_1\big(\boldsymbol{x}, \boldsymbol{o}^{(\mathrm{b})}_1\big) \wedge \mathrm{BNN}_2\big(\boldsymbol{x}, \boldsymbol{o}^{(\mathrm{b})}_2\big) \ \wedge \boldsymbol{o}^{(\mathrm{b})}_1 \neq \boldsymbol{o}^{(\mathrm{b})}_2.$$

We translate the inequality $\boldsymbol{o}^{(\mathrm{b})}_1 \neq \boldsymbol{o}^{(\mathrm{b})}_2$ over vectors of Boolean variables into

$$\neg\big(o^{(\mathrm{b})}_{1,1} \Leftrightarrow o^{(\mathrm{b})}_{2,1}\big) \vee \cdots \vee \neg\big(o^{(\mathrm{b})}_{1,s} \Leftrightarrow o^{(\mathrm{b})}_{2,s}\big)$$

which can then be further translated to a set of clauses by using Tseitin transformation.

4. Encoding of clauses and Boolean cardinality constraints

In Section 3, we proposed an encoding of BNNs into *clauses* $l_1 \vee \cdots \vee l_n$ as well as *equivalences* over Boolean cardinality constraints in the form

$$l \ \Leftrightarrow \ \sum_{i=1}^{n} l_i \geq c, \tag{4.1}$$

where $l, l_1, \ldots, l_n$ are literals and $c \in \mathbb{N}$ is a constant where $0 \leq c \leq n$. Note that our encoding applies *"AtMost"* Boolean cardinality constraints as well. Such a constraint $\sum_{i=1}^{n} l_i \leq c$ can always be translated to an *"AtLeast"* constraint $\sum_{i=1}^{n} \neg l_i \geq n - c$.

Depending on the approaches one wants to apply to the satisfiability checking of those constraints, they have to be encoded in different ways.

4.1. Encoding into SAT

There are various existing, well-known approaches expressing Boolean cardinality constraints into Boolean logic, for example by using sequential counters [35], cardinality networks [1] or modulo totalizers [30, 33].

Sequential counters [35] encode an "AtLeast" Boolean cardinality constraint into the following Boolean formula:

$$\begin{aligned} & (l_1 \Leftrightarrow v_{1,1}) \\ \wedge \quad & \neg v_{1,j} && \text{for } j \in [2,c], \\ \wedge \quad & (v_{i,1} \Leftrightarrow l_i \vee v_{i-1,1}) && \text{for } i \in [2,n], \\ \wedge \quad & \big(v_{i,j} \Leftrightarrow (l_i \wedge v_{i-1,j-1}) \vee v_{i-1,j}\big) && \text{for } i \in [2,n], j \in [2,c]. \end{aligned}$$

All the Boolean variables $v_{i,j}$ are introduced as fresh variables and the formula above can be converted into its CNF [35]. On the top of that, to encode the constraint (4.1), we only need to additionally encode the formula $l \Leftrightarrow v_{n,c}$.

Cardinality networks [1] yield another, refined approach for encoding Boolean cardinality constraints. For improving reasoning about cardinality constraints encoded, for example, using sequential counters, a cardinality network encoding of a cardinality constraint divides the cardinality constraint into multiple instances of the base operations *half sorting* and *simplified half merging*, which basically work as building blocks.

The *modulo totalizer* cardinality encoding [33] and its variant for k-cardinality [30] improve the above described approach based on cardinality network, especially in connection with MaxSAT solving. The modulo totalizer approach of [33] addresses limitations of the half sorting cardinality network approach from [1], by using totalizer encodings from [3] in order to reduce the number of variables during CNF encodings. The modulo totalizer cardinality encoding of [33] decreases the number of clauses used in [3], and hence improves cardinality network encodings during constraint propagation.

4.2. Encoding into SMT

It is straightforward to encode clauses and constraints (4.1) into SMT over the logic QF_LIA. We would like to note that bit-vector constraints (3.5), (3.6) are bit-blasted into CNF in our tool and then added as clauses, even when being encoded into SMT. As future work, one could try to solve all the constraints over the logic QF_BV.

