Counter-Example Guided Neural Network Quantization Refinement (CEG4N)

João Batista Pereira Matos Júnior[1], Iury Bessa[1], Edoardo Manino[2], Xidan Song[2], and Lucas C. Cordeiro[2,1]

[1] Federal University of Amazonas, Manaus-AM, Brazil
jbpmj@icomp.ufam.edu.br and iurybessa@ufam.edu.br
[2] Univeristy of Manchester, Machester, United Kingdom
{lucas.cordeiro,eduardo.manino,xidan.song}@manchester.ac.uk

Abstract. Neural networks are essential components of learning-based software systems. However, their high compute, memory, and power requirements make using them in low resources domains challenging. For this reason, neural networks are often quantized before deployment. Existing quantization techniques tend to degrade the network accuracy. We propose Counter-Example Guided Neural Network Quantization Refinement (CEG4N). This technique combines search-based quantization and equivalence verification: the former minimizes the computational requirements, while the latter guarantees that the network's output does not change after quantization. We evaluate CEG4N on a diverse set of benchmarks, including large and small networks. Our technique successfully quantizes the networks in our evaluation while producing models with up to 72% better accuracy than state-of-the-art techniques.

Keywords: Robust Quantization, Neural Network Quantization · Neural Network Equivalence · Counter Example Guided Optimization

1 Introduction

Neural networks (NNs) are becoming essential in many applications such as autonomous driving [6], security, medicine, and business [2]. However, current state-of-the-art NNs often require substantial compute, memory, and power resources, limiting their applicability [9].

In this respect, quantization techniques help reduce the network size and its computational requirements [9,24,16]. Here, we focus on *quantization* techniques, which aim at reducing the number of bits required to represent the neural network weights [16]. A desirable quantization technique produces the smallest neural network possible from the quantization perspective. However, at the same time, quantization affects the functional behavior of the resulting neural network by making them more prone to erratic behavior due to loss of accuracy [18]. For this reason, existing techniques monitor the degradation in the accuracy of the quantized model with statistical measures defined on the training set [16].

However, statistical accuracy measures do not capture the network's vulnerability to malicious attacks. Indeed, there may exist some specific inputs for

which the network performance degrades significantly [19,27,3]. For this reason, we reformulate the goal of guaranteeing the accuracy of a quantized model under the notion of *equivalence* [12,17,11,20]. This formal property requires that two neural network models both produce the same output for every input, thus ensuring that the two networks are functionally equivalent [28,30].

We are the first to explore the combination of quantization techniques and equivalence checking in the present work. Doing so guarantees that the quantized model is functionally equivalent to the original one. More specifically, our main scientific contributions are the following:

- We model the equivalence quantization problem as an iterative optimization-verification cycle.
- We propose CEG4N, a counter-example guided neural network quantization technique that provides formal guarantees of NN equivalence.
- We evaluate CEG4N on both large (ACAS Xu [23] and MNIST [26]) and small (Iris [13] and Seeds [8]) benchmarks.
- We demonstrate that CEG4N can successfully quantize neural networks and produce models with similar or better accuracy than a baseline state-of-the-art quantization technique (up to 72% better accuracy).

2 Preliminaries

2.1 Neural Network

NNs are non-linear mapping functions $f : \mathcal{I} \subset \mathbb{R}^n \rightarrow \mathcal{O} \subset \mathbb{R}^m$ consisting of a set of L linked layers, organized as a *direct graph*. Each layer l is connected with the directly preceding layer $l-1$, *i.e.*, the output of the layer $l-1$ is the input of the layer l. Exceptions are the first and last layers. The first layer is just a placeholder for the input for the NN while the last layer holds the NN function mapping f. A layer l is composed by a matrix of weights $\mathbf{W_l} \in \mathbb{R}^{n \times m}$ and a bias vector $\mathbf{b_l} \in \mathbb{R}^m$.

The output of a layer is computed by performing the combination of an affine transformation, followed by the non-linear transformation on its input $\mathbf{x_l} \in \mathbb{R}^n$(see Eq. (1)). Formally, we can describe the function $y_l : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that computes the output of a layer l as follows:

$$y_l(\mathbf{x_l}) = \mathbf{W_l} \cdot \mathbf{x_l} + \mathbf{b_l} \tag{1}$$

and the function that computes the activated output of a layer l as follows:

$$y_l^{\sigma}(\mathbf{x_l}) = \sigma(y_l(\mathbf{x_l})) \tag{2}$$

where $\sigma : \mathbb{R}^m \rightarrow \mathbb{R}^m$ is the *activation function*. In other words, the output l is the result of the activation function σ applied to the dot product between weight and input, plus the bias. The most popular activation functions are: namely, ReLU, sigmoid (Sigm), and the re-scaled version of the latter known as

hyperbolic tangent(TanH) [10]. We focus on the *rectified linear unit* activation function $ReLU = \max\{0, \mathbf{y_1}\}$.

