UNIVERSIDADE FEDERAL DO AMAZONAS-UFAM
FACULDADE DE TECNOLOGIA-FT
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA-PPGEE

Automated Verification and Refutation of Quantized Neural Networks

Luiz Henrique Coelho Sena

MANAUS-AM
2021

Luiz Henrique Coelho Sena

Automated Verification and Refutation of Quantized Neural Networks

Qualification document presented to the Postgraduate Program of Electrical Engineering (PPGE) of the Federal University of Amazonas (UFAM) as one of the prerequisites for Msc title obtention.

Mentor: Lucas Carvalho Cordeiro

MANAUS-AM
2021

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S474a

Sena, Luiz Henrique Coelho
Automated verification and refutation of quantized neural networks / Luiz Henrique Coelho Sena, Lucas Carvalho Cordeiro. 2021
55 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro
Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. Model checking. 2. Neural networks. 3. Quantized neural networks. 4. Artificial intelligence. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas III. Título

LUIZ HENRIQUE COELHO SENA

AUTOMATED VERIFICATION AND REFUTATION OF QUANTIZED NEURAL NETWORKS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração Controle e Automação de Sistemas.

Aprovada em 04 de março de 2022.

BANCA EXAMINADORA

Prof. Dr. Lucas Carvalho Cordeiro, Presidente

Universidade Federal do Amazonas

Prof. Dr. Eddie Batista de Lima Filho, Membro

TP Vision

Profª. Drª. Eulanda Miranda dos Santos, Membro

Universidade Federal do Amazonas

Abstract

Artificial Neural Networks (ANNs) are being deployed for an increasing number of safety-critical applications, including autonomous cars and medical diagnosis. However, concerns about their reliability have been raised due to their black-box nature and apparent fragility to adversarial attacks. These concerns are amplified when ANNs are deployed on restricted system, which limit the precision of mathematical operations and thus introduce additional quantization errors. Here, we develop and evaluate a novel symbolic verification framework using software model checking (SMC) and satisfiability modulo theories (SMT) to check for vulnerabilities in ANNs and mainly in Multilayer Perceptron (MLP). More specifically, here is proposed several ANN-related optimizations for SMC, including invariant inference via interval analysis, slicing, expression simplifications, and discretization of non-linear activation functions. With this verification framework, we can provide formal guarantees on the safe behavior of ANNs implemented both in floating- and fixed-point arithmetic. In this regard, the current verification approach was able to verify and produce adversarial examples for 52 test cases spanning image classification and general machine learning applications. Furthermore, for small- to medium-sized ANN, this approach completes most of its verification runs in minutes. Moreover, in contrast to most state-of-the-art methods, the presented approach is not restricted to specific choices regarding activation functions and non-quantized representations. Experiments show that this approach can analyze larger ANN implementations and substantially reduce the verification time compared to state-of-the-art techniques that use SMT solving.

Keywords: Model checking, Neural networks, Quantized neural networks.

Contents

1 Introduction **1**
1.1 Problem Description . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 3
1.4 Dissertation Organization . . . 3

2 Preliminaries **4**
2.1 Artificial Neural Networks (ANNs) . . . 4
2.2 Quantized Neural Networks (QNNs) . . . 5
2.3 Safety properties for ANNs and QNNs . . . 6
2.4 Satisfiability Modulo Theories (SMT) . . . 7
2.5 Existing SMT approaches for ANNs and QNNs . . . 7

3 Methodology **9**
3.1 ANN code generation . . . 9
3.2 Models for fixed-point ANN implementations . . . 10
3.3 Discretization of non-linear activation functions . . . 12
3.4 Introducing safety properties in ANN code . . . 14
3.5 Invariant inference via interval analysis . . . 15
3.6 Model checking ANN implementations . . . 16
3.7 Incremental verification using lemma learning via SMT . . . 17
3.8 Constant folding, slicing and expression balancing for search-space reduction 18
3.9 Illustrative example: robustness to adversarial images . . . 20

4 Experimental Evaluation **24**
4.1 Quantization aspects and data adaptation . . . 25
4.2 Description of the benchmarks . . . 25
4.3 Ablation study . . . 27
4.4 Verification of quantized ANNs . . . 34
4.5 Comparison with state-of-the-art verification tools . . . 39
4.6 Limitations . . . 42

5 Related Work **44**

6 Conclusion **46**
6.1 Future Works . . . 47

Bibliography **48**

Chapter 1

Introduction

1.1 Problem Description

Artificial neural networks (ANNs) are soft computing models usually employed for regression, machine learning, decision-making, and pattern recognition problems [1], which have been recently used to perform various safety-critical tasks. For instance, ANNs are employed for Covid-19 diagnosis [2], and for performing steering commands in self-driving cars [3]. Unfortunately, in such contexts, incorrect classifications can cause serious problems. Indeed, adversarial disturbances can make ANNs misclassify objects, thus causing severe damage to users of safety-critical systems. For instance, Eykholt *et al.* [4] showed that noise and disturbances, such as graffiti on traffic signals, could result in target misclassification during the operation of computer vision systems. Moreover, given that ANNs are notorious for being difficult to interpret and debug, the whole scenario becomes even more problematic [5], which then claims for techniques able to assess their structures and verify results and behaviors. For this reason, there is a growing interest in verification methods for ensuring safety, accuracy, and robustness for neural networks. The approaches for ANN verification may be divided into three groups: optimization [6, 7, 8, 9], reachability [10, 11, 12, 13, 14, 11, 15], and satisfiability [16, 17, 18, 19].

On the one hand, optimization-based algorithms pose the safety verification problem as an optimization one, in which safety properties are usually treated as constraints, as described by Tjeng *et al.* [20]. The main difficulty of optimization methods, such as mixed-integer linear programming [21, 20, 8], branch and bound [7], and semi-definite programming [6], is to deal with constraints that are non-linear and non-convex due to a network's complex structure and its activation functions. Indeed, it is still possible to employ dual optimization for simplifying those constraints and then obtaining a convex problem [22]; however, completeness tends to be lost due to relaxations. On the other hand, reachability-based approaches aim at computing the reachable set of an ANN by propagating input sets through it, layer-by-layer, while checking whether some unsafe state (violation) belongs or not to that same reachable set. The main advantage of those methods is that they are usually sound, i.e., if the algorithm indicates that a network is unsafe, its safety property is violated. However, the computational cost to compute exact reachable sets becomes unreasonable for more complex ANNs and more extensive input spaces. In order to avoid such a problem, a reachable set is over-approximated by using symbolic [14, 18, 15] and/or set-theoretic methods [11, 12]. Although those tools effectively reduce the computational cost of reachability sets, it is still challenging to over-approximate ANN's non-linear elements, particularly their activation functions. There

are some symbolic techniques suitable for dealing with over-approximation of activation functions [13, 10]; however, most of the approaches available in literature are only able to approximate piecewise-linear and rectified linear unit (ReLU) activation functions.

Finally, satisfiability modulo theories (SMT) encode both ANN and desired safety property into a single logic formula using a decidable fragment of first-order logic, and then check whether a counterexample exists. In this regard, only binarized neural networks [23, 24] can be encoded into boolean logic and verified with existing SAT solvers [25, 16]. More complex ANNs, whether implemented in floating- or fixed-point [26, 27], the latter aiming at efficiency and simplicity, require the use of first-order logic instead of propositional logic to exploit more abstract and less expensive techniques to solve the problem at hand. For example, SMT solvers often integrate a simplifier, which applies standard algebraic reduction rules and contextual simplification to simplify the logical formula. Regarding these, several SMT-based approaches have been proposed [28, 17, 18, 14, 29, 30, 19]. While SMT background theories allow those approaches to model the semantic of neural operations exactly using word-level theories, the resulting verification problem is challenging to solve [28]. In this respect, quantization, i.e., a representation with a lower number of bits, has been proven to make this problem even computationally harder [19]. As a consequence, most existing approaches specialize in simple piecewise-linear activation functions [18, 29, 30], focus on the floating-point scenario only [14], or require domain-specific abstractions [17].

1.2 Objectives

This work aims to design a tool capable of efficiently verifying any desirable safety property in quantized and nonquantized ANNs. Our overall objective can be achieved by fulfilling the following specific objectives:

- Train ANNs with consolidated training algorithms, *e.g.*, backpropagation along cross-validation.
- Model ANNs, *e.g.*, vocalic recognition and iris recognition ANNs are some of the ANNs that are modeled here. Basically, ANNs must be converted to linear equations and activation functions model that ESBMC supports.
- Model quantized ANNs. Quantized ANN models will be obtained from converting floating point ANN models into fixed point models.
- Model safety properties. These properties are based in how safe a classification can be regarding range errors and then they are modeled as `assume` and `assert` statements.
- Find the best parameters in ESBMC that provide the best verification performance to our models. An ablation study is further presented in order to reach the best ESBMC parameters for each ANN verification.
- Analyze counterexamples and adversarial examples. This objective relies on comparing our approach with some of the state of the art techniques and analyzing the impact of our models and techniques on the verification process.

1.3 Contributions

We propose a novel approach to verify both fixed- and floating-point ANN implementations. Our main idea is to look at the source code of an ANN rather than the abstract mathematical model behind it. By doing so, we can then leverage many recent advances in software verification that can dramatically increase the computational efficiency of verification processes, as observed in our experimental evaluation. More specifically, in this research, we make the following original contributions:

- We cast the ANN verification problem into a software verification one dealing with real ANNs code implementation. On the one hand, we propose a method to represent ANN safety properties as pairs of `assume` and `assert` instructions. On the other hand, we explain how to represent fixed- and floating-point operations in a quantized ANN, using direct implementations of their behavior, i.e., representations that consider a target precision.

- We introduce several pre-processing steps to increase the efficiency of downstream software verification tools. Namely, we give a principled method to discretize non-linear activation functions and replace them with lookup tables. Furthermore, we show how to bound the feasible range of each variable with interval analysis and how to represent those bounds with additional `assume` instructions.

- We detail which existing techniques for search-space reduction can be borrowed from the software verification literature, and we empirically evaluate their individual and cumulative effects.

- We evaluate our approach on fixed- and floating-point ANNs and give empirical evidence on its computational efficiency. In particular, we show that we can verify ANNs with hundreds of neurons in less than an hour.

- We compare our approach with state-of-the-art (SOTA) techniques, including quantized and floating-point tools. According to the comparison, since our method applies various optimization techniques before invoking the SMT solver, we have better performance than other SMT-based verification tools.

1.4 Dissertation Organization

. In Chapter 2, we introduce the ANN verification problem and present existing satisfiability modulo theories. In Chapter 3, we detail all the steps involved in our code-level verification approach for ANNs. In Chapter 4, we empirically test our approach on ANN classifiers trained on the classic Iris dataset and an image recognition dataset. In Chapter 5, we give a broader review of the recent trends in verifying ANNs. In Chapter 6, we conclude and outline possible future work.

Chapter 2

Preliminaries

Before introducing the details of our verification approach, let us review some important concepts related to the verification of artificial neural networks.

2.1 Artificial Neural Networks (ANNs)

Modern ANNs are universal function approximators built by composing multiple copies of the same basic building block, called neuron [1]. In other words, they provide a way of constructing system models with a set of sample observations, in such a way that the joint behavior of existing neurons is correctly adjusted. In their most common form, each neuron k is itself the composition of two functions, as illustrated in Fig. 2.1. The first one is an affine projection of the m local inputs, often referred to as the *activation potential* u_k. The second one is a non-linear transformation of the resulting potential, often referred to as *activation function* $\mathcal{N}_k$. Together, they define the following mapping $n_k : \mathbb{R}^m \to \mathbb{R}$:

$$y_k = \mathcal{N}_k(u_k), \tag{2.1}$$

where

$$u_k\left(x\right) = \sum_{j=1}^{m} w_{j,k} x_j + b_k. \tag{2.2}$$

Finally, b_k provides a way of directly shifting a given activation function.

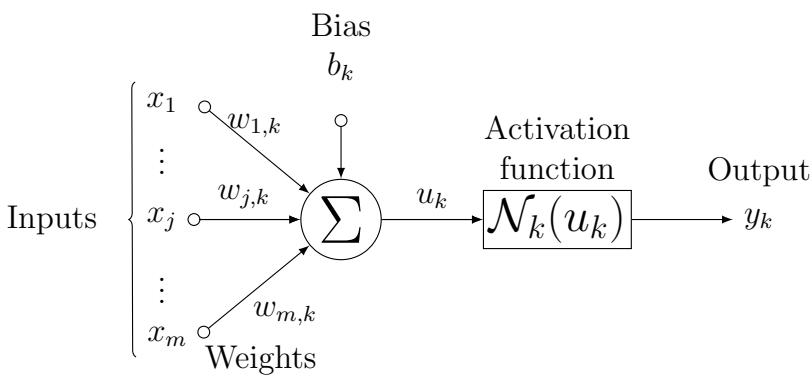


Figure 2.1: The detailed view of a single neuron n_k.

