Quantitative Symbolic Robustness Verification for Quantized Neural Networks

Mara Downing[1] William Eiers[2] Erin DeLong[1] Anushka Lodha[1] Brian Ozawa Burns[1] Ismet Burak Kadron[1] and Tevfik Bultan[1]

[1] University of California Santa Barbara {maradowning,erindelong,anushkalodha, brianozawaburns,kadron,bultan}@ucsb.edu
[2] Stevens Institute of Technology weiers@stevens.edu

Abstract. With the growing prevalence of neural networks in computer systems, addressing their dependability has become a critical verification problem. In this paper, we focus on quantitative robustness verification, i.e., whether small changes to the input of a neural network can change its output. In particular, we perform quantitative symbolic analysis, where the goal is to identify how many inputs in a given neighborhood are misclassified. We target quantized neural networks, where all values in the neural network are rounded to fixed point values with limited precision. We employ symbolic execution and model counting to achieve quantitative verification of user-defined robustness properties where the verifier will report not only whether the robustness properties are satisfied for the given neural network, but also how many inputs violate them. This measure enables comparison of non-robust networks by assessing the level of robustness, which is not possible with existing quantized network verifiers. We implement and evaluate our approach as a tool called *VerQ*^{2}. To the best of our knowledge, *VerQ*^{2} is the first quantitative verifier for quantized neural networks.

Keywords: Formal Verification · Model Counting · Neural Networks.

1 Introduction

Neural networks are becoming increasingly common in safety critical domains like medicine and automotive industry. For safety critical applications, traditional techniques for evaluating effectiveness of neural networks, such as accuracy on a previously unseen data set, are not sufficient. It is necessary to evaluate the dependability of neural networks—for example, checking if (potentially adversarial) small changes to the input can change the output of a network, and, furthermore determining how many inputs exist within a perturbation region that can trigger an unexpected output.

A full precision neural network uses floating point values, which is computationally intensive and may not be feasible in cases where storage space or processing power is limited [5]. Quantized networks address this limitation by

This research is supported by the NSF under Awards #2124039 and #2008660.

using fixed point numbers. These networks can thus be implemented using less storage, and can also be computed faster [22]. The most extreme example of this type of network is a binary neural network, where each value is 1 or 0. Quantized networks are commonly used in mobile, embedded, and IoT devices where memory and power are limited [22].

In the safety critical healthcare domain, the use of machine learning models is expanding [28] including systems that are expected to make and execute a decision without physician involvement, such as a device that is implanted into a patient's body. The FDA maintains strict requirements for device approval [1]. Additionally, implanted devices have strict power and size requirements which make quantized networks a valuable method of storing and executing neural networks in the medical domain.

Thus, automated verification of quantized neural networks is a critically important area of research. In this paper, we focus particularly on quantitative verification of quantized networks. Given a correctly classified input and an allowed perturbation around that input, traditional robustness verification evaluates whether or not any inputs exist in the region that are classified differently (incorrectly). Quantitative robustness verification goes further to count how many inputs in that region are classified differently. As we mention in our related work discussion, there are traditional verifiers for floating point networks and quantized networks, and there are quantitative verifiers for floating point networks and binary precision networks. However, to the best of our knowledge, this paper is the first to address quantitative robustness for quantized networks with greater than binary precision.

In this paper, we present a quantitative verifier for quantized neural networks, capable of handling different levels of precision. Our major contributions, implemented in our tool *VerQ*^{2} (VERifier for Quantitative robustness of Quantized neural networks), are:

- A quantitative robustness verification approach for quantized neural networks based on symbolic execution and model counting.
- Improvements to constraint solving during symbolic execution for neural networks based on 1) Abstract symbolic execution, and 2) Model generation at symbolic execution tree nodes.
- Translation rules from fixed point arithmetic computations to integer constraints.
- Experimental evaluation of *VerQ*^{2} and empirical comparison of model counters on constraints generated by neural networks.

We test our quantitative robustness verification tool *VerQ*^{2} on networks trained from two medical datasets from the UCI Machine Learning Repository [16]. Our experiments show the techniques we present improve the performance of quantitative symbolic execution over our baseline implementation and perform better than sampling-based approaches including PROVERO [8].

Motivating Example. Let us look at two networks, both trained with the Parkinson's dataset [16, 27] to detect Parkinson's disease from voice data. The networks

take in inputs with 22 input features, each one corresponding to a different measurement gathered from the voice of a patient. Network A has one hidden layer of size 60, and an accuracy of 84.21%. Network B has two hidden layers of size 15, and an accuracy of 89.47%. We evaluate robustness of these two networks for a robustness region using data generated from a patient who has Parkinson's disease, where two of the 22 input features are perturbed and can take on any possible value within the expected input range.

Within the given perturbation region, both networks classify some inputs incorrectly. For a traditional verifier, this would be the extent of the robustness check results. However, with our quantitative verifier $VerQ^2$, we can further show that out of 1089 possible inputs in the perturbation region (33 allowed values per perturbed input feature), network A misclassifies only 12, whereas network B misclassifies 273. We can alternately analyze an average of multiple robustness regions using the quantitative robustness definition we provide in this paper (Section 2.1). Our quantitative robustness definition provides a robustness value between 0 and 1 (1 corresponds to 100% robustness—no input in the robustness region violates the robustness property). When we compare 11 robustness regions for networks A and B using $VerQ^2$, we find network A has an average robustness of 0.904, whereas network B has an average robustness of 0.954. Depending on the importance of an individual input, we may decide either to prioritize overall higher robustness or higher robustness in a key instance when choosing which network to use. Finally, we can also compare the least robust input from a set of inputs—network A has an input with 0.427 robustness whereas for network B, for the set of inputs we analyzed, the minimum robustness is 0.749. Quantitative robustness analysis enables us to make these types of comparisons among networks, which are not possible with traditional robustness analysis.

Related Work. There is a significant amount of prior work on non-quantitative verification of full precision neural networks [13, 14, 17, 18, 26, 31] and there is a limited amount of prior work on quantitative verification of full precision neural networks [8, 12, 21, 29]. These employ a variety of techniques, including formal verification, sampling, and abstraction.

One of the techniques used for scalability in symbolic verification of full-precision networks is the approximation of floating point computations using real values. However, this approach can lead to incorrect verification results [25]. Our approach avoids this type of erroneous analysis by avoiding real approximations of the quantized values and taking into account how the values will behave with rounding (Section 3.2).

For quantitative verification of full precision networks, one approach is to use statistical sampling methods to obtain a probabilistically sound result for quantitative robustness [8, 12]. We choose one such approach [8] for experimental comparison as it uses higher precision in published analysis.

There is also some prior work on verification methods for binarized neural networks [3, 9, 24, 36]. These have a computationally simpler task compared to higher precision quantized network verifiers such as ours.