Considering the above, let us denote the input of a NN with L layers as $\mathbf{x} \in \mathbb{I}$, and $f(x) \in \mathcal{O}$ as the output; thus, we have that:

$$f(\mathbf{x}) = \sigma\left(y_L(\sigma\left(y_{L-1}(...(\sigma\left(y_1(\mathbf{x})\right)))\right)\right)\right) \tag{3}$$

2.2 Quantization

Quantization is the process of constraining high precision values (*e.g.*, single-precision floating-point values) to a finite range of lower precision values (*e.g.*, a discrete set such as the integers) [1,16]. The quantization quality is usually determined by a scalar n (the available number of bits) that defines the lower and upper bounds of the finite range. Let us define quantization as a mapping function $\mathcal{Q}_n : \mathbb{R}^{m\times p} \rightarrow \mathbb{I}^{m\times p}$, formulated as follow:

$$Q\left(n, A\right) = clip\left(\left\lfloor \frac{A}{q(A,n)} \right\rceil, -2^{n-1}, 2^{n-1} - 1\right) \tag{4}$$

where $A \in \mathbb{R}^{m\times\ p}$ denotes the continuous value– notice that A can be a single scalar, a vector, or a matrix; n denotes the number of bits for the quantization, $q(A, n)$ denotes a function that calculates the scaling factor for A in respect to a number of bits n, and $\lfloor\cdot\rceil$ denotes rounding to the nearest integer. Defining the scaling factor (see Eq. 5) is an important aspect of uniform quantization [22,25].

The scaling factor is essentially what divides a given range of real values A into an arbitrary number of partitions. Thus, let us define a scaling factor function by $q_n(A)$, a number of bits (bit-width) to be used for the quantization by n, a clipping range by $[\alpha, \beta]$, the scaling factor can be defined as follow:

$$q(A, n) = \frac{\beta - \alpha}{2^n - 1} \tag{5}$$

The min/max of the signal are often used to determining the clipping range values, *i.e.*, $\alpha = \min A$ and $\beta = \max A$. But as we are using symmetric quantization, the clipping values are defined as $\alpha = \beta = \max([|\min A|, |\max A|])$. In practice, the quantization process can produce an integer value that lies outside the range of $[\alpha, \beta]$. To prevent that, the quantization process will have an additional clip step.

Eq. (6) shows the corresponding de-quantization function, which computes back the original floating-point value. However, we should note that the de-quantization approximates the original floating-point value.

$$\hat{A} = q(A, 2)Q\left(n, A\right) \tag{6}$$

2.3 NN quantization

In this section, we discuss how a convolutional or fully-connected NN layer can be quantized in the symmetric mode. Considering l to be any given layer in a

NN, let us denote $\mathbf{x_l}$, $\mathbf{W_l}$, and $\mathbf{b_l}$ as the original floating-point input vector, the original floating-point weight matrix, and the original floating-point bias vector, respectively, of the layer l. And applying the de-quantization function from Eq. (6), where, we assume that $A = \hat{A}$. Borrowing from notations used in Sections 2.1 and 2.2. We can formalize the quantization of a NN layer l as follows:

$$\begin{aligned} y_1(\mathbf{x_l}) &= \mathbf{W_l} \cdot \mathbf{x_l} + \mathbf{b_l} \\ &\approx q(\mathbf{W_l}, n_l) Q\left(n_l, \mathbf{W_l}\right) \cdot \mathbf{x_l} + q(\mathbf{b_l}, n_l) Q\left(n_l, \mathbf{b_l}\right) \end{aligned} \tag{7}$$

Notice that the bias does not need to be re-scaled to match the scale of the dot product. Since we consider maximum scaling factor between $q(\mathbf{W_l}, n_l)$ and $q(\mathbf{b_l}, n_l)$), both the weight and the bias share the same scaling factor in Eq. (7). With that in mind, the formalization of a NNf in Eq. (3) can be reused to formalize a quantized NN as well.

2.4 NN Equivalence

Let $\mathcal{F}$ and $\mathcal{T}$ be two arbitrary NNs, and let $\mathcal{I} \in \mathbb{R}^n$ be the common input space of the two NNs and $\mathcal{O} \in \mathbb{R}^m$ be their common output space. Thus, NN equivalence verification is the problem of proving that $\mathcal{F}$ and $\mathcal{T}$, or more specifically, their corresponding mathematical functions $f : \mathcal{I} \to \mathcal{O}$, $t : \mathcal{I} \to \mathcal{O}$ are equivalent. In essence, by proving the equivalence between two neural networks, one can prove that both NNs produce the same outputs for the same set of inputs. Currently, the literature reports the following definition of equivalence.

Definition 1 (Top-1-Equivalence [7,30]). *Two NNs f and t are Top-1-equivalent, if* $\arg\max\ f(x) = \arg\max\ t(x)$*, for all* $x \in \mathcal{I}$.

Let us formalise the notion of *Top-1 Equivalence* in first-order logic. This is necessary for the comprehension of the equivalence verification explained in the following sections of the paper. But first, we formalize some essential assumptions for the correctness of the equivalence properties.

Assumption 1 *Let $f(x)$ be the output of the NN $\mathcal{F}$ in real arithmetic (without quantization). It is assumed that* $\arg\max f(x) = y$ *such that* $x \in \mathcal{H}$.

Assumption 2 *Let $f^q(x)$ be the output of the NN $\mathcal{F}$ in a quantized form. There is set of numbers of bits $\mathcal{N}$ such that* $\arg\max f(x) = \arg\max f^q(x) = y$ *for all* $x \in \mathcal{H}$.

Note that the quantization of the NN f that results in the NN $f^q(x)$ depends on the number of bits N. Refer to Eq. (7) to understand the relationship between $\mathcal{N}$ and f^q.