The behavior of the basic neuron in Fig. 2.1 depends on the values of its weights w_k and also on the chosen activation function $\mathcal{N}_k$. In this regard, researchers have experimented with a wide range of functions, including non-monotonic [31, 32], non-continuous [1], and unbounded ones [33, 34]. In our experiments, which are available in Chapter 4, we cover the most popular activation functions: namely, ReLU, sigmoid (Sigm), and the re-scaled version of the latter known as hyperbolic tangent (TanH):

$$\mathcal{N}_{\text{ReLU}}(u_k) = \max(0, u_k) \tag{2.3}$$

$$\mathcal{N}_{\text{Sigm}}(u_k) = \left(1 + e^{-u_k}\right)^{-1} \tag{2.4}$$

$$\mathcal{N}_{\text{TanH}}(u_k) = 2\mathcal{N}_{\text{Sigm}}(2u_k) - 1. \tag{2.5}$$

At the same time, one may notice that many state-of-the-art verification tools for ANNs are only compatible with ReLU and similar piece-wise linear activation functions [18, 29, 30]. Moreover, those that do support more activation functions [13, 10, 14] often incur a significant performance hit, when solving the resulting non-linear verification problem. In contrast, the discretization technique we propose in Section 3.3 allows us to efficiently verify ANNs with any form of activation function.

Besides, our verification methodology is general enough to be applied to a large variety of ANN architectures. Specifically, we support any feedforward, feedforward DNN and feedforward MLP that are built from the composition of the basic neuron model in Fig. 2.1. Similar to what has been reported in existing ANN verification studies [35, 17, 18], the primary factor influencing our verification time is the number of non-linearities, in a neural network, rather than its architecture (see Section 4).

2.2 Quantized Neural Networks (QNNs)

As the deployment of ANNs in software applications becomes widespread, concerns about power consumption and complexity of large models increase. In this light, one of the main techniques to reduce energy requirements related to ANN inference is *quantization* [26], which further restrict operations required to compute the output of each neuron (see (2.1) and (2.2)) to integer [27] or even binary representations [23, 24]. State-of-the-art methods to perform such a transformation significantly improve the low-power feature of ANNs while retaining the original predictive accuracy [36].

At the same time, the discretized nature of quantized neural networks (QNNs) generates unique challenges regarding their verification [19]. More specifically, the output and intermediate computations performed by a network may differ from their floating-point counterparts. Thus, verification tools that operate on non-quantized ANN may return incorrect results.

We demonstrate this with the following motivating example. Assume that we want to verify the neural network in Fig. 2.2, which relies on the activation function ReLU and whose output can be directly computed as:

$$f(x_1, x_2) = A + B = \text{ReLU}(2x_1 - 3x_2) + \text{ReLU}(x_1 + 4x_2). \tag{2.6}$$

Furthermore, assume that, in our example application, the output of this ANN must never fall below $f(x_1, x_2) \geq 2.7$, and that we want to verify whether this is true for the input $(x_1, x_2) = (0.749, 0.498)$.

Now, if we run an experiment with real numbers $\mathbb{R}$ (from the mathematical domain), the result is $f(0.749, 0.498) = 2.745$, which satisfies our safety property $f(x_1, x_2) \geq 2.7$.

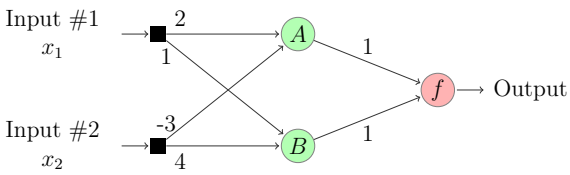


Figure 2.2: A simple fully-connected neural network with ReLU activations and biases set to zero (not shown).

However, if the same ANN is quantized to a lower precision, this is not the case anymore. Indeed, for a QNN with 4-bit integer and 6-bit fractional precision, its output becomes $\hat{f}(0.749, 0.498) = 2.6867$, which violates our property. It is worth mentioning that such discrepancies can be even worse when larger ANNs are employed, due to cumulative error in long computation chains. Thus, in our verification approach, we make sure the actual implementation model used in an ANN implementation is captured (see Section 3.2).

Besides, we could formulate another research question of interest: what is the deepest quantization that can be applied to a given ANN so that it makes correct decisions? This way, for instance, we would be able to target heavily restricted devices while still keeping the implementation correctness based on formal guarantees. Although that is not the focus of the present work, it provides the first step towards that goal. Moreover, it paves the way for a complete verification framework suitable to ANN implementations in embedded devices.

2.3 Safety properties for ANNs and QNNs

Let us now formalize the concept of safety property we briefly mentioned in the previous Section 2.2. In general, a safety property defines the set of states that a system is designed to reach safely. In software verification, such properties are usually defined according to a user's domain knowledge, which allows him to state which program behaviors are safe [37]. In ANN verification, the black-box nature of their associated computation means that safety properties are usually defined on the inputs and outputs alone [38, 39]. In this research, we often refer to safety properties in the following form:

$$\mathbf{x} \in \mathcal{H} \implies f(\mathbf{x}) \in \mathcal{G}, \tag{2.7}$$

where $\mathbf{x}$ is an input vector, $\mathcal{H}$ is an input region, $f(\mathbf{x})$ is their corresponding output, and $\mathcal{G}$ is an output region. However, one may notice that our verification method supports any safety property that can be expressed in first-order logic (see Section 3.4).

A powerful and general way to define an input region $\mathcal{H}$ is choosing a center point $\mathbf{x} \in \mathcal{D}$ in the input domain $\mathcal{D}$, and letting the set $\mathcal{H}(\mathbf{x}, d_{in})$ cover the whole neighborhood of points around it that are within a given distance $d_{in}(\mathbf{x}, \mathbf{x}') \leq 1$ [38, 39]. As an example, in the field of image classification, robustness properties are defined in this way [40]. For continuous input domains $\mathcal{D} \equiv \mathbb{R}^m$, such a distance is often defined in terms of the family of p-norms as follows:

$$d_p(\mathbf{x}, \mathbf{x}') = ||\mathbf{x}, \mathbf{x}'||_p = \Big(\sum_{i=1}^{m} |x_i - x_i'|^p \Big)^{\frac{1}{p}}, \qquad \textit{with } p \in [1, \infty), \tag{2.8}$$

where $p = 1$ is the Manhattan distance and $p = 2$ is the Euclidean distance. Furthermore, this definition can be extended to $p = \infty$ by introducing the so-called infinity or maximum

norm $d_\infty(x, x') = \max_i(|x_i - x'_i|)$. Note that input regions defined through p_∞ can be described by a set of linear constraints, a fact that makes them attractive to the verification community for efficiency reasons [18, 15, 41]. Also, input vectors can be re-scaled using a diagonal matrix Z, allowing us to define hyper-ellipsoids (if $p = 2$) and hyper-rectangles (if $p = \infty$) in the input space:

$$\hat{d}_p(\mathbf{x}, \mathbf{x}', Z) = d_p(Z\mathbf{x}, Z\mathbf{x}'). \tag{2.9}$$

Moreover, further attention is required if the input domain $\mathcal{M}$ is discrete in nature, for instance, in natural language processing (NLP) applications. However, a mapping to a continuous space is often available [42].

Once we establish a definition for the input set $\mathcal{H}$ in (2.7), we can complete the definition of our safety property by choosing the corresponding output set $\mathcal{G}$ [39]. For regression tasks, we can again define a safe neighborhood around an output point $f(\mathbf{x})$ within a given distance $d_{out}(f(\mathbf{x}), f(\mathbf{x}')) \leq 1, \forall \mathbf{x}' \in \mathcal{H}(\mathbf{x}, d_{in})$. For classification tasks, the output set $\mathcal{G}$ often comprises all points that assign the highest score to the desired class, e.g., $\mathcal{G} \equiv \{\mathbf{y} | (\mathbf{y} = f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{D}) \wedge (y_i > y_j, \forall j \neq i)\}$ for output class i. In Section 3.4, we show how to define this kind of safety properties inside our verification tool.

2.4 Satisfiability Modulo Theories (SMT)

Once we have defined a safety property P, according to (2.7), we need to verify that it always holds for our (quantized) neural network. As we mentioned in Chapter 1, there exist many approximate techniques to do so. However, in this research, we focus on bit-precise verification via *satisfiability modulo theories* (SMT) solvers [43].

Similar to Boolean Satisfiability (SAT) solving [44], the SMT approach to verification works by converting a verification problem at hand into a logic formula and then checking whether it is satisfiable. However, SMT extends SAT beyond boolean logic and allows us to model a verification problem as a decidable subset of first-order logic. At the same time, the interpretation of these models is restricted to a combination of *background theories*, which are written in first-order logic with equality. More formally, given a first-order formula F, encoding a verification problem, and a background theory T, we say that F is T-satisfiable if and only if there exists an assignment such that the union $F \cup \{T\}$ is satisfiable.

The modeling power of SMT comes from the variety of background theories T that we can use. Those theories model the semantic of common mathematical objects like real, floating-point, and integer numbers, arrays, lists, bit vectors, and the operations defined on them for computational problems [45]. While modeling capabilities of SMT are still being extended to new domains (e.g., the work of de Salvo Braz [46]), mainstream SMT solvers (e.g., Z3 [47], CVC4 [48], and Boolector [49]) already offer native support for all theories above.

2.5 Existing SMT approaches for ANNs and QNNs

SMT approaches have been applied to an extensive range of verification problems [43]. In this section, we review existing approaches for ANNs and QNNs. One may notice that due to the SMT paradigm flexibility, such approaches vary in the abstraction level at which they tackle a verification problem.

Early research applied existing SMT solvers to the verification of real-valued ANNs and showed some difficulties in scaling beyond toy examples [28]. More recently, Katz *et al.* proposed to extend the background theory of real numbers and include an extra predicate for the ReLU activation function [18]. Since each ReLU doubles the number of verification formulas, they introduced a dedicated lazy solver, called Reluplex, which only visits a relevant subset of formulas. Their algorithm was subsequently extended to arbitrary piecewise-linear activation functions [14]. An alternative approach by Huang et al. asks a user to define a problem-dependent set of micro-manipulations that the SMT solver chains to search a state space [17]. Thus, they can scale to medium-sized ANNs for image classification. Furthermore, verification methods based on real number computation can be extended to cover floating-point implementations of ANNs [18, 41].

In contrast, SMT methods to verify QNNs have to contend with a more challenging computational problem, from the theoretical perspective [19]. In this respect, Giacobbe *et al.* chose to represent QNN operations with the bit-vector background theory and showed that the associated verification results can be very different from their real and floating-point counterparts [30]. Similarly, Baranowski *et al.* proposed a new fixed-point background theory and tested it on some small QNNs [29]. In general, low-level optimizations in SMT encoding of QNNs are shown to speed up verification processes considerably [30, 19]. In the extreme case of binarized neural networks, where quantization only allows two binary states for each variable, a verification problem can be reduced to SAT solving [16]. In addition, hardware-level optimizations are crucial for efficiency too [25].

In summary, our methodology is a generalization of Sena *et al.* focused on SMT verification of CUDA implementations of ANNs [50]. As we expound in Chapter 3, we take advantage of existing techniques in software verification to model both ANNs and QNNs as SMT formulas. Our novelty lies in the encoding of fixed-point operations and the efficient treatment of non-linear activation functions, which allows us to verify networks beyond the simple ReLU function.

Chapter 3

A methodology for Verifying Quantized Neural Networks

While we usually think of neural networks as mathematical models, their implementation is actually written in source code, in a given language. Thus, in this respect, neural networks can be treated like any other piece of software. The advantage of this strategy is twofold. First, we can readily adapt many existing software verification techniques to ANNs and QNNs. Second, we give a user access to these highly technical verification tools in a familiar coding framework.

This section lists the sequence of steps required to verify ANNs in such a way. To this end, we assume that an ANN is given as input in the form of a piece of single-threaded C code (see Section 3.1). Furthermore, we explain how to represent a quantized ANN by calling our finite-word length (FWL) implementation models, which are discussed in Section 3.2. Likewise, we show how to discretize each activation function with the algorithm in Section 3.3.

Once the code has been prepared in this way, the user can specify the desired safety property with `assume` and `assert` statements, as detailed in Section 3.4. Then, we compute a reachable set of values for each variable, using the invariant inference techniques in Section 3.5. Finally, we verify the safety property via SMT model checking, as explained in Section 3.6. All the techniques we use to reduce the search space of the SMT solver are listed in Sections 3.7 and 3.8.

Our whole verification methodology is summarized in Fig. 3.1. Furthermore, we conclude in Section 3.9 with a complete walk-through example of our workflow.

3.1 ANN code generation

The current approach to ANN development uses machine learning libraries such as TensorFlow and PyTorch to define the architecture of the neural network and train its weights. Despite those frameworks are efficient for training ANNs, the current approach deals with the trained ANN regardless how it was trained. Once the development phase is over, a final implementation of the ANN is produced, targeting a specific computer architecture, e.g., AMDx64, CUDA-enabled GPUs [51], embedded systems running on FPGAs [52]. Depending on the application, these implementations are optimized with several objectives in mind, ranging from speed of inference to energy consumption and memory required [53, 54].