To the best of our knowledge, there is no prior work on quantitative verification for quantized networks. In terms of traditional (non-quantitative) verification of quantized networks, there are some verifiers that use SMT constraint solving [20, 23]. However, as these are traditional verifiers, the information they can provide is limited. We discuss this drawback in Section 2 and show how our quantitative verification approach offers more valuable results.

As *VerQ*^{2} is the first tool to produce quantitative robustness results for quantized networks, experimental comparison with existing tools is limited. Our approach is faster and can handle larger networks than [21], which computes exact counts for constraints on floating point networks. For comparison, we implement an algorithm proposed for quantitative verification of full precision networks [8] (with modifications for quantized networks), and show that our approach performs better in Section 4.

2 Quantitative Robustness Formalization

We use $\mathbb{F}$ to denote the set of all floating point numbers, and $\mathbb{I}$ to denote the set of fixed point numbers, with subscripts q and d indicating the total bit length and the number of fractional bits, e.g., $\mathbb{I}_{q,d}$.

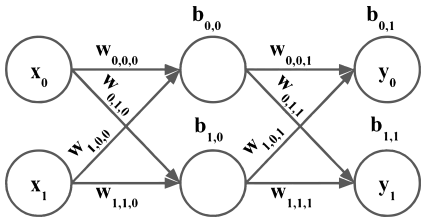

Fig. 1: A feedforward network with two input features (x_0, x_1), two outputs (y_0, y_1).

Fig. 1 shows a small example network where weights are marked along their respective arrows, biases are marked above respective neurons. A neural network $\mathcal{N}$ takes an input X consisting of N features, $\langle x_0, \cdots, x_{N-1}\rangle$, has K hidden layers where each hidden layer k with size J_k consists of neurons $h_{0,k}, \cdots, h_{(J_k-1),k}$, and returns values of J output neurons $y_0, \cdots, y_{J-1}$. In a full precision network, $x_0, \cdots, x_{N-1} \in \mathbb{F}$, $y_0, \cdots, y_{J-1} \in \mathbb{F}$, and all $h \in \mathbb{F}$. For computing the values of the hidden and output neurons, the connections between each neuron in different layers have weights $w_{i,j,k}$ where the ith neuron in kth layer and jth neuron in $(k+1)$th layer is connected, and each neuron i in layer j has bias value $b_{i,j}$. We use the *ReLU* activation function in our model: $ReLU(x) = \max(x, 0)$. For classification tasks, the output neuron with the highest value determines the class—if y_j has the highest value, the classification is class j.

The equations below describe how the values are computed in a non-quantized (using floating point numbers, $\mathbb{F}$) neural network:

$$h_{j,0} = \ ReLU\left(\Sigma_{i=0}^{n} w_{i,j,0} x_i + b_{j,0}\right) \tag{1}$$

$$h_{j,k} = \ ReLU\left(\Sigma_{i=0}^{n} w_{i,j,k} h_{i,(k-1)} + b_{j,k}\right) \tag{2}$$

$$y_j = \ \Sigma_{i=0}^{n} w_{i,j,K} h_{i,K-1} + b_{j,K} \tag{3}$$

Quantization maps variables of the network (neuron values, weights, bias values, etc.) to a limited precision. We use $L_{min}, L_{max} \in \mathbb{I}$ to denote the limits

of the quantized values (minimum and maximum possible values represented by the chosen fixed point size): $L_{min} = -(2^{q-d-1})$ and $L_{max} = 2^{q-d-1} - 2^{-d}$ for quantization $\mathbb{I}_{q,d}$. When working with quantized networks, the computation of nodes must not only take into account the ReLU activation functions, but also the rounding of results to maintain limited precision and avoid overflow. Due to the higher precision to avoid overflow during the computation, the ReLU piece of the above equations (1) and (2) is computed as follows:

$$ReLU(expr) = \begin{cases} 0, & \text{if } expr \leq 0 \\ L_{max}, & \text{if } expr \geq L_{max} \\ expr, & \text{otherwise} \end{cases} \tag{4}$$

Additionally, Equation (3) is replaced with the following for quantized networks:

$$y_j = \begin{cases} L_{min}, & \text{if } \Sigma_{i=0}^{n} w_{i,j,K} h_{i,K-1} + b_{j,K} \leq L_{min} \\ L_{max}, & \text{if } \Sigma_{i=0}^{n} w_{i,j,K} h_{i,K-1} + b_{j,K} \geq L_{max} \\ \Sigma_{i=0}^{n} w_{i,j,K} h_{i,K-1} + b_{j,K}, & \text{otherwise} \end{cases} \tag{5}$$

Due to the multiplication of fixed points, results of $\Sigma_{i=0}^{n} w_{i,j,k} h_{i,k-1} + b_{j,k}$ will contain double the original fractional bits. The rounding to account for this will be shown in Section 3.2.

2.1 Quantitative Robustness

In neural network verification, local robustness is measured by checking if any small perturbations made to the input change the output (classification result). If there exists such a perturbation that changes the output, then the network is not robust on that input. However, this yes/no answer does not give any information about how many of the perturbations change the output. For example, in Fig. 2, both examples would be determined not robust for region 2 by a traditional verifier, whereas a quantitative verifier can distinguish the levels of robustness for these two networks.

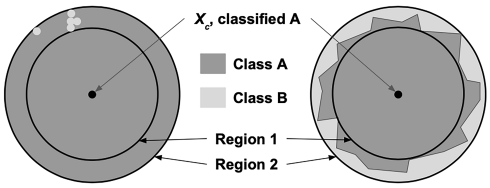

Fig. 2: Two hypothetical perturbation regions for the same input for two different networks. Both networks are robust for region 1 but not region 2. However, the network on the right is less robust than the network on the left.

We provide a definition for quantitative robustness which measures the proportion of inputs within a given perturbation region which do not change the output. For a given neural network $\mathcal{N}$ and a perturbation region containing input X_c within the region with known correct classification, we define the quantitative robustness measure $R(\mathcal{N}, S_{PerturbRegion})$ as follows:

$$R(\mathcal{N}, S_{PerturbRegion}) = |S_{RobustSet}|/|S_{PerturbRegion}| \quad \text{where}$$
$$S_{RobustSet} = \{\tilde{X} \mid \arg\max \mathcal{N}(\tilde{X}) = \arg\max \mathcal{N}(X_c) \wedge \tilde{X} \in S_{PerturbRegion}\}$$

In the definition above, $S_{PerturbRegion}$ denotes the set of all inputs within the perturbation region, and $S_{RobustSet}$ denotes the subset of those where the output of $\mathcal{N}$ does not change. We call remaining inputs potentially adversarial inputs, and we define $S_{AdversarialSet}$ as follows: $S_{AdversarialSet} = S_{PerturbRegion} \setminus S_{RobustSet}$. This is a general form of robustness definition that can represent attacks such as one or two-pixel attacks for images, or general L_∞ ball constraints over the input [32, 33].