An instance of a equivalence verification is given by a conjunction of constraints on the input $\psi_x(x)$, the output $\psi_y(y)$ and the NNs f and f^q. $\psi(f, f^q, x, y) =$

$\psi_x(x) \rightarrow \psi_y(y)$. We denote $\psi_y(y)$ the equivalence constraint. Let $\bar{x} = x + \hat{x}$ such that $|x + \hat{x}|_\infty \leq \epsilon$, consider $\bar{x} \in \mathcal{H}$ and $y \in \mathcal{G}$. Taking from Definition 1, we have that:

- $\psi_x(x)$ is an equivalence property such that $\psi_x(x) \leftrightarrow \bar{x} \in \mathcal{H}$
- $\psi_y(y)$ is an equivalence property such that $\psi_y(y) \leftrightarrow \arg\max f^q(x) = y$

Note that, to prove the equivalence of f and f^q, one may prove that the property $\psi(f, f^q, x, y)$ holds for any given x and y. This approach may not be feasible. But proving that $\psi(f, f^q, x, y)$ does not hold for some x and y is a more tractable approach. If we do so, we can provide a counter-example.

2.5 Verification of NN properties

In this paper, we use the classic paradigm of SMT verification. In this paradigm, the property to check (e.g., equivalence) and the computational model (e.g., the neural networks) are encoded as a first-order logic formula, which is then checked for satisfiability. Moreover, to keep the problem decidable, SMT restricts the full expressive power of first-order logic to a decidable fragment.

SMT formulas can capture the complex relationship between variables, holding real, integer values and other data types. If it is possible to assign values to such variables that a formula is evaluated as true, then the formula is said to be *satisfiable*. On the other hand, if it's not possible to assign such values, the formula is said to be *unsatisfiable*.

Given a NN $\mathcal{F}$ and its mathematical function f, a set of safe input instances $\mathcal{H} \in \mathbb{R}^n$, and a safe domain $\mathcal{G} \subseteq \mathcal{O}^m$– both defined as a set of constraints, safety verification is concerned with the question of whether there exist an instance $x \in \mathcal{H}$ such that $f(x) \notin \mathcal{G}$. An instance of a safety verification is given by a conjunction of constraints on the input $\psi_x(x)$, the output $\psi_y(y)$ and the NN f. $\psi(f, x, y) = \psi_x(x) \rightarrow \psi_y(y)$ is said to be satisfiable if there exists some $x \in \mathcal{H}$ such that $f(x)$ returns y for the input x and $\psi(f, x, y)$ does not hold.

3 Counter-Example Guided Neural Network Quantization Refinement (CEG4N)

We define *robust quantization (RQ)* to describe the problem of maximizing the quantization of a NN while keeping the equivalence between the original model and the quantized one (see Definition 2). Borrowing from the notations used in Section 2, we formally define RC as follows.

Definition 2 (Robust Quantization). *Let f be the reference NN and $\mathcal{H} \in \mathbb{R}^n$ be a set of inputs instances. We define robust quantization as a process that performs the quantization of f hence resulting in a quantized model f^q such that* $\arg\max f(x) \iff \arg\max f^q(x)\ \forall\ x \in \mathcal{H}$.

From the definition discussed in Section 2.4, we preserve the equivalence between the mathematical functions f and f^q associated with the NNs. In the RC, we shift the focus from the original NN to the quantized NN, *i.e.*, we assume that f is safe (or robust) and use it as a reference to define the safety properties we expect for f^q. By checking the equivalence of f and f^q, we can state that f^q is robust, and therefore, we achieve a *robust quantization*. In more details, consider a NN f with L layers. The quantization of f assumes there is a set $\mathcal{N} = \{n_1, n_2, \cdots, n_L\}$, where $n_l \in \mathcal{N}$ represents the number of bits that should be used to quantize the l-th layer in f. In our robust quantization problem, we obtain a sequence $\mathcal{N}$ for which each $n \in \mathcal{N}$ is minimized (e.g., one could minimize the sum of all $n \in \mathcal{N}$) and the equality between f and f^q is satisfied.

3.1 Robust quantization as a minimization problem

We consider the robust quantization of a NN as an iterative minimization problem. Each iteration is composed of two complementary sub-problems. First, we need to minimize the quantization bit widths, that is, finding a candidate set $\mathcal{N}$. Second, we need to verify the equivalence property, that is, checking if a NN quantized with the bit widths in $\mathcal{N}$ is equivalent to the original NN. If the latter fails, we iteratively return to the minimization sub-problem with additional information. More specifically, we formalize the first optimization sub-problem as follows.

