Annales Mathematicae et Informaticae
57 (2023) pp. 36–48
DOI: https://doi.org/10.33039/ami.2023.04.003
URL: https://ami.uni-eszterhazy.hu

Formal verification for quantized neural networks

Gergely Kovásznai[a], **Dorina Hedvig Kiss**[a], **Péter Mlinkó**[b]

[a]Department of Computational Science, Eszterházy Károly Catholic University
kovasznai.gergely@uni-eszterhazy.hu
k.dorina33@gmail.com

[b]Pázmány Péter Catholic University
peter.mlinko@gmail.com

Abstract. Despite of deep neural networks are being successfully used in many fields of computing, it is still challenging to verify their trustiness. Previously it has been shown that binarized neural networks can be verified by being encoded into Boolean constraints. In this paper, we generalize this encoding to quantized neural networks (QNNs). We demonstrate how to implement QNNs in Python, using the Tensorflow and Keras libraries. Also, we demonstrate how to implement a Boolean encoding of QNNs, as part of our tool that is able to run a variety of solvers to verify QNNs.

Keywords: Artificial Intelligence, Deep Learning, Neural Network, Formal Verification, SAT, SMT, Constraint Programming, Python, Keras

AMS Subject Classification: 68T07, 68T27, 68Q60

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 [7]. However, many concerns have been raised about the decision-making process behind deep learning technology, in particular, deep neural networks [4, 8]. To address this problem, one can define properties and then verify whether the given neural network satisfies these properties [1, 13, 19, 21, 23].

There exist approaches that formulate the verification of neural networks to Satisfiability Modulo Theories (SMT) [3, 9, 13], while others do the same to Mixed-Integer Programming (MIP) [2, 5, 22].

Submitted: July 14, 2022
Accepted: April 20, 2023
Published online: July 17, 2023

One important family of deep neural networks is the class of *Binarized Neural Networks* (BNNs) [10]. Since these networks are memory efficient and computationally efficient, as their parameters and activations are predominantly binary, BNNs are useful in resource-constrained environments, like embedded devices or mobile phones [15, 17]. Moreover, BNNs allow a compact representation in Boolean logic, enabling verification approaches based on SAT or SMT solving, or 0-1 Integer Linear Programming [1, 14, 19].

Some approaches [10, 14, 19] describe the structure of a BNN in terms of sequential composition of blocks of layers rather than individual layers. While the blocks can produce real-typed intermediate values, each of them takes a binary input vector and outputs a binary vector, except for the output block. Figure 1 shows a common construction of a BNN [10, 14, 19]. Each *internal block* is composed of three layers: linear transformation with binarized weights (BLin), batch normalization (BN), and binarization (Bin). The *output block* produces the classification decision for a given binary input vector. It consists of two layers: a BLin that outputs a vector of integers, one for each output label class, followed by an ArgMax layer.

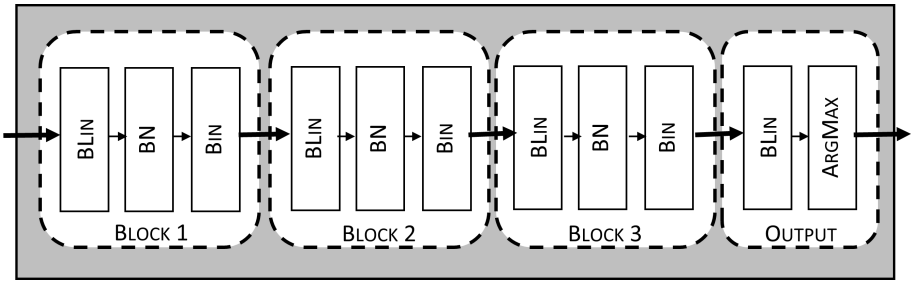

Figure 1. A schematic view of a binarized neural network.

In this paper, we propose a very similar, generalized structure of a Quantized Neural Network (QNN), which applies quantization instead of binarization. Since BNNs are often not that robust, therefore it has a great potential to apply QNNs instead of binarized ones in order to achieve higher robustness [11], while keeping the possibility of applying logic-based verification in an efficient way. Figure 2 shows the proposed structure where the linear transformation layer (QLin) applies quantization to the weights, and so does the activation layer (Qnt).