A network with a higher number of misclassified inputs (i.e., larger $|S_{AdversarialSet}|$) in a given perturbation region is less robust (and thus more prone to adversarial attacks) than a network with fewer misclassified inputs in the same region.

3 Quantitative Symbolic Robustness Verification

In this section, we present our quantitative symbolic verification approach for quantized neural networks. The two major techniques we use for quantitative verification of neural networks are symbolic execution and model counting. Symbolic execution is a verification technique in which a program is evaluated by constructing a tree of all of the possible paths through the program, where each node in the tree corresponds to a branch condition in the program [6]. A symbolic state and path condition are recorded at each node of the tree; the symbolic state keeps track of the value of each variable at that point in the program and the path condition keeps track of all of the constraints necessary to reach that point in the program. We denote the path condition as a set of constraints C_p, comprised of clauses $c_0, \cdots, c_{t-1}$. At a given branch, c_t is the new clause being introduced. It is possible that at any branch point, one or more of the possible paths may be infeasible, and if a path is infeasible there is no need to traverse further down. The feasibility of a path is checked with a constraint solver, which checks if the path constraints at each node are satisfiable.

The next technique that we use is model counting. A model counting constraint solver returns the number of solutions to a satisfiable constraint, or 0 if the constraint is unsatisfiable. Using symbolic execution and model counting one can determine which execution paths are more likely than others [19].

3.1 Symbolic Analysis for Quantitative Robustness

Our quantitative verifier uses symbolic execution to explore all possible paths of a neural network and model counting to compute the quantitative robustness measure. In this paper we focus only on networks with ReLU (Rectified Linear Unit) activation functions, which is a common activation function [26]. We use Z3's linear integer arithmetic SMT solver [15] to evaluate the path conditions and determine if each branch is feasible. The fixed point values are converted into integers, and we create SMT formulas that are equivalent to the fixed point conditions they are describing (Section 3.2). We chose this encoding, rather than bit-vectors [10], to allow for the use of linear integer arithmetic model counters.

Algorithm 1 QuantSymRobustness(C_{in}, C_{out}, $\mathcal{N}$)
▷ Symbolic analysis for computing quantitative robustness of a neural network
▷ Calls QuantSymExec and a model counter (Count)

Input: C_{in}: set of all user constraints on inputs, derived from $S_{PerturbRegion}$; C_{out}: set of all user constraints on outputs, representing the robustness constraints from $S_{RobustSet}$; $\mathcal{N}$: the neural network under analysis
Output: Quantitative robustness metric R

1: $PerturbRegionSize \leftarrow$ Count(C_{in}) ▷ Computing $|S_{PerturbRegion}|$
2: $RobustSetSize \leftarrow$ QuantSymExec($\mathcal{N}$, C_{in}, C_{out}) ▷ Computing $|S_{RobustSet}|$
3: **return** $RobustSetSize/PerturbRegionSize$ ▷ Computing $|S_{RobustSet}|/|S_{PerturbRegion}|$

We use syntax C_{in} and C_{out} to represent constraints on input perturbations and expected output, respectively. We use a model counting constraint solver on C_{in} to get the total number of possible inputs within that region. Then, we use symbolic execution to capture all possible behaviors of the neural network $\mathcal{N}$ for all inputs that satisfy C_{in}.

We show the overall quantitative verification approach in Algorithm 1, with symbolic execution detailed in Algorithm 2.

The symbolic execution process described in Algorithm 2 is as follows:

1. On line 1, a symbolic expression is created for the node value: the result of multiplying weights and adding the bias, before the ReLU is applied.
2. Lines 3–20 show the branching that occurs for all nodes except the last output node. There are three possible branches—*expr* could be $\leq$ the lower limit L_l (0 for ReLU activated nodes, L_{min} for output nodes), *expr* could be $\geq$ the upper limit L_{max}, or *expr* could be between the two limits. In each case, the stored expression for the node is updated and a recursive call is made to continue exploring the tree given that outcome for the value of *node*. The if statement in lines 16–19 includes the addition of constraints to the path constraint which make sure Expr(*node*) is constrained to the proper result of rounding *expr* (Section 3.2).
3. Lines 22–35 mimic 3–20 in structure, but the gathered constraint is conjoined with C_{out} and a call is made to Count to get a count of how many distinct input vectors ($\vec{X}$) satisfy the gathered constraint.

The helper functions employed in Algorithm 2 are as follows: NewSymVar() creates a new symbolic variable name unique to the current node of the tree; Next(*node*) returns the next node from the network, nodes are ordered by layer (from beginning to end), and from 0 to j ascending within each layer; Expr(*node*) returns the symbolic expression associated with the node, and can be used to change the symbolic expression (this accesses and modifies the symbolic state); isOutput(*node*) returns True only if the node is an output node; isLast(*node*) returns True only if the node is the last node in the order; ConstructExpr is used to compute the multiplication of previous layer nodes by weights and add the bias value, and is shown in Algorithm 4.

We use four different model counting tools to obtain the number of satisfying solutions to a constraint. Three of these are model counters: ABC [4],

Algorithm 2 QuantSymExec($\mathcal{N}$, C_p, C_{out}, *node*)
▷ Quantitative symbolic execution of a neural network
▷ Called by QuantSymRobustness (Algorithm 1); Calls IsSat (Algorithms 3, 5), CreateExpr (Algorithm 4), and Count; Uses helper functions NewSymVar, Next, Expr, isOutput, isLast

Input: C_p: Current path constraint on symbolic input values; C_{out}: Constraint on output nodes; $\mathcal{N}$: the neural network under analysis; *node*: current node with indices j, k
Output: Number of robust inputs within the perturbation region