In this research, we use the C language as an abstraction of all these possible system

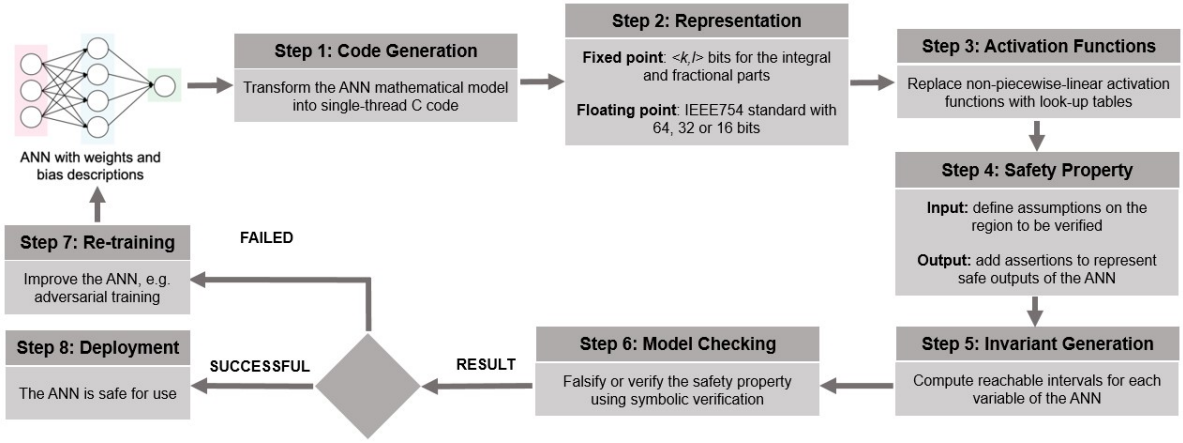


Figure 3.1: The proposed verification workflow for fixed- and floating-point ANNs.

realizations, with the addition of implementation models to represent fixed-point arithmetic (see Section 3.2). Furthermore, we limit our scope to single-threaded code, and leave the verification of concurrent implementations of ANNs (e.g. CUDA) for future work.

At the same time, given the mathematical model of a specific ANN (see Section 2.1), there exist multiple possible sequential implementations of it. This is because neural networks are highly parallel, since the output of all neurons in a single layer can be computed independently. Furthermore, the activation potential u_k of each neuron (see Equation 2.2) is the result of a sequence of multiply-and-accumulate (MAC) operations, whose order can be changed arbitrarily.

In our experiments (cf. Section 4.3), we show that our verification framework is insensitive to changes in the order of the basic operations performed by the ANN. As a result, all equivalent implementations of the same ANN will yield the same verification performance in terms of time, memory usage, and outcome. Thus, for the remainder of this section, we assume that specific implementation is given and detail the sequence of processing steps required for its verification.

3.2 Models for fixed-point ANN implementations

In this section, we discuss how our implementation models work to support fixed-point verification of neural network implementations.

There exist two ways of supporting fixed-point neural network implementations [30]. First, convert inputs into fixed-point and perform all the underlying steps, e.g., training and validation, also in fixed-point. Second, convert trained models and neural network operations, e.g., realization, from floating-point representation into fixed-point, which is then followed by a check of the desired properties. The former is likely to produce better representations, but the latter is likely to be more practical [55] because datasets are provided in floating-point representation. Besides, such a conversion should keep the dynamic range noticed in weights and neuron outputs, given that those values obtained from training are matched to the entire resulting ANN model. Here, we have chosen the latter. Moreover, such a method, also known as network compression or quantization, is the usual way of deploying neural networks on restricted devices, reinforcing its use.

Our goal is to transform an existing model (and its constraints) defined in the C

programming language into a fixed-point representation. Here, a fixed-point format is specified as $\langle k, l \rangle$, where k denotes the number of bits to encode its sign and integral part, resulting in a representation I, and l indicates the number of bits to encode its fractional part, resulting in F. Furthermore, given a rational number, we can represent it in fixed-point by using $k + l$ bits, which is interpreted as $I + \frac{F}{2^l}$. Such representation allows us to take a hardware platform's limitations, where a specific model will be executed, into account, in such a way that a more suitable implementation is provided. Moreover, in the present context, two's complement is used for value representation and arithmetic operations, due to some advantages, such as the wrap-around effect [56]. For instance, if we want to encode number +3.25 into format $\langle 5, 3 \rangle$, it will give rise to the following representation in memory: $\{00011|010\}$, with the most significant bit (i.e., 0) indicating the sign "+", $I = 3$, and $F = 2$.

To model the quantization effect on ANN's computation steps, we convert each arithmetic operation (addition, subtraction, multiplication, or division) from floating-point to their respective fixed-point counterparts. In particular, these operations and conversions must consider the parameters k and l, along with the sign bit. We achieve this goal with the implementation models proposed by Chaves *et al.* [56], which have been extensively validated in the digital controller domain. Indeed, they replace the mentioned arithmetic operations (i.e., "+", "−", "∗", and "/") and then return results according to a specific precision. Furthermore, these implementation models formally define a set of methods and values that precisely represent fixed-point operations' behavior.

In Fig. 3.2 we show an example of how to convert a piece of floating-point source-code into a fixed-point representation with the proposed implementation models. Here, we have a code snippet that computes the activation potential of a single neuron, one of the basic operations in ANNs. One may notice how the types and operations have been changed in the fixed-point version. In particular, `fxp_float_to_fxp` transforms a type `float` into a type `fxp_t` (fixed point), and both `fxp_add` and `fxp_mult` make sure that the addition and multiplication arithmetic operations are performed in fixed-point and take into account the previously defined desired precision.

In summary, the fixed-point version of an ANN's code references the appropriate implementation models, thus ensuring that the behavior of each fixed-point arithmetic operation is carried out correctly. Our experiments, in Section 4.4, show the impact of different levels of quantization granularity in ANNs.

Finally, another aspect is worth mentioning: for an entirely correct implementation, when a fixed-point format is chosen, one should still represent the dynamic range associated with the target data. If that is not done, a mismatch between trained model and processing arises, which leads to overflow and introduces errors that can jeopardize an ANN's decision.

In other words, if a given variable holds values that range from -15.5 to 15.5, for instance, a format $\langle 3, 2 \rangle$ should not be used because that would lead to frequent overflow events. Specifically, values above 3.75 and below -4.00 would not be represented. Consequently, in this specific case, a format $\langle 5, 2 \rangle$ (note the dynamic range provided by the integer part), for instance, would be suitable, then keeping correct representation in all associated operations.

3.3 Discretization of non-linear activation functions

As mentioned in Section 2.1, the choice of an activation function can have a considerable impact on verification times. While piece-wise linear functions can be readily represented as a (sequence of) if-then-else instructions, non-linear activation functions require careful adjustments to avoid severe performance degradation. This section presents an well known approach [57] to convert such non-linear functions into look-up tables, thus significantly speeding up verification processes.

Assume that the non-linear activation function $\mathcal{N} : \mathcal{U} \mapsto \mathbb{R}$ is a piece-wise Lipschitz continuous function [58], thus there is a finite set of a locally Lipschitz continuous functions $\mathcal{N}_i : \mathcal{U}_i \mapsto \mathbb{R}$ for $i \in \mathbb{N}_{\leq a}$, the so-called selection functions, such that the sets $\mathcal{U}_i \subset \mathbb{R}$ are disjoint intervals, $\mathcal{N}(u) \in \{\mathcal{N}_1(u), \ldots, \mathcal{N}_a(u)\}$ holds for all $u \in \mathbb{U}$, $\mathcal{U} = \bigcup_{i \in \mathbb{N}_{\leq a}} \mathcal{U}_i$, and

$$\|\mathcal{N}_i(u_1) - \mathcal{N}_i(u_2)\| \leq \lambda_i \|u_1 - u_2\|, \quad \forall u_1, u_2 \in \mathcal{U}_i, \tag{3.1}$$

where λ_i denotes the Lipschitz constant of $\mathcal{N}_i$.

The proposed discretisation is applied to each subset $\mathcal{U}_i$. The idea consists in discretising the $\mathcal{U}_i$ to obtain the discrete and countable set $\tilde{\mathcal{U}}_i \subset \mathcal{U}_i$. We build a lookup table for rounding the evaluation of $\mathcal{N}_i(u)$ to $\tilde{\mathcal{N}}_i(u) : \mathcal{U}_i \mapsto \mathcal{R}$, and consequently rounding $\mathcal{N}(u)$ to $\tilde{\mathcal{N}}(u) \in \left\{\tilde{\mathcal{N}}_1(u), \ldots, \tilde{\mathcal{N}}_a(u)\right\}$. This lookup table contains uniformly distributed N_i samples within $\mathcal{U}_i$, including interval limits, to ensure the accuracy $\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \epsilon$. Let L_i be defined as the length of the interval $\mathcal{U}_i$, i.e.,

$$L_i \triangleq \sup_{u \in \mathcal{U}_i} u - \inf_{u \in \mathcal{U}_i} u. \tag{3.2}$$

The following theorem can choose the number of samples N_i to ensure the desired accuracy ϵ.

Theorem 3.3.1. *Let the non-linear function $\mathcal{N} : \mathcal{U} \mapsto \mathbb{R}$, $\mathcal{N} \in \{\mathcal{N}_1(u), \ldots, \mathcal{N}_a(u)\}$, be piecewise Lipschitz continuous such that each selection function $\mathcal{N}_i(u) : \mathcal{U}_i \mapsto \mathcal{R}$ presents the Lipschitz constant λ_i, and consider the approximation $\tilde{\mathcal{N}}(u) \in \left\{\tilde{\mathcal{N}}_1(u), \ldots, \tilde{\mathcal{N}}_a(u)\right\}$, where each selection function $\tilde{\mathcal{N}}_i : \mathcal{U}_i \mapsto \mathbb{R}$, for $i \in \mathbb{N}_{\leq a}$ is obtained with $\mathcal{U}_i \subset \mathcal{U}$ containing N_i samples. The approximation error is bounded as*

$$\|\tilde{\mathcal{N}}(u) - \mathcal{N}(u)\| \leq \epsilon, \tag{3.3}$$

for a given ϵ, if

$$N_i \geq 1 + \frac{L_i \lambda_i}{\epsilon}, \forall i \in \mathbb{N}_{\leq a} \tag{3.4}$$

holds.

Proof. Given that the length of each interval $\mathcal{U}_i$ is L_i (cf. (3.2)), the length of each subinterval, obtained by uniformly dividing $\mathcal{U}_i$ at the N_i samples, is $\frac{L_i}{N_i - 1}$. Considering the Lipschitz continuity in (3.1), the rounding error for $\tilde{\mathcal{N}}_i(u)$ is bounded as

$$\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \frac{L_i}{N_i - 1} \lambda_i. \tag{3.5}$$

If (3.3) holds for all $i \in \mathbb{N}_{\leq a}$, the inequality

$$\frac{L_i}{N_i - 1}\lambda_i \leq \epsilon \tag{3.6}$$

and, consequently, (3.4) holds. Moreover, from (3.5) and (3.6), $\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \epsilon$ for all $i \in \mathbb{N}_{\leq a}$. □

Based on Theorem 3.3.1, the number of samples used in the discretization of nonlinear activation functions, e.g., $\mathcal{N}_{\text{TanH}}$ and $\mathcal{N}_{\text{Sigm}}$, described respectively in (2.4) and (2.5), can be computed to ensure desired accuracy. Without loss of generality, the approximation $\tilde{\mathcal{N}}_i(u)$ can be defined as

$$\tilde{\mathcal{N}}_i(u) = \mathcal{N}_i\left(\mathcal{A}_i(u)\right), \tag{3.7}$$

where $\mathcal{A}_i : \mathcal{U}_i \mapsto \tilde{\mathcal{U}}_i$ is an arbitrary approximation operator, e.g., rounding and quantization.

For instance, consider that we want to obtain the function $\tilde{\mathcal{N}}_{\text{Sigm}}$, which approximates $\mathcal{N}_{\text{Sigm}}$ based on a discrete domain $\tilde{\mathcal{U}}$, with target accuracy $\epsilon = 0.01$. It is clear that $\mathcal{N}_{\text{Sigm}}$ is globally Lipschitz continuous with constant $\lambda_{\text{Sigm}} = 0.25$ since the $\sup_{u \in \mathbb{U}} |\mathcal{N}_{\text{Sigm}}(u)| = 0.25$, and $\mathcal{U} = \mathbb{R}$. Moreover, let us choose the following three intervals to define the approximation $\tilde{\mathcal{N}}_{\text{Sigm}}(u)$:

$$\mathcal{U}_1 = (-\infty, -20], \tag{3.8}$$

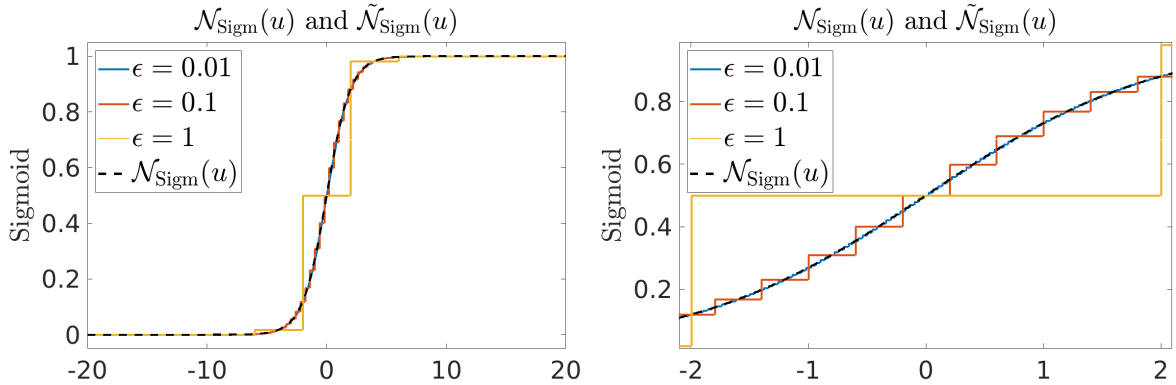
$$\mathcal{U}_2 = (-20, 20), \tag{3.9}$$

and

$$\mathcal{U}_3 = [20, \infty), \tag{3.10}$$

since the derivative of $\mathcal{N}_{\text{Sigm}}(u)$ is negligible for $u \in \mathcal{U}_1 \cup \mathcal{U}_3$, i.e., $\lambda_1 \approx 0$ and $\lambda_3 \approx 0$, while the constant λ_2 is equivalent to the global Lipschitz constant, i.e., $\lambda_2 = \lambda_{\text{Sigm}} = 0.25$. Now, we can use (3.4) to compute the number of samples in each interval necessary to ensure the desired accuracy $\epsilon = 0.01$. Accordingly, the numbers of samples are $N_1 = N_3 = 1$ and $N_2 = 1001$ since $L_2 = 40$ (cf. (3.2)). Note that the approximators $\mathcal{A}_i$ can be arbitrarily chosen. Here, it is suggested to choose $\mathcal{A}_1 = -20$ and $\mathcal{A}_3 = 20$ because it is not necessary to have more samples than the limits of the intervals for $\mathcal{U}_1$ and $\mathcal{U}_3$. Finally, $\mathcal{A}_2$ can be chosen as the half-towards-zero rounding with 3 decimal digits for floating-point and real ANNs, and as the underlying quantization function for fixed-point ANNs.