4.3. Encoding into Boolean cardinality constraints

The encoding that we proposed for BNNs consists of *clauses* on the one hand, and *equivalences* over Boolean cardinality constraints in the form (4.1) on the other hand. We show how to encode both type of constraints into a set of Boolean cardinality constraints.

A clause $l_1 \vee \cdots \vee l_n$ can be encoded as the Boolean cardinality constraint $\sum_{i=1} l_i \geq 1$.

A constraint (4.1) can be unfolded into two implications (assume $c > 0$):

$$\begin{aligned} l &\Rightarrow \sum_{i=1} l_i \geq c, \\ \neg l &\Rightarrow \sum_{i=1} l_i \leq c - 1. \end{aligned} \tag{4.2}$$

By following the idea on the GitHub page[3] of the SAT solver MiniCARD [27], an implied Boolean cardinality constraints (4.2) can be translated to a (non-implied) Boolean cardinality constraint

$$\sum_{i=1} l_i + \underbrace{\neg l + \cdots + \neg l}_{c} \geq c, \tag{4.3}$$

which can then be solved by cardinality solvers with duplicated-literal handling, such as MiniCARD.

4.4. Encoding into pseudo-Boolean constraints

The Boolean cardinality encoding from Section 4.3 can be fed into pseudo-Boolean solvers as well. The Boolean cardinality constraint (4.3) can naturally be translated to a pseudo-Boolean constraint $\sum_{i=1} l_i + \neg l \cdot c \geq c$.

5. Implementation

All the encodings described in the previous sections are implemented in Python, as part of our solver. Since our solver is a portfolio solver, it executes different kind of solvers (SAT, SMT, MIP) in parallel, by instantiating `ProcessPool` from the Python module `pathos.multiprocessing` [29], which can run jobs with a non-blocking and unordered map.

The Python package PySAT [21] provides a unified API to several SAT solvers such as MiniSat [12], Glucose [2] and Lingeling [6]. PySAT also supports a lot of encodings for Boolean cardinality constraints, including sequential counters [35], cardinality networks [1] and modulo totalizer [30, 33]. Furthermore, PySAT offers API to the SAT solver MiniCARD [27], which handles Boolean cardinality constraints *natively* on the level of watched literals and conflict analysis, instead of translating them into CNF.

In a similar manner, the Python package PySMT [16] provides a unified API to several SMT solvers, such as MathSAT [8], Z3 [9], CVC4 [4] and Yices [10].

The Python package MIP provides tools to solve mixed-integer linear programming instances and provides a unified API to MIP solvers such as CLP, CBC and Gurobi.

[3] `https://github.com/liffiton/minicard`

When running our portfolio solver, one can easily choose the solvers to execute in parallel, by using the following command-line arguments:

`-sat-solver.` Choose any SAT solver supported by the PySAT package such as MiniSat, Glucose, etc., including MiniCARD, or disable this option by using the value `none`.

`-smt-solver.` Choose any SMT solver supported by PySMT such as Z3, MathSAT, etc., or disable this option by using the value `none`. Note that you might need to install the corresponding SMT solver for PySMT by using the `pysmt-install` command.

`-mip-solver.` Choose any MIP solver supported by the MIP package, most importantly Gurobi, or disable this option by using the value `none`. Note that you might need to purchase a license for Gurobi.

`-card-enc.` Choose any cardinality encoding supported by the PySAT package such as sequential counters, cardinality networks, modulo totalizer, k-cardinality modulo totalizer, etc., or disable this option by using the value `none`.

`-timeout.` Set the timeout in seconds.

Our solver consists of two Python programs `bnn_adv_robust_check.py` and `bnn_eq_check.py` to check adversarial robustness and network equivalence, respectively. If `bnn_adv_robust_check.py` returns UNSAT, then the given input image is considered to be robust under the given maximal perturbation value passed as a command-line argument. In case of SAT answer, the tool displays the perturbated input values and the label resulted by misclassification.