1: $expr \leftarrow$ConstructExpr(j, k)
2: **if** $\neg$isLast(*node*) **then**
3: $RobSS \leftarrow 0$
4: $L_l \leftarrow 0$
5: **if** isOutput(*node*) **then**
6: $L_l \leftarrow L_{min}$
7: **end if**
8: **if** IsSat(C_p, $expr < 2^d L_l + 2^{d-1}$) **then** ▷ $expr_{rounded} \leq L_l$
9: Expr(*node*) $\leftarrow L_l$
10: $RobSS \leftarrow RobSS +$ QuantSymExec($\mathcal{N}$, $C_p \wedge expr < 2^d L_l + 2^{d-1}$, C_{out}, Next(*node*))
11: **end if**
12: **if** IsSat(C_p, $expr \geq 2^d L_{max} - 2^{d-1}$) **then** ▷ $expr_{rounded} \geq L_{max}$
13: Expr(*node*) $\leftarrow L_{max}$
14: $RobSS \leftarrow RobSS +$ QuantSymExec($\mathcal{N}, C_p \wedge expr \geq 2^d L_{max} - 2^{d-1}, C_{out},$ Next(*node*))
15: **end if**
16: **if** IsSat(C_p, $expr \geq 2^d L_l + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1}$) **then** ▷ $expr_{rounded} > L_l \wedge expr_{rounded} < L_{max}$
17: Expr(*node*) $\leftarrow$ NewSymVar() ▷ create a new symbolic variable for *node*
18: $RobSS \leftarrow RobSS +$ QuantSymExec($\mathcal{N}$, $C_p \wedge expr \geq 2^d L_l + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1} \wedge expr < 2^d$Expr$(node) + 2^{d-1} \wedge expr \geq 2^d$Expr$(node) - 2^{d-1}$, C_{out}, Next(*node*))
19: **end if**
20: **return** *RobSS*
21: **else**
22: $RobSS \leftarrow 0$
23: **if** IsSat(C_p, $expr < 2^d L_{min} + 2^{d-1}$) **then** ▷ $expr_{rounded} \leq L_l$
24: Expr(*node*) $\leftarrow L_l$
25: $RobSS \leftarrow RobSS +$ Count$(C_p \wedge expr < 2^d L_{min} + 2^{d-1} \wedge C_{out})$
26: **end if**
27: **if** IsSat(C_p, $expr \geq 2^d L_{max} - 2^{d-1}$) **then** ▷ $expr_{rounded} \geq L_{max}$
28: Expr(*node*) $\leftarrow L_{max}$
29: $RobSS \leftarrow RobSS +$ Count$(C_p \wedge expr \geq 2^d L_{max} - 2^{d-1} \wedge C_{out})$
30: **end if**
31: **if** IsSat(C_p, $expr \geq 2^d L_{min} + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1}$) **then** ▷ $expr_{rounded} > L_l \wedge expr_{rounded} < L_{max}$
32: Expr(*node*) $\leftarrow$ NewSymVar() ▷ create a new symbolic variable for *node*
33: $RobSS \leftarrow RobSS +$ Count$(C_p \wedge expr \geq 2^d L_l + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1} \wedge expr < 2^d$Expr$(node) + 2^{d-1} \wedge expr \geq 2^d$Expr$(node) - 2^{d-1} \wedge C_{out})$
34: **end if**
35: **return** *RobSS*
36: **end if**

Barvinok [34], and LattE [7]. The fourth is Z3 [15], which can be used to count satisfying models by generating a satisfying model, adding the negation of the model to the constraint, and looping until the constraint is unsatisfiable. We describe the model counters in more detail in Section 3.4.

3.2 Fixed Point Rounding Constraints

As discussed in Section 2, after the multiplication of node values by weights, the result has double the fractional bits, and must be rounded. We translate

the fixed-point inequality expressions (used when deciding how the ReLU impacts each internal node value) into equivalent integer expressions for symbolic execution of the network.

The fixed point values themselves are translated to integers—for example, a concrete fixed point value 0010.1100 (2.75) can be represented as a concrete integer value 00101100 (44) by eliminating the decimal point. We use this approach in transforming fixed point computations to equivalent integer constraints during symbolic execution of the network.

For the following formulas, d will represent the number of fractional bits in the chosen fixed point representation and $expr \in \mathbb{I}_{q,2d}$ will represent the value to be rounded. The rounding rule is as follows, with & representing bitwise AND operation, and $\gg$ representing arithmetic right shift:

$$expr_{rounded} \in \mathbb{I}_{q,d} = \begin{cases} expr \gg d, & \text{if } expr \ \& \ (2^d - 1) < 2^{d-1} \\ (expr \gg d) + 1, & \text{if } expr \ \& \ (2^d - 1) \geq 2^{d-1} \end{cases} \tag{6}$$

The following four equations provide equivalences between the expression we wish to evaluate, using the rounded result of $expr$, and the equivalent expression without using the rounded result.

$$\begin{aligned} expr_{rounded} < x &\Leftrightarrow expr < 2^d x - 2^{d-1} \\ expr_{rounded} \leq x &\Leftrightarrow expr < 2^d x + 2^{d-1} \\ expr_{rounded} > x &\Leftrightarrow expr \geq 2^d x + 2^{d-1} \\ expr_{rounded} \geq x &\Leftrightarrow expr \geq 2^d x - 2^{d-1} \end{aligned} \tag{7}$$

It is also necessary during the symbolic execution of the network to use $expr_{rounded}$ as an argument in future layer computations. This is handled by setting $expr_{rounded}$ equal to a new symbolic variable n, and then using n in any place $expr_{rounded}$ would appear:

$$expr_{rounded} = n \Leftrightarrow expr < 2^d n + 2^{d-1} \wedge expr \geq 2^d n - 2^{d-1} \tag{8}$$

3.3 Constraint Solving Optimizations

The most time consuming computation during symbolic execution of neural networks is in evaluating the satisfiability of path constraints C_p (which we do using Z3 [15]). Therefore, we implement a few strategies to help optimize this computation. The basic constraint solving algorithm (before optimization) is shown in Algorithm 3, where we conservatively return the result as satisfiable if the constraint solver returns unknown. Note that this does not result in imprecision during symbolic execution since the constraint is preserved. It ensures that we do not eliminate paths that might be feasible but where the constraint solver is unable to prove a definite result at that stage of symbolic execution.

Abstract Symbolic Execution. The first optimization we have implemented is abstract symbolic execution, in which an abstract state is kept alongside the typical symbolic state. In this abstract state, each variable can have one of eight

Algorithm 3 IsSat$_{ORIG}$(C_p,c_t)
▷ Path constraint checking (simplest version)
▷ Calls an SMT solver (SMTSolve)

Input: C_p: prior path constraint; c_t: new constraint to be added to the path constraint
Output: Satisfiability of $C_p \wedge c_t$

```
1: if SMTSolve(C_p ∧ c_t) = SAT ∨ SMTSolve(C_p ∧ c_t) = UNKNOWN then
2:     return SAT
3: else
4:     return UNSAT
5: end if
```

values {⊥, -, 0, +, -0, -+, 0+, ⊤} indicating what is known about the variable's sign, which are arranged in a complete lattice with the partial order ⊂ such that ⊥ is the meet of all elements of the lattice and ⊤ is the join of all elements of the lattice. We show the Hasse diagram for this lattice in Figure 3.

We chose this particular abstract domain due to the ReLU activation functions: the ReLU choice is determined by the sign of the input, and the ReLU determines the sign of its output. These abstract values are updated alongside the symbolic values held in the symbolic state; at any given program point, the abstract value for each variable will be an over-approximation of what is known about its symbolic value.

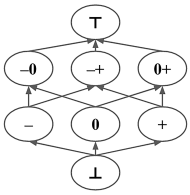

Fig. 3: Lattice for the abstract domain used in abstract symbolic execution.