The focus of this paper is on what is necessary for applying logic-based verification to QNNs. In Section 2, we define the proposed QNN structure in an exact way, together with all the necessary concepts for the verification task based on Boolean logic. Section 3 gives some ideas how to implement QNNs, using the Tensorflow and Keras libraries. Section 4 proposes an encodings of internal blocks of QNNs into a set of Boolean constraints, for the sake of formal verification. Finally, in Section 5, we show some aspects of how to implement a tool for verifying the encoded QNNs.

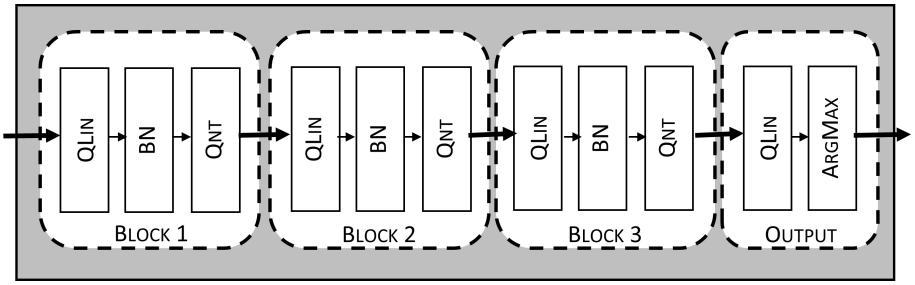

Figure 2. A schematic view of a quantized neural network.

2. Preliminaries

A *literal* is a Boolean variable x or its negation $\neg x$. 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 $0 \leq c \leq n$.

Reifying a constraint C creates a new constraint $l \Leftrightarrow C$ where l is a Boolean literal. An *indicator constraint* means almost the same, except for that it applies implication instead of equivalence, in the form of $l \Rightarrow C$. Note that a reified constraint can always be translated to a conjunction of two indicator constraints, namely $(l \Rightarrow C) \wedge (\neg l \Rightarrow \neg C)$.

According to the visualization of a BNN in Figure 1, Table 1 presents the formal definition of a BNN structure [14, 19]. We have $m-1$ internal blocks, $\text{BLOCK}_1, \ldots, \text{BLOCK}_{m-1}$ that are placed consecutively. Let n_k denote the number of input values to BLOCK_k. The output of the last internal block, $\boldsymbol{x}_m$, is passed to the output block OUTPUT to obtain one of the s labels.

Table 1. BNN structure. A_j and b_j are parameters of the BLIN layer, whereas $\beta_j, \gamma_j, \mu_j, \sigma_j$ are parameters of the BN layer, where μ_j and σ_j correspond to mean and standard deviation, respectively.

Structure of k^{th} internal block, $\text{BLOCK}_k : \{-1,1\}^{n_k} \to \{-1,1\}^{n_{k+1}}$ on $\boldsymbol{x}_k \in \{-1,1\}^{n_k}$	
BLIN	$\boldsymbol{y} = A_k \boldsymbol{x}_k + \boldsymbol{b}_k$, 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	$z_i = \gamma_{k_i}\left(\frac{y_i - \mu_{k_i}}{\sigma_{k_i}}\right) + \beta_{k_i}$, where $\boldsymbol{\beta}_k, \boldsymbol{\gamma}_k, \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k, \boldsymbol{z} \in \mathbb{R}^{n_{k+1}}$. 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, $\text{OUTPUT} : \{-1,1\}^{n_m} \to [1,s]$ on input $\boldsymbol{x}_m \in \{-1,1\}^{n_m}$	
BLIN	$\boldsymbol{w} = A_m \boldsymbol{x}_m + \boldsymbol{b}_m$, 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 $o \in [1,s]$