Fig. 3.3 illustrates the effect of the discretization when evaluating the sigmoid function. Note that the approximation fits well for $\epsilon = 0.01$, and it becomes poor when ϵ increases. It is worth mentioning that a look-up table is fundamentally a trade-off between speed and memory. If the latter is not a restriction, verification processes may benefit from such a strategy. Another interesting point is that such a discretization strategy should be under the final desired fixed-point format so that a safety-property verification is not compromised. This way, ϵ should be arbitrarily small and also much lower than the quantization step incurred by a fixed-point format.



(a) (b)

Figure 3.3: Comparison between the real sigmoid $\mathcal{N}_{\text{Sigm}}$ and its discretizations $\tilde{\mathcal{N}}_{\text{Sigm}}$ for $\epsilon = 0.01$ ($N_2 = 1001$), $\epsilon = 0.1$ ($N_2 = 101$), $\epsilon = 1$ ($N_2 = 11$): (a) sigmoid activation function together with its approximations within the range $[-20, 20]$ and (b) a zoom in to show the interval $[-2, 2]$.

3.4 Introducing safety properties in ANN code

As we explain in Section 2.3, verifying an ANN means proving that a given safety property holds. Such a safety property is a falsifiable mathematical relation defined on the values of an ANN's variables. Since we are considering software implementation of ANNs here, in this section, we then show how to annotate ANN code and also how to specify a desired safety property.

As a preliminary step, we annotate code by replacing the concrete input to the neural network with a general non-deterministic input. We do so by assigning a non-deterministic value to each input variable as in the following example (another example is shown in Figure 3.4):

```
float x_1 = nondet_float()
float x_2 = nondet_float()
```
(3.11)

where we use the notation `nondet_float()` prescribed by our underlying verification tool ESBMC [59, 60]. With this, our verification tool knows to expect any possible input, and it is then the role of the safety property (see Equation 3.14 below) to restrict the input space to the sub-domain of interest.

Let us consider safety properties in the general form $\mathbf{x} \in \mathcal{H} \implies \mathbf{y} \in \mathcal{G}$, where knowing an input vector $\mathbf{x}$, belonging to $\mathcal{H}$, guarantees that the output vector $\mathbf{y} = f(\mathbf{x})$ belongs to $\mathcal{G}$. Consequently, we encode the premise of this implication with a pre-condition instruction `assume`, specifying the set of values $\mathcal{H}$ that each $x_i \in \mathbf{x}$ can take. For example, a rectangular domain for the input variables $x_1 \in [0, 2]$ and $x_2 \in [-\frac{1}{2}, +\frac{1}{2}]$ can be encoded as

```
assume(x_1 >= 0 && x_1 =< 2)
```
(3.12)

and

```
assume(x_2 >= - 0.5 && x_2 =< 0.5).
```
(3.13)

It is also important to mention that comparison between floating point values must consider the smallest quantization step e, e.g., when checking if a floating point variable d

equals to 1 the condition should be written as

```
if(abs(d - 1) < e)
```
(3.14)

This notation instructs the subsequent SMT model checking to search only the inputs that satisfy the conditions specified in the `assume` instruction, as it ignores an execution when being false (e.g., see `__ESBMC_assume` [61]), thus making sure that the premise of the safety property $\mathbf{x} \in \mathcal{H}$ is satisfied. Note also that the instruction `assume` is general and supports any boolean condition as its argument.[1] This way, any form of input region $\mathcal{H}$ can be specified, as long as it is valid C code syntax. At the same time, hyper-rectangular input domains tend to lead to faster verification times, as mentioned in Section 2.3.

In contrast, we encode the conclusion of the implication with the post-condition instruction `assert`, specifying the set of values $\mathcal{G}$ that each variable $y_i \in \mathbf{y}$ can safely range in. For instance, if we have a binary classification network with two outputs y_1 and y_2 indicating the score of each class, we can encode the conclusion of a robustness safety property for the second class as

```
assert(y_2 > y_1).
```
(3.15)

Consequently, it requires that when that premise is satisfied, our binary network always predicts the second class. As for the input region $\mathcal{H}$, the `assert` instruction can be used to specify a variety of output regions $\mathcal{G}$, but now making an assessment of what is expected.

3.5 Invariant inference via interval analysis

Once the safety property has been specified, as we explain in Section 3.4, we can inject further `assume` instructions in the code and reduce the model checker's search space. Indeed, given the sequential nature of ANN computation, the set $\mathcal{H}$ of values allowed by the premise of a safety property also constrains the range of the following intermediate computation steps. Thus, if we can explicitly derive and unfold these additional constraints onto intermediate variables, in such a way that we propagate constraints and benefit from them on subsequent operations, we can more succinctly tell a model checker where to look for counterexamples.

In general, deriving additional (over-approximated) constraints on intermediate computation steps falls under the umbrella of *invariant inference* [62]. It is based on the discovery of an assertion that holds during the execution a given piece of code, which can then be used in verification procedures. For neural network code, which can easily be implemented without loops and dynamic memory allocation, we find that an interval invariant analysis suffices [63]. Such a method of invariant analysis computes lower and upper bounds on the values of each program variable (e.g., $a \leq x \leq b$, where a, b are constants and x is a variable), by propagating the initial set $\mathcal{H}$ through an ANN with interval arithmetic rules. One may notice that more complex constraint propagation methods (e.g., zonotopes and polyhedra) exist in the literature [12], but whether reduction in search space justifies the additional computational cost is an open problem. Moreover, given that neural network quantization, as tackled here, already increases computational cost, then simple and efficient constraints are desired.

[1] When working with quantized representations, whether fixed or floating point, extra care should be taken in checking that the specified constants are rounded in a way that does not break the desired condition.

On the more practical side, there are many tools to perform interval analysis of C code. In our experiments, in Section 4, we have used the evolved value analysis (EVA) plugin of the open-source tool FRAMA-C [64]. We then inject intervals into ANN code as additional pre-condition instructions `assume` on intermediate variables, thus covering the entire processing chain. Finally, we have compared this method with the native interval analysis support provided by the state-of-the-art verification tool ESBMC [65], in Section 4.3, and found that combining them (both enabled) yields the best results.

3.6 Model checking ANN implementations

Given the annotated C code from Sections 3.4 to 3.5, we are now tasked with answering the following verification question: do *all* inputs that satisfy pre-conditions `assume` also satisfy associated `assert` post-conditions, in a specific ANN implementation? In other terms, are we able to find at least `one` specific input that violates a safety property, given an ANN implemented with a specific precision? In this section, we explain how to answer this question with state-of-the-art *symbolic model checking* techniques.

In general, model checking is concerned with verifying whether a given property ϕ holds for a finite state transition system M, which is typically represented by a triple (S, I, T) [66]. More formally, these mathematical objects are defined as follows:

- S is the set of states a system can be in, where each state consists of the value of the program counter (PC), local and global variables;
- $I : S \rightarrow \{0, 1\}$ is an indicator function for a set of initial states;
- $T : s_i \rightarrow s_j$, with $s_i, s_j \in S$, is a transition function describing a system's evolution, i.e., pairs of states specifying how a system can move from state to state;
- $\phi : S \rightarrow \{0, 1\}$ is an indicator function for safe states.

In our case, the annotated ANN code defines these objects implicitly. S represents all possible value assignments to a set of program variables, including the PC. I indicates all assignments that satisfy existing `assume` pre-conditions. T holds the semantic of each instruction in code, defining how to go from one state to another, which allows checking for reachability (cf. Definition 1 below). Finally, ϕ represents a safety property encoded with existing `assert` post-conditions.

Definition 1. *Let M be a transition system. A state $s_r \in S$ is called a reachable state in M if there exists a finite sequence of N state transitions starting from an initial state s_0 and ending in state s_r, i.e., $s_0 \xrightarrow{T_0} s_1 \xrightarrow{T_1} \ldots \xrightarrow{T_N} s_{N+1} = s_r$, where $s_n \xrightarrow{T_n} s_{n+1}$ denotes a state transition when applying T_n.*

In practice, several state-of-the-art model checkers accept C code as input [67, 68, 59, 60]. Frequently, input code is readily converted into static single assignment (SSA) form before further processing [69], which has the advantage of making underlying finite-state transition systems more explicit. We show an example, in Fig. 3.4, parts (a) and (b), of such a conversion procedure.

Note that, in all experiments in Section 4, we use ESBMC for this model checking step [59, 60]. Like any other state-of-the-art model checker, ESBMC has been heavily optimized to reduce verification times. However, not all of these optimization techniques apply to feed-forward neural network code, which does not contain loops and recursions. In the following Sections 3.7 and 3.8, we clarify which techniques do apply to ANN code.

3.7 Incremental verification using lemma learning via SMT

The SMTLIB logic format introduced an assertion stack concept and the ability to push and pop assertions of it [70]. In particular, some SMTLIB compliant SMT solvers have an internal stack of assertions, which we can add new assertions to or remove old ones from. The main idea here is to enable assertion retraction and lemma learning incrementally. The former allows one to add assertions to a formula, evaluate the individual result, and then return the same formula to its original form. The latter happens when the SMT solver stores facts (in the form of lemmas over a formula's variables). In summary, it has already determined a formula, which may prove helpful in future checks.

Here, we enable the underlying SMT solver to use lemmas determined during previous checks for future ones, thereby optimizing search procedures and potentially eliminating a large amount of formula state-space to be searched. Note that previous studies report encouraging results using incremental (bounded) model checking for software, increasing the search depth without leading to the overhead of restarting a verification process from scratch [71]. This way, we apply incremental SMT solving to verify neural net implementations, where a formula is built up in stages, and lemmas are learned, along the way, about that same formula.

In particular, this incremental verification is beneficial to exploring neural net implementations by ESBMC since they contain various *ite* operators (e.g., to represent ReLU activation functions). The existing operation of the SMT solver follows directly from ESBMC. Indeed, once we build the directed acyclic graph (DAG) and produce an SSA program by symbolic execution, from a neural net's implementation, that program is converted to a fragment of first-order logic and translated into a form acceptable for the SMT solver. Then, after checking the satisfiability of a given formula, the latter is discarded. Here, many *ite* operations will be converted, solved, and discarded during a neural net's verification procedure. Since each variable in an *ite* operation is assigned only once along each path in SSA form, this requires a case split to evaluate the activation function, e.g., $z = g \ ? \ x : y$. As a result, we call the SMT solver during a symbolic execution to check the satisfiability of the guard g and then determine the value of variable z. Using *ite* retraction to build and deconstruct a formula has the potential to reduce SMT-conversion overhead, and lemma learning could lead to swifter verification times. The SMT solvers supported by ESBMC (i.e., Z3 [47], Yices [72]) claim lemma learning as a feature, thereby allowing us to evaluate its impact for verifying neural-net implementations.

To use incremental SMT, during neural net verification, we must identify ways to reuse an SMT formula by pushing and popping *ite* operations into the solver. In particular, we retain the formula produced for an *ite* operator, identify the common prefix between it and the next *ite* operator produced, and retract all the *ite* operations that can be evaluated. Then, we place the *ite* operators that could not be evaluated on top of the remaining formula. Fig. 3.4 illustrates this approach. In particular, in Fig. 3.4(a), we have two inputs x and y in lines 4 and 5, respectively; three assignments in lines 6, 8, and 10; three *ite* operators, which represent ReLU activation functions, in lines 7, 9, and 11; and one assertion representing a safety property, in line 12. Fig. 3.4(b) illustrates the program of Fig. 3.4(a) converted into SSA form (i.e., each variable is assigned exactly once), which is the format we use for incremental learning.

During the symbolic execution of this neural net implementation, based on Fig. 3.4(a), we check the satisfiability of guard "$a < 0$", in line 7, and conclude that it could either

be evaluated as "true" or "false" since "a" can assume values between -3 (lowest) and 2 (highest). As a result, we cannot simplify this expression before checking the safety property in line 12 of Fig. 3.4(a). However, we can learn from this assignment and place its *ite* operation on top of the remaining formula, which can then be used to check the mentioned safety property. After that, we check the satisfiability of guard "$b < 0$", in line 9, and conclude that it always evaluates to "false" since "b" can assume only positive numbers between 0 (lowest) and 5 (highest). So, we are thus able to remove this expression and the respective assertion. Similarly, we check the satisfiability of guard "$f < 0$", in line 11, and also conclude that it always evaluates to "false" since "f" can assume only positive values between 0 (lowest) and 4 (highest).