Before checking any SMT formula for satisfiability, the new clause c_t is checked abstractly using its abstract counterpart $c_{t,a}$, and if $c_{t,a}$ is unsatisfiable, then the branch is infeasible and there is no need to invoke Z3. If $c_{t,a}$ is a tautology, (e.g., $- < +$), $C_p \wedge c_t$ is satisfiable and c_t is not added to the path constraint. Only if $c_{t,a}$ is undetermined then Z3 is invoked on the full constraint.

The method we use to construct symbolic expressions for internal nodes can be augmented to construct these abstract values as well, as shown in Algorithm 4. The three lines we add specifically for this abstract value computation are marked with "Abstract"; without these lines, the function computes solely the symbolic values necessary for symbolic execution. The abstract satisfiability check is added as lines 1–2 and 13–14 in Algorithm 5.

Model Generation and Checking. In the symbolic execution tree, we define model m to be a model that satisfies the path constraint of the node under consideration and m_{prev} as the model that satisfies the path constraint of its parent node. When checking the satisfiability of a node's path constraint, first we check if m_{prev} satisfies that constraint—if so, the constraint is satisfiable and m is set to be m_{prev}. If not, or if there is no m_{prev} available, we evaluate using Z3 and if it is satisfiable generate a new model m for this node. The parent model m_{prev} must satisfy one of the new clauses (together, the clauses cover the entire solution space) which marks that clause as satisfiable. Note that checking if a model

Algorithm 4 ConstructExpr(j, k)
▷ Computes $\Sigma_{i=0}^{n} w_{i,j,k} h_{i,(k-1)} + b_{j,k}$
▷ Uses helper function AbsVal to access and modify the abstract values associated with nodes and concrete values, and AbsEval to compute an abstract mathematical operation on two abstract values.

Input: j: index of node within layer; k: index of layer

1: $expr \leftarrow$Multiply(Expr$(h_{0,k-1}), w_{0,j,k})$
2: AbsVal$(expr) \leftarrow$AbsEval(AbsVal$(h_{0,k-1})$ × AbsVal$(w_{0,j,k}))$ ▷ Abstract
3: **for** i in range$(1,n)$ **do**
4: $expr \leftarrow$Add($expr$, Multiply(Expr$(h_{i,k-1}), w_{i,j,k})$)
5: AbsVal$(expr) \leftarrow$ AbsEval(AbsVal$(expr)$ +
6: AbsEval(AbsVal$(h_{i,k-1})$ × AbsVal$(w_{i,j,k}))$) ▷ Abstract
7: **end for**
8: $expr \leftarrow$ Add$(expr, b_{j,k})$
9: AbsVal$(expr) \leftarrow$AbsEval(AbsVal$(expr)$ + AbsVal$(b_{j,k}))$ ▷ Abstract
10: **return** ($expr$, AbsVal$(expr)$)

Algorithm 5 IsSat$_{FINAL}(C_p, c_t, c_{t,a}, m_{PREV})$
▷ Path constraint checking
▷ Calls an SMT solver (SMTSolve) and a model generation tool (GetModel)

Input: C_p: set of all path constraints from prior branches, c_t: new path constraint to check, m_{prev}: model that satisfies C_p, $c_{t,a}$: equivalent abstract constraint to c_t (details of the construction of $c_{t,a}$ are provided in the appendix).
Output: Satisfiability of $C_p \wedge c_t$

1: **if** $c_{t,a}$ = Tautology **then return** SAT ▷ Next constraint C_p, passes model m_{prev}
2: **else if** $c_{t,a}$ = Sat **then**
3: **if** $m_{prev} \models c_t$ **then return** SAT ▷ Next constraint $C_p \wedge c_t$, passes model m_{prev}
4: **else**
5: **if** SMTSolve$(C_p \wedge c_t)$ = SAT **then**
6: $m \leftarrow$ GetModel$(C_p \wedge c_t)$
7: **return** SAT ▷ Next constraint $C_p \wedge c_t$, passes model m
8: **else if** SMTSolve$(C_p \wedge c_t)$ = UNKNOWN **then**
9: **return** SAT ▷ Next constraint $C_p \wedge c_t$, no model
10: **else return** UNSAT
11: **end if**
12: **end if**
13: **else return** UNSAT
14: **end if**

satisfies a given constraint is faster to compute than checking satisfiability of a constraint. This approach uses concrete values during symbolic execution similar to concolic execution [30].

This model generation and check is added as lines 3 and 6 of Algorithm 5.

Overall Constraint Solving Algorithm. Algorithm 5 incorporates both abstract symbolic execution and model generation as discussed above.

3.4 Model Counting Approaches

We use two model counting techniques in *VerQ*^{2}: symbolic and constraint-loop model counting.

Symbolic Model Counting. Symbolic model counters ABC [4], Barvinok [34], and LattE [7] compute the full model count for constraints without explicitly enumerating all solutions. In *VerQ*^{2} we use the model counting constraint solver ABC for part of the model counting. ABC is an automata-based model counter which constructs a finite-state automaton characterizing the set of solutions to a constraint. We also test LattE and Barvinok, both of which compute the model count using Barvinok's algorithm [11].

Constraint-Loop Model Counting. In this approach, a satisfiability solver is used iteratively to find the solutions to a constraint formula F by first solving F (using an SMT-solver such as Z3) to get the model m, then re-solving $F \wedge \neg m$ ($\neg m$ indicates that m cannot be a solution). This iteration continues until either no more solutions exist or the solver returns UNKNOWN or times out. In general, if $|F|$ is the model count of F, then the solver is invoked $|F|$ times. We name this approach constraint-loop model counting. This approach still uses symbolic constraint solving just like symbolic model counting, but it iteratively produces one model at a time and must run in a loop to produce a count.

Model Counting Implementation. In our quantitative verification approach, there are two places where model counting is needed. The first is before symbolic execution, to obtain a model count of the input constraints (C_{in}) so that the robustness can be calculated as a proportion of the total number of perturbed inputs. The second is at the leaves of the symbolic execution tree, where path constraints are conjoined with C_{out}: at each leaf, we count satisfying solutions to $C_p \wedge C_{out}$.

For the first model count, the constraints are relatively small. For the second model count, the constraints are generated by symbolic execution to represent the behavior of the network, and can be large and complex, which impacts the cost of model counting. However, the initial model count can be leveraged to allow an inverse model count ($|S_{AdversarialSet}|$) to compute robustness, which for a fully or nearly fully robust network can yield small model counts from the leaves. This property can aid the performance of constraint-loop model counting. This constitutes counting solutions to $C_p \wedge \neg C_{out}$ at each leaf.

4 Experimental Evaluation and Discussion

In our experimental evaluation we investigate the following research questions:

RQ1: Which of four integer model counting approaches is the best choice for counting the constraints generated by neural networks?

RQ2: Do the optimizations we propose improve the symbolic verification time for neural networks?

RQ3: Does quantitative symbolic robustness verification of neural networks produce results faster than brute force testing?