Optimization sub-problem o:

$$\begin{aligned} \textbf{Objective:} \quad & \mathcal{N}^o = \underset{n_1^o,\ldots,n_L^o}{\arg\min} \sum_{l \in \mathbb{N}_{l \leq L}} n_l \\ \textbf{s.t:} \quad & \arg\max f(x) = \arg\max f^q(x),\ \forall\ x \in \mathcal{H}^o_{\text{CE}} \\ & n_l \geq \underline{N}\ \forall\ n_l \in \mathcal{N}^o \\ & n_l \leq \overline{N}\ \forall\ n_l \in \mathcal{N}^o \end{aligned} \tag{8}$$

where f is the mathematical function associated with the NN $\mathcal{F}$ and f^q is the quantized mathematical function associated with the NN $\mathcal{F}$, $\mathcal{H}^o_{\text{CE}}$ is a set of counter-examples available at iteration o. Consider $\underline{N}$ and $\overline{N}$ as the minimum and the maximum bit width allowed to be used in the quantization; these parameters are constant. $\overline{N}$ ensures two things, it gives an upper bound to the quantization bit width, and provides a termination criteria, if a candidate $\mathcal{N}^o$ such that $n_l = \overline{N}$ for every $n_l \in \mathcal{N}^o$, the optimization is stopped because it reached our **Assumption 2**. In particular, our **Assumption 2** ensures the termination of CEG4N, and it is build over the fact that there is a set of $\overline{N}$ for which the quantization introduces a minimal amount of error to NN. In any case, if CEG4N proposes a quantization solution equal to the $\overline{N}$, this solution is verified as well, and in case the verification returns a counter-example, CEG4N finishes with failure. Finally, note that $\mathcal{H}^o_{\text{CE}}$ is an iterative parameter, meaning its value is updated at each iteration o. This is done based on the verification sub-problem (formalized below).

Verification sub-problem o:
In the verification sub-problem o, we check whether the $\mathcal{N}^o$ generated by the optimization sub-problem o satisfies the following equivalence property:

$$\psi(f, f^q, x, y) = \psi_x(x) \rightarrow \psi_y(y)$$

if $\psi_x(x) \rightarrow \psi_y(y)$ holds for the candidate $\mathcal{N}^o$, the optimization halts and $\mathcal{N}^o$ is declared as solution; otherwise, a new counter-example x_{CE} is generated. Iteration $o+1$ starts where iteration o stopped. That is, the optimization sub-problem $o+1$ receives as parameter a set of $\mathcal{H}_{\text{CE}}^{o+1}$ such that $\mathcal{H}_{\text{CE}}^{o+1} = \mathcal{H}_{\text{CE}}^{o} \cup x_{\text{CE}}$.

3.2 The CEG4N framework implementation

We propose CEG4N framework, which is a counterexample-guided optimization approach to solve the robust quantization problem. In this approach, we consider combining two main modules to solve the two sub-problems presented in Section 3.1: the optimization of the bit widths for the quantization and the verification of the NN equivalence. The first module that solves the optimal bit width problem roughly takes in a NN and generates quantized NN candidates. Then, the second module takes in the candidates and verifies their equivalence to the original model.

Figure 1 illustrates the overall architecture of the CEG4N framework. It also shows how each framework's module interacts with the other and in what sequence. The *GA module* is an instance of a Genetic Algorithm. The GA module expects two main parameters, NN and a set of counter-examples $\mathcal{H}_{\text{CE}}$ We can also specify a maximum number of generations the algorithm is allowed to run and lower and upper bounds to restrict the possible number of bits. Once the GA module produces a candidate, that is, a sequence of bit widths, for each layer of the neural network, CEG4N generates the C-Abstraction code for the original model and the quantized candidate and then checks their equivalence. Each check for this equivalence property is exported to a unique verification test case. Then, it triggers the execution of the verifier for each verification test case and awaits the verifier output. Here, *Verifier module* is an instance of a formal verifier (i.e., a Bounded Model Checker (BMC), namely, ESBMC [15]). This step is done sequentially, meaning each verification is run once the last verification terminates.

Once all verification test cases terminate, CEG4N collect and process all outputs and checks whether any counter-example has been found. If so, it updates the set of counter-examples $\mathcal{H}_{\text{CE}}$ and triggers the GA module execution again, thus initiating a new iteration of CEG4N. If no counter-example is found, CEG4N considers the verification successful and terminates the quantization process outputting the found solution.

We work with two functional versions of the NN. The GA module works with a functional NN written in Python, while the verifier module works with a functional version of the NN written in C. The two models are equivalent since they share the same parameters; the python model loads the parameters

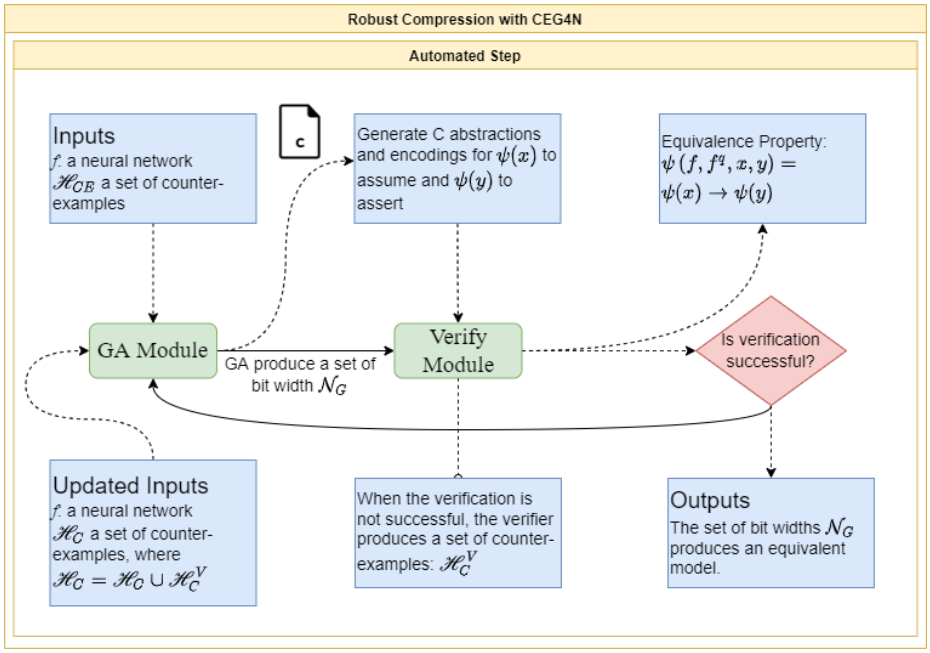

Fig. 1. CEG4N architecture overview, highlighting the relationship between the main modules, and their inputs and outputs.