In this paper, we propose a generalization of the above structure, in order to come up with a similar QNN structure. For this, we first have to define what we mean by *quantization*, similar to the DoReFa-Net method in [24]. Given a

quantization bit-width $\mathsf{bw} \in \mathbb{N}$, we quantize $\mathbb{R}$ into the finite set

$$\mathcal{V}_{\mathsf{bw}} = \left\{ V_{\mathsf{bw}}(q) \;\middle|\; q = 0, \ldots, 2^{\mathsf{bw}} \right\}, \quad \text{where} \quad V_{\mathsf{bw}}(q) = \frac{q}{2^{\mathsf{bw}-1}} - 1,$$

along the threshold values

$$T_{\mathsf{bw}}(q) = V_{\mathsf{bw}}(q) - \frac{1}{2^{\mathsf{bw}}}, \quad \text{where} \quad q = 1, \ldots, 2^{\mathsf{bw}}.$$

For instance, $\mathsf{bw} = 1$ results in a ternary quantization into $\mathcal{V}_{\mathsf{bw}} = \{-1, 0, 1\}$ along the threshold values $-0.5, 0.5$. As another example, $\mathsf{bw} = 2$ quantizes into $\mathcal{V}_{\mathsf{bw}} = \{-1, -0.5, 0, 0.5, 1\}$ along the threshold values $-0.75, -0.25, 0.25, 0.75$. Note furthermore that binarization is a special case of quantization, where $\mathsf{bw} = 0$.

A value $x \in \mathbb{R}$ is quantized by the function [11]

$$\mathrm{quant}_{\mathsf{bw}}(x) = \mathrm{clip}\left(\frac{\mathrm{round}(x 2^{\mathsf{bw}-1})}{2^{\mathsf{bw}-1}}, -1, 1 \right),$$

where $\mathrm{clip}(y, a, b)$ clips the value of y into $[a, b]$, and $\mathrm{round}(\cdot)$ applies rounding half up.

The proposed QNN structure uses quantization (QNT) instead of binarization. Furthermore, the weights in the linear transformation layers (QLIN) can take quantized values. In our approach, we propose to use ternary weights $-1, 0, 1$, to strive for sparse weight matrices and a more straightforward encoding into Boolean constraints. The format definition of this QNN structure is shown in Table 2.

Table 2. QNN structure for quantization bit-width $\mathsf{bw} \in \mathbb{N}$. The QLIN layer applies ternary quantization. The QNT layer uses quantization as activation with respect to the bit-width bw.

Structure of k^{th} internal block, $\textsc{Block}_k : \mathcal{V}_{\mathsf{bw}}^{n_k} \to \mathcal{V}_{\mathsf{bw}}^{n_{k+1}}$ on $\boldsymbol{x}_k \in \mathcal{V}_{\mathsf{bw}}^{n_k}$	
QLIN	$\boldsymbol{y} = A_k \boldsymbol{x}_k + \boldsymbol{b}_k$, where $A_k \in \mathcal{V}_1^{n_{k+1} \times n_k}$ and $\boldsymbol{b}_k, \boldsymbol{y} \in \mathbb{R}^{n_{k+1}}$
BN	$z_i = \gamma_{k_i}\left(\frac{y_i - \mu_{k_i}}{\sigma_{k_i}}\right) + \beta_{k_i}$, where $\boldsymbol{\beta}_k, \boldsymbol{\gamma}_k, \boldsymbol{\mu}_k, \boldsymbol{\sigma}_k, \boldsymbol{z} \in \mathbb{R}^{n_{k+1}}$. Assume $\sigma_{k_i} > 0$.
QNT	$\boldsymbol{x}_{k+1} = \mathrm{quant}_{\mathsf{bw}}(\boldsymbol{z})$ where $\boldsymbol{x}_{k+1} \in \mathcal{V}_{\mathsf{bw}}^{n_{k+1}}$
Structure of output block, $\textsc{Output} : \mathcal{V}_{\mathsf{bw}}^{n_m} \to [1, s]$ on input $\boldsymbol{x}_m \in \mathcal{V}_{\mathsf{bw}}^{n_m}$	
QLIN	$\boldsymbol{w} = A_m \boldsymbol{x}_m + \boldsymbol{b}_m$, where $A_m \in \mathcal{V}_1^{s \times n_m}$ and $\boldsymbol{b}_m, \boldsymbol{w} \in \mathbb{R}^s$
ARGMAX	$o = \mathrm{argmax}(\boldsymbol{w})$, where $o \in [1, s]$

3. Implementing quantized neural networks

Our Python implementation of a QNN is based on a publicly available BNN implementation[1], using the Tensorflow and Keras libraries.