RQ4: Does the quantitative symbolic robustness verification approach we propose in this paper and implement in *VerQ*^{2} perform better than the existing tool Provero [8]?

Table 1: Quantitative robustness results for different inputs computed by *VerQ*^{2} for a network trained from the Parkinson's dataset with 2 hidden layers, size 15.

$\lvert S_{PerturbRegion}\rvert$	$\lvert S_{AdversarialSet}\rvert$	Exact	R	Explanation
3,373,232,128	0	Yes	1	The network is fully robust for the region.
2,951,578,112	374	Yes	0.9999999	There are exactly 374 misclassifications in the region.
3,855,122,432	10,179	No	0.9999974	There are at least 10,179 misclassifications in the region.

We trained neural networks using Tensorflow [2] on three different datasets described below, all obtained from the UCI Machine Learning Repository [16]. The networks are trained with full precision, and then converted to fixed-point. All tests use values in $\mathbb{I}_{8,4}$.

Dataset Specifics: The Iris dataset [16] contains four real-valued input variables (normalized to the [0,1] range), and three output classifications. We use this smaller dataset for comparison to explain our choice of model counting strategies, without the need to run slower model counters on large networks for comparison. *The Parkinson's dataset* [16, 27] contains 22 real-valued input variables (normalized to the [-1,1] range) and two output classifications. *The Wisconsin Breast Cancer dataset: abbrev. Cancer* [16, 35] contains 30 real-valued input variables (normalized to the [-1,1] range) and two output classifications. Accuracies of all of the tested networks are presented in our code repository.[3]

VerQ^2 *Output:* Our tool has two ways in which it may produce a robustness result—it will either report the robustness R as an exact result or as a sound upper bound. To demonstrate the output produced by *VerQ*^{2}, we present a few examples in Table 1, all from the same network.

Comparison of Model Counters: We first compare 2 model counting approaches for the initial count of the user's input constraints: constraint-loop model counting via Z3, and symbolic model counting via ABC. The results are shown in Figure 4a. These tests use ten constraints of each size from the Iris constraint set and only compare the time taken to complete the model count.

A radius of ∞ indicates that values are bounded by the range the inputs were normalized to when training the network.

These results are intuitive—the constraint-loop approach with Z3 needs to call Z3 once per model, so a larger count is slower. However, as ABC does not use this loop approach, all of these constraints can be counted quickly. For all future tests, we use ABC for the initial model count.

For the leaf counts, we also use the Iris networks and all constraints are of the form of a two input feature attack, where both input features can be any value within the normalized range. Results are shown in Figure 4b. The results are an average of 10 tests with a 600s time bound each (ABC can exceed this

[3] All of the data from the experiments in this paper is available at https://github.com/mara-downing/ver-q2

(a) Comparison of constraint-loop and symbolic model counting for input constraints of various sizes for the Iris dataset.

Constraint Parameters		Time (s)	
# inputs perturbed	radius	Z3	ABC
4	0.1	2.92	**0.01**
4	0.2	80.38	**0.01**
1	∞	0.22	**0.01**
2	∞	2.80	**0.01**

(b) Comparison of constraint-loop and symbolic model counting for the leaf counts of 2 input feature perturbations of the Iris networks.

Network Parameters			Z3		ABC	
# HL	HL Size	Accuracy	# E	Time (s)	# E	Time (s)
1	20	73.33	10	6.53	10	122.92
1	30	73.33	10	13.55	6	371.80
1	40	60.00	10	28.44	0	895.28
1	50	73.33	10	17.39	7	331.48
1	60	93.33	10	19.09	8	357.78
1	70	100.00	10	25.48	2	964.88
2	10	93.33	8	30.42	10	65.02
2	15	93.33	10	14.09	10	49.24
2	20	100.00	10	47.77	10	81.14
2	25	100.00	9	73.44	10	149.66
2	30	100.00	8	94.28	10	164.80
2	35	100.00	3	111.22	10	169.82
3	5	100.00	10	16.23	10	32.63
3	10	93.33	9	86.54	10	145.18
3	15	100.00	1	110.08	10	166.61
3	20	100.00	7	84.23	10	173.01
		Average:	**8.44**	**48.67**	8.31	265.08

Fig. 4: Model counting performance comparison for input and leaf constraints.

since time is checked after each counting task is completed). The # E column indicates how many are exact counts. Incomplete counts are a sound upper bound on robustness measure R using Z3, as Z3 is counting misclassifications, and a sound lower bound on R using ABC. Z3 is able to complete all of the tests faster than ABC. Both tools report some incomplete counts. We additionally tested the leaf counts using the Barvinok and LattE model counters, but both were unable to solve simple leaf constraints in under an hour.

These experiments answer **RQ1**, identifying the best tool choice for both places where model counting is required: ABC performs better than Z3 for the initial count, and Z3 performs better than ABC for the leaf counts. All further experiments use the model counters in this configuration.

Evaluating Effectiveness of Optimization Strategies: For this set of experiments, we used the networks trained on the Iris dataset [16]. Each test uses 2 perturbed input features, each allowed to take on any value in the normalized range. Results are shown in Fig. 5, where HL stands for Hidden Layer(s). The categories are *Base*: Time with no optimizations, *Abs*: Time with abstract symbolic execution only, *MG*: Time with model generation only, and *All*: Time with all optimizations.

For all tests in Fig. 5, both abstract symbolic execution and model generation show improvement over the base solving time. The combination of the two shows an even larger improvement than either one alone. The effectiveness of these optimizations increases with a higher # of internal nodes, which is expected as more nodes means more branch points which can benefit from optimization.

We additionally tested these improvements on networks trained from the Parkinson's dataset. We allow 3 perturbed input features, which can take on any

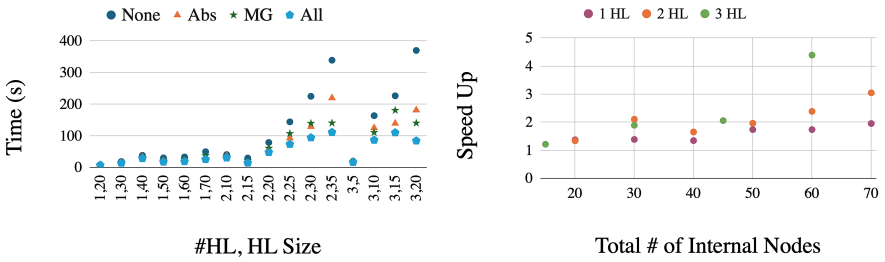

Fig. 5: Comparison of constraint solving optimization strategies on the Iris networks. Left shows times for all four levels of optimization, right shows speed up (as in *value* times speed up) caused by both optimizations.

value in the normalized range. We remove any tests where both optimized and unoptimized verification time out. These tests show an average 1.43x speedup.