If `bnn_eq_check.py` returns UNSAT, then the two given BNNs are considered to be equivalent. In case of SAT answer, the tool displays the common input values for which the BNNs return different outputs, which are also displayed. Note that an output is displayed as a list of Boolean literals among which the single positive literal represents the output label.

6. Experiments and results

Our experiments were run on Intel i5-7200U 2.50 GHz CPU (2 cores, 4 threads) with 8 GB memory. The time limit was set to 300 seconds.

In our experiments, the BNN architecture is the same as in the experiments in [31]: it consists of 4 internal blocks and 1 output block. Each internal block contains a Lin layer with 200, 100, 100 and 100 neurons, respectively. We use an additional HardTanh layer only during the training of the network. We trained the network on the MNIST dataset [26]. The accuracy of the resulting network is 93%.

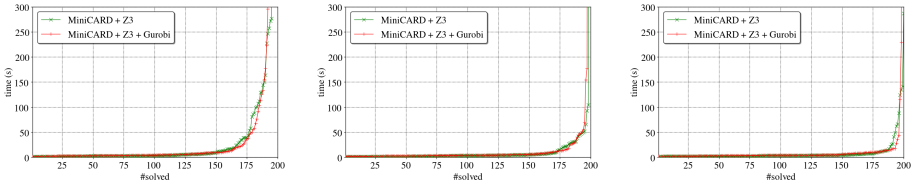
6.1. Verifying adversarial robustness

In the first set of experiments, we focused on the important problem of checking adversarial robustness under the L_∞ norm. From the MNIST dataset, we randomly picked 20 images (from the test set) that were correctly classified by the network for each of the 10 classes. This resulted in a set of 200 images that we consider in our experiments on adversarial robustness. We experimented with three different maximum perturbation values by varying $\epsilon \in \{1, 3, 5\}$.

To process the inputs, we add a BnBin block to the BNN before Block_1. The BnBin block applies binarization to the grayscale MNIST images. We would like emphasize that our experiments did not apply any additional preprocessing, as opposed to the experiments in [31] that first try to perturb only the top 50% of highly salient pixels in the input image. Furthermore, our solver does not apply any additional search procedure on the top of the solvers being run in parallel, as opposed to the experiments in [31] that apply a counterexample-guided (CEG) search procedure based on Craig interpolation. In this sense, our solver explores the search space without applying any additional procedures.

Figure 2 shows some of the results of our experiments. Each column shows the number of solved instances out of the 200 selected instances and the average runtime in seconds. The bar chart under certain cells shows the distribution of different solvers providing the results. The bottom charts present the results in a more detailed way, where the distribution of runtimes suggests that our solver can solve ca. 85–95% of the instances in less than 30 seconds.

Solvers	$\epsilon = 1$	$\epsilon = 3$	$\epsilon = 5$
MiniCARD + Z3	195 (26.8)	198 (10.4)	200 (8.7)
MiniCARD + Z3 + Gurobi	192 (26.4)	197 (12.7)	198 (9.8)



(a) $\epsilon = 1$ (b) $\epsilon = 3$ (c) $\epsilon = 5$

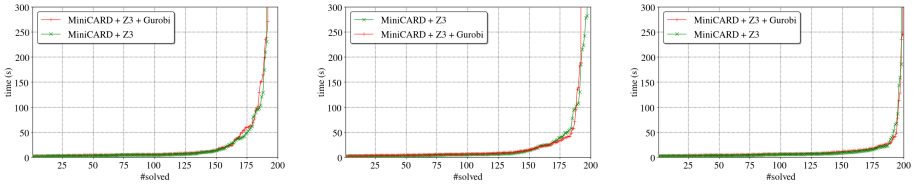
Figure 2. Results on checking adversarial robustness of 4-Block BNN on MNIST dataset, for different maximum perturbation values ϵ. Colors represent the ratio of solved instances by different solvers: purple for MiniCARD, green for Z3, blue for Gurobi.