[1] https://github.com/Haosam/Binary-Neural-Network-Keras

First, a method needs to be implemented to quantize not just a single number, but even a matrix of real numbers. Additionally, the method takes the quantization bit-width `bw` as parameter. The source code of this function can be seen in Listing 1.

```
def round_quantize(x, bw):
    q_pow = 2**(bw-1)
    numerator = q_pow * x
    numerator = numerator + K.stop_gradient(K.round(numerator) - numerator)
    return numerator / q_pow
```

Listing 1. Quantization function.

Quantization is a mathematical function that has several points of discontinuity. Although this seems an unimportant detail, that should be taken into consideration since no gradient can be computed in such points. Due to TensorFlow, `stop_gradient` can be used to handle a function of this kind, as it disables gradient calculation.

The `round_quantize` function is called inside the `QDense` layer, which is derived from the `Dense` class of Keras. In addition to the properties inherited from its base class, `QDense` includes the quantization bit-width `bw`, which will later be passed to the `round_quantize` function when being called inside the `call` method of `QDense`, as shown in Listing 2. This function calculates the quantized kernel using the given quantization bit-width.

```
def call(self, inputs):
    output = K.dot(inputs,round_quantize(self.kernel, self.bw))
    if self.use_bias:
        return K.bias_add(output, self.bias)
    if self.activation is not None:
        return self.activation(output)
```

Listing 2. Inside the `QDense` layer.

Listing 3 shows how to start to assemble a sequential QNN model. Note that most layers must be uniquely labeled in order to access their parameters later on.

```
model = Sequential()
model.add(QDense(bw = 1, name='qlayer0', ...))
model.add(BatchNormalization(name='bnlayer0', ...))
model.add(Activation(lambda x: round_quantize(x, quantizationBw)))
```

Listing 3. Structure of the QNN network.

For the sake of formal verification, the parameters of all the `QDense` and batch normalization layers must be extracted. This can easily be done by using the `save_weights` procedure of Keras, to save the stored weights, bias values and other parameters to a file. By using the unique labels of layers, the corresponding parameters can be accessed as shown in Listing 4. Note that since the kernels do not store the quantized weights, we must quantize them after reading from the file. Notice, furthermore, that the quantization bit-width for the kernels is 1.

```
model.save_weights(datafile)
with h5py.File(datafile, 'r+') as hdf:
  kernel0 = round_quantize(np.array(hdf.get('/qlayer0/qlayer0/kernel:0')), 1)
```

```
bias0 = np.array(hdf.get('/qlayer0/qlayer0/bias:0'))
variance0 = np.array(hdf.get('/bnlayer0/bnlayer0/moving_variance:0'))
mean0 = np.array(hdf.get('/bnlayer0/bnlayer0/moving_mean:0'))
beta0 = np.array(hdf.get('/bnlayer0/bnlayer0/beta:0'))
gamma0 = np.array(hdf.get('/bnlayer0/bnlayer0/gamma:0'))
```

Listing 4. Extracing the parameter of the layers `qlayer0` and `bnlayer0`.

4. Encoding quantized internal blocks

In this section, we show how to encode the internal blocks into a set of Boolean constraints. In order to make it easier to distinguish Boolean variables from non-Boolean ones, we will use the $(\cdot)^{\mathsf{bl}}$ notation.