to a framework built over Pytorch [29]. The C version loads the weights into a framework designed and developed in C to work correctly with the verifier idioms and annotations. We provide more details regarding the C implementations of the NNs in Section A.2.

4 Experimental Evaluation

This section describes our experimental setup and benchmarks, defines our objectives, and presents the results.

4.1 Description of the Benchmarks

We evaluate our methodology on a set of feedforward NN classification models extracted from the literature [10,23,26]. We chose these specific ones based on their popularity in previous NN robustness and equivalence verification studies [30,10]. Additionally, we include a few other NN models to cover a broader range of NN architectures (e.g., NN size, number of neurons).

ACAS Xu The airborne collision avoidance system for unmanned aircraft ACAS Xu dataset [23] is derived from 8 specifications (features boundaries and expected outputs). ACAS Xu features are sensor data indicating the speed, present course of the aircraft, and the position and speed of any nearby intruder aircraft. An ACAS Xu NN is expected to give appropriate navigation advisories for a given input sensor data. The expected outputs indicate that either the aircraft is clear-of-conflict, or it should take soft or hard turns to avoid the collision. We evaluated CEG4N on 5 pre-trained NNs, each containing 8 layers and 300 ReLU nodes each. The pre-trained NNs were obtained from the VNN-COMP2021 [5] benchmarks[3]

MNIST MNIST is a popular dataset [26] for image classification. The dataset contains 70,000 gray-scale images with uniform size of 28x28 pixels, where the original pixel values from the integer range $[0, 255]$ are rescaled to the floating-point range $[0, 1]$. We evaluated CEG4N on two NNs with 2 layers, one with 10 ReLU nodes each and another with 25 and 10 ReLU nodes. The NNs followed the architecture of models described by the work of Eleftheriadis et al. [10].

Seeds The Seeds dataset [8] consists of 210 samples of wheat grain belonging to three different species, namely Kama, Rosa and Canadian. The input features are seven measurements of the wheat kernel geometry scaled between [0,1]. We evaluated CEG4N on 2 NNs, containing 1 layer, one containing 15 ReLU nodes, and the other containing 2 ReLU nodes. Both NNs were trained for the CEG4N evaluation.

Iris The Iris flower dataset [13] consists of 50 samples from three species of Iris flower (*Iris setosa, Iris virginica and Iris versicolor*). The dataset is a popular benchmark in machine learning for classification, and the data is composed of records of real value measurements of the width and length of sepals and petals of the flowers. The data was scaled to [0,1]. We evaluated CEG4N on 2 NNs, one of them containing 2 layers with 20 ReLU nodes and the other having only one layer with 3 ReLU nodes. Both NNs were trained for the CEG4N evaluation.

4.2 Setup

Genetic Algorithm. As explained in Section 3.1, we quantize the NNs with a NSGA-II Genetic Algorithm module. We set the upper and lower bounds for the allowed bit widths to 2 and 52 in all experiments. The lower bound was chosen because 2 is the first valid integer that does not break our quantization formulas. The upper bound was chosen to match the significand of the double-precision float format IEEE 754-1985 [21]. The upper bound value could be higher depending on the precision of weights parameters of the NN, as the scaling

[3] The pre-trained weight for the ACAS Xu benchmarks can be found in the following repository: `https://github.com/stanleybak/vnncomp2021`

factor could lead the quantization to large integer values. However, as we wanted the framework to work on every NN in our experimentation setup without further steps, we restricted the clipping range to a comfortable number to avoid integer overflow.

Furthermore, we allow the GA to run for 110 generations for each layer in the NN. This number of generations was defined after extensive preliminary tests, which confirmed that GA could reach the optimal solution in most cases (see Table 3 in Appendix A.4). Lastly, we randomly select the initial set of counter-examples $\mathcal{H}$ from the benchmark set of each case study. The samples in $\mathcal{H}$ do not necessarily have to be *counter-examples*, and any valid concrete input can be specified. Our choice is justified by the practical aspect of using samples from the benchmark set.

Equivalence Properties. One input sample was selected for each output class and used to define the equivalence properties. Due to the high dimensional number of the features in the MNIST study case, we proposed a different approach when specifying the equivalence properties for the equivalence verification. We considered three different approaches: 1) one in which we considered all features in the input domain; 2) another one in which we considered only a subset of 10 out of the 784 features in the input domain; 3) a last one in which we considered only a subset of 4 out of the 784 features in the input domain. The subset of features in cases 2 and 3 was randomly selected.

Availability of Data and Tools. Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and results of our evaluation are available on a supplementary web page `https://zenodo.org/record/6791964`.