As the figure shows, our solver produced the best results when running

MiniCARD as a SAT solver and Z3 as an SMT solver in parallel. Since, in our preliminary experiments, Gurobi had showed promising performance, we also ran experiments with Gurobi parallel to MiniCARD and Z3. Of course, we also tried different combinations of solvers in our experiments, but we found the ones in the table the most promising.

In order to investigate how our solver scales for larger BNNs, we constructed another BNN with 5 internal blocks containing Lin layers of size 300, 200, 150, 100 and 100, respectively, and trained it on the MNIST dataset. The accuracy of the resulting network is 94%. Figure 3 shows the results of our corresponding experiments.

Solvers	$\epsilon = 1$	$\epsilon = 3$	$\epsilon = 5$
MiniCARD + Z3	191 (29.3)	197 (24.2)	198 (13.6)
MiniCARD + Z3 + Gurobi	192 (31.6)	192 (26.0)	199 (14.3)



(a) $\epsilon = 1$ (b) $\epsilon = 3$ (c) $\epsilon = 5$

Figure 3. Results on checking adversarial robustness of 5-Block BNN on MNIST dataset.

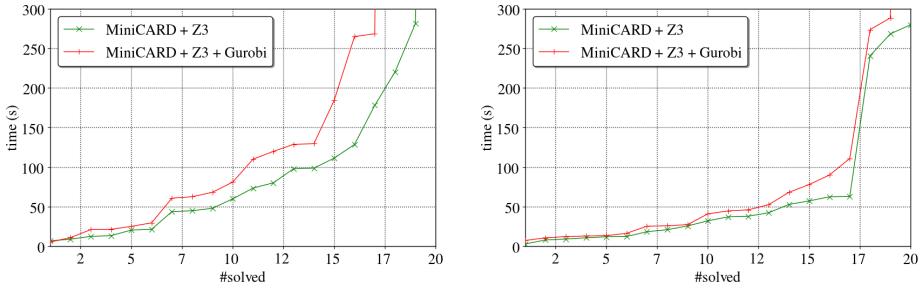
6.2. Verifying network equivalence

In the second set of experiments, we focused on the problem of checking network equivalence. From our 4-Block BNN trained to classify MNIST images, we generated 20 slightly different variants by altering a few weights in the network. For this sake, we randomly flip $\delta > 0$ weights in A_m. Then, we run our solver to check if the original BNN is equivalent with an altered variant. Since the aim was to generate difficult benchmark instances, i.e., which are "almost UNSAT", we chose small values for δ. Figure 4 shows the results of our corresponding experiments.

6.3. Side notes

In our solver's source code, there exist implemented features that are not yet accessible due to the lack of API features of certain Python packages. Although PySAT's CNF encodings of Boolean cardinality constraints are accessible via our

Solvers	$\delta = 2$	$\delta = 5$
MiniCARD + Z3	19 (92.3)	20 (64.5)
MiniCARD + Z3 + Gurobi	17 (124.5)	19 (77.1)



(a) $\delta = 2$ (b) $\delta = 5$

Figure 4. Results on checking network equivalence, for different δ values.

solver's command-line argument `--card-enc`, equivalences (4.1) cannot directly be dealt with PySAT since the output variable of a CNF encoding cannot be accessed through PySAT's API. For instance, we would need to access the Boolean variable $v_{n,c}$ when using sequential counter encoding as described in Section 4.1. Therefore, in our solver's current version, each equivalence (4.1) is first encoded into a pair of Boolean cardinality constraints as described in Section 4.3, and the resulting cardinality constraints are then encoded into CNF. Note that encoding equivalences (4.1) directly into Boolean logic would result in more easy-to-solve instances, once PySAT allows. In the latter case, on the other hand, the encoding into CNF might dominate the runtime, since millions of variables and millions of clauses are generated even for our 4-Block BNN.