Given the quantization bit-width $\mathsf{bw} \in \mathbb{N}$, let us introduce the simplified notation of threshold constants $T_q := T_{\mathsf{bw}}(q)$ for all $q = 1, \ldots, 2^{\mathsf{bw}}$. The quantized output $o_i \in [-1, 1]$ is represented by a vector $\boldsymbol{o}_i^{\mathsf{bl}} = \left(o_{i,1}^{\mathsf{bl}}, \ldots, o_{i,2^{\mathsf{bw}}}^{\mathsf{bl}}\right)$ of Boolean variables, i.e., $o_{i,q}^{\mathsf{bl}} \in \{0, 1\}$ for all $q = 1, \ldots, 2^{\mathsf{bw}}$. Let $o_{i,q}^{\mathsf{bl}}$ be set to true iff the block's i^{th} output exceeds the threshold value T_q:

$$\gamma_i \frac{\langle \boldsymbol{a}_i, \boldsymbol{x} \rangle + b_i - \mu_i}{\sigma_i} + \beta_i \geq T_q \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}}. \tag{4.1}$$

Here, $\boldsymbol{x}$ denotes the input vector to this internal block, $\boldsymbol{a}_i$ the i^{th} row vector of the kernel A, b_i the bias value, and $\beta_i, \gamma_i, \mu_i, \sigma_i$ the parameters of batch normalization. (4.1) can be reorganized into

$$\langle \boldsymbol{a}_i, \boldsymbol{x} \rangle \circ_{\text{rel}} C_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}}, \tag{4.2}$$

where

$$C_{i,q} = \frac{\sigma_i}{\gamma_i}(T_q - \beta_i) + \mu_i - b_i,$$

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

Optionally, the variables $o_{i,q}^{\mathsf{bl}}$ can be further constrained if this makes propagation faster:

$$o_{i,q+1}^{\mathsf{bl}} \;\Rightarrow o_{i,q}^{\mathsf{bl}} \quad \text{for all} \quad q = 1, \ldots, 2^{\mathsf{bw}}.$$

A quantized input $x_j \in [-1, 1]$ is represented by a vector $\boldsymbol{x}_j^{\mathsf{bl}} = \left(x_{j,1}^{\mathsf{bl}}, \ldots, x_{j,2^{\mathsf{bw}}}^{\mathsf{bl}}\right)$ of Boolean variables. The sum of vector elements can be calculated as $\mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}}$. The actual input value x_j can be calculated:

$$x_j = \frac{\mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}}}{2^{\mathsf{bw}-1}} - 1.$$

Let $\neg \boldsymbol{x}_j^{\mathsf{bl}} = \left(\neg x_{j,1}^{\mathsf{bl}}, \ldots, \neg x_{j,2^{\mathsf{bw}}}^{\mathsf{bl}}\right)$ denote the piecewise negation of vector elements. Now, let us plug each x_j into (4.2), as follows:

$$\begin{aligned} \sum_{j=1}^{n_k} a_{ij} \left(\frac{\mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}}}{2^{\mathsf{bw}-1}} - 1 \right) \circ_{\mathrm{rel}} C_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}} \\ \sum_{j} a_{ij}\ \mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}} \circ_{\mathrm{rel}} C'_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}}, \end{aligned} \tag{4.3}$$

where

$$C'_{i,q} = 2^{\mathsf{bw}-1}\Big(C_{i,q} + \sum_j a_{ij}\Big).$$

Since $a_{i,j} \in \{-1, 0, 1\}$, we can further translate (4.3) to

$$\sum_{j \in J_i^+} \mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}} - \sum_{j \in J_i^-} \mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}} \circ_{\mathrm{rel}} C'_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}}, \tag{4.4}$$

where

$$\begin{aligned} J_i^+ &= \{j \mid a_{ij} > 0\}, \\ J_i^- &= \{j \mid a_{ij} < 0\}. \end{aligned}$$