4.3 Objectives

Considering the benchmarks given in Section 4.1, our evaluation has the following two experimental goals:

EG1 (**robustness**) Show that the CEG4N framework can generate robust quantized NNs.
EG2 (**accuracy**) Show that the quantized NNs do not have a significant drop in accuracy compared to other quantization techniques.

4.4 Results

In our first set of experiments, we want to achieve our first experimental goal **EG1**. We want to show that our technique CEG4N can successfully generate

quantized NNs that are verifiably equivalent to the original NNs. As a secondary goal, we want to perform an empirical scalability study to help us evaluate the computational demands for quantizing and verifying the equivalence of NNs models. Our findings are summarized in Table 1.

Table 1. Summary of the CEG4N executions, including the models, number of features, the number of bits per layer, and the status.

Model	Features	Equivalence Properties	Iterations	Bits	Status
iris_3	4	3	1	4, 3	*completed*
seeds_2	7	3	1	4, 3	*completed*
seeds_15	7	3	1	4, 2	*completed*
acasxu_1	5	6	1	6, 8, 7, 7, 9, 7, 6	*completed*
acasxu_2	5	7	1	10, 9, 9, 9, 7, 7, 10	*completed*
acasxu_3	5	7	1	5, 9, 10, 7, 8, 8, 5	*completed*
acasxu_4	5	7	1	8, 9, 14, 9, 10, 10, 7	*completed*
acasxu_5	5	7	1	6, 12, 8, 8, 10, 10, 10	*completed*
	5	10	1	4, 3	*completed*
mnist_10	10	10	1	4, 3	*completed*
	784	10	0	4, 3	*timeout*
	5	10	1	3, 3	*completed*
mnist_25	10	10	1	3, 3	*completed*
	784	10	0	3, 3	*timeout*

All the CEG4N runs that were completed successfully took only 1 iteration to find a solution. However, we observed that four of the CEG4N attempts to find a solution for MNIST models resulted in a timeout. We attribute this observation to a mix of factors. First is the high number of features in the MNIST problem. Second, the network's overall architecture requires many arithmetic operations to compute the model's output. Finally, we also observed that it took only a few minutes for CEG4Nto find a solution to the Iris, Seeds, and Acas Xu benchmarks. In contrast, on MNIST, it took hours to either find a solution or fail with a timeout.

These results answer our **EG1**: overall, these experiments show that CEG4N can successfully produce robust quantized models. Although, one should notice that for larger NNs models, scalability should be a point of concern due to our verifier stage.

In our second set of experiments, we want to achieve our second experimental goal **EG2**. We primarily want to understand the impact of the quantization performed by CEG4N on the accuracy of the NNs compared to other quantization techniques. Due to our research's novelty, no existing techniques lend themselves to a fair comparison. For this reason, we take a recent post-training

quantization technique called GPFQ [31] and modify it to our needs. GPFQ [31] is a greedy path-following quantization technique that also produces quantized models with floating/double-precision values. It works by iterating over each layer of the NN and quantizing each neuron sequentially. More specifically, a greedy algorithm minimizes the error between the original neural output and the quantized neuron.

Table 2 summarizes the accuracy of the models quantized using CEG4N and GPFQ. Note that we do not report the accuracy of the Acas Xu models because the original training and test datasets are not public.

Table 2. Comparison of Top-1 accuracy for NNs quantized using CEG4N and GPFQ

Model	Method	Ref Acc (%)	Quant Acc (%)	Acc Drop (%)
iris_3	CEG4N	93.33	83.33	10.0
	GPFQ		23.33	70.0
seeds_2	CEG4N	88.09	85.71	2.38
	GPFQ		64.28	23.81
seeds_15	CEG4N	90.04	85.71	4.33
	GPFQ		40.47	49.57
mnist_10	CEG4N	91.98	86.7	5.28
	GPFQ		91.29	0.69
mnist_25	CEG4N	93.68	92.57	1.11
	GPFQ		92.59	1.09

Our findings show that the highest drops in accuracy happen on the Iris benchmark (10% for CEG4N and 70% drop for GPFQ). In contrast, the lowest drops in accuracy happen on mnist_25 for CEG4N and on mnist_10 for GPFQ. Overall, the accuracy of models quantized with CEG4N are better on the Iris and Seeds benchmarks, while the accuracy of models quantized with GPFQ are better on the mnist benchmarks, but only by a small margin. Our understanding is that GPFQ shows high drops in accuracy for smaller NNs because the number of neurons in each layer is small. As GPFQ focuses on each neuron individually, it may not be able to find a good global quantization.

These results answer our **EG2**: overall, these experiments show that CEG4N can successfully produce quantized models with superior or similar accuracy to other state-of-the-art techniques.

4.5 Limitations

Although we showed in our evaluation that the CEG4N framework can generate a quantized neural network while keeping the equivalence between the original NN and the quantized NN, we note that the architecture of the NN used in the

evaluation does not fully reflect state-of-the-art NN architectures. The NNs used in our evaluation have few layers and only hundreds of ReLU nodes, while state-of-the-art NNs may have hundreds of layers and thousands of ReLU nodes. The main bottleneck is state-of-the-art verification algorithms, which currently do not scale to large neural networks. As it is, our technique could only quantized 80% of the NN in our experimental evaluation.