Finally, we tested these improvements on networks trained on the Cancer dataset. We allow 5 perturbed input features, which can take on any value in the normalized range. Additionally, we allow one comparative perturbation—one specific input value must be greater than another. We remove any tests where both optimized and un-optimized verification time out. These tests show an average 1.13x speedup.

These results answer **RQ2**, showing that our optimizations produce improvements to symbolic verification time. Furthermore, our optimizations are more effective with increasing network size.

Comparing VerQ^2 with Random Sampling and Exhaustive Enumeration: In this section, we show that our approach can perform better than random sampling without replacement and exhaustive concrete enumeration.

For this section, *exhaustive concrete enumeration* refers to the approach where all valid quantized inputs within the perturbation region are tested. Meanwhile, *random sampling without replacement* (once an element has been sampled, it cannot be chosen again) functions very similarly, but is constrained by a time bound rather than a sample number and the order in which samples are taken is randomized. *Exhaustive concrete enumeration* can thus achieve an exact robustness result by taking the time to check every input, whereas *random sampling without replacement* can find a number of correctly classified and incorrectly classified samples (incorrectly classified divided by $|S_{PerturbRegion}|$ forms a sound upper bound on the robustness R).

We begin with a set of experiments comparing our approach to *exhaustive concrete enumeration*. Our results are shown in Table 2. For this table, we take 160 tests (10 tests per network size, 16 networks) and divide them into three categories by result, corresponding to the three rows of the table. The first two rows indicate that *VerQ*^{2} obtained an exact robustness result for R and are split by whether or not $R = 1$. The last row indicates that *VerQ*^{2} obtained a sound upper bound on R.

Table 2: Comparison of $VerQ^2$ evaluation time with Exhaustive Concrete Enumeration for 16 Parkinson's networks, 10 constraints each.

	# Tests	$VerQ^2$ time (s)	Exhaustive Enum time (s)
$VerQ^2$ exact, $R = 1$	114	55.51	80,820.98
$VerQ^2$ exact, $R < 1$	5	73.50	90,262.19
$VerQ^2$ sound upper bound	41	1,087.67	91,522.52

Time reported for $VerQ^2$ is an average of all tests in that category, whereas time reported for *exhaustive concrete enumeration* is an extrapolation using the average time per input multiplied by the total number of inputs in the perturbation region. We used the same 16 network sizes used in Figure 4b, trained on the Parkinson's dataset [16, 27], 10 perturbation regions per network, and a 30 minute timeout. Each constraint allows 11 input features to be perturbed with a radius of 0.2.

In Table 2 we see that in cases where $VerQ^2$ can obtain an exact result, it is 3 orders of magnitude faster than *exhaustive concrete enumeration*. To further analyze the 41 cases for which $VerQ^2$ produces a sound upper bound, we construct an additional experiment in which *random sampling without replacement* is given the exact amount of time as $VerQ^2$ took for each given test to produce as many misclassifications as possible (if $VerQ^2$ found an exact result for a test in 2000ms, *random sampling without replacement* is given 2000ms for that test).

This random sampling strategy iteratively chooses inputs from $S_{PerturbRegion}$ at random, runs them in the network, and records whether or not they are classified as expected until the specified time limit. Results are shown in Fig. 6.

With these results, we answer **RQ3** affirmatively. Moreover, $VerQ^2$ performs better than random sampling even excluding cases where the tested region is fully robust (where $VerQ^2$ obviously outperforms random sampling).

Comparing $VerQ^2$ with PROVERO*:* We compare $VerQ^2$ with the sampling based tool PROVERO [8]. Given a threshold θ of proportion of adversarial (misclassified) inputs within a perturbation region, PROVERO can report with a degree of confidence measured by parameters η and δ whether or not the proportion of adversarial inputs is above or below a threshold θ. η is an additive precision on the threshold θ and δ is the level of probabilistic certainty necessary to consider a threshold proved or disproved. PROVERO is not specifically designed for our

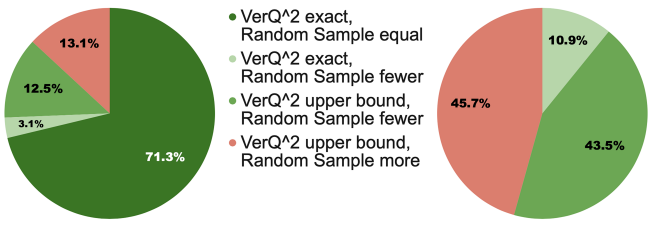

Fig. 6: Comparison with *random sampling* approach. $VerQ^2$ shows improvement in the green sections. Left: All tests; Right: All nonrobust ($R < 1$) tests.

Table 3: Comparison of $VerQ^2$ with Provero, using parameters $\eta = 0.001$ and $\delta = 0.01$ for Provero as well as a binary search loop to find the closest probabilistically provable thresholds.

	# Tests	$VerQ^2$	Provero ($\eta = 0.001,\ \delta = 0.01$)		
		time (s)	time (s)	Avg. Rob. Range	Avg. Upper Bound Diff.
$VerQ^2$ exact, $R = 1$	114	55.51	1,800.00	0.00195	NA
$VerQ^2$ exact $R < 1$	5	73.50	1,800.00	0.00195	NA
$VerQ^2$ sound upper bound	41	1,087.67	1,715.24	NA	0.056

purpose, but it is an existing and effective robustness tool which does not rely on properties of floating point or binary neural networks and thus can be adapted as a baseline comparison.

For this comparison, we re-implement the Provero algorithm since the original tool does not support inequality-based perturbation region constraints and fixed-point input feature values. Additionally, since Provero requests an expected robustness threshold to be given by the user (θ), and our approach produces this threshold automatically, we set up a binary search loop where θ starts at 0.5 and then is modified as thresholds are proven or disproven.

We report our results for comparison with Provero in Table 3 and divide the rows of this table in the same way as Table 2. We report our results for the first two rows with the time taken by Provero and the distance between the upper and lower R discovered using the binary search. For example, if Provero can probabilistically prove that the robustness measure R is bounded by $0.751 \leq R < 0.764$, we report 0.013 difference between the upper and lower bound (robustness range in Table 3, averaged across all 114 or 5 cases in the row).

In Table 3, we also show the 41 cases for which $VerQ^2$ reports a sound upper bound, and display here the average difference obtained by subtracting Provero's upper bound on R from our upper bound on R. For example, if Provero can probabilistically prove that the robustness measure R is bounded by $0.751 \leq R < 0.764$ and $VerQ^2$ reports a sound upper bound on the robustness at 0.771, we report $0.771 - 0.764 = 0.017$ as the upper bound difference in Table 3 (averaged across all 41 cases in the row).

We use δ as 0.01, the level used in [8]. We test η as 0.001, the most precise value used in [8]. For all tests, we give a 30 minute timeout to match the timeout for $VerQ^2$ on these networks, and we report the results at timeout (or stop early if the robustness range becomes $\leq \eta$).