We show the simplified neural net implementation using our incremental verification via lemma learning in Fig. 3.4(c). One may notice that we have safely removed two ReLU activation functions represented by the variables $b2$ and $f2$, initially present in Fig. 3.4(b), which thus reduce the formula's size to be checked by the underlying SMT solver. Note further that we have learned that variable "a" can assume values between -3 (lowest) and 2 (highest), which can be used to check the assert statement specified in line 9 of Fig. 3.4(b). Consequently, that same assert can not be identified in Fig. 3.4(c) anymore because the knowledge of its range allowed such a simplification. The assertions $b1 \leq 5$ and $f1 \leq 4$ were also removed since we previously learned the intervals for the variable b and f. Lastly, we can observe the ability to perform a query at any neuron using incremental verification, which can help prune neural net implementation before deploying it to an embedded device with time, memory, and energy constraints.

3.8 Constant folding, slicing and expression balancing for search-space reduction

Our employed verification engine implements general code optimizations, when converting a neural net implementation to SMT. These include *constant folding*, *slicing* and expression balancing [73], which we briefly introduce here.

Constant folding evaluates constants, including nondeterministic symbols, and propagates them throughout the resulting formula, during encoding. In particular, we exploit the constant propagation technique to reduce the number of expressions associated with specific neuron computation procedures and activation function. Thus, we simplify the SSA representation, using local and recursive transformations, to remove functionally redundant expressions (for neuron computation procedures and activation functions) and redundant literals (for safety properties), as

$$\begin{array}{ll} a \wedge \mathit{true} = a & a \wedge \mathit{false} = \mathit{false} \\ a \vee \mathit{false} = a & a \vee \mathit{true} = \mathit{true} \\ a \oplus \mathit{false} = a & a \oplus \mathit{true} = \neg a \\ \mathit{ite}\,(\mathit{true}, a, b) = a & \mathit{ite}\,(\mathit{false}, a, b) = b \\ \mathit{ite}\,(f, a, a) = a & \mathit{ite}\,(f, f \wedge a, b) = \mathit{ite}\,(f, a, b)\,. \end{array}$$

We apply such simplifications to reduce the size of the resulting formula and consequently achieve simplification within each time step and across time steps, during the encoding procedure of a neural net's implementation. In our experimental evaluation, in Section 4.3, we have noticed substantial improvements using these simplifications in formulas, but we have not identified improvements using the constant propagation approach itself. It

happens because neural net inputs are typically symbolic ones and not constants, as can be noticed in the illustrative example in Fig. 3.4, where incremental learning removed the activation functions for neuron b and output f.

Slicing removes expressions that do not contribute to the checking procedure of a given safety property. It is an essential step to improve a program's verification procedure, considerably, in some cases [74]. Our verification engine implements two slicing strategies in combination. First, it removes all instructions after the last assert in the set of SSA. Second, it collects all symbols (and their dependent symbols) in assertions and removes instructions that do not contribute to them. When used in combination, both slicing strategies ensure that unnecessary instructions are ignored during SMT encoding. As an example, the code in Fig. 3.4(a) can be considered. If we are interested in checking that neural net's output only, we could rewrite the final assert statement, in line 12, as $f <= 4$. Consequently, such a modification do indicate that everything not involving f does not cause an impact on the conclusion of the intended safety property. Based on such a scenario, the resulting SSA for the code in Fig. 3.4(a) would be sliced as

$$x1 == nondet_symbol(nondet0) \wedge y1 == nondet_symbol(nondet1) \wedge$$
$$f1 == 3 * (int)x1 + (int)y1 \wedge f2 == (f1 < 0\,?\,0 : f1) \wedge f2 <= 4,$$

where there is no presence of information (states) regarding neurons a and b. In our experimental evaluation, in Section 4.3, we have observed that *slicing* can significantly reduce the resulting SMT solving time.

Expression balancing reduces the size of SMT formulae by reordering long chains of operations with the associative rule. This technique has been recently applied to neural networks by Giacobbe *et al.* [30], but has been used in compilers for decades. In brief, the computation of neuron potentials in ANNs requires a linear combination of the neuron inputs (see Equation 2.2). Depending on the specific implementation, the resulting sequence of multiply-and-accumulate operations (MAC) in code is translated to SMT formulae of different sizes. In the worst case, which is portrayed in Fig. 3.5a, the formula size is linear in the number of MAC operations (associating to the right). Expression balancing then ensures that SMT formulae are reordered according to a balanced fashion, as shown in Fig. 3.5b. Consequently, the sequence of MAC operations is split over multiple accumulators in a divide-and-conquer fashion, yielding a set of semantically equivalent, but smaller SMT formulae. As a result, it can avoid, for instance, solver time-out events. In Section 4.3, we show that this associative balancing step is crucial in making ANN verification viable. One may also notice that this result is consistent with that presented by Giacobbe *et al.* [30]. Furthermore, in Section 4.3, we show that thanks to this balancing step, the performance of our verification methodology is stable across different implementations of the same ANN.

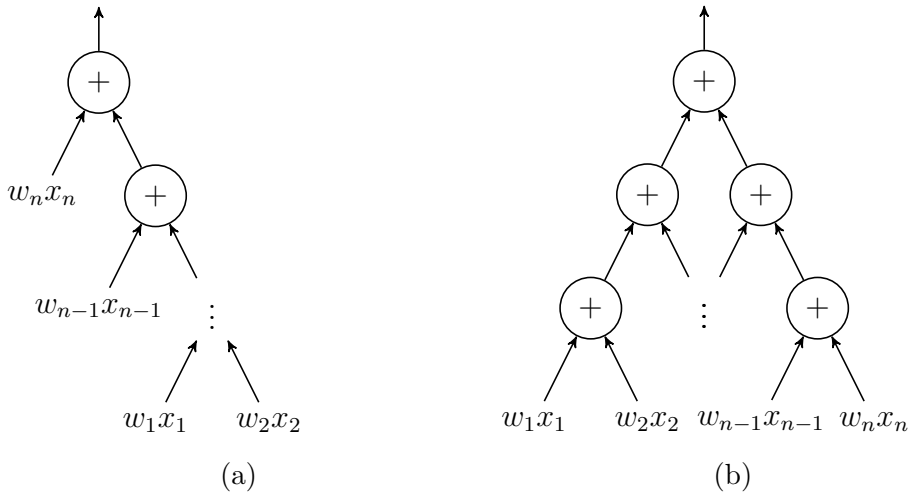


Figure 3.5: Effect of expression balancing on abstract syntax trees: (a) linear and (b) balanced association layouts.

3.9 Illustrative example: robustness to adversarial images

We conclude this section with an illustrative example of our verification methodology. We do so in order to clarify the user's side of the workflow illustrated in Fig. 3.1. Later, in Section 4, we report more details on the range of ANNs and safety properties that can be verified with our methodology, as well as the efficiency of doing so.

The present example, illustrated in Fig. 3.6, shows how to verify a character recognition ANN. First, given a network's architecture and weights, in a high-level representation, as in Fig. 3.6a, such elements should be converted into single-threaded C code. This task can be achieved through the popular machine learning libraries PyTorch [75] and Tensorflow [76], or, like in many of our experiments, in Section 4, by converting from the mid-level representation NNet[2]. Here, the converting process is implemented in C language and it consists in translating the NNet file into a single-threaded C code. In this example, we use the neural network from our Vocalic benchmark (see Section 4.2) quantized to a fixed-point representation with 8 integer (including sign) and 8 fractional bits.

(a) ANN structure and weights. (b) $\mathcal{H}$ center. (c) $\mathcal{H}$ lower. (d) $\mathcal{H}$ upper. (e) Counterex.

Figure 3.6: Inputs and outputs of our verification approach: (a) ANN implementation; safety property, where the (b) center, (c) lower, and (d) upper extremes of the input region are shown; and (e) a counterexample that violates the safety property.

[2] github.com/sisl/NNet

Second, the ANN's source code undergoes a further sequence of transformations. Initially, we replace all floating-point arithmetic operations with the corresponding fixed-point implementation models (see Section 3.2), given that our ANN is quantized. Then, we also replace any sigmoid, hyperbolic tangent, or piecewise-linear activation function with its corresponding discretized look-up table (see Section 3.3).

Third, a safety property is encoded by adding the corresponding pair of `assume` and `assert` instructions. In the present example, we check for robustness around a specific input image, which we show in Fig. 3.6b. More formally, we define the input region of our safety property (premise) as a set $\mathcal{H} = \{x : |x - x^d|_\infty \leq c\}$, where the centre point x^d corresponds to the 5×5 pixel values in the image of the ideal character "A", i.e., without deviation, in Fig. 3.6b, and $c = 80$. Since ANNs are designed to solve specific problems, the region and its parameters must vary. For reference, we report the lower and upper bounds of $\mathcal{H}$ that precisely defines the minimum and maximum values of the defined hyper-rectangle. It is illustrated in the gray image pixel domain, in Figs. 3.6c and 3.6d, respectively. In this case, it is possible to see that $c = 80$ is a suitable value for the vocalic recognition problem, since it produced an accurate adversarial example. It is important to mention that there is no formal algorithm that produces suitable values of c, however it can be found by running experiments and analysing its counterexamples. However, c can also be set by analyzing a specific system and determining the maximum input error range, for instance. Further we provide a set of benchmarks that evidences how the value of c interferes on the counterexample proximity.

Likewise, we set the output region of the safety property (conclusion) as the set of all outputs that assign a higher score $y_A > y_k, \forall k \neq A$ to class "A" than to any other output classes. Note that the final softmax layer, typically included in classification ANNs, can be omitted for our purposes since it is a monotonic function of the score of each class [1]. After this, a static analysis tool such as FRAMA-C [64] propagates the input region $\mathcal{H}$ through the associated ANN code and annotates it with additional `assume` instructions, representing the reachable values-interval of each intermediate variable (see Section 3.5).

Fourth, annotated C code goes through a model checker that tries to falsify a safety property. In our experiments (see Section 4), we have used ESBMC to do so, as it is a good representative of state-of-the-art SMT model checkers [59, 60]. If a given safety property can not be verified, ESBMC returns a counterexample that falsifies it, which represents a potential adversarial attack on a neural network. In the present example, ESBMC does indeed report such a counterexample, which we show in Fig. 3.6e. More adversarial examples can be seen in Figs. 4.10a, 4.10b, and 4.10c, for a wide range of safety properties and quantization granularities of our character recognition ANN.

```c
float potential(float *w,
                unsigned int w_len,
                float *x,
                unsigned int x_len,
                float b) {

  if (w_len != x_len) {
    return 0;
  }

  float result = 0;

  for (unsigned int i = 0; i < w_len; ++i) {
    result += w[i] * x[i];
  }

  result += b;

  return result;
}
```

(a)

```c
fxp_t potential(float *w,
                unsigned int w_len,
                float *x,
                unsigned int x_len,
                float b) {
  if (w_len != x_len) {
    return 0;
  }
  fxp_t result = 0;
  for (unsigned int i = 0; i < w_len; ++i) {
    fxp_t w_fxp = fxp_float_to_fxp(w[i]);
    fxp_t x_fxp = fxp_float_to_fxp(x[i]);
    result = fxp_add(result, fxp_mult(w_fxp, x_fxp
       ));
  }
  fxp_t b_fxp = fxp_float_to_fxp(b);
  result = fxp_add(result, b_fxp);
  return result;
}
```

(b)

Figure 3.2: A method to compute the activation potential of neurons in C: (a) floating- and (b) fixed-point.

```c
int main() {
  _Bool x, y;
  int a, b, f;
  x = nondet_bool();
  y = nondet_bool();
  a = ((2*x) - (3*y));
  a = a < 0 ? 0 : a;
  b = (x + (4*y));
  b = b < 0 ? 0 : b;
  f = ((3*x) + y);
  f = f < 0 ? 0 : f;
  assert(a <= 2 && b <= 5
     && f <= 4);
  return 0;
}
```

(a)

```
x1 == nondet_symbol(nondet0
   )
y1 == nondet_symbol(nondet1
   )
a1 == 2 * (int)x1 - 3 * (
   int)y1
a2 == (a1 < 0 ? 0 : a1)
b1 == (int)x1 + 4 * (int)y1
b2 == (b1 < 0 ? 0 : b1)
f1 == 3 * (int)x1 + (int)y1
f2 == (f1 < 0 ? 0 : f1)
(assert) a2 <= 2
(assert) b2 <= 5
(assert) f2 <= 4
```

(b)

```
x1 == nondet_symbol(nondet0)
y1 == nondet_symbol(nondet1)
a1 == 2 * (int)x1 - 3 * (int)y1
a2 == (a1 < 0 ? 0 : a1)
```

(c)

Figure 3.4: (a) A simple neural net implemented in C, where variables “a”, “b”, and “c” range from -3 to 2, 0 to 5, and 0 to 4, respectively. (b) The initial neural-net C program converted into SSA form. (c) A simplified version of the SSA form using incremental learning.

Chapter 4

Experimental Evaluation

In this chapter, we test the performance of the verification approach we introduced in Chapter 3. In this regard, we are mainly interested in the following research questions:

RQ1 - Ablation study - Is it possible to establish the role of each of the enhancement techniques introduced in Chapter 3 and also define an optimal setup, both regarding total verification time and performance?

RQ2 - Quantization effects - How does a quantization choice influence our verification process and the safety of a neural network?

RQ3 - Comparison with SOTA techniques - What is the performance of our verification approach when compared to the existing literature?