(4.4) can be further translated to

$$\begin{aligned} \sum_{j \in J_i^+} \mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}} - \sum_{j \in J_i^-} \mathbf{1} \cdot \left(\mathbf{1} - \neg \boldsymbol{x}_j^{\mathsf{bl}}\right) \circ_{\mathrm{rel}} C'_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}} \\ \sum_{j \in J_i^+} \mathbf{1} \cdot \boldsymbol{x}_j^{\mathsf{bl}} + \sum_{j \in J_i^-} \mathbf{1} \cdot \neg \boldsymbol{x}_j^{\mathsf{bl}} \circ_{\mathrm{rel}} D_{i,q} \;\Leftrightarrow\; o_{i,q}^{\mathsf{bl}}, \end{aligned} \tag{4.5}$$

where

$$D_{i,q} = \begin{cases} \lceil C'_{i,q} \rceil + 2^{\mathsf{bw}} |J_i^-|, & \text{if } \gamma_i > 0, \\ \lfloor C'_{i,q} \rfloor + 2^{\mathsf{bw}} |J_i^-|, & \text{if } \gamma_i < 0. \end{cases} \tag{4.6}$$

Note that the left-hand side of (4.5) is a sum of Boolean literals, therefore (4.5) represents a set of *reified Boolean cardinality constraints*.

5. Verification for quantized neural networks

In the previous section, we showed how to transform the QNN blocks into Boolean constraints, which can now fed into a constraint solver, for the sake of formal verification. In this section, we demonstrate how to implement this. Our implementation is written in Python and it leverages a range of different solver packages such as PySAT [12], PySMT [6] or Google's OR-Tools [20]. Our tool is able to run those solvers in parallel, due to applying `ProcessPool` from the module `pathos.multiprocessing` [18].

5.1. Generating bounds and constraints

To implement the encoding of an internal block, we need to generate all the bounds $D_{i,q}$ from (4.6), as shown in Listing 5.

```
quantizationCount = 1 << quantizationBitWidth

D = []
for q in range(quantizationCount):
    C = sigma[k][i] / gamma[k][i] *
        (quantizationBound(q) - beta[k][i]) + mu[k][i] - b[k][i]

    Cprime = (C + sum(A[k][i])) * (quantizationCount >> 1)

    D.append(int(math.ceil(Cprime) if gamma[k][i] > 0 else math.floor(Cprime)))

offset = sum(1 for a in A[k][i] if a < 0) * quantizationCount
D = [d + offset for d in D]
```

Listing 5. Generating bounds for an internal block.

Note that Listing 5 only shows how to generate those bounds for the k^{th} block and its i^{th} output. Of course, this has to be done for each k and each i, thus the corresponding bounds are going to be stored in a 3-dimensional matrix and can be accessed within the vector `D[k][i]`, as Listing 6 shows, which is about generating the constraints (4.5).

```
lits = []
for j in range(len(inputVars[k])):
    if A[k][i][j] > 0:
        lits.extend(inputVars[k][j])
    elif A[k][i][j] < 0:
        lits.extend([solver.negateVar(x) for x in inputVars[k][j]])

solver.addConstraint(Constraint(
    lits = lits,
    relation = Relations.GreaterOrEqual if gamma[k][i] > 0 else Relations.
     LessOrEqual,
    bounds = D[k][i],
    resLits = inputVars[k + 1][i]
))
```

Listing 6. Generating constraints for an internal block.

Note that the class `Constraint` represents the reifed Boolean cardinality constraints (4.5) to add to the underlying solver. It is important to note that a `Constraint` instance represents a set of actual constraints, as a list of bounds is associated with the same left-hand side (`lits`) and relation. Notice furthermore that, via the `resLit` parameter, the value of each inequality is made equivalent with the corresponding input of the subsequent block.

5.2. Solver interface

One of our attempts is to extend the number of available solvers in our tool. For easier usage and addition of the different solver packages, a `Solver` base class was

introduced. It defines an interface through which the derived solver classes can be used uniformly, but it also provides the possibility to handle the solver packages differently. The common interface includes functions to generate Boolean variables, negate them, or feed constraints to the underlying solver. After defining a set of constraints for satisfiability checking, a solver can be called through the interface to solve the problem and to return a satisfying model.

5.2.1. Gurobi's solver interface as example

Each solver package has a different interface. They also differ in the possibility of adding different constraints. In this section, we describe a few issues with Gurobi's Python API and show how to overcome them.