From Table 3 we can see that our approach outperforms Provero both by time and by precision for cases where we get an exact result—in all of these cases, we are able to produce a sound and exact result, whereas Provero is only able to bound the result and is producing probabilistic guarantees rather than fully sound guarantees on these bounds. For the cases where we get an upper bound, we show that our approach is faster than Provero as well, and additionally that, while it is expected Provero's approach will produce a more precise upper bound by being able to sample and prove probabilistic results instead of counting individual models, ours is not much higher on average.

With these results, we answer **RQ4** and show that our approach performs better than PROVERO on cases where we can produce an exact result and comparable on cases where we produce a sound upper bound.

5 Discussion

Within our experiments, we evaluate different model counting approaches and find the most effective for quantized networks (**RQ1**) and we show how our constraint solving optimizations improve our quantitative verification (**RQ2**). We additionally demonstrate how our technique can perform better than two forms of brute force testing (**RQ3**) and an existing published tool [8] (**RQ4**).

It is known that worst case complexities of many symbolic verification techniques are exponential. However, worst case exponential complexity has not excluded verification techniques from transition to practice. For our approach, we have a worst-case complexity of $O(3^{|\mathcal{N}|})$ calls to a constraint solver, where $|\mathcal{N}|$ is the number of nodes in $\mathcal{N}$ minus the input nodes. However, despite this high worst-case complexity, within existing networks and local perturbation regions the actual calls to the constraint solver are far fewer.

6 Conclusion

We present a symbolic execution and model counting based approach for quantitative verification of quantized neural networks, and its implementation in $VerQ^2$. Given a user-defined robustness property for a network, we compute the proportion of inputs in the perturbation region that do not change the output of the network, which provides a quantitative measure of robustness. We present translations from fixed-point constraints to equivalent integer constraints so that we can use integer model counting to produce quantitative results. We have compared the performance of different model counting approaches on the quantized network constraints, and also our own improvements to constraint solving within symbolic execution. Additionally, we have compared our approach against two brute-force sampling approaches and an existing tool for quantitative floating point neural network verification modified for quantized values and found our approach performs favorably against all three. To the best of our knowledge, $VerQ^2$ is the first quantitative verifier for quantized networks with more than binary precision.

References

1. Artificial intelligence and machine learning (ai/ml)-enabled medical devices. https://www.fda.gov/medical-devices/software-medical-device-samd/artificial-intelligence-and-machine-learning-aiml-enabled-medical-devices, accessed: 4/12/24
2. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). pp. 265–283 (2016)

3. Amir, G., Wu, H., Barrett, C., Katz, G.: An smt-based approach for verifying binarized neural networks. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Proceedings, Part II 27. pp. 203–222 (2021)
4. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: International Conference on Computer Aided Verification. pp. 255–272 (2015)
5. Bacchus, P., Stewart, R., Komendantskaya, E.: Accuracy, training time and hardware efficiency trade-offs for quantized neural networks on fpgas. In: International symposium on applied reconfigurable computing. pp. 121–135 (2020)
6. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
7. Baldoni, V., Berline, N., De Loera, J.A., Dutra, B., Köppe, M., Moreinis, S., Pinto, G., Vergne, M., Wu, J.: A user's guide for latte integrale v1.7.2. Optimization **22**(2) (2014)
8. Baluta, T., Chua, Z.L., Meel, K.S., Saxena, P.: Scalable quantitative verification for deep neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 312–323 (2021)
9. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264 (2019)
10. Baranowski, M., He, S., Lechner, M., Nguyen, T.S., Rakamarić, Z.: An smt theory of fixed-point arithmetic. In: International Joint Conference on Automated Reasoning. pp. 13–31 (2020)
11. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research **19**(4), 769–779 (1994), http://www.jstor.org/stable/3690312
12. Bu, H., Sun, M.: Measuring robustness of deep neural networks from the lens of statistical model checking. In: 2023 International Joint Conference on Neural Networks (IJCNN). pp. 1–8. IEEE (2023)
13. Bunel, R., Mudigonda, P., Turkaslan, I., Torr, P., Lu, J., Kohli, P.: Branch and bound for piecewise linear neural network verification. Journal of Machine Learning Research **21** (2020)
14. Chen, S., Wong, E., Kolter, J.Z., Fazlyab, M.: Deepsplit: Scalable verification of deep neural networks via operator splitting. IEEE Open Journal of Control Systems **1**, 126–140 (2022)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
16. Dua, D., Graff, C.: Uci machine learning repository (2017), http://archive.ics.uci.edu/ml
17. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI. vol. 1, p. 3 (2018)
18. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy. pp. 3–18 (2018)
19. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: International Symposium on Software Testing and Analysis. pp. 166–176 (2012)

20. Giacobbe, M., Henzinger, T.A., Lechner, M.: How many bits does it take to quantize your neural network? In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. vol. 12079 (2020)
21. Gopinath, D., Pasareanu, C.S., Usman, M.: Quantifyml: How good is my machine learning model? In: 30th International Symposium on Software Testing and Analysis (ISSTA) (2021)
22. Guo, Y.: A survey on methods and theories of quantized neural networks. arXiv preprint arXiv:1808.04752 (2018)
23. Henzinger, T.A., Lechner, M., Žikelić, Đ.: Scalable verification of quantized neural networks. In: Proceedings of the AAAI conference on artificial intelligence. vol. 35, pp. 3787–3795 (2021)
24. Jia, K., Rinard, M.: Efficient exact verification of binarized neural networks. Advances in neural information processing systems **33**, 1782–1795 (2020)
25. Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: International Static Analysis Symposium. pp. 191–205 (2021)
26. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al.: The marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452 (2019)
27. Little, M., McSharry, P., Roberts, S., Costello, D., Moroz, I.: Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. Nature Precedings pp. 1–1 (2007)
28. Lyell, D., Coiera, E., Chen, J., Shah, P., Magrabi, F.: How machine learning is embedded to support clinician decision making: an analysis of fda-approved medical devices. BMJ health & care informatics **28**(1) (2021)
29. Păsăreanu, C., Converse, H., Filieri, A., Gopinath, D.: On the probabilistic analysis of neural networks. In: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 5–8 (2020)
30. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)
31. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
32. Su, J., Vargas, D.V., Sakurai, K.: One pixel attack for fooling deep neural networks. IEEE Transactions on Evolutionary Computation **23**(5), 828–841 (2019)
33. Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 109–119 (2018)
34. Verdoolaege, S.: The barvinok model counter (2017)
35. Wolberg, W., Mangasarian, O., Street, N., Street, W.: Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository (1995)
36. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: Bdd4bnn: a bdd-based quantitative analysis framework for binarized neural networks. In: International Conference on Computer Aided Verification. pp. 175–200 (2021)