In addition, the field of research on NN equivalence is relatively new and there is no well-established set of benchmarks that works in this field could benefit from [10]. Furthermore, our work is the first to propose a framework that mixes NN quantization and NN equivalence verification. There is no comparable methodology in the literature we could compare our approach with.

5 Conclusion

We presented a new method for NN quantization, called CEG4N, a post-training NN quantization technique that provides formal guarantees of NN equivalence. This approach leverages a counter-example guided optimization technique, where an optimization-based quantizer produces quantized model candidates. A state-of-the-art C verifier then checks these candidates to prove the equivalence of the quantized candidates and the original models or refute that equivalence by providing a counter-example. This counter-example is then passed back to the quantized to guide it to search for a feasible candidate.

We evaluate the CEG4N method on four benchmarks, including large models (ACAS Xu and MNIST) and smaller models (Iris and Seeds). We successfully demonstrate the application of the CEG4N for NN quantization, where it could successfully quantize the networks while producing models with up to 72% better accuracy than state-of-the-art techniques. However, CEG4N can only handle a restricted set of NNs models, and further work needs to scale the CEG4N applicability on a broader set of NNs models (e.g., NNs models with a more significant number of layers and neurons and higher numbers of input features).

For future work, we could explore other quantization techniques, which are not limited to search-based quantization and other promising equivalence verification techniques using a MILP approach [30] or an SMT-based approach [10]. Combining different quantization and equivalence verification techniques can enable CEG4N to achieve better scalability and quantization rates. Another interesting future work relates to the possibility of mixing quantization approaches that generate quantized models, which operate entirely on integer arithmetic; this can potentially improve the verification step scalability of the CEG4N.

Acknowledgment

The work is partially funded by EPSRC grant EP/T026995/1 entitled “EnnCore: End-to-End Conceptual Guarding of Neural Architectures” under *Security for all in an AI-enabled society.*

References

1. Abate, A., Bessa, I., Cattaruzza, D., Cordeiro, L.C., David, C., Kesseli, P., Kroening, D.: Sound and automated synthesis of digital stabilizing controllers for continuous plants. In: Frehse, G., Mitra, S. (eds.) Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, Pittsburgh, PA, USA, April 18-20, 2017. pp. 197–206. ACM (2017). https://doi.org/10.1145/3049797.3049802
2. Abiodun, O.I., Jantan, A., Omolara, A.E., Dada, K.V., Mohamed, N.A., Arshad, H.: State-of-the-art in artificial neural network applications: A survey. Heliyon **4**(11), e00938 (2018). https://doi.org/https://doi.org/10.1016/j.heliyon.2018.e00938, https://www.sciencedirect.com/science/article/pii/S2405844018332067
3. Albarghouthi, A.: Introduction to neural network verification. ArXiv **abs/2109.10317** (2021)
4. Bai, J., Lu, F., Zhang, K., et al.: Onnx: Open neural network exchange. https://github.com/onnx/onnx (2019)
5. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (vnn-comp 2021): Summary and results (2021)
6. Bojarski, M., del Testa, D.W., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., Zieba, K.: End to end learning for self-driving cars. ArXiv **abs/1604.07316** (2016)
7. Büning, M.K., Kern, P., Sinz, C.: Verifying equivalence properties of neural networks with relu activation functions. In: Simonis, H. (ed.) Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12333, pp. 868–884. Springer (2020). https://doi.org/10.1007/978-3-030-58475-7_50, https://doi.org/10.1007/978-3-030-58475-7_50
8. Charytanowicz, M., Niewczas, J., Kulczycki, P., Kowalski, P.A., Łukasik, S., Żak, S.: Complete Gradient Clustering Algorithm for Features Analysis of X-Ray Images, pp. 15–24. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
9. Cheng, Y., Wang, D., Zhou, P., Zhang, T.: A survey of model compression and acceleration for deep neural networks. ArXiv **abs/1710.09282** (2017)
10. Eleftheriadis, C., Kekatos, N., Katsaros, P., Tripakis, S.: On neural network equivalence checking using smt solvers. ArXiv **abs/2203.11629** (2022)
11. Esser, S.K., Appuswamy, R., Merolla, P., Arthur, J.V., Modha, D.S.: Backpropagation for energy-efficient neuromorphic computing. In: Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 28. Curran Associates, Inc. (2015), https://proceedings.neurips.cc/paper/2015/file/10a5ab2db37feedfdeaab192ead4ac0e-Paper.pdf
12. Farabet, C., LeCun, Y., Kavukcuoglu, K., Martini, B., Akselrod, P., Talay, S., Culurciello, E.: Large-scale fpga-based convolutional networks (2011)
13. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Annals of Eugenics **7**, 179–188 (1936)
14. Gadelha, M.R., Menezes, R.S., Cordeiro, L.C.: ESBMC 6.1: automated test case generation using bounded model checking. Int. J. Softw. Tools Technol. Transf. **23**(6), 857–861 (2021). https://doi.org/10.1007/s10009-020-00571-2
15. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: Esbmc 5.0: An industrial-strength c model checker. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 888–891 (2018). https://doi.org/10.1145/3238147.3240481

16. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W., Keutzer, K.: A survey of quantization methods for efficient neural network inference. ArXiv **abs/2103.13630** (2022)
17. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural network. ArXiv **abs/1506.02626** (2015)
18. Hooker, S., Courville, A.C., Dauphin, Y., Frome, A.: Selective brain damage: Measuring the disparate impact of model pruning. ArXiv **abs/1911.05248** (2019)
19. Huang, X., Kroening, D., Kwiatkowska, M., Ruan, W., Sun, Y., Thamo, E., Wu, M., Yi, X.: Safety and trustworthiness of deep neural networks: A survey. ArXiv **abs/1812.08342** (2018)
20. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. ArXiv **abs/1609.07061** (2017)
21. IEEE: Ieee standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (2019). https://doi.org/10.1109/IEEESTD.2019.8766229
22. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A.G., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. CoRR **abs/1712.05877** (2017), http://arxiv.org/abs/1712.05877
23. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J.: Policy compression for aircraft collision avoidance systems. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–10 (2016). https://doi.org/10.1109/DASC.2016.7778091
24. Kirchhoffer, H., Haase, P., Samek, W., Müller, K., Rezazadegan-Tavakoli, H., Cricri, F., Aksu, E., Hannuksela, M.M., Jiang, W., Wang, W., Liu, S., Jain, S., Hamidi-Rad, S., Racapé, F., Bailer, W.: Overview of the neural network compression and representation (nnr) standard. IEEE Transactions on Circuits and Systems for Video Technology pp. 1–1 (2021). https://doi.org/10.1109/TCSVT.2021.3095970
25. Krishnamoorthi, R.: Quantizing deep convolutional networks for efficient inference: A whitepaper. CoRR **abs/1806.08342** (2018), http://arxiv.org/abs/1806.08342
26. LeCun, Y., Cortes, C.: The mnist database of handwritten digits (2005)
27. Liu, C., Arnon, T., Lazarus, C., Barrett, C.W., Kochenderfer, M.J.: Algorithms for verifying deep neural networks. Found. Trends Optim. **4**, 244–404 (2021)
28. Narodytska, N., Kasiviswanathan, S.P., Ryzhyk, L., Sagiv, S., Walsh, T.: Verifying properties of binarized deep neural networks. In: AAAI (2018)
29. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
30. Teuber, S., Buning, M.K., Kern, P., Sinz, C.: Geometric path enumeration for equivalence verification of neural networks. 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI) (Nov

2021). https://doi.org/10.1109/ictai52525.2021.00035, http://dx.doi.org/10.1109/ICTAI52525.2021.00035
31. Zhang, J., Zhou, Y., Saab, R.: Post-training quantization for neural networks with provable guarantees. arXiv preprint arXiv:2201.11113 (2022)

A Appendices

A.1 Implementation of NNs in Python.

The NNs were built and trained using the Pytorch library [29]. Weights of the trained models were then exported to the ONNX [4] format, which can be interpreted by Pytorch and used to run predictions without any compromise in the NNs performance.

A.2 Implementation of NNs abstract models in C.

In the present work, we use the C language to implement the abstract representation of the NNs. It allows us to explicitly model the NN operations in their original and quantized forms and apply existing software verification tools (e.g., ESBMC [14]). The operational C-abstraction models perform double-precision arithmetic. Although, we must notice that the original and quantized only diverge on the precision of the weight and bias vectors that are embedded in the abstractions code.

A.3 Encoding of Equivalence Properties

Suppose, a NN F, for which $x \in \mathcal{H}$ is a safe input and $y \in \mathcal{G}$ is the expected output of f the input. We now show how one can specify the equivalence properties. For this example, consider that the function f can produce the outputs of F in floating-point arithmetic, while fq produces the outputs of F in fixed-point arithmetic (*i.e.* quantization). First, the concrete NN input x is replaced by a non-deterministic one, which is achieved using the command **nondet_float** from the ESBMC.

Listing 1.1. Definition of concrete and symbolic input domain in *EBMC*.

```
float x0 = -1.0;
float x1 = 1.0;
float s0 = nondet_float();
float s1 = nondet_float();
```

Listing 1.2. Definition of input constraints in *EBMC*.

```
const float EPS = 0.5;
__ESBMC_assume(x0 - EPS <= s0 && s0 <= x0 + EPS);
__ESBMC_assume(x1 - EPS <= s1 && s1 <= x1 + EPS);
```

Listing 1.3. Definition of output constraints in *EBMC*.

```
__ESBMC_assert(f(s0, s1) == fq(s0, s1));
```

A.4 Genetic Algorithm Parameters Definition

In Table 3, we report a summary of experiments conducted to tune the parameters of the Genetic Algorithm, more precisely, the number of generations. For example, a NN with 2 layers would require a brute force algorithm to search for 52^2 combinations of bits widths for the quantization. Similarly, a NN with 7 layers would require a brute force algorithm to search for 52^7 combinations of bits widths. We conducted a set of experiments where we ran the GA one hundred times with a different number of generations options ranging from 50 to 1000. In addition, we fixed the population size to 5. From our findings, the GA needs about 100 to 110 generations per layer to find the optimal bit width solution for each run.

Table 3. Summary of experiments for tuning Genetic Algorithm Parameters.

Number of Layers	Generations	Population	Percentage of optimal solutions
7	800	5	100
7	750	5	100
7	700	5	98
7	50	5	0
2	250	5	100
2	200	5	100
2	150	5	96
2	50	5	30