Regarding **RQ1**, since those techniques were first introduced for software verification in general, we are interested, in particular, in finding their optimal configuration to verify ANNs, including contribution and general setup. In addition, **RQ2** is related to quantization of ANNs, which is in the core of the present work and have the potential to provide a methodology regarding integration into target platforms. Moreover, if we were to verify the same property for different quantization levels, would we observe any difference in verification time or outcome? Finally, regarding **RQ3**, it is always of paramount importance to position a given approach among the existing scientific knowledge.

We present our answers to those questions in the following way. In Section 4.1, we discuss a configuration step regarding quantization and also general data processing to provide adaptation and avoid overflow in ANN operations. In Section 4.2, we describe the datasets and ANNs that constitute our verification benchmarks, including the necessary minimum number of bits for correct data-range representation. In Section 4.3, we isolate the contribution of each component of our verification approach and propose the configuration that yields the best results performance-wise, which answers **RQ1**. In Section 4.4, we compare the performance and output of our verification approach across different quantization levels of the same problem, which addresses **RQ2**, while analyzing important aspects and general behavior. In Section 4.5, we compare our verification framework with the most popular SOTA approaches, which fulfills **RQ3**. Finally, in Section 4.6, we list the remaining limitations towards large-scale verification of fixed-point ANNs. All benchmarks, tools, and results associated with the current evaluation are available for download at `https://tinyurl.com/6y7e49vk`.

4.1 Quantization aspects and data adaptation

As mentioned at the end of Section 3.2, when correctness comes into play, not every quantization format can be used. Indeed, suppose a format that is unsuitable to a target ANN is chosen, aiming to keep the dynamic range of its data. In that case, overflow will likely occur, compromising operation results and general ANN output. Nonetheless, a designer can also incur severe quantization and suppose that errors due to wrong operations are an acceptable side effect (even under frequent overflow). Still, our goal is to provide compression that results in quantization error only, then preserving an ANN's associated dynamic range and correct computation of operations in neurons.

Another aspect is that input data may present a broad diversity of dynamic ranges. As a consequence, they are usually processed in scaled format. In our framework, input data is first normalized to the range $[0, 1]$ and then fed to a given ANN (also for training). This way, the initial (input) dynamic range is always known as well as all neurons output values.

Consequently, it is essential to analyze neurons in a given ANN and then identify the minimum and maximum associated values resulting from their processing, given input data in the range $[0, 1]$, which will define the minimum number of bits for the integer part of a given representation. It does not specify maximum compression because it only intends to represent the existing dynamic range and avoid overflow correctly. In this way, all the worst case values could be represented by the specified number of bits keeping any possible neuron output inside the dynamic range and no overflow would occur. Hence, saturation would be avoided. Besides, we should also check the number of bits for the fractional part to provide the desired accuracy.

Note that the discovery of the minimum number of bits for the integer part is made by using Eq. (2.8), with $p = 1$, and taking into account all weights of each neuron to find the maximum magnitude. Alternatively, FRAMA-C [64] can also be used, as it reveals intervals associated with variables in ANN code.

4.2 Description of the benchmarks

In our evaluation, we consider ANNs trained on two datasets: the UCI Iris dataset [77] and a vocalic character recognition dataset [50]. This section gives the details regarding the employed datasets, the neural networks we trained on top of them, the safety properties that we used to test our verification approach, and, finally, our general experimental setup.

Iris benchmark

The Iris dataset [77] consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica, and Iris versicolor). This dataset contains both the length and width of the sepals and petals in centimeters (our inputs) and the iris specie label (our output). Here, we use TensorFlow version 1.4 [76] and keras [78] to train a feedforward neural network with layers of $4 \times 7 \times 3$ neurons, hyperbolic tangent activation functions, and softmax output layer. This architecture presented a good training performance when compared to $4 \times 5 \times 3$ and $4 \times 9 \times 3$ architectures under test accuracy. We train such a neural network to predict the correct Iris species with the backpropagation algorithm and cross-validation [1]. When quantizing the ANN to fixed-point arithmetic, we followed what was presented in Section 4.1. We found that the maximum neuron output was bounded, in

modulus, by 23.3. Consequently, we allow for 6 integer bits, including sign, as they are required to avoid overflow. In terms of safety properties, we specify hyper-rectangular input regions, since it is quite used in ANN safety properties [39]. Hyper-rectangular inputs are defined for each species: *setosa*, *versicolor*, and *virginica*. We identify the center of these regions from the dataset with the granular fuzzy clustering algorithm in [79]. Then, for each of the four input variables, we computed its maximum range. With it, we generated nine regions R_s for each class, sharing the same center but with different sizes $s \in \{1, 2, 5, 8, 10, 20, 30, 40, 50\}$ of the hyperrectangle surrounding it, where s is a percentage representing the fraction of the maximum input range.

Vocalic benchmark

The vocalic dataset [50] consists of 200 gray-scale images with dimensions 5×5 pixels. Half of the dataset consists of the base images illustrated in Fig. 4.1 and also noisy versions of them. In contrast, the other half presents non-vocalic images. With it, we have trained a feedforward neural network with architecture $25 \times 10 \times 4 \times 5$ and sigmoid activation functions. As Fig. 4.1 shows, there are five output classes that this network learned to discriminate via backpropagation algorithm and cross-validation. Once again, we have followed what was presented in Section 4.1 and found 53.9 as maximum neuron output. Consequently, we have quantized this ANN to fixed-point arithmetic with a minimum of 7 integer bits, including sign, as they are required to avoid overflow. As far as the safety properties are concerned, we specify five hypercubic input regions corresponding to the vocalic labels. The centers are defined by the base images in Fig. 4.1. Similarly to the Iris benchmark, we generate five instances L_s of these regions with different sizes $s \in \{10, 20, 40, 80, 120\}$, where s represents the hypercube's side length.

Figure 4.1: Vocalic images in benchmarks.

AcasXu benchmark

The Acas Xu benchmark [80] is the result of avionics research in airborne collision avoidance systems (ACAS) for unmanned aircrafts (Xu). In particular, when avoiding a nearby aircraft, some specific piloting decisions must be taken. These are recorded in a large state-action table that is impractical to store on-board due to its memory requirements. The Acas Xu benchmark splits and compresses such a table into a set of 45 neural networks. The split is done by discretizing the following two input dimensions: time until loss of vertical separation (9 intervals), and previous advisory action (5 actions). The remaining 5 inputs are fed into a fully-connected feedforward neural network with ReLU activation functions and architecture $5 \times 300 \times 300 \times 300 \times 300 \times 300 \times 300 \times 5$, which outputs a prediction for each of the 5 possible actions. We quantize all these 45 ANNs with 27 integer bits, which is the least number of bits required to avoid overflow in the worst-case scenario, i.e., with a neuron output of 72142560.0, as pointed out by FRAMA-C [64]. More details on its associated safety properties can be found in the work developed by Katz *et al.* [18] and in Section 4.5.

Experimental setup

We have conducted our experimental evaluation on a Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with 128 GB of RAM and Linux OS. All presented execution times are CPU times, i.e., only the elapsed periods spent in allocated CPUs, which was measured with the `times` system call [81]. All experimental results reported here were obtained by executing ESBMC v6.6.0[1] with the following command line parameters, unless specifically noted: `esbmc <file.c> -I <path-to-OM> --force-malloc-success --no-div-by-zero-check --no-pointer-check --yices --no-bounds-check --interval-analysis --fixedbv`. In general, we let ESBMC run without time or memory limits. The timeouts reported in the following experiments are all due to exceedingly high memory consumption. All of our benchmarks have been annotated with the reachable intervals provided by FRAMA-C, unless specifically noted. In particular, we executed FRAMA-C using the following command: `frama-c -eva -eva-plevel 255 -eva-precision 11`.

4.3 Ablation study

This section aims at evaluating the impact of different aspects of our approach on the total verification time. Here, our aim is both to discover the best configuration for our verification tool and shed some light on the importance of each technique for reducing the search space of the verification problem. Specifically, we address four choices in our verification approach: SMT solver, optional parameters offered by the ESBMC verification engine, interval analysis technique and expression balancing strategy.

SMT solvers comparison

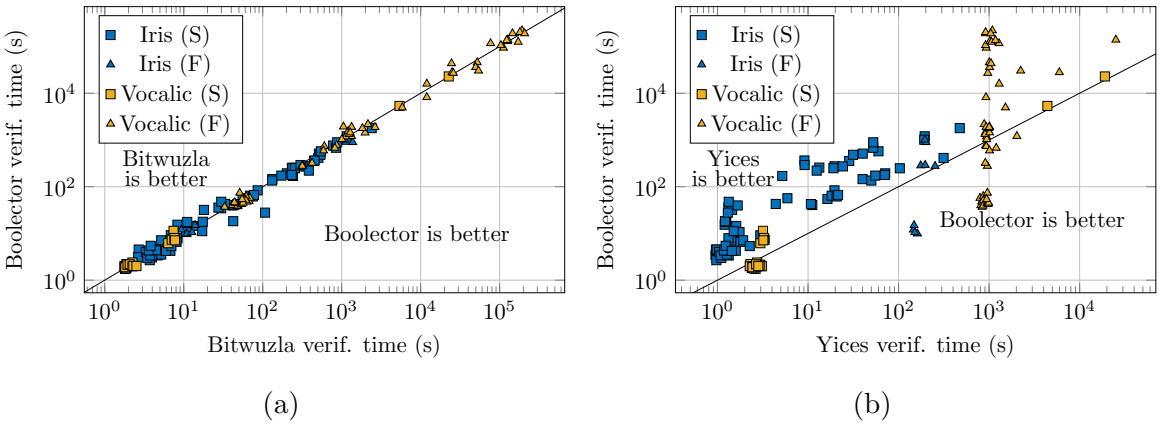
As mentioned in Chapter 3, our approach relies on model checking to reason about the satisfiability of a given safety property concerning an ANN implementation. For the experiments of the present section, we have chosen ESBMC as our verification engine since it has been extensively evaluated at various SV-Comp [82] competitions, where it has consistently achieved state-of-the-art results [60]. More in detail, the ESBMC model checker takes care of converting input C code into SMT formulae and then calls an external SMT solver. Currently, ESBMC supports four solvers: Bitwuzla, Boolector, Yices, and Z3. In general, they yield different verification results, both in terms of the generated counterexample (if any) and verification time.

Here, we are interested in comparing the performance of such solvers in verifying ANN implementations. To this end, we run them on all our fixed-point benchmarks, with word lengths of 8, 16, and 32 bits. With this choice, we cover the most popular quantization lengths, and observe the behaviour of our verification methodology on a varied test suite. We use these experimental settings all throughout our ablation study (see also Sections 4.3, 4.3 and 4.3).

The results of our comparison are summarized in Fig. 4.2. There, we can see that solvers Bitwuzla and Boolector have nearly identical performance, in terms of verification time (Fig. 4.2a). In contrast, Yices exhibits a considerable advantage across the whole verification suite, being, in some specific cases, even two orders of magnitude faster (Fig. 4.2b). Finally, solver Z3 struggled to complete the majority of verification runs, and

[1] Available at http://esbmc.org/

it is, in general, orders of magnitude slower than the other three solvers. For this reason, we do not portray its results in Fig. 4.2.



(a) (b)

Figure 4.2: Comparison with different SMT solvers, regarding verification time (in seconds), when handling the fixed-point Iris and Vocalic benchmarks. On the left, (a) Bitwuzla and Boolector show similar performance; on the right, (b) Yices is considerably faster than Boolector, in most instances. In both plots, we discriminate between successful verification outcomes (S) and falsifiable safety properties that admit a counterexample (F).

Given the results in Fig. 4.2, we choose Yices as our underlying SMT solver for the rest of this experimental section. While it is impossible to know exactly why Yices is the best-performing solver on our test suite, we speculate it is a consequence of the fact that ESBMC encodes verification problems into SMT formulae with the formalism of *QF_AUFBV* logic.[2] Here, *QF* stands for quantifier-free formulas, *A* stands for the theory of arrays, *UF* stands for uninterpreted functions, and *BV* stands for the theory of fixed-sized bit-vectors. For this type of formulae, Yices represents the state-of-the-art SMT solver. [3]

Comparison regarding ESBMC's parameters

In Sections 3.7 and 3.8, we have presented a number of state-of-the-art software verification techniques that apply to ANN implementations. From our prior experience of participating in software verification and testing competitions (e.g., SV-COMP and Test-Comp), such techniques play an essential role in optimizing the performance of ESBMC on a given set of benchmarks [83, 60, 84]. In the present section, we quantify their individual impact on verification times of our test suite and comment on their relative performance.

Here, we rely on the fact that the ESBMC's verification engine allows us to toggle each separate technique via command-line parameters. More specifically, the list of verification techniques and corresponding ESBMC parameters are as follows:

- **Constant propagation.** It can be disabled with the option `no-propagation`. Otherwise, it will generate a minimal set of SSAs in the symbolic engine.
- **Slicing.** It can be disabled with the option `no-slice`. Otherwise, it will eliminate redundant or irrelevant portions of a program [85]. In ESBMC, this is applied to

[2] https://smtlib.cs.uiowa.edu/logics.shtml
[3] https://smt-comp.github.io/2020/results/qf-aufbv-single-query

the SSA program before it is encoded to SMT to reduce the number of variable assignments by identifying variables not used to evaluate any property assertion.

- **Incremental verification.** Activated with the (experimental) options `smt-during-symex` and `smt-symex-guard`. The former enables incremental SMT solving using the SMT solvers Yices or Z3, the latter allows calls to the solver during symbolic execution to check the satisfiability of the guards.

- **Expression simplification.** It can be disabled with the option `no-simplify`, effectively neutering constant propagation so that no fact is statically determined to be true or false, and always end up exploring to the top of the unwind bound.