For example, the Gurobi solver lacks the possibility of adding "greater than" and "less than" constraints. As a Boolean cardinality constraint is defined over Boolean variables and integer numbers, "less than" and "greater than" Boolean cardinality constraints can be transformed as follows:

$$\sum_i l_i < c \;\longrightarrow\; \sum_i l_i \leq c-1$$
$$\sum_i l_i > c \;\longrightarrow\; \sum_i l_i \geq c+1.$$

Since the constraints that we got from encoding quantized blocks in Section 4 assign multiple bounds to the same left-hand sides, the above transformation of a "greater than" constraint can be implemented as Listing 7 shows.

```
constraint.relation = Relations.GreaterOrEqual
for i in range(len(constraint.bounds)):
    constraint.bounds[i] += 1
```

Listing 7. The translation of a "greater than" constraint for Gurobi's API.

Another issue that had to be handled with Gurobi's API is the lack of adding reified constraints, or using "not equal to" relation for a constraint. However, Gurobi supports *indicator constraints*. Therefore, a reified constraint $A \Leftrightarrow \sum_i l_i = c$ can be split into two equations:

$$A \Rightarrow \sum_i l_i = c \tag{5.1}$$
$$\neg A \Rightarrow \sum_i l_i \neq c. \tag{5.2}$$

Since Gurobi cannot natively deal with "not equal to" constraints, (5.1) has to be transformed by introducing two new Boolean variables A_1, A_2 as follows:

$$A_1 \Rightarrow \sum_i l_i \leq c$$

$$A_2 \Rightarrow \sum_i l_i \geq c$$
$$A \Rightarrow A_1 \wedge A_2.$$

In a similar way, we transform (5.2) to

$$\neg A_1 \Rightarrow \sum_i l_i \geq c + 1$$
$$\neg A_2 \Rightarrow \sum_i l_i \leq c - 1$$
$$\neg A \Rightarrow \neg A_1 \vee \neg A_2.$$

The above translation can be implemented by calling the `addGenConstrIndicator` method from Gurobi's API, which takes as parameters a Boolean variable *var*, a Boolean value *val*, and a constraint *constr*, and then it adds the indicator constraint $(var = val) \Rightarrow constr$ to the solver. The implementation is shown in Listing 8.

```
leftHandSide = sum(constraint.lits)
for i in range(len(constraint.bounds)):
    if constraint.relation == Relations.Equal:
        [a1, a2] = self.generateVar(2)

        self.model.addGenConstrIndicator(
                a1, True, leftHandSide <= constraint.bounds[i])
        self.model.addGenConstrIndicator(
                a1, False, leftHandSide >= constraint.bounds[i] + 1)

        self.model.addGenConstrIndicator(
                a2, True, leftHandSide >= constraint.bounds[i])
        self.model.addGenConstrIndicator(
                a2, False, leftHandSide <= constraint.bounds[i] - 1)

        self.model.addGenConstrIndicator(
                constraint.resLits[i], True, a1 + a2 == 2)
        self.model.addGenConstrIndicator(
                constraint.resLits[i], False, a1 + a2 < 2)
```

Listing 8. The translation of a reified "equal to" constraint for Gurobi's API.

5.3. Experiments

Our preliminary 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 1200 seconds.

In our experiments, the QNN architecture consisted of 3 internal blocks that contain QLin layers with 200, 100 and 100 neurons, respectively. The quantization bit-width was set to 2. We trained the network on the MNIST dataset [16] with an accuracy of 91%. To process the inputs, we added an additional preprocessing block to the QNN before Block 1. The preprocessing block consisted of a BN layer and a Qnt layer, and it applied quantization to the grayscale MNIST images.

The use case for our experiments was to verify the adversarial robustness of the resulting QNN, meaning that it might have misclassified inputs if we allowed to add perturbation in the range $[-\epsilon, \epsilon]$ to individual input values. For this, we randomly picked 20 images that were correctly classified by the network and then we experimented with three different maximum perturbation values by varying $\epsilon \in \{1, 3, 5\}$. Figure 3 shows the results of our experiments. As the figure suggests, our tool produced the best results when running MiniCARD as an underlying solver. All the benchmark instances were proved to be satisfiable and our tool were able to generate the corresponding perturbation matrices. Notice that MiniCARD timed out for only one input image when $\epsilon = 3$.