7. Conclusions

We introduced a new portfolio-style solver to verify important properties of binarized neural networks such as adversarial robustness and network equivalence. Our solver encodes those BNN properties, as we propose SAT, SMT, cardinality and pseudo-Boolean encodings in the paper. Our experiments demonstrated that our solver was capable of verifying adversarial robustness of medium-sized BNNs on the MNIST dataset in reasonable time and seemed to scale for larger BNNs. We also ran experiments on network equivalence with impressive results on the SAT instances.

After we submitted this paper, K. Jia and M. Rinard have recently published a paper about a framework for verifying robustness for BNNs [22]. They devel-

oped a SAT solver with native support for reified cardinality constraints and, also, proposed strategies to train BNNs such that weight matrices were sparse and cardinality bounds low. Based on their experimental results, their solver might outperform our solver on their benchmarks. As part of future work, we would like to run experiments with both solvers on those benchmarks.

We will try to overcome the problems that originate in using the PySAT Python packages, in order to make already implemented "hidden" features accessible for users. Furthermore, we are planning to extend the palette of solvers with Google's OR-Tools, which look promising based on our preliminary experiments.

References

[1] R. Asín, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell: *Cardinality Networks: a theoretical and empirical study*, Constraints 16.2 (2011), pp. 195–221.

[2] G. Audemard, L. Simon: *Lazy Clause Exchange Policy for Parallel SAT Solvers*, in: Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 8561, Lecture Notes in Computer Science, Springer, 2014, pp. 197–205.

[3] O. Bailleux, Y. Boufkhad: *Efficient CNF Encoding of Boolean Cardinality Constraints*, in: Proc. 9th International Conference on Principles and Practice of Constraint Programming (CP), 2003, pp. 108–122.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli: *CVC4*, in: Proc. Int. Conf. on Computer Aided Verification (CAV), vol. 6806, Lecture Notes in Computer Science, Springer, 2011, pp. 171–177.

[5] C. Barrett, P. Fontaine, C. Tinelli: *The Satisfiability Modulo Theories Library (SMT-LIB)*, `www.SMT-LIB.org`, 2016.

[6] A. Biere: *CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017*, in: Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions, vol. B-2017-1, Department of Computer Science Series of Publications B, University of Helsinki, 2017, pp. 14–15.

[7] C. Cheng, G. Nührenberg, H. Ruess: *Verification of Binarized Neural Networks* (2017), arXiv: `1710.03107`.

[8] A. Cimatti, A. Griggio, B. Schaafsma, R. Sebastiani: *The MathSAT5 SMT Solver*, in: Proc. Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), ed. by N. Piterman, S. Smolka, vol. 7795, Lecture Notes in Computer Science, Springer, 2013, pp. 93–107.

[9] L. De Moura, N. Bjørner: *Z3: An Efficient SMT Solver*, in: Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), TACAS'08/ETAPS'08, Springer-Verlag, 2008, pp. 337–340.

[10] B. Dutertre: *Yices 2.2*, in: Proc. Int. Conf. on Computer-Aided Verification (CAV), ed. by A. Biere, R. Bloem, vol. 8559, Lecture Notes in Computer Science, Springer, 2014, pp. 737–744.

[11] S. Dutta, S. Jha, S. Sankaranarayanan, A. Tiwari: *Output Range Analysis for Deep Feedforward Neural Networks*, in: NASA Formal Methods, Springer, 2018, pp. 121–138.

[12] N. Eén, N. Sörensson: *An Extensible SAT-solver*, in: Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 2919, Lecture Notes in Computer Science, Springer, 2004, pp. 502–518.

[13] R. Ehlers: *Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks*, in: Automated Technology for Verification and Analysis, Springer, 2017, pp. 269–286.

[14] EU Data Protection Regulation: *Regulation (EU) 2016/679 of the European Parliament and of the Council*, 2016.

[15] M. Fischetti, J. Jo: *Deep Neural Networks and Mixed Integer Linear Optimization*, Constraints 23 (3 2018), pp. 296–309, doi: https://doi.org/10.1007/s10601-018-9285-6.

[16] M. Gario, A. Micheli: *PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms*, in: International Workshop on Satisfiability Modulo Theories (SMT), 2015.