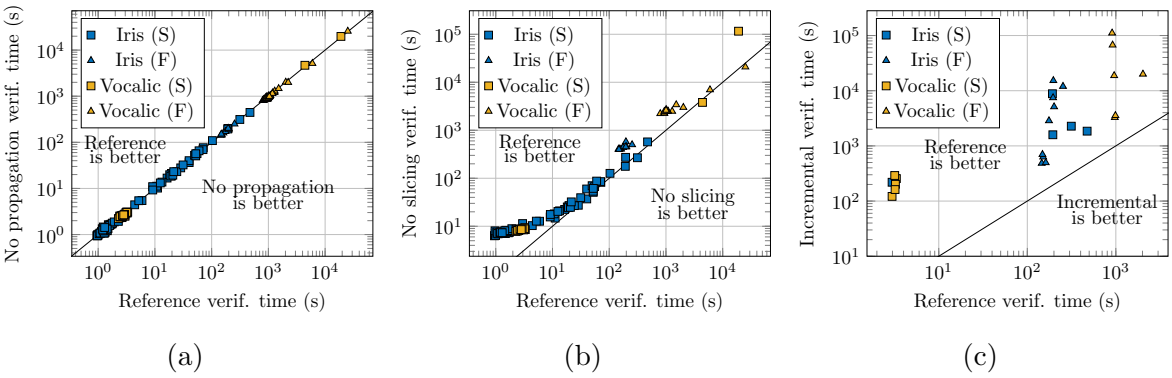


Figure 4.3: Comparison of verification times of ESBMC with different parameters settings on the fixed-point Iris and Vocalic benchmarks. In each figure, one individual technique has been changed from the best (reference) configuration: (a) disabling constant propagation, (b) disabling slicing, and (c) enabling incremental verification. In the plots, we discriminate between successful verification outcomes (S) and falsifiable safety properties that admit a counterexample (F).

Here, we quantify the impact of each technique on the same test suite of Section 4.3. We do so by setting a reference configuration and toggling one verification technique at a time. For reasons that become clear from the results shown in Fig. 4.3, our reference configuration of ESBMC has constant propagation, slicing and expression simplification enabled. In contrast, we choose to keep incremental verification disabled.

As the results in Fig. 4.3a show, constant propagation makes no difference on our test suite. This is because we are verifying a specific kind of safety property, namely robustness to adversarial examples, which allows all input variables to be modified. As such, there is no constant input that can be propagated through the ANN code, thus yielding no reduction in the SMT formulae size. At the same time, we believe that constant propagation is a useful technique for safety properties that restrict the attack surface to just a subset of the input variables, as the ones identified by Karmon, Zoran, and Goldberg [86].

In contrast, Fig. 4.3b shows that slicing yields a small improvement in performance, which becomes the more significant the shorter the verification time is. We speculate that this is because neural networks are usually redundant (e.g., see dropout [1]), and thus the majority of neurons contribute to the ANN output. As a consequence, only a small number of expressions can be removed with slicing.

Interestingly, incremental verification (cf. Fig. 4.3c) does not improve verification time as expected. We believe this happens because the cost of deriving and storing new facts during the verification process outweighs the reduction in search space they induce since it performs various calls to the solver. Still, we hypothesize that incremental verification may offer some advantages when verifying not only one but also a whole set of safety properties since it allows incrementally remembering important facts across properties, whose net contribution may pay off. For example, we could perform a query at any neuron using incremental lemma learning, which could help prune neural net implementation before deploying it to an embedded device with time, memory, and energy constraints. However, we leave the exploration of such a hypothesis for future work.

Finally, expression simplification is crucial in making the verification of our test suite practical. Indeed, without expression simplification, none of the safety properties could be checked before hitting our machine memory limit of 128GB, despite letting the verification process run without any time limit.

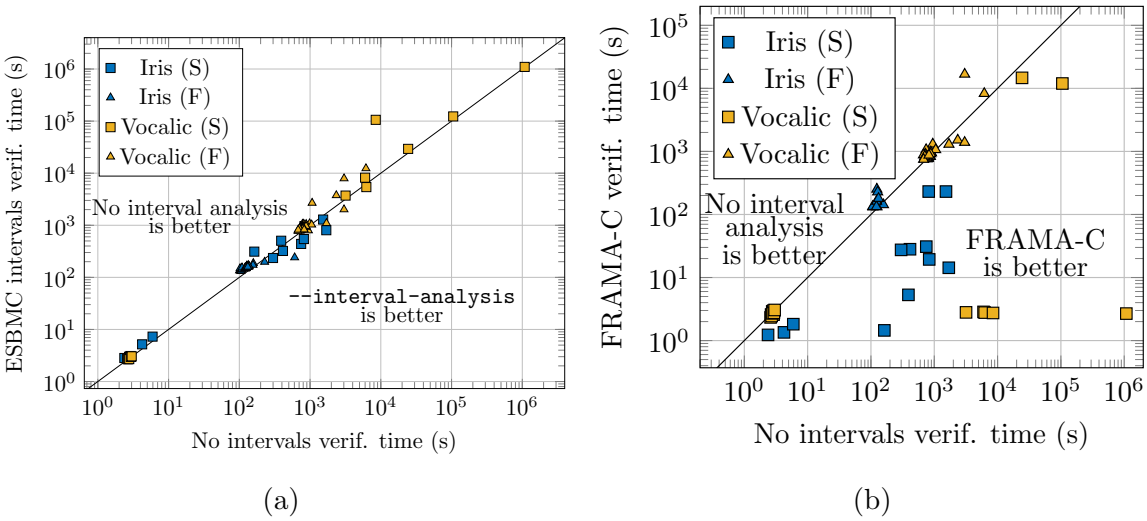
Interval analysis comparison

In Section 3.5, we introduced interval analysis as an essential pre-processing stage before running the verification engine on ANN code. Here, we show the effect of disabling such an important step on total verification times. Furthermore, we compare two approaches to interval analysis and discuss their results. The first requires FRAMA-C [64] to annotate ANN code with additional `assume` instructions. In contrast, the second requires running ESBMC with the extra `--interval-analysis` option enabled. Note that both of them compute hyper-rectangular constraints over program variables.

For consistency with the previous experiments, we evaluate the impact of these two interval analysis options on the same test suite as in Section 4.3. We present the results in Fig. 4.4, where the native `--interval-analysis` option and the externally computed intervals by FRAMA-C are compared with our reference configuration of ESBMC without any form of interval analysis. Note how the former has almost no impact on the verification time, while the latter can improve it by up to two orders of magnitude. Still, regarding the use of FRAMA-C, it is interesting to notice that we only observe improvement on successful safety properties (S), i.e., those that do not admit a counterexample. This way, the verification time of falsifiable properties (F) does not appear to be improved by interval analysis on our test suite.

On the one hand, as no counterexample is found, the FRAMA-C's more sophisticated interval analysis indeed pays off, given the apparent reduction in the state space that must be explored. On the other hand, when a property is falsifiable, that seems to be easily identified in the proposed framework and adopted test suite. As future work, we can perform a deep analysis of that matter and then even propose improvements in this interval analysis focused on ANN code and properties. Note that the intervals produced by ESBMC work only for integer variables [87], while Frama-C can make intervals for integer and floating-point ones [88]. Since our benchmarks contain heavily floating-point computations, we expected Frama-C to improve our verification results considerably compared to the interval analysis implemented in ESBMC, particularly for safe neural nets due to the state-space size.

Such performance improvement is in line with our previous experiments over a large set of open-source software benchmarks when enabling invariant generation [60]. In particular, in the mentioned study, invariant generation based on intervals allowed us to verify 7% more programs using a k-induction proof rule. Therefore, we chose to use both the



(a) (b)

Figure 4.4: Comparison of verification times with and without interval analysis on the fixed-point Iris and Vocalic benchmarks. On the left, (a) enabling the native `--interval-analysis` option in ESBMC does not yield much improvement; on the right, (b) adding the intervals computed by FRAMA-C reduces verification times of a large number of safety properties. In both plots, we discriminate between successful verification outcomes (S) and falsifiable safety properties that admit a counterexample (F).

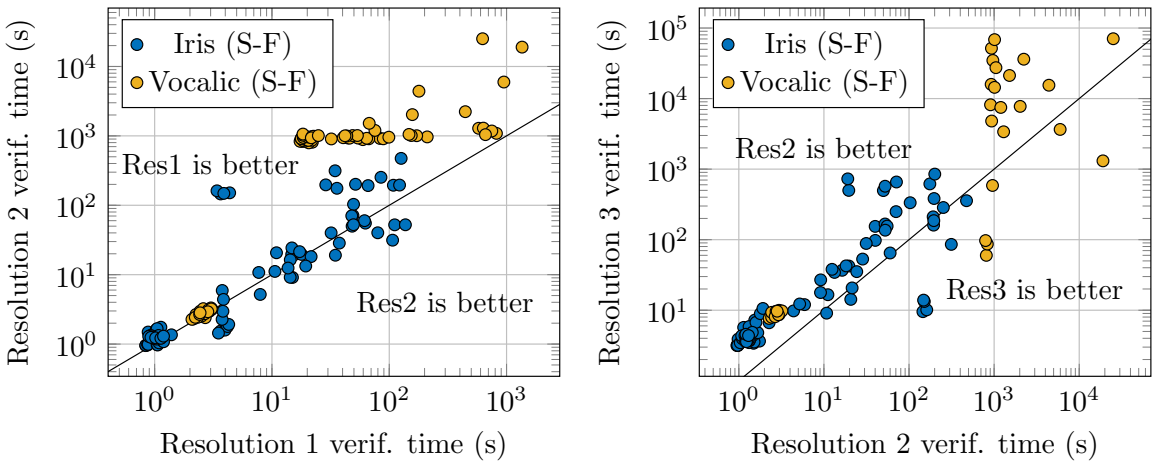
`--interval-analysis` option in ESBMC and the FRAMA-C's intervals for the upcoming experiments.

Activation function discretization comparison

The Iris and Vocalic benchmarks we use in the present ablation study are based on neural networks with sigmoid and hyperbolic tangent activation functions (see detailed descriptions in Sections 4.2 and 4.2). An important step in our verification methodology is the discretization of such functions, as explained in Section 3.3. In practical terms, it means replacing the non-linear mathematical expression of the activation function with a look-up table. Here, we show the impact of the resolution of such look-up table on verification times, and how the error we introduce with the discretization influences the verification outcome.

To this end, we compare three different resolutions of our look-up tables, which we call *Res1*, *Res2*, and *Res3*. These discretize the input interval $[-6, +6]$ with one, two, or three decimal fractional places, respectively. Outputs for inputs that fall outside that range are automatically saturated to 0 or 1 for the sigmoid function and -1 or $+1$ for the hyperbolic tangent one. We report the corresponding results on the Iris and Vocalic benchmarks with 8, 16, and 32 bits, all condensed in Fig. 4.5. Although coarser resolutions usually result in faster verification times, as expected, given the inherent speed-up in operations, one may also notice some outliers: all regarding the Iris benchmark, when comparing *Res1* with *Res2*, and a mixture of Iris and Vocalic benchmarks, when comparing *Res2* with *Res3*. This is because different look-up table resolutions affect the computation of each neuron's output, and, in some cases, even the ANN's output itself (see example in Section 2.2). Consequently, a given violation that happened early during state-space exploration may then occur later or may not be even identified anymore, thus introducing a lot of variability in the verification time.

A more outcome-oriented comparison is presented in Table 4.1. As one can notice, the verification outcome is indeed affected by the resolution choice. In fact, comparing *Res1* and *Res2* on the Vocalic benchmark yields one instance where the two verification runs disagree: *Res1* reports a falsifiable property with a counterexample (F), whereas suh counterexample disappears with the finer resolution *Res2* and the property is declared safe (S). Unfortunately, if we increase the resolution further to *Res3*, the additional computational requirements overwhelm our verification setup, and we begin to observe a number of time-outs. This is more noticeable for the Vocalic benchmarks, because they employ a larger ANN. *Res3* would be suitable in a more accurate system, since *Res2* already provides reliable counterexamples.



(a) (b)

Figure 4.5: Comparison of verification times with different discretization resolutions for activation functions on the fixed-point Iris and Vocalic benchmarks. On the left, (a) comparison between one and two decimal places; on the right, (b) comparison between two and three decimal places. In both plots, we only report benchmarks that did not incur in timeout.

In conclusion, choosing the right discretization resolution is a trade-off between verification time and possible errors in verification outcomes. In the ablation study in Section 4.3 and the later quantization experiments in Section 4.4, we choose the intermediate resolution *Res2*, based on two main reasons. First, it is the finest resolution that does not incur in large amounts of timeout when verifying our benchmarks. Second, all the counterexamples generated with it are valid, as we confirmed by running them through a non-discretized MATLAB implementation of the corresponding neural networks.

Code generation comparison

In Section 3.1 we mentioned that a single ANN can be implemented in multiple ways. In fact, due to the intrinsic parallelism of neural architectures, the order of many mathematical operations can be shuffled arbitrarily. Here, we show that our verification methodology produces the same result (time and outcome) for very different orderings of these mathematical operations, and thus its performance is stable across them.