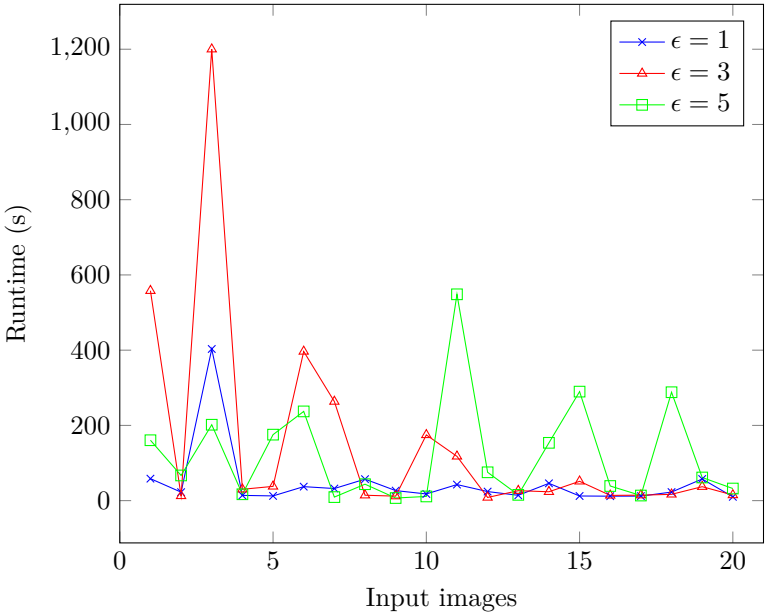

Figure 3. Runtimes of VerBiNe when running MiniCARD on 20 MNIST images with maximum perturbation $\epsilon \in \{1, 3, 5\}$.

6. Summary

In this paper, we proposed a structure of Quantized Neural Networks (QNNs) consisting of blocks of suitable layers. Dense layers use ternary weights to strive for sparse weight matrices, while activation layers apply quantization of arbitrary bit-width. The goal is to make neural networks efficient and robust enough, while making them suitable subjects for logic-based verification. We showed how to implement QNNs in Python, using the Tensorflow and Keras libraries. For the sake of the formal verification of QNNs, we demonstrated how to encode the internal blocks

of a QNN into a set of reified Boolean cardinality constraints. We discussed some aspects of implementing a tool for verifying the encoded QNNs, also in Python, where the constraints that we have specified are to be passed to the underlying solvers. Finally, we reported on the results of our preliminary experiments.

As future work, we will define a Boolean encoding for other types of blocks, including blocks with convolutional layers. We are about finishing the development of our verification tool, after which we will run a thorough experimentation.

References

[1] C. Cheng, G. Nührenberg, H. Ruess: *Verification of Binarized Neural Networks via Inter-Neuron Factoring*, CoRR (2017), arXiv: 1710.03107.

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio: *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*, Journal of Machine Learning Research 18.187 (2018), pp. 1–30.

[12] 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.

[13] 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.

[14] G. Kovásznai, K. Gajdár, N. Narodytska: *Portfolio Solver for Verifying Binarized Neural Networks*, Annales Mathematicae et Informaticae 53 (2021), pp. 183–200, issn: 1787-6117, doi: https://doi.org/10.33039/ami.2021.03.007.

[15] 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.

[16] 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.

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

[18] M. M. McKerns, L. Strand, T. Sullivan, A. Fang, M. A. Aivazis: *Building a Framework for Predictive Science*, CoRR (2012), arXiv: 1202.1056.

[19] 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.

[20] L. Perron, V. Furnon: *OR-Tools*, version 9.3, Google, Mar. 15, 2022, url: https://developers.google.com/optimization/.

[21] 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.

[22] 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.

[23] 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.

[24] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, Y. Zou: *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*, CoRR (2016), arXiv: 1606.06160.