[17] I. Goodfellow, Y. Bengio, A. Courville: *Deep Learning*, The MIT Press, 2016, isbn: 0262035618.

[18] B. Goodman, S. R. Flaxman: *European Union Regulations on Algorithmic Decision-Making and a "Right to Explanation"*, AI Magazine 38.3 (2017), pp. 50–57.

[19] X. Huang, M. Kwiatkowska, S. Wang, M. Wu: *Safety Verification of Deep Neural Networks*, in: Computer Aided Verification, Springer, 2017, pp. 3–29.

[20] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio: *Binarized Neural Networks*, in: Advances in Neural Information Processing Systems 29, Curran Associates, Inc., 2016, pp. 4107–4115.

[21] A. Ignatiev, A. Morgado, J. Marques-Silva: *PySAT: A Python Toolkit for Prototyping with SAT Oracles*, in: Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 10929, Lecture Notes in Computer Science, Springer, 2018, pp. 428–437.

[22] K. Jia, M. Rinard: *Efficient Exact Verification of Binarized Neural Networks* (2020), arXiv: 2005.03597 [cs.AI].

[23] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, M. J. Kochenderfer: *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*, in: CAV, 2017, pp. 97–117, doi: https://doi.org/10.1007/978-3-319-63387-9_5.

[24] G. Kovásznai, A. Fröhlich, A. Biere: *Complexity of Fixed-Size Bit-Vector Logics*, Theory of Computing Systems 59 (2016), pp. 323–376, issn: 1433-0490, doi: https://doi.org/10.1007/s00224-015-9653-1.

[25] J. Kung, D. Zhang, G. Van der Wal, S. Chai, S. Mukhopadhyay: *Efficient Object Detection Using Embedded Binarized Neural Networks*, Journal of Signal Processing Systems (2017), pp. 1–14.

[26] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner: *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE 86.11 (Nov. 1998), pp. 2278–2324.

[27] M. H. Liffiton, J. C. Maglalang: *More Expressive Constraints for Free*, in: Proc. International Conference on Theory and Applications of Satisfiability Testing (SAT), vol. 7317, Lecture Notes in Computer Science, Springer, 2012, pp. 485–486.

[28] B. McDanel, S. Teerapittayanon, H. T. Kung: *Embedded Binarized Neural Networks*, in: EWSN, Junction Publishing, Canada / ACM, 2017, pp. 168–173.

[29] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, M. A. Aivazis: *Building a framework for predictive science* (2012), arXiv: 1202.1056.

[30] A. Morgado, A. Ignatiev, J. Marques-Silva: *MSCG: Robust Core-Guided MaxSAT Solving*, JSAT 9 (2014), pp. 129–134.

[31] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, T. Walsh: *Verifying Properties of Binarized Deep Neural Networks*, in: 32nd AAAI Conference on Artificial Intelligence, 2018, pp. 6615–6624.

[32] NIPS IML Symposium: *NIPS Interpretable ML Symposium*, Dec. 2017.

[33] T. Ogawa, Y. Liu, R. Hasegawa, M. Koshimura, H. Fujita: *Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers*, in: 25th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE, 2013, pp. 9–17.

[34] G. Singh, T. Gehr, M. Püschel, M. T. Vechev: *Boosting Robustness Certification of Neural Networks*, in: 7th International Conference on Learning Representations, OpenReview.net, 2019.

[35] C. Sinz: *Towards an Optimal CNF Encoding of Boolean Cardinality Constraints*, in: Proc. Principles and Practice of Constraint Programming (CP), Springer, 2005, pp. 827–831.

[36] V. Tjeng, K. Y. Xiao, R. Tedrake: *Evaluating Robustness of Neural Networks with Mixed Integer Programming*, in: 7th International Conference on Learning Representations, OpenReview.net, 2019.

[37] T. Weng, H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. S. Boning, I. S. Dhillon: *Towards Fast Computation of Certified Robustness for ReLU Networks*, in: ICML, 2018, pp. 5273–5282.