Specifically, we focus on the order of operations required to compute the activation potential of each neuron, one of the basic building blocks of ANNs (see (2.2)). In this

Iris Dataset		Res2		
		S	F	TO
Res1	S	72	0	0
	F	0	9	0
	TO	0	0	0

Vocalic Dataset		Res2		
		S	F	TO
Res1	S	21	0	0
	F	1	53	0
	TO	0	0	0

(a)

Iris Dataset		Res3		
		S	F	TO
Res2	S	70	0	2
	F	0	0	9
	TO	0	0	0

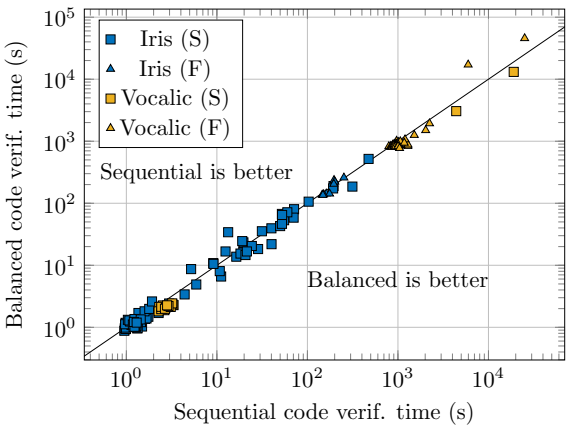
Vocalic Dataset		Res3		
		S	F	TO
Res2	S	20	0	2
	F	0	0	53
	TO	0	0	0

(b)

Table 4.1: Comparison of verification outcomes with different discretization resolutions of activation functions on the fixed-point Iris and Vocalic benchmarks. On the left, (a) comparison between one and two decimal places; on the right, (b) comparison between two and three decimal places. Both tables are structured as confusion matrices: entries on the main diagonal represent benchmarks with the same outcome under both resolutions. There, we discriminate between successful verification outcomes (S), falsifiable properties that admit a counterexample (F), and properties that incurred in timeout (TO).

regard, we compare two opposite implementations of it that we exemplify in Fig. 4.6. On the one hand, we have run a fully sequential version of that ANN code, where each multiply-and-accumulate (MAC) operation in (2.2) is executed in the same order as the input vector $\mathbf{x}$. We implement this version of the code with simple loops as in the example of Fig. 4.6a. On the other hand, we have also run a *balanced* version of the ANN code, where the MAC operations are reordered in a divide-and-conquer sequence to minimize the number of additions, as in the example of Fig. 4.6b. Such associative rebalancing procedures are common optimizations performed by compilers, as they reduce the total number of machine instructions and improve execution time on out-of-order processors [89, 90, 91].

As we have done in the previous Section 4.3, we compare the verification performance of these two code generation approaches on the fixed-point Iris and Vocalic benchmarks, considering the word lengths 8, 16 and 32 (bits). One code generation relies on generating a C code file where the mathematical operations follows the linear layout and the second one generates a C code file where the mathematical operations follows the balanced layout described in Section 3.8. The results in Fig. 4.7 show very little difference in verification time and identical verification outcomes (except for one single timeout with balanced code). The reason for such a behavior lies in the understanding that ESBMC performs several aggressive expression-simplification steps including associative techniques, as explained in Section 3.8. As such, the final set of SMT formulae that are fed into the solver are quite insensitive to the order of operations in ANN code. Thus, we can conclude that the performance of our verification methodology is consistent across different implementations of the same ANN.



(a)

Iris Dataset		Balanced		
		S	F	TO
Sequential	S	72	0	0
	F	0	9	0
	TO	0	0	0

Vocalic Dataset		Balanced		
		S	F	TO
Sequential	S	22	0	0
	F	0	52	1
	TO	0	0	0

(b)

Figure 4.7: Comparison of verification performance with different ANN code generation techniques on the fixed-point Iris and Vocalic datasets. On the left, (a) verification time; on the right, (b) verification outcome. In all plots and tables, we discriminate between successful verification outcomes (S) and falsifiable safety properties that admit a counterexample (F).

The results presented here successfully answer **RQ1 - Ablation study**: we have identified an optimal configuration for the ESBMC verification engine within our framework, which consists of using the SMT solver Yices and the `interval-analysis` option in conjunction with FRAMA-C intervals. Moreover, we quantified the individual importance and associated influence of a number of related techniques: constant propagation, expression simplification, slicing, incremental verification, discretization of non-linear activation functions, and code generation.

4.4 Verification of quantized ANNs

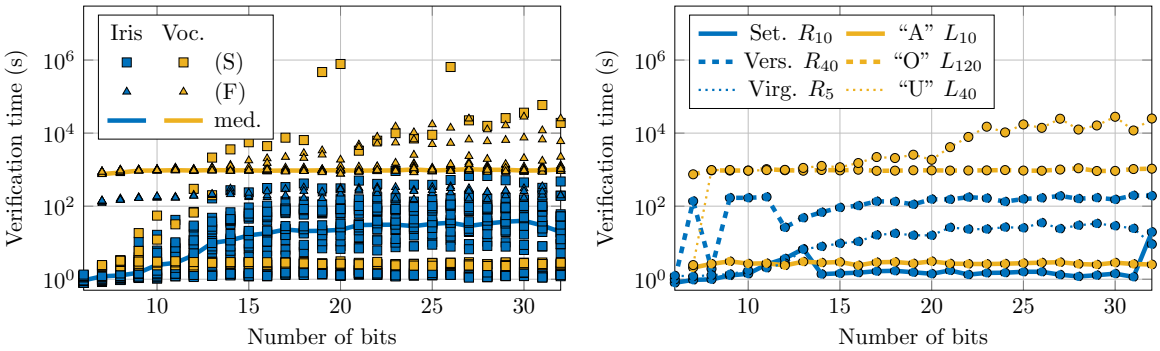
In Section 4.3, we established what the best configuration of our verification method is by comparing its runtime under different scenarios. Similarly, in the present section, we compare its verification time and output along another dimension: the quantization level of ANNs. Our main result is that the granularity of ANN quantization may influence verification performance, but that may even be considered minor, depending on the specific aspect being evaluated. Here, we show that this is true both for verification time and verification outcome. Consequently, ANN quantization can be regarded as a viable and effective tool for adaptation towards a given target platform, as long as some evaluation is performed.

Effects of quantization on verification time

First, let us comment on how the quantization of an ANN affects its verification time of its safety properties. First, recall that verifying quantized neural networks is PSPACE-hard, as proven by [19]. However, this is a theoretical worst case, and existing empirical results in [30] show a positive correlation between the number of bits used in a quantized representation and the total verification time. Here, we show that this correlation holds

only for small number of bits and specific safety properties, and there is no general trend for word lengths equal or longer than 16 bits.

To this end, we run our Iris and Vocalic benchmarks with a broad range of quantization levels, covering the span between the common word lengths of 8, 16 and 32 bits, and extending to smaller word lengths with zero fractional bits. We present such results in Fig. 4.8a. Note that there is a general upwards trend in verification time for short word lengths (from $6-7$ to 15 bits), but this phenomenon almost disappears for longer word lengths (16 bits and above). Moreover, results are spread across six orders of magnitude, thus it is difficult to prove the existence of a true correlation in the associated data. In fact, applying common summary statistics (e.g., median verification time like in [30]) shows a correlation between time and quantization for the Iris and Vocalic benchmarks.



(a) (b)

Figure 4.8: Comparison of verification times with different quantization levels on the fixed-point Iris and Vocalic benchmarks. On the left, (a) a scatter plot of all the safety properties in our benchmarks with their respective median times; on the right, (b) a selection of six safety properties shows a very limited correlation between number of bits and verification time.

A better understanding can be extracted by selecting individual safety properties and comparing their verification time across different quantization levels. We do so in Fig. 4.8b, where we choose six properties from Fig. 4.8a that showcase the full range of behaviors. More specifically, we broadly observe three different behaviors. First, properties like Vocalic "A" L_{10} and Vocalic "O" L_{120} exhibit almost identical verification time across all quantization levels. Second, properties like Iris Setosa R_{10} and Iris Versicolor R_{40} are somewhat erratic across quantization levels. However, their verification time falls into a limited range, where no systematic trend emerges. Third, properties like Vocalic "U" L_{40} and Iris Virginica R_5 have verification time that is mildly correlated with the quantization level.

Overall, we believe that the quantization level has only a minor impact on the hardness of the verification problem from a practical perspective. Other factors, like the number of active neurons or the size of input regions of a given safety property, are probably better predictors regarding verification time. However, since these are beyond the scope of the present work, we leave a thorough exploration of them to future work, where we might establish predictors and bounds.

Effects of quantization on verification outcome

Another aspect regards verification outcomes, where narrower bit widths deserve some discussion. Here, we take the results of the same experiments shown in Section 4.4 and plot, in Fig. 4.9, a summary of how many safety properties are declared safe (S), generate a counterexample (F), or result in timeout (TO). As the figure shows, the percentage of successful safety properties is stable across quantization levels. The only noticeable differences happen in the Iris and Vocalic benchmarks for small word lengths. In the former, we observe a sudden drop in the number of safe properties between 6 and 7 bits, which goes through behavior that resembles transient responses in control systems [92], until a more suitable representation is achieved (12 bits). In addition, with 6 bits, all safety properties are declared safe, which is indeed due to differences caused by computation with quantized values (see Section 2.2). Moreover, one may notice a clear trend related to more comprehensive formats, indicating an increasing number of correct operations.

We observe a higher incidence of undecidable safety properties regarding the Vocalic benchmarks that lead to a timeout. Note, however, that the Vocalic ANN is larger than Iris. Thus, more timeout events are expected due to the additional computational complexity, which is also worsened by the chosen representation. Again, stability regarding verification outcome is only achieved when a more suitable representation is used (14 bits).

In this context, some conclusions can be drawn. Indeed, there is a clear relationship between data representation and safety-property verification when using restricted formats. In addition, it becomes negligible when more bits are used. Moreover, arbitrarily small representations should not be carelessly used, as erratic behavior may be experienced.

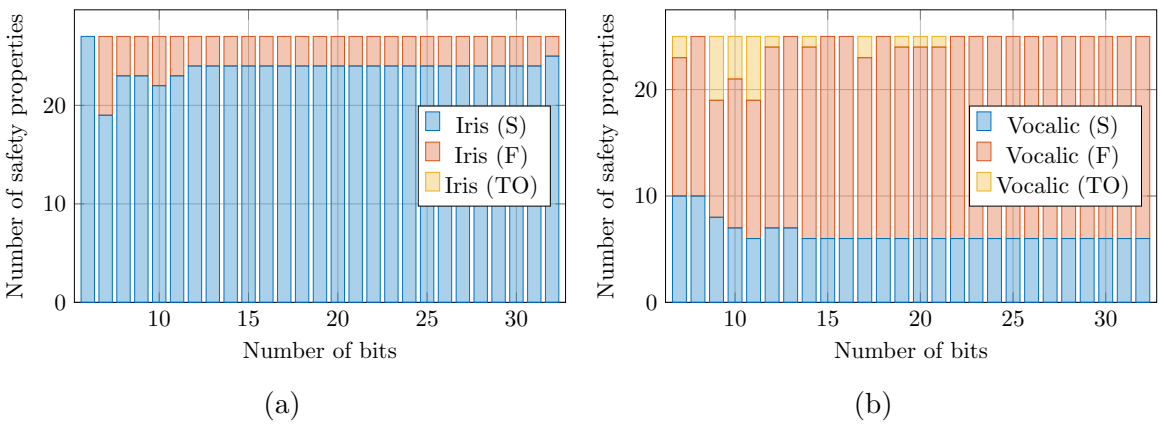


Figure 4.9: Comparison of verification outcomes with different quantization levels on the fixed-point Iris and Vocalic benchmarks. On the left, (a) Iris dataset; on the right, (b) Vocalic dataset. In both histograms, we discriminate between successful verification outcomes (S), falsifiable properties that admit a counterexample (F), and properties that resulted in timeout (TO).

A more focused picture of the relationship between quantization and verification outcome can be extracted by looking at individual safety properties. To this end, we report, in Tables 4.2 and 4.3, all safety properties that have different outcomes across quantization levels. There, we can see two completely opposite behaviors. On the one hand, properties like Vocalic "A" L_{20}, Vocalic "I" L_{10}, Iris Versicolor R_{50} and Iris Virginica L_{50} are only safe for very short word lengths. On the other hand, properties like Vocalic "U"

L_{20} and Iris Versicolor R_{40} tend to be safe as the word length increases.

Iris		Number of bits																										
Property		6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Set.	R_{40}	S	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{50}	S	S	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	S
Vers.	R_{20}	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{30}	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{40}	S	F	S	F	F	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{50}	S	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
Virg.	R_{20}	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{30}	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{40}	S	F	S	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	R_{50}	S	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Table 4.2: Iris safety properties with different verification outcomes across quantization levels.

Vocalic		Number of bits																									
Property		7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
A	L_{20}	S	S	S	S	S	S	S	F	F	F	TO	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F
	L_{40}	S	F	S	F	F	F	F	F	F	F	TO	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{80}	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{120}	F	F	F	F	F	F	F	TO	F	F	F	F	F	F	TO	F	F	F	F	F	F	F	F	F	F	F
E	L_{40}	TO	F	TO	TO	TO	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{80}	F	F	TO	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{120}	F	F	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
I	L_{10}	S	S	S	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{20}	F	S	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
O	L_{20}	TO	S	TO	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{40}	F	F	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{80}	F	F	TO	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
U	L_{20}	S	S	F	S	F	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
	L_{40}	S	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{80}	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	L_{120}	F	F	F	TO	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Table 4.3: Vocalic safety properties with different verification outcomes across quantization levels.

A possible explanation of this behavior is that the properties listed in Tables 4.2 and 4.3 are on the verge of breaking the ANN