An MILP Encoding for Efficient Verification of Quantized Deep Neural Networks

Samvid Mistry, Indranil Saha, *Member, IEEE* and Swarnendu Biswas

***Abstract*—Quantized Deep Neural Networks (DNNs) have the potential to find wide applications in safety-critical cyber-physical systems implemented on processors supporting only integer arithmetic. The significant challenge therein is to ensure the correctness of the operation of the network with its approximated computation. To address this verification challenge formally, we present a methodology to encode the verification problem into a Mixed-Integer Linear Programming (MILP) problem. Our encoding is based on the bit-precise semantics of quantized neural networks, which ensures the soundness of our method. We implement our verification methodology using the Gurobi MILP solver and evaluate it on several widely used DNN benchmarks. We compare our method with state-of-the-art bit-vector encodings, which are outperformed by our MILP-based verification methodology by an order of magnitude in most cases. These experimental results establish our MILP-based verification technique as a powerful tool for developing formally verified safety-critical systems with quantized DNNs as a component.**

***Index Terms*—quantized neural networks, fixed-point arithmetic, mixed-integer linear programming, formal verification.**

I. Introduction

Sophisticated cyber-physical systems involve complex sensing and control that can be achieved by employing Deep Neural Networks (DNNs). For example, in an autonomous vehicle, DNNs are used for perception, which enables the vehicle to apply appropriate control to make progress towards the goal by following traffic rules and avoiding collisions [1]–[4]. For a UAV, a DNN controller can help in reducing the energy consumption due to control computations and thus can enhance the flight time [5]. It is widely believed that the DNNs have the potential to revolutionize the state-of-the-art of cyber-physical systems and IoTs in the near future.

A major challenge in deploying a DNN for safety-critical systems is its opaque operation, which makes it challenging to provide guarantees about their behavior. Recent work on formal verification of DNNs [6]–[9] try to address this reliability issue by employing mathematical reasoning and tools to establish that the DNN satisfies some formally captured specifications. An example of such a specification for a classification network is that the result produced by a network should *not* change due to small perturbations in its input. The result of this verification process is either a formal guarantee that the network satisfies the property for all possible inputs or a concrete counterexample demonstrating that it violates the property.

Our focus in this paper is the *formal verification of systems employing fixed-point neural networks* [10], [11]. Fixed-point neural networks have the potential to find wide applications in safety-critical systems implemented on low-cost processors supporting only integer arithmetic as well as powerful accelerators such as GPUs and FPGAs. Systems that use fixed-point arithmetic are faster, consume less power and memory, and are less expensive as the computation can happen on low-cost integer-only processors. As noted in prior work [10], quantization is the standard practice for the deployment of neural networks on real-time embedded devices. Though the networks implemented using fixed-point arithmetic are known to introduce little degradation to a network's accuracy [12], they are *not* immune to malicious misclassifications caused by adversarial attacks, and verification of real-valued neural networks is inadequate for establishing their correctness [10].

Recently, the verification problem for DNNs implemented as fixed-point networks has been addressed by reducing the problem to an SMT-solving problem involving the theory of quantifier-free bit-vector arithmetic [10], [11]. However, the poor scalability of the method motivates us to explore alternative verification approaches. We introduce a method for verification of quantized neural networks that reduces the problem to a *decision problem* in the form of a *Mixed-Integer Linear Program (MILP)* [13]. MILP solvers cannot be used directly to verify quantized neural networks because they do not support shifting and rounding operations, which are the most basic primitives needed to perform arithmetic operations with fixed-point numbers. Our method captures the bit-precise semantics of quantized neural networks accurately by encoding the fixed-point primitives as operations in a MILP program.

Based on our MILP encoding, we develop a verification tool that uses Gurobi [14] as the back-end MILP solver. We evaluate our verification tool on several benchmarks, including different variants of networks solving the classification problem on the MNIST dataset and three other popular benchmarks. We also compare the performance of our tool with the state-of-art SMT approaches involving the theory of bit-vectors [10], [11], [15]. Experimental results show that our methodology can solve significantly more verification problem instances for most of the benchmarks than the SMT approaches. Furthermore, our tool takes an order of magnitude less computation time than its competitors for the instances that both the methods can solve. These experimental results establish our MILP-based verification technique as a powerful tool for developing

Samvid Mistry is with GitHub. The work was carried out when the author was with the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur.

Indranil Saha and Swarnendu Biswas are with the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur.

Email: mistrysamvid@gmail.com, {isaha, swarnendu}@cse.iitk.ac.in

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

formally verified safety-critical systems with quantized DNNs as a component.

Contributions: This paper makes the following contributions.

- We present a sound and scalable verification technique for DNNs implemented using fixed-point arithmetic. Our verification methodology is based on a novel encoding of fixed-point arithmetic to mixed-integer linear programs.
- We implement our verification methodology in an automated tool using the Gurobi MILP solver. The tool can automatically solve a wide range of verification problems for systems involving fixed-point networks.
- We apply our tool to several benchmark DNN applications to demonstrate the efficacy. We also compare our method with two recent techniques [11], [15] addressing the same problem, showing significant performance improvement.

II. Related Work

The major focus on applying formal methods to DNNs has been to verify the network against the *robustness property*. Initially, the robustness issue was concerned with small perturbations of data to deal with measurement errors and other factors leading to noisy input data [16]. As more and more networks started being deployed in safety-critical applications, it quickly became a security issue. Recent advances in formal verification methodologies such as Satisfiability Modulo Theories (SMT) [17] and Mixed Integer Linear Programming (MILP) [13] have made it possible to use general-purpose solvers to be applied to the problem of verification of neural networks. However, these solvers were too slow to verify any practical network [18]–[20]. Thus, additional DNN-level reasoning was required to make any verification procedure scale. Toward that end, DNN verification tools such as DeepPoly [21] and ReluVal [6] use interval arithmetic with an iterative refinement of bounds. Reluplex [22] and its successor Marabou [8] have augmented the SMT theory of reals to add support for ReLU and other piece-wise linear activation functions to provide better reasoning through nonlinear constraints. NNV [9] can perform verification of cyber-physical systems having DNNs as major components through reachability analysis.

Recently, there has been significant effort in verifying quantized neural networks. A Binarized deep Neural Network (BNN) [23] is a network that uses binary weights to decrease the latency and memory required by the network. Several authors have addressed the verification problem for BNNs through a reduction to a SAT solving problem [24]–[26] or OBDD manipulation [27]. Recently, [28] has extended Marabou to make it possible to reason about networks with both binary and non-binary weights. To the best of our knowledge, the first method for verification of quantized neural networks implemented using fixed-point arithmetic was introduced by [10]. It was followed by [15] giving formal semantics for fixed-point arithmetic for SMT solvers by providing reduction to the theory of bitvectors and reals. Recently, [11] has provided optimizations for improving the results with SMT solvers, building on the work by [10]. However, verification of several properties of important benchmarks remains out of reach for these methods due to poor scalability.

III. Background

In this section, we briefly review fixed-point arithmetic and quantized deep neural networks.

A. Fixed-point arithmetic

In fixed-point arithmetic, all numbers are represented in 2's complement form over B bit words, where F bits are reserved for the fractional part. To denote a fixed-point type, we use the Q-point notation from [29]. A Q-point notation is denoted as Q[QI].[QF], where QI and QF denote the number of integer and fractional bits allocated to a value. Adding the number of integer bits QI with the number of fractional bits QF yields the total number of bits used to represent a number, denoted by B. For signed fixed-point numbers, the sign bit is also included in the integer part of the number. Hence, the largest value of QF can be $B-1$.

Due to limited precision, not all floating-point values can be represented exactly using fixed-point arithmetic. We may need to perform a rounding operation while converting a floating-point number to a fixed-point number.

Definition III.1 (Rounding down or rounding towards $-\infty$)**.** Let η' be a floating-point number with a non-zero fractional part. We can represent η' as $\eta' = \eta'_i + \eta'_f$, where η'_i is an integer and η'_f is a floating-point number such that $0 < \eta'_f < 1$. We refer to η'_i as the "rounded down" version of η' and denote it as $\texttt{int}(\eta')$.

In order to convert a floating-point number to a fixed-point number, we use the following equation [29], where int is a function that rounds the number towards $-\infty$:

$$\texttt{fixedpoint} = \texttt{int}(\texttt{floatingpoint} \cdot 2^F). \tag{1}$$

For example, we can convert 6.125 from floating-point to fixed-point type Q4.2 using Equation (1), such that $\texttt{int}(6.125 \cdot 2^2) = \texttt{int}(24.5) = 24$.

The integer representations of two fixed-point numbers can be directly added together, as long as the radix point is aligned for both numbers. Otherwise, one of the numbers must be converted to be compatible with the other type. For example, a $Q1.7$ and a $Q2.6$ number cannot be added directly. In this case, we can either shift $Q1.7$ right to get a $Q2.6$ number or shift $Q2.6$ left to get a $Q1.7$ number. Shifting a number left will introduce imprecision in the integer part of the variable, which can result in a much larger error between the floating-point and the fixed-point numbers. Thus, it is preferable to shift a number right and lose precision from the fractional part of a number.

Multiplication between two fixed-point numbers can be accomplished by directly multiplying the underlying integers. Unlike addition, the radix point need not be aligned before multiplication. Multiplying two numbers of type $Q[a].[b]$ and $Q[p].[q]$ results in a number of type $Q[a+p].[b+q]$. For example, $Q1.7 \cdot Q2.6 = Q3.13$. After multiplication, the bit-width required to hold the result gets doubled. To bring the result back into the original bit-width, we can shift the number right until the result lies in the range. We have to shift the $Q3.13$ number right by 8 places to get a $Q3.5$ number, which

can be stored in the original bit-width and can be used in further computations.

A fixed-point number can be converted back to floating-point using the following equation:

$$\texttt{floatingpoint} = \texttt{fixedpoint}/2^F \tag{2}$$

We can convert the fixed-point value 24 with type Q4.2 to the floating-point value 6.0 using Equation (2). Note that converting a floating-point number (e.g., 6.125) to a fixed-point number (i.e., 24) and re-converting the fixed-point number to a floating-point number (i.e., 6.0) may lead to a loss of precision.

B. Deep Neural Networks

A *neural network* is a collection of neurons connected by edges. The neurons are organized in layers such that one neuron belongs to exactly one layer, and layers are connected with other layers using edges. All edges have a *weight* associated with them, and all nodes have an optional value associated with them, called *bias*. A *deep* neural network (DNN) consists of an input layer, multiple *hidden* layers, and an output layer. More formally, for a DNN f, the number of layers are denoted by n, and the size of a layer i is denoted by s_i. Out of these n layers, the first layer is the input layer and the n^{th} layer is the output layer. All the layers in between are called hidden layers. The output of the j^{th} node of the i^{th} layer is denoted by $a_{i,j}$. Consequently, the output of layer i can be packed into a vector $[a_{i,1}, \ldots, a_{i,s_i}]^\mathsf{T}$, which is denoted by $\mathbf{A}_i$.

Running inference on a DNN f consists of calculating the value of $\mathbf{A}_n$ from the given value of $\mathbf{A}_1$. Each layer i has a weight matrix W_i of size $s_{i-1} \times s_i$ and a bias vector $\mathbf{B}_i$ of size s_i associated with it, where $2 \leq i \leq n$. In order to calculate $\mathbf{A}_n$, values from $\mathbf{A}_1$ are propagated through the network. This propagation is carried out in terms of a dot product between the values of layer $\mathbf{A}_{i-1}$ and weight matrix W_i, and then $\mathbf{B}_i$ is added to the resulting value, creating the *preactivation* values for all nodes in the layer i. A *nonlinear* activation function, such as ReLU [30] or sigmoid [31], is applied element-wise on the resulting vector to get the value of $\mathbf{A_i}$. Thus,

$$\mathbf{A_i} = fn_i(W_i \cdot \mathbf{A}_{i-1} + \mathbf{B}_i), \tag{3}$$

where fn_i is the activation function for layer i. Overall, a DNN implements a function $f : \mathbb{R}^{s_1} \to \mathbb{R}^{s_n}$. The result of the DNN is given by the values for the nodes in the output layer.

C. Quantized Neural Network

Quantization converts neural networks with real numbers, that require floating-point hardware to work, into networks over integers, whose semantics follow fixed-point arithmetic [12]. Quantization allows computation to be carried out by integer-only architectures, widening the applicability of DNNs. Furthermore, quantization allows DNN computations to use small words, e.g., 8-bits and 16-bits, that help reduce the training time, inference latency, and storage overhead.

We will now present how the computation in each layer of a quantized network is carried out using fixed-point arithmetic. First, we consider the case where the outputs of any layer and the weights have the *same* fixed-point type. The following equation, modified from Equation (3), represents the inner workings of a quantized neural network having the same fixed-point type for each computation in the network:

$$\overline{\mathbf{A}_i} = \overline{fn_i}(\texttt{int}(2^{-F}(\overline{W_i} \cdot \overline{\mathbf{A}_{i-1}} + \overline{\mathbf{B}_i}))). \tag{4}$$

Variables with bar and without bar represent fixed-point and floating-point variables, respectively. For a weight matrix W_i, the quantized weight is $\overline{W_i} = \texttt{rnd}(2^F W_i)$, where $\texttt{rnd}(\cdot)$ stands for some rounding scheme to an integer and is applied element-wise on vectors and matrices. In Equation (4), the output of $\overline{W_i} \cdot \overline{\mathbf{A}_{i-1}}$ has $2F$ bits in the fractional part. Consequently, we scale $\mathbf{B}_i$ by 2^{2F}. For a bias vector $\mathbf{B}_i$, the quantized bias is $\overline{\mathbf{B}_i} = \texttt{rnd}(2^{2F}\mathbf{B}_i)$. After the dot product and the sum, the resulting value would have $2F$ number of bits in the fractional part. To get the result with F bits in the fractional part, we multiply by 2^{-F} (i.e., right shift). In Equation (4), $\texttt{int}(\cdot)$ is a specialized version of function $\texttt{rnd}(\cdot)$ that rounds towards $-\infty$ and $\overline{fn_i}$ is the fixed-point version of fn_i.

Now we consider the case where the outputs of the neurons in a layer have *different* fixed-point datatypes. Equation (4) has to be modified if we want to support *different* number of bits for all nodes. We use the following procedure to calculate the value of $\overline{\mathbf{A}_i}$ when we have different types associated with each neuron's output. We first convert all nodes to the same fixed-point type. We achieve this by converting the fixed-point numbers to floating-point and then converting them to the required target fixed-point type. Let T and L be matrices containing fractional bits and integer bits for each node of the network. $T_{i-1} = [t_1, t_2, \ldots, t_{s_{i-1}}]^\mathsf{T}$ is a vector containing the number of fractional bits, and $L_{i-1} = [l_1, l_2, \ldots, l_{s_{i-1}}]^\mathsf{T}$ is a vector containing the number of integer bits for each node in layer $i-1$. Let $m_i = \min(T_{i-1})$ denote the minimum value in the vector T_{i-1}. With this data, the following equations can align all values from the previous layer, and each resulting value will have m number of fractional bits:

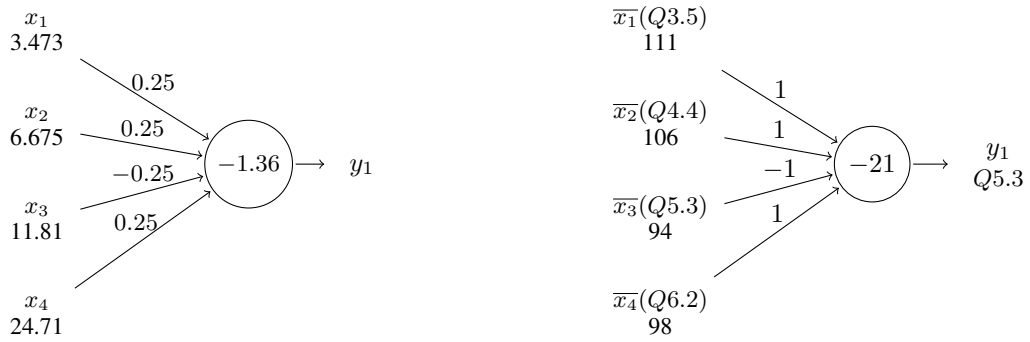
$$\overline{\mathbf{A}_{i-1}}' = [\overline{\mathbf{A}_{i-1,1}}', \overline{\mathbf{A}_{i-1,2}}', \ldots, \overline{\mathbf{A}_{i-1,s_{i-1}}}']^\mathsf{T}, \tag{5}$$

$$\overline{\overline{\mathbf{A}_{i-1}}} = \texttt{int}(2^{m_i}\overline{\mathbf{A}_{i-1}}'). \tag{6}$$

where $\overline{\mathbf{A}_{i-1,\alpha}}' = 2^{-t_\alpha}\overline{\mathbf{A}_{i-1,\alpha}}$, $1 \leq \alpha \leq s_{i-1}$, and $\overline{\overline{A_{i-1}}}$ is a vector of fixed-point numbers with m_i bits in the fractional part. Equation (5) converts all fixed-point values to equivalent floating-point values by multiplying each value with 2^{-t_α}, where t_α is the number of fractional bits in α. Equation (6) converts the floating-point values to fixed-point values with m_i bits in the fractional part. Similarly, all weights in $\overline{W_i}$ can now be quantized using m_i bits in the fractional part, and all biases in $\overline{\mathbf{B}_i}$ can be quantized using $2m_i$ bits in the fractional part. For the resulting node, we need $T_{i,j}$ bits in the fractional part, and we already have $2m_i$ bits. Let $\mathbf{H}_i = [2^{-h_1}, 2^{-h_2}, \ldots, 2^{-h_{s_i}}]^\mathsf{T}$ be a vector where $h_j = 2m_i - T_{i,j}$. Finally, the post-activation value for all nodes can be computed as follows.

$$\overline{\mathbf{A}_i} = \overline{fn_i}(\texttt{int}(\mathbf{H}_i \cdot (\overline{W_i} \cdot \overline{\overline{\mathbf{A}_{i-1}}} + \overline{\mathbf{B}_i}))) \tag{7}$$

Equation (7) is a modification of Equation (4) which can work with networks where each node in the network can have



(a) A toy floating-point neural network

(b) Fixed-point version of Figure 1a

Fig. 1: An example of floating-point to fixed-point network conversion

a different type. First, we multiply the fixed-point weights $\overline{W_i}$ with the incoming values from the previous layer $\overline{\overline{\mathbf{A}_{i-1}}}$, followed by the addition of bias $\overline{\mathbf{B}_i}$. We then multiply $\mathbf{H}_i$, which converts all the incoming values to the types specified in the upcoming layer. Then, we round down the results by applying the `int` rounding function. Finally, we apply an activation function. The output of this expression will serve as the input of the next layer, i.e., $\overline{\mathbf{A}_i}$.

Example 1. *Figure 1a shows a toy neural network with floating-point values. Figure 1b shows the converted network with fixed-point types and values, where each input node in the network has a different type. From Figure 1b, we can observe that $\overline{x}_1$ has type Q3.5. Since $\mathbf{T}_1$ is the vector containing fractional parts of the inputs, its first element is the fractional part of $\overline{x}_1$, i.e.,* 5*. Similarly, $\mathbf{L}_1$ is the vector containing integer parts of the inputs. Its first element is the integer part of $\overline{x}_1$, i.e.,* 3*. With values from all inputs, we can see that $\mathbf{T}_1 = [5, 4, 3, 2]^\mathsf{T}$ and $\mathbf{L}_1 = [3, 4, 5, 6]^\mathsf{T}$. The weights in the floating-point network are $W_2 = [0.25, 0.25, -0.25, 0.25]^T$. The fractional part of the fixed-point representations of the weights is decided as the minimum fractional part in any type in the previous layer, i.e., $min(T_1) = 2$. The fractional part is then subtracted from the total number of bits to get the integer part, which in this case is* 6*. With type Q6.2, we can convert W_2 to $\overline{W_2} = [1, 1, -1, 1]^\mathsf{T}$.*

Let $\mathbf{A}_1 = [x_1, x_2, x_3, x_4]^\mathsf{T} = [3.473, 6.675, 11.81, 24.71]^\mathsf{T}$ and $\mathbf{B}_2 = [-1.36]$ (chosen arbitrarily as shown in Figure 1a). Based on the fixed-point datatypes of the nodes in the input layer, $\overline{\mathbf{A}_1} = [\overline{x}_1, \overline{x}_2, \overline{x}_3, \overline{x}_4]^\mathsf{T} = [111, 106, 94, 98]^\mathsf{T}$. From the fixed-point data types of the input nodes, we can see that $m_1 = \min(T_1) = 2$. Consequently, $\mathbf{H}_2 = [2^{-(2m_1 - T_{2,1})}] = [2^{-1}]$. From Equation (5)*, we can calculate*

$$\begin{aligned}\overline{\mathbf{A}_1}' =&[2^{-T_{1,1}}\overline{\mathbf{A}_{1,1}}, 2^{-T_{1,2}}\overline{\mathbf{A}_{1,2}}, 2^{-T_{1,3}}\overline{\mathbf{A}_{1,3}}, 2^{-T_{1,4}}\overline{\mathbf{A}_{1,4}}]^\mathsf{T}\\ =&[3.46875, 6.625, 11.75, 24.5]^\mathsf{T},\\ \overline{\overline{\mathbf{A}_1}} =&\texttt{int}(2^2 \cdot [3.46875, 6.625, 11.75, 24.5]^\mathsf{T})\\ =&[13, 26, 47, 98]^\mathsf{T}.\end{aligned}$$

The type of all values in $\overline{\overline{\mathbf{A}_1}}$ is Q6.2. Since all values in $\overline{W_2}$ also have type Q6.2, the type of the resulting vector will be Q12.4 after multiplication. To be able to add bias to the resulting vector, we need all values of the bias vector to be of type Q12.4. Consequently, $\overline{\mathbf{B}_2} = \texttt{int}(2^4 \cdot \mathbf{B}_2) = [-21]$.

$$\begin{aligned}\overline{W_2} \cdot \overline{\overline{\mathbf{A}_1}} + \overline{\mathbf{B}_2} =&[13 \cdot 1 + 26 \cdot 1 + 47 \cdot -1 + 98 \cdot 1] + [-21]\\ =&[69]\end{aligned}$$

With that result, we can now scale the Q12.4 value to the resulting type Q5.3. To match the fractional part, we need to shift the result to the right by 1 bit. Using 8 bits, we can only represent integers in the range [-256, 255]. So we need to clip the result between those bounds if it overflows, i.e., any value less than -256 will be replaced with -256, and any value greater than 255 will be replaced with 255.

$$\begin{aligned}\texttt{int}(\mathbf{H}_2 \cdot (\overline{W_2} \cdot \overline{\overline{\mathbf{A}_1}} + \overline{\mathbf{B}_2})) =&\texttt{int}([2^{-1}] \cdot [69])\\ =&\texttt{int}(34.5) = 34.\end{aligned}$$

Since 34 ∈ [-256, 255], we do not need to clip the result. Lastly, we will apply `ReLU` as our activation function $\overline{fn_i}$.

$$\overline{\mathbf{A}_2} = \overline{fn_i}(\texttt{int}(\mathbf{H}_2 \cdot (\overline{W_2} \cdot \overline{\overline{\mathbf{A}_1}} + \overline{\mathbf{B}_2}))) = \max(0, 34) = 34$$

IV. Problem Definition

A DNN verification query checks a property against a DNN. The property can be any set of arbitrary constraints involving the nodes and the edges of the neural networks. Typically, a property constrains the input range and implies that the output nodes or some combination of output nodes will be in some range. The verification problem involves guaranteeing that none of the valid inputs in the range violates the output property.

Let $\overline{f}$ be the quantized version of neural network f. Let $fxp(\mathbf{v}, \boldsymbol{\tau}, \boldsymbol{\iota})$ be a function which converts $\mathbf{v}$ to fixed-point vector $\overline{\mathbf{v}}$ such that $\overline{v_i}$ has $\boldsymbol{\tau}_i$ number of fractional bits and $\boldsymbol{\iota}_i$ number of integer bits. Let $fp(\overline{\mathbf{v}}, \boldsymbol{\tau})$ be a function that converts a fixed-point vector $\overline{\mathbf{v}}$ back to floating-point. Given a bounded input domain D and a property P, we want to prove that

$$\begin{aligned}\forall \mathbf{x} \in D.\ (\overline{\mathbf{x}} = fxp(\mathbf{x}, \mathbf{T}_1, \mathbf{L}_1)) \wedge\\ (\overline{\mathbf{y}} = \overline{f}(\overline{\mathbf{x}})) \wedge (\mathbf{y} = fp(\overline{\mathbf{y}}, \mathbf{T}_n)) \implies P(\mathbf{y})\end{aligned} \quad (8)$$

where P is a predicate that takes the output of a network as input and checks for the satisfaction of some arbitrary constraints. In other words, it consists of checking an input/output relationship. Equation (8) checks the property P over the real-valued output

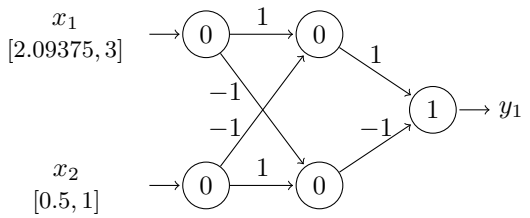


Fig. 2: Example neural network where a property holds with floating-point values but not with fixed-point values

$\mathbf{y}$ since the properties are generally defined by domain experts for the original domain, and the domains generally use floating-point values for computation. For example, consider a closed-loop control system where the controller implemented using fixed-point arithmetic gets floating-point inputs from the sensors and has to produce floating-point output for the actuators. Given the fixed-point datatype of the output, from the quantized output $\bar{y}$, the floating-point output y can be determined uniquely. The above problem formulation also works with minor changes if the inputs and outputs of the quantized network and the property are given in fixed-point arithmetic. This formulation of the verification problem is very abstract and generalizes over various verification queries, such as queries involving perturbation of inputs and queries comparing values of different output nodes.

Example 2. *We give a concrete example to discuss the verification problem. Consider the neural network in Figure 2. The values inside the nodes in the floating-point network f indicate biases, and the values on the edges indicate weights. All values are floating-point numbers. The numbers below the input node labels indicate the range of valid values for that node. D is considered as $[2.09375, 3] \times [0.5, 1]$.*

The property we want to prove is $P(\mathbf{y}) = y_1 \geq 2.09375$. The network f satisfies the property for any input from the chosen ranges for x_1 and x_2. Interestingly, when we convert the network to fixed-point with type Q4.4, the same property gets violated. For the chosen range of inputs, the smallest value that such a fixed-point network can get for y_1 is 2.0625. This shows that even if a property holds on the floating-point network, it may not hold on a fixed-point network [10].

V. MILP Encoding

In this section, we present how we solve the verification problem introduced in Section IV through a reduction to an MILP problem. Our goal is to encode the verification condition in Equation (8) as a mixed-integer linear program such that the verification result can be concluded from its solution.

The difficulty with the formulation in Equation (8) is that it cannot be directly encoded into an MILP solver, as an MILP solver cannot solve a validity problem. The validity problem can be reduced to a satisfiability problem by taking a negation of the input formula, which can be solved using an MILP solver. If the negated formula is unsatisfiable, then the input formula is valid (verification is successful). On the other hand, if the negated formula is satisfiable, then the input formula is not valid (verification fails), which is evident by the counterexample generated from the solution produced by the solver.

To convert the validity problem to a satisfiability problem, we take the negation of the formula in Equation (8), which leads to the following verification condition[1]:

$$\forall \mathbf{x} \in D.\ (\overline{\mathbf{x}} = fxp(\mathbf{x}, \mathbf{T}_1, \mathbf{L}_1)) \wedge \tag{9a}$$
$$(\overline{\mathbf{y}} = \overline{f}(\overline{\mathbf{x}})) \wedge \tag{9b}$$
$$(\mathbf{y} = fp(\overline{\mathbf{y}}, \mathbf{T}_n)) \wedge \tag{9c}$$
$$\neg P(\mathbf{y}) \tag{9d}$$

This verification condition can be directly encoded into an MILP solver since negation and conjunction are natively supported by the MILP solvers.

In the rest of this section, we will show how we encode this verification condition into a mixed-integer linear program. Note that though the MILP solvers are routinely used for solving optimization problems, we use the MILP solvers to check the feasibility of the equations and not for optimization.

Running inference on the fixed-point network $\overline{f}$ involves passing input values to $\mathbf{A}_1$ and calculating the values at $\mathbf{A}_n$ by repeatedly applying Equation (7) between layers. $\overline{W_i}$ and $\overline{\mathbf{B}_i}$ along with $\overline{\mathbf{A}_1}$ can be obtained by quantizing respective floating-point values before running the MILP solver, and the quantized values can be directly used in the MILP encoding. Values of T and L are known in advance (e.g., using range analysis), so all values of $\mathbf{H}$ can be computed. However, Equation (7) involves multiplication by $\mathbf{H}_i$ followed by the $\texttt{int}(\cdot)$ operation, which is equivalent to shifting right F places where 2^{-F} is any element from $\mathbf{H}_i$. In the following, we discuss challenges in designing the MILP encoding and their solutions.

A. Encoding Shift Operation

MILP solvers do *not* provide primitives to shift numbers. We must divide by 2^F to simulate shifting. The problem with division is that MILP solvers carry out all arithmetic operations in floating-point, and we may be left with some fractional part after division since fixed-point operations are to be carried out using integers only. MILP solvers also do not provide any primitives to take away this fractional part. The following theorem introduces a set of constraints to encode rounding in a program. They are designed to make sure that there is only one integer that the solver can choose as the result of division, and that is the rounded-down version of the result of division.

Theorem 1. *Let η be a floating-point variable and $\overline{\zeta}$ and $\overline{\eta}$ be fixed-point variables such that $\overline{\zeta} = \texttt{int}(\eta)$ and $\eta = \overline{\eta} \cdot 2^{-F}$, i.e., $\overline{\zeta}$ is the result of shifting and rounding $\overline{\eta}$. Let $\texttt{offset} \in \mathbb{R}$ be a constant chosen satisfying the following constraint: $(1 - 2^{-F}) \leq \texttt{offset} < 1$. Then the constraints*

$$\eta - \texttt{offset} \leq \overline{\zeta}, \tag{10}$$
$$\overline{\zeta} \leq \eta. \tag{11}$$

shift $\overline{\eta}$ by F places and provide sound rounding towards $-\infty$.

Proof. To prove the theorem, we consider two different possibilities for the value of η: (i) η is an integer, (ii) η is

[1] $\neg(a \Rightarrow b) \Leftrightarrow \neg(\neg a \vee b) \Leftrightarrow a \wedge \neg b$

a non-integer number with some positive fractional part. We show that in both cases, Equation (10) and Equation (11) ensure that $\overline{\zeta}$ is assigned a unique value correctly.

Case (i) : In case η is an integer, the value of $\overline{\zeta}$ should be η. As $\texttt{offset} < 1$, Equation (10) gives us $\overline{\zeta} > \eta - 1$. On the other hand, Equation (11) ensures that $\overline{\zeta} \leq \eta$. Thus, the two equations in the theorem ensure that the only possible value for $\overline{\zeta}$ is η when η is an integer.

Case (ii) : In case η is a non-integer number with some fractional part, we can write $\eta = \eta_i + \eta_f$, where η_i is an integer and $0 < \eta_f < 1$. In this case, the value of $\overline{\zeta}$ should be η_i.

Since fixed-point numbers are distributed evenly on the number line, we can calculate the largest value η_f can take by subtracting the smallest nonzero positive value it can take from 1. The smallest nonzero positive value η_f can take is 2^{-F}. Consequently, the largest value for η_f is $1 - 2^{-F}$. As the value of $\texttt{offset}$ is chosen to be greater than or equal to $(1 - 2^{-F})$, $\eta - \texttt{offset} \leq \eta_i$ and Equation (10) ensures that η_i is one of the feasible values for $\overline{\zeta}$. Moreover, as $\texttt{offset} < 1$, we get from Equation (10) that $\eta_i - 1 < \eta - \texttt{offset} \leq \overline{\zeta}$. Thus, the value of $\overline{\zeta}$ cannot be equal to or lower than $\eta_i - 1$. As Equation (11) requires the value of $\overline{\zeta}$ to be less than or equal to η, the maximum possible value for $\overline{\zeta}$ is η_i (as $\overline{\zeta}$ is an integer). Thus, the only value $\overline{\zeta}$ can assume in this case is η_i. □

The next example shows an application of the above theorem.

Example 3. *Let $\overline{\eta} = 55$ be an integer representing a fixed-point number with type $\texttt{Q4.4}$. We would like to shift $\overline{\eta}$ 2 places to the right. That essentially means we want to do $\overline{\zeta} = \texttt{int}(\eta')$ where $\eta' = \overline{\eta} \cdot 2^{-2} = 13.75$. According to Theorem 1, we want to choose $\texttt{offset}$ such that $1 - 2^{-2} \leq \texttt{offset} < 1 \rightarrow 0.75 \leq \texttt{offset} < 1$. Let $\texttt{offset} = 0.75$. The equations from Theorem 1 will now become*

$$\eta' - \texttt{offset} \leq \overline{\zeta} \rightarrow 13.75 - 0.75 \leq \overline{\zeta} \tag{12}$$

$$\overline{\zeta} \leq \eta' \rightarrow \overline{\zeta} \leq 13.75 \tag{13}$$

Since subtracting 0.75 brings 13 into the allowed range of values, any value greater than 0.75 and less than 1 will also bring 13 into the allowed range of values and produce the correct set of equations. The example shows that the only feasible value for $\overline{\zeta}$ that satisfies both equations is 13, which is $\texttt{int}(\eta')$. We can also see that if we chose $\texttt{offset} < 0.75$, then no value would be able to satisfy the equations, and the problem would become infeasible.

B. Encoding the Verification Condition in MILP

Here we present the MILP constraints for encoding the verification problem captured in Equation (9). A SAT result for Equation (9) indicates that a counter-example violating the property is found. An UNSAT indicates successful verification. In the encoding, we write $[|n|]$, $n \in \mathbb{N}$, to denote the set $\{1, 2, \ldots, n\}$.

1) Input nodes: We start with presenting the MILP constraints corresponding to the first conjunct in the verification condition given in Equation (9a). For all input nodes, we have some constraints bounding the value that the node can take. Let $\mathbf{UB}$ and $\mathbf{LB}$ be vectors containing upper bounds and lower bounds for floating-point inputs, respectively. We can have fixed-point vectors $\overline{\mathbf{UB}} = fxp(\mathbf{UB}, \mathbf{T}_1, \mathbf{L}_1)$ and $\overline{\mathbf{LB}} = fxp(\mathbf{LB}, \mathbf{T}_1, \mathbf{L}_1)$ containing upper and lower bounds for fixed-point inputs, respectively. There are typically only 2 types of constraints present when bounding the input, and the encoding for both of them is shown below.

$$\forall r \in [|T_1|].\ \overline{\mathbf{A}_{1,r}} \leq \overline{\mathbf{UB}_r},\ \overline{\mathbf{A}_{1,r}} \geq \overline{\mathbf{LB}_r} \tag{14}$$

2) Hidden nodes: Now we present the MILP constraints corresponding to the second conjunct in the verification condition given in Equation (9b). These constraints capture the internal processing of the quantized neural network through the hidden layers. For any node in a hidden layer, the following set of constraints can be used to encode its operation. Let us assume the node for which we are encoding the constraints is the j^{th} node of the i^{th} layer. If all the incoming edges into a node do not have the same type, then we first align the radix points. We introduce a vector $\mathbf{F}$ with floating-point type and encode the following constraints into the solver.

$$\forall r \in [|T_{i-1}|].\ \mathbf{F}_r = \overline{\mathbf{A}_{i-1,r}} / 2^{T_{i-1,r}}. \tag{15}$$

Equation (15) converts all fixed-point values coming from the previous layer back into the floating-point values. Now these values can be converted back to fixed-point with desired number of fractional bits, i.e., having m_i bits in the fractional part where $m_i = \min(T_{i-1})$. Following the quantization procedure and Theorem 1, we can use the following constraints to quantize $\mathbf{F}$ back to fixed-point. We introduce a vector $\overline{\mathbf{X}^{i,j}}$ to hold the quantized values.

$$\forall r \in [|T_{i-1}|].\ \mathbf{F}_r \cdot 2^{m_i} - \texttt{offset} \leq \overline{\mathbf{X}_r^{i,j}} \tag{16}$$

$$\forall r \in [|T_{i-1}|].\ \overline{\mathbf{X}_r^{i,j}} \leq \mathbf{F}_r \cdot 2^{m_i} \tag{17}$$

If all the incoming edges in a node have the same type, then we can directly assign the incoming values to $\overline{\mathbf{X}^{i,j}}$.

$$\forall r \in [|T_{i-1}|].\ \overline{\mathbf{X}_r^{i,j}} = \overline{\mathbf{A}_{i-1,r}} \tag{18}$$

With the aligned inputs with us, we simply need to perform a dot product with the weights and add the bias to get the pre-activation value for this node. We introduce a variable $\overline{pr}$ to hold the value of the result of this computation.

$$\overline{pr_{i,j}} = \overline{W_{i,j}} \cdot \overline{\mathbf{X}^{i,j}} + \overline{B_{i,j}} \tag{19}$$

The value in $\overline{pr_{i,j}}$ currently has $2m$ number of bits in the fractional part. As noted in Section III, we need to shift by $\mathbf{H}_{i,j}$ bits. We introduce a variable $\overline{\gamma}$ to hold the pre-activation value of this neuron. We can shift by $\mathbf{H}_{i,j}$ bits using the following constraints.

$$\overline{pr_{i,j}} \cdot \mathbf{H}_{i,j} - \texttt{offset} \leq \overline{\gamma_{i,j}} \tag{20}$$

$$\overline{\gamma_{i,j}} \leq \overline{pr_{i,j}} \cdot \mathbf{H}_{i,j} \tag{21}$$

An issue with Equations (20) and (21) is that the output value can be larger than the fixed-point type can hold. There are mainly two ways to handle this overflow. One way is to let the hardware take its natural path and wrap around the value.

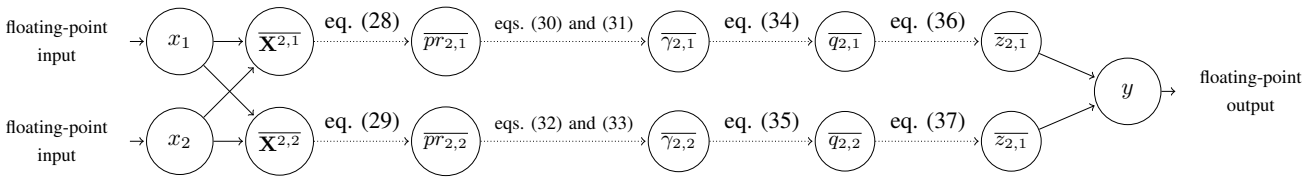


Fig. 3: Fixed-point encoding of Figure 2 with intermediate steps.

Since the theory of bitvectors in SMT natively supports wrap-around as a mode of rounding, this type of rounding can be handled in SMT solvers easily, while MILP solvers offer no such primitive, and hence it would be difficult to accurately implement the semantics of wrap-around in MILP encoding. The other would be to saturate the value to the maximum or minimum that the fixed-point type can hold. We use saturation as the mechanism to handle overflow. Let $lb = -2^{L_{i,j}+T_{i,j}-1}$ and $ub = 2^{L_{i,j}+T_{i,j}-1} - 1$ be the lowest and highest value representable by this node. We introduce a variable $\overline{q_{i,j}}$ to hold the result of this computation.

$$\overline{q_{i,j}} = \min(ub, \max(lb, \overline{\gamma_{i,j}})) \tag{22}$$

Finally, we can apply any piecewise-linear activation function on $\overline{q}$. We use ReLU as the activation function. We introduce a variable $\overline{z_{i,j}}$ to hold the value of post-activation.

$$\overline{z_{i,j}} = \max(0, \overline{q_{i,j}}) \tag{23}$$

Since linear programming cannot reason about functions that are not linear or piecewise-linear, our methodology supports only a limited set of activation functions. Nonlinear activation functions such as sigmoid and hyperbolic tangent are beyond the scope of the current paper.

3) Output nodes: We now present the MILP constraints corresponding to the third conjunct in the verification condition given in Equation (9c). In the final layer, we return the value of the output node without applying any activation function, i.e., $\overline{z} = \overline{q}$. Once we have collected all the outputs in a vector $\overline{\mathbf{y}}$, we convert the vector back to floating-point.

$$\mathbf{y} = fp(\overline{\mathbf{y}}, \mathbf{T}_n) \tag{24}$$

We can encode $\neg P(\mathbf{y})$ (the last component in the verification condition in Equation (9d)) into the solver to find a counter-example for the property. The presented method encodes the semantics of fixed-point arithmetic exactly and completely.

Example 4. *To illustrate the encoding procedure with a concrete example, we are going to encode the network in Figure 2. All nodes in the fixed-point network will have type Q4.4. Figure 3 shows the intermediate variables and the constraints used to generate them visually. The constraints for input nodes will be as follows.*

$$\overline{\mathbf{A}_{1,1}} = x_1 \leq 48, \overline{\mathbf{A}_{1,1}} = x_1 \geq 33 \tag{25}$$
$$\overline{\mathbf{A}_{1,2}} = x_2 \leq 16, \overline{\mathbf{A}_{1,2}} = x_2 \geq 8 \tag{26}$$

Since all nodes in the network have the same type, we need not align the incoming values for the intermediate layer. We are directly going to assign the values to intermediate vector $\overline{\mathbf{X}}$ for each node.

$$\forall j \in [|T_2|].\forall r \in [|T_1|].\overline{\mathbf{X}_r^{2,j}} = \overline{\mathbf{A}_{1,r}} \tag{27}$$

Once we have the values in the intermediate vector, we can encode the dot product with the weights and addition of the bias.

$$\overline{pr_{2,1}} = [16, -16]^\intercal \cdot \overline{\mathbf{X}^{2,1}} + 0 \tag{28}$$
$$\overline{pr_{2,2}} = [-16, 16]^\intercal \cdot \overline{\mathbf{X}^{2,2}} + 0 \tag{29}$$

The result of the dot product will be of type $Q8.8$. The resulting type we want is $Q4.4$. We need to shift the result 4 times to the right to get the desired number of fractional bits. Therefore, $\mathbf{H}_i = [2^{-4}, 2^{-4}]$. Let `offset` $= 1 - 2^{-2*8} = 0.999984741211$. *The procedure to choose a sound value for* `offset` *is explained in Section V-C. The shifting and rounding can be encoded using the following constraints.*

$$\overline{pr_{2,1}} \cdot 2^{-4} - \texttt{offset} \leq \overline{\gamma_{2,1}} \tag{30}$$
$$\overline{\gamma_{2,1}} \leq \overline{pr_{2,1}} \cdot 2^{-4} \tag{31}$$
$$\overline{pr_{2,2}} \cdot 2^{-4} - \texttt{offset} \leq \overline{\gamma_{2,2}} \tag{32}$$
$$\overline{\gamma_{2,2}} \leq \overline{pr_{2,2}} \cdot 2^{-4} \tag{33}$$

We need to saturate the result so that it fits in the resulting type $Q4.4$. The largest value a $Q4.4$ type fixed-point number can hold is 255, and the smallest is -256.

$$\overline{q_{2,1}} = \min(255, \max(-256, \overline{\gamma_{2,1}})) \tag{34}$$
$$\overline{q_{2,2}} = \min(255, \max(-256, \overline{\gamma_{2,2}})) \tag{35}$$

We now need to apply the ReLU activation function to the result.

$$\overline{z_{2,1}} = \max(0, \overline{q_{2,1}}) \tag{36}$$
$$\overline{z_{2,2}} = \max(0, \overline{q_{2,2}}) \tag{37}$$

The constraints to encode the output layer are identical, except that we do not encode the activation function. We instead encode the negation of the property that we want to prove. In this case, the output of the final layer before the activation function will be denoted by $\overline{q_{3,1}}$. As per Equation (2)*, we can convert the result to floating point by multiplying 2^{-4}. The floating-point result will be denoted by y.*

$$y = \overline{q_{3,1}} \cdot 2^{-4} \tag{38}$$

Finally, we encode the negation of the property as follows

$$y < 2.09375 \tag{39}$$

C. Choosing `offset` for a network

In a program with multiple types, the value of `offset` must be chosen carefully to make sure that all different types get rounded in a sound manner. The solution is to choose an `offset` which works for the largest divisor in the network, and it will work for all instances of rounding. The constraint system defined in this section has 2 instances of rounding, namely in Equations (16) and (17) and Equations (20) and (21).

Equations (16) and (17) involves division by $2^{T_{i-1,r}}$ which can take the largest value of 2^{B-1} where B is the total number of bits. Equations (20) and (21) involve division by $\mathbf{H}_{ij} = 2^{-(2m-\mathbf{T}_{i-1,j})}$ which can take the largest value of $2^{-(2B-2)}$. Hence, choosing `offset` such that $(1 - 2^{-(2B-2)}) \leq \texttt{offset} < 1$ will work for all instances of rounding. Avoiding all the complexity, we recommend choosing the value of $\texttt{offset} = 1 - 2^{-2B}$. Since $1 - 2^{-(2B-2)} < 1 - 2^{-2B} < 1$, the equations will be sound with $\texttt{offset} = 1 - 2^{-2B}$. Given that we are going to work with small values of B, we are unlikely to run into precision issues while storing $1 - 2^{-2B}$ in a floating-point variable.

D. Implementation of `max` in MILP Solvers

There are mainly two ways to implement $z = \texttt{max}(x, lb)$ in MILP solvers, where x is a variable and lb is a constant. The implementation of `min` is also analogous.

1) Big-M constraints: Let b_{lb} and b_x be two binary variables. Taking M to be a very large positive constant, we can use the following assertions to encode the `max` operation.

$$b_{lb} + b_x = 1 \tag{40a}$$
$$z \geq lb \tag{40b}$$
$$x - z - M \cdot b_{lb} \leq 0 \tag{40c}$$
$$x - z + M \cdot b_{lb} \geq 0 \tag{40d}$$
$$z - M \cdot b_x \leq lb \tag{40e}$$
$$x - M \cdot b_x \leq lb \tag{40f}$$

The set of constraints works in the following way to make sure the maximum is assigned to z. Equation (40a) makes sure that exactly one of b_{lb} and b_x is 1. M is supposed to be a constant so large that it can be treated as ∞. When $b_{lb} = 1$ and $b_x = 0$, Equations (40c) and (40d) will be trivially satisfied as $-\infty \leq 0$ and $\infty \geq 0$. Equations (40b), (40e) and (40f) will make sure that $z = lb$ and $x \leq lb$. When $b_{lb} = 0$ and $b_x = 1$, Equations (40e) and (40f) will be trivially satisfied as $-\infty \leq lb$. Equations (40c) and (40d) will make sure that $x = z$ and Equation (40b) will make sure that $z \geq lb$ so transitively $x \geq lb$.

2) Indicator constraints: Solvers such as Gurobi [14] and CPLEX [32] offer an alternative to Big-M constraints in the form of indicator constraints. With indicator constraints, we can store the result of a constraint in a binary variable. This binary variable can be further used to conditionally apply a constraint. Let b be a binary variable. We can encode the max function using the following set of constraints.

$$x > lb = b \tag{41a}$$
$$b = 0 \rightarrow z = lb \tag{41b}$$
$$b = 1 \rightarrow z = x \tag{41c}$$

The set of constraints works in the following way to make sure that the maximum is assigned to z. Equation (41a) assigns 0 to b if $x \leq lb$ else assigns 1 to b. Based on the value of b, either Equation (41b) or Equation (41c) is applied as a constraint and the other is ignored. If $b = 0$, it implies that $x \leq lb$ and $z = lb$ should be applied. If $b = 1$, it implies that $x > lb$ and $z = x$ should be applied.

VI. Evaluation

In the following, we compare the efficiency and robustness of our approach with closely-related prior work [11], [15].

A. Experimental Setup

1) Implementation: We implement our verification methodology in Python, which is available at https://github.com/iitkcpslab/QNNV. Our implementation takes neural network models in the NNet [33] format. To decide on the target solver, we ran a preliminary comparison between solvers using the *Collision Avoidance* benchmark (described below) to decide on the solver to use with our experiments. The results are reported in Table I. Gurobi performed the best among all the solvers we tried. It should be noted that ≈ 1033 seconds are spent in preprocessing and generating the MILP constraints from the neural network. So in most cases, the solving takes only a fraction of a second with Gurobi. Thus, it was a natural choice for us to implement our tool using Gurobi as the backend solver. Our implementation uses Gurobi v9.0.3 [14] to encode the constraints provided in Section V-B. Table II shows the Gurobi parameters we configure and their influence on the solver. In the rest of the paper, we refer to our work as the MILP encoding.

Prior work by Baranowski et al. [15] provides a set of primitives for describing fixed-point computation by extending PySMT [34]. These primitives can be rewritten to use an existing theory supported by current SMT solvers [35]–[37]. It can be reduced to either the theory of bitvectors or the theory of reals. We use the reduction to the theory of bitvectors since it performed better in their evaluation, and we refer to this encoding as the BV-SMT encoding. We use Boolector as the underlying SMT solver with PySMT, which is the same solver used by [15]. We do not test the reduction to the theory of reals since Boolector does not support the theory of reals. Furthermore, prior work [15] shows that the reduction to the theory of bitvectors performed better than the theory of reals.

2) Benchmarks: For the first benchmark, we use the MNIST dataset [38] containing images of handwritten digits which need to be classified. A *classifier* maps a n-dimensional input to one out of μ target classes. If the outputs of a classification network are $o_1, \ldots, o_\mu$, then the prediction of the network is given by $\texttt{class}(o_1, \ldots, o_\mu) = \arg\max_i \ o_i$. We trained three different

TABLE I: Preliminary comparison of solvers on the *Collision Avoidance* benchmark

Solver	Solved	Timeout	Exception	Total time (s)	Total time excluding encoding time (s)
Gurobi	500	0	0	1079	46
Gurobi (8-Threads)	500	0	0	1087	54
GLPK	309	191	0	12621	11588
CBC	495	3	2	2626	1593
CBC (8-Threads)	495	1	4	1795	762

TABLE II: Impact of Gurobi parameters on the solving algorithm

Variable	Purpose
`SolutionLimit = 1`	Stop after 1 solution is found
`MIPFocus = 1`	Focus on finding feasible solutions quickly
`IntFeasTol = 1e-9`	Tolerance for considering a floating-point value as integer
`DualReductions = 0`	Do not perform dual reduction. Useful for debugging
`InfUnbdInfo = 1`	Extra information when solution is infeasible or unbounded for debugging
`Threads = 8`	Number of threads to use for solving
`ConcurrentMIP = 8`	Spawn 8 independent MILP solvers, with different configurations

networks for verification using the TensorFlow framework [39]. The first network has an architecture of $784 \times 16 \times 16 \times 10$, referred to as MNIST-S (i.e., a small network). The second network has 10 hidden layers, each having 10 nodes, referred to as MNIST-D (i.e., a short and deep network). The third network has an architecture of $784 \times 256 \times 256 \times 10$, referred to as MNIST-T (i.e., a tall but shallow network). Along with our own trained benchmarks, we use a $784 \times 256 \times 256 \times 10$ network from the *mnistfc* benchmark from VNN-COMP [40], referred to as MNIST-FC.

We use the first 100 images from the test set of the MNIST dataset from the Keras [41] library to check for the *robustness* property. Prior work [10] defines robustness as follows: A sample s is robust when, for all perturbations within distance ε, the sample gets classified in the same class as s, denoted by c. $\varepsilon > 0$ is some notion of distance. If we denote $\mathbf{x}$ as a perturbation of s within ε distance and class $\circ$ f be the result of classification, then we can define the following property for the verification of robustness for s

$$|s - \mathbf{x}|_\infty \leq \varepsilon \implies c = \text{class} \circ f(\mathbf{x}). \tag{42}$$

We use three other popular benchmarks in our evaluation. *Collision Avoidance* [7] contains attributes about two vehicles, and the network predicts whether the two vehicles will collide. The network contains 40 linear nodes in the first layer, followed by a MaxPool layer where each node has 4 incoming edges, followed by a layer with 19 ReLU nodes, and a final layer with 2 ReLU nodes. There are 500 properties defined on the network based on *safety margins* around the tuples in the dataset.

In *TwinStream* [42], a network has two streams of data coming into the final layer. Both streams have the same architecture, weights, and inputs. The final layer computes the difference between the two streams and adds a positive bias; the output of the network is always equal to the bias added in the final layer. The benchmark contains 81 randomly-generated networks. The networks vary in depth, number of hidden nodes, number of inputs, and the value of the margin. The only property to check on all the 81 networks is that the output at the final layer is positive, which is true by construction.

TABLE III: Number of variables in each benchmark

Benchmark	Number of variables
MNIST-S	18,805
MNIST-D	15,229
MNIST-T	2,78,485
MNIST-FC	2,78,485
Collision Avoidance	5,253
ACAS Xu	15,480
TwinStream	10,608

The *ACAS Xu* benchmark [43] contains 45 DNNs to control an aircraft. Each network has six hidden layers of 50 nodes each. We use four properties that are applicable on all networks. The properties describe scenarios about the position and speed of both the aircraft and what advisories must not be given in those cases. The properties are described in detail in [22].

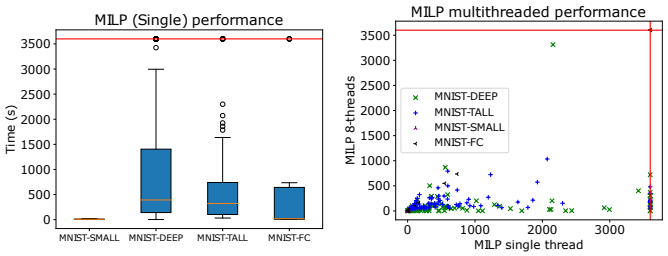
All the networks used in our evaluation have 4 bits allocated to the integer part and 4 bits allocated to the fractional part, i.e., Q4.4 except for MNIST-FC. MNIST-FC uses ϵ value of 0.05, which cannot be represented by a Q4.4 type. Consequently, it uses a Q3.5 type. To convert and encode floating-point networks into fixed-point, we have followed the semantics described in Section III. Table III contains the number of variables in the MILP encoding in our implementation. TwinStream benchmarks contain networks of various sizes. The number of variables reported in the table corresponds to the largest network in the TwinStream benchmark.

Platform. The experiments were run on a system with an Intel® Core™ i7-4770 processor, having 4 physical cores (8 logical with hyperthreading), running at 3.40 GHz and with 16 GB of memory. The single-threaded experiments were run simultaneously in groups of eight, as was done in [10]. For multithreaded experiments, we used 8 threads.

B. Results

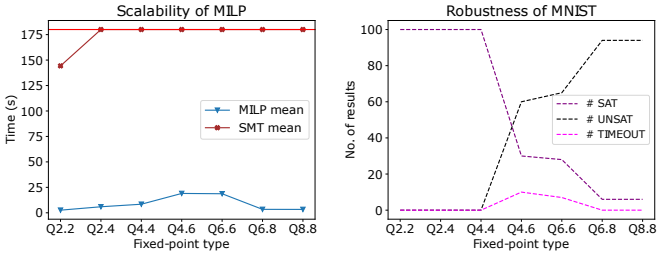
The evaluation results for comparison between MILP and BV-SMT encoding are summarized in Table IV. The detailed results are described below.

1) MNIST Dataset: The results for the 4 MNIST networks are shown in Figure 4. The timeout for the MNIST-S is set to



(a) (b)

Fig. 4: Comparison of the MILP implementations and BV-SMT on the MNIST networks. Red lines indicate timeout.



(a) (b)

Fig. 5: Scalability and robustness of the MILP encoding.

60s, while it is 3600s for the networks MNIST-D, MNIST-T, and MNIST-FC. Figure 4a and Table IV show that our approach is able to verify 290 out of 315 problems using single-threaded MILP and verify *all* 315 problems within the timeout limit with multithreaded MILP. The SMT solver timed out on *all* instances of the problem in the case of MNIST-S, MNIST-D, and MNIST-T. In case of MNIST-FC, BV-SMT consistently ran out of memory. In our experiments, it seems that the memory consumption in BV-SMT is highly sensitive to the number of bits in the fractional part. Our technique using MILP uses a relatively consistent amount of memory since the bit-width that the MILP solver works with does not change with changes in fixed-point types. These benchmarks show that our tool is well-equipped to handle short and deep networks (e.g., MNIST-D) as well as tall and shallow networks (e.g., MNIST-T and MNIST-FC). Figure 4b compares the performance of single-threaded and multithreaded MILP on the 4 networks. The red lines in Figure 4b indicate timeout values. The performance improvement on average for the MNIST-S network is modest ($\sim$36%) because the run time with a single thread is low. The speedup is more significant for the two larger networks: 4.5X for MNIST-D and 2.7X for MNIST-T on average.

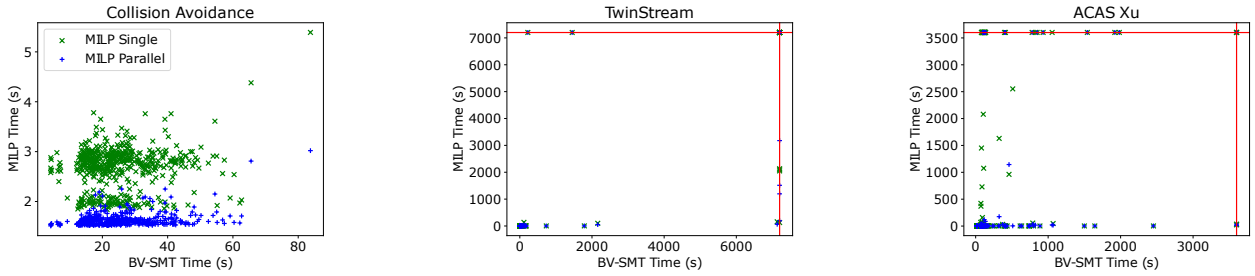
Scalability of MILP encoding: After Q2.2, all instances of BV-SMT time out because the run time of BV-SMT encoding increases quasi-exponentially [10]. The MILP encoding scales well, as the run time stays relatively stable with the increase in the number of bits. The reason is that the MILP solver does not work with the numbers of different bit widths. MILP solvers model variables with a double-precision floating-point type. The bit width for the variables stays the same for all problems while changing the fixed-point type for the network results in a change of bounds only. Figure 5b shows the detailed information about the instances verified by MILP encoding. Given the scalability of the MILP encoding, it can be used to search the space of fixed-point types and find robust types, which also minimize the number of bits required to achieve that level of robustness.

2) Collision Avoidance, TwinStream, and ACAS Xu: Figures 6a and 6b compare the performance of the MILP tool with BV-SMT on the benchmarks Collision Avoidance and TwinStream. The figure includes results for both single-threaded and multithreaded implementation of our proposed MILP approach. For Collision Avoidance, the single-threaded MILP implementation verified the properties 10X faster on average than BV-SMT. With multiple threads, the speedup increases to 15X on average. In the case of TwinStream, single-threaded MILP verified the properties 2.26X faster than BV-SMT, and parallel MILP verified the properties 2.5X faster than BV-SMT, on average. For TwinStream, BV-SMT timed out on 17 instances, while the single-threaded and the multithreaded MILP solver run timed out on 15 instances each.

Figure 6c shows the verification performance of ACAS Xu properties with the MILP solver configurations and BV-SMT. Single-threaded MILP verified the properties 2.5X faster, and the multithreaded MILP verified the properties 13X faster, on average, excluding instances that timed out. However, the MILP solver timed out on more instances than BV-SMT. Single-threaded implementation timed out on 40 instances while multithreaded MILP timed out on 20 instances, compared to BV-SMT, which timed out for 5 instances. We hypothesize that the MILP solver with a single thread got stuck in the wrong part of the search space and was unable to identify feasible solutions and hence timed out. We ran multiple different instances of the MILP solver using `ConcurrentMIP` with slightly different configurations to explore *different* areas of the search tree. Consequently, the multithreaded MILP was a significant improvement ($\sim$5.25X faster) on the single-threaded runs. The number of timeouts also *decreased* from 40 to 20 with multiple threads, validating our hypothesis. We are investigating other reasons for the performance difference.

3) Comparison with [11]: In a recent work [11], the authors present an improved bitvector encoding for SMT, which we refer to as BV2-SMT in this paper. In order to compare with BV2-SMT, we run the MILP implementation on the networks used by [11], where the network trained on the MNIST dataset is referred to as MNIST-C, and the network trained on the Fashion-MNIST dataset is referred to as FASHION-C. We run the benchmark with type Q4.2 for all the nodes using a single thread for a fair comparison, same as [11]. We verified the same set of properties. More specifically, we used the first 400 MNIST and fashion MNIST images where $\epsilon = 1$ for first 100, $\epsilon = 2$ for next 100, $\epsilon = 3$ for the next 100 and $\epsilon = 4$ for the last 100. Here ϵ represents the maximum deviation allowed for any node in the input, i.e., L_∞ norm.

A comparison of MILP encoding with BV2-SMT is provided in Table V. MILP encoding performs 15X faster in case of MNIST benchmarks and 8.5X faster in case of fashion MNIST benchmark, on average. The difference between mean and medians in the case of MILP encoding is negligible, showing that MILP encoding consistently performs better than BV2-



(a) (b) (c)

Fig. 6: Comparison of MILP with the theory of bitvectors in SMT on the Collision Avoidance, TwinStream, and ACAS Xu benchmarks. Red lines indicate timeout.

TABLE IV: Performance comparison of MILP and BV-SMT on all the benchmarks. TO = timeout.

Benchmarks	**# Props**	**TO**	**Computation time** (s) (Mean | Median)			**# Timeouts**		
			MILP		BV-SMT	MILP		BV-SMT
			Single	Parallel		Single	Parallel	
MNIST-S	100	60s	8.429|7.920	6.183|4.560	NaN	0	0	100
MNIST-D	100	3600s	592.3|319.1	128.8|51.04	NaN	14	0	100
MNIST-T	100	3600s	450.5|261.1	162.2|106.8	NaN	8	0	100
MNIST-FC	15	3600s	427.1|15.71	253.8|30.99	–	3	0	–
CollAvoid	500	3600s	2.682|2.770	1.630|1.610	25.83|22.80	0	0	0
TwinStream	81	7200s	105.1|1.710	96.07|1.330	238.3|12.41	15	15	17
ACAS Xu	180	3600s	88.90|3.380	16.94|4.160	229.6|97.55	40	20	5

TABLE V: Comparison with [11]. TO = 180s

Benchmark	**# Props**	**Time** (s) (Mean | Median)		**# Timeouts**	
		MILP	BV2	MILP	BV2
MNIST-C	400	5.53|5.4	90|5	0	82
FASHION-C	400	5.73|5.46	49|4	0	206

SMT and not just on average. Moreover, MILP encoding verified all 800 instances, while BV2-SMT timed out on 288 instances. Please note that our experimental setup is slightly different, but is comparable to [11].

VII. Conclusion

To the best of our knowledge, this work is the first to present a methodology to encode the quantized DNN verification problem into an MILP problem. We present a sound round-down procedure for MILPs and prove its correctness. We also present a set of constraints for encoding a fixed-point network as an MILP problem. We compare our results with closely-related prior work and show that our MILP encoding is faster by an order of magnitude in most cases.

Though our MILP encoding provides significant performance improvement compared to prior work, its performance in solving some problems may be less than satisfactory, as evident by the number of timeouts experienced for the ACAS Xu benchmark. This is because the DNN verification problem is NP-complete, and there may be instances where the heuristics employed by the MILP solver are not effective in finding a solution quickly. In future work, we would like to investigate the reason for the timeouts encountered for ACAS Xu. Furthermore, we would like to incorporate more network-level reasoning into the MILP encoding. For example, one can perform a range analysis before verification to disable those ReLUs that will not take values on both sides of 0. Such preprocessing is expected to speed up the verification process, as decreasing the amount of nonlinearity is known to help verification tools significantly.

References

[1] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[2] G. Liu, A. Siravuru, S. P. Selvaraj, M. M. Veloso, and G. Kantor, "Learning End-to-end Multimodal Sensor Policies for Autonomous Navigation," in *1st Annual Conference on Robot Learning (CoRL)*, ser. Proceedings of Machine Learning Research, vol. 78, 2017, pp. 249–261.

[3] H. Xu, Y. Gao, F. Yu, and T. Darrell, "End-to-End Learning of Driving Models from Large-Scale Video Datasets," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 3530–3538.

[4] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, "End-to-End Race Driving with Deep Reinforcement Learning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 2070–2075.

[5] P. Varshney, G. Nagar, and I. Saha, "DeepControl: Energy-Efficient Control of a Quadrotor using a Deep Neural Network," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 43–50.

[6] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal Security Analysis of Neural Networks using Symbolic Intervals," in *USENIX Conference on Security Symposium (SEC)*, 2018, pp. 1599–1614.

[7] R. Ehlers, "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks," in *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2017, pp. 269–286.

[8] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. L. Dill, M. J. Kochenderfer, and C. W. Barrett, "The Marabou Framework for Verification and Analysis of Deep Neural Networks," in *Computer Aided Verification - 31st International Conference (CAV)*, 2019, pp. 443–452.

[9] H.-D. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems," in *International Conference on Computer Aided Verification (CAV)*, 2020, pp. 3–17.

[10] M. Giacobbe, T. A. Henzinger, and M. Lechner, "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 (TACAS)*, 2020, pp. 79–97.

[11] T. A. Henzinger, M. Lechner, and D. Zikelic, "Scalable Verification of Quantized Neural Networks," in *Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, 2021, pp. 3787–3795.

[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[13] G. Sierksma and Y. Zwols, *Linear and Integer Optimization: Theory and Practice*. CRC Press, 2015.

[14] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: http://www.gurobi.com

[15] M. Baranowski, S. He, M. Lechner, T. S. Nguyen, and Z. Rakamarić, "An SMT Theory of Fixed-Point Arithmetic," in *International Joint Conference on Automated Reasoning (IJCAR)*, 2020, pp. 13–31.

[16] M.-Y. Chow and S. O. Yee, "A Measure of Relative Robustness for Feedforward Neural Networks Subject to Small Input Perturbations," *International Journal of Neural Systems*, vol. 3, no. 03, pp. 291–299, 1992.

[17] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[18] L. Pulina and A. Tacchella, "Checking Safety of Neural Networks with SMT Solvers: A Comparative Evaluation," in *AI*IA 2011: Artificial Intelligence Around Man and Beyond*, 2011, pp. 127–138.

[19] C. Cheng, G. Nührenberg, and H. Ruess, "Maximum Resilience of Artificial Neural Networks," in *Automated Technology for Verification and Analysis - 15th International Symposium (ATVA)*, 2017, pp. 251–268.

[20] W. Xiang, H. Tran, and T. T. Johnson, "Output Reachable Set Estimation and Verification for Multilayer Neural Networks," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 29, no. 11, pp. 5777–5783, 2018.

[21] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An Abstract Domain for Certifying Neural Networks," *Proceedings of the ACM on Programming Languages (POPL)*, vol. 3, pp. 1–30, 2019.

[22] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *International Conference on Computer Aided Verification (CAV)*, 2017, pp. 97–117.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS)*, 2016, pp. 4114–4122.

[24] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying Properties of Binarized Deep Neural Networks," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2018, pp. 6615–6624.

[25] N. Narodytska, H. Zhang, A. Gupta, and T. Walsh, "In Search for a SAT-friendly Binarized Neural Network Architecture," in *International Conference on Learning Representations (ICLR)*, 2019.

[26] K. Jia and M. Rinard, "Efficient Exact Verification of Binarized Neural Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[27] A. Shih, A. Darwiche, and A. Choi, "Verifying Binarized Neural Networks by Angluin-Style Learning," in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2019, pp. 354–370.

[28] G. Amir, H. Wu, C. W. Barrett, and G. Katz, "An SMT-Based Approach for Verifying Binarized Neural Networks," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2021, pp. 203–222.

[29] E. Oberstar, "Fixed-Point Representation and Fractional Math (revision 1.2)," 2007, http://darcy.rsgc.on.ca/ACES/ICE4M/FixedPoint/FixedPointRepresentationFractionalMath.pdf.

[30] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010, pp. 807–814.

[31] J. Han and C. Moraga, "The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning," in *International workshop on artificial neural networks*, 1995, pp. 195–201.

[32] IBM ILOG Cplex, "User's Manual for CPLEX V12.1," *International Business Machines Corporation*, 2009.

[33] K. Julian, "NNet file format," 2018, https://github.com/sisl/NNet.

[34] M. Gario and A. Micheli, "PySMT: A Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms," in *SMT Workshop*, 2015.

[35] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[36] B. Dutertre and L. De Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *International Conference on Computer Aided Verification (CAV)*, 2006, pp. 81–94.

[37] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014.

[38] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST Database of handwritten digits," 2009, http://yann.lecun.com/exdb/mnist.

[39] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[40] "2nd International Verification of Neural Networks Competition (VNN-COMP'21)," 2021, https://sites.google.com/view/vnn2021.

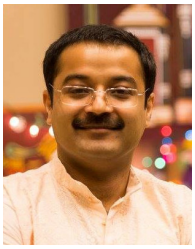
[41] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[42] R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar, "Piecewise Linear Neural Networks Verification: A Comparative Study," Technical Report, 2018, https://arxiv.org/abs/1711.00455v1.

[43] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy Compression for Aircraft Collision Avoidance Systems," in *IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–10.

Samvid Mistry is a Software Engineer at GitHub. He obtained his Diploma in Computer Engineering from Institute of Diploma Studies, Nirma University in 2016, B.E. degree in Computer Engineering from Vishwakarma Government Engineering College, Ahmedabad in 2019, and M.Tech. degree in Computer Science and Engineering from Indian Institute of Technology Kanpur in 2021. After completing M.Tech., he joined GitHub as a Software Engineer. His research interests lie in formal verification, high performance computing, and machine learning.

Indranil Saha (M'12) is an Associate Professor in the Department of Computer Science and Engineering at IIT Kanpur. Prior to joining IIT Kanpur in 2015, he was a postdoctoral researcher affiliated with the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley and the Department of Computer and Information Science at the University of Pennsylvania. He obtained his B.Tech. degree in Electronics and Communication Engineering from Kalyani Government Engineering College in 2003, M.Tech. degree in Computer Science from Indian Statistical Institute, Kolkata in 2005, and Ph.D. degree in Computer Science from the University of California Los Angeles in 2013. From 2005 to 2008, he was a research scientist at Honeywell, Bangalore. His research interest lies in the application of formal methods to embedded and cyber-physical systems and robotics. He was a recipient of the Best Paper Award at the ACM SIGBED International Conference on Embedded Software (EMSOFT) in 2010. He received the ACM SIGBED Frank Anger Memorial Award in 2012 for his contribution in the intersection of embedded systems and software engineering.



Swarnendu Biswas is an Assistant Professor at the Department of Computer Science and Engineering at the Indian Institute of Technology Kanpur. Swarnendu has a BE in Computer Science and Engineering from the National Institute of Technology Durgapur, an MS in Computer Science and Engineering from Indian Institute of Technology Kharagpur, and a Ph.D. from the Ohio State University. Swarnendu was a postdoctoral fellow at the University of Texas at Austin before joining Kanpur, and has also worked as a software developer at Wipro Technologies for three years. His research interests are Programming Languages, Compilers and Runtime Systems, and parallel Software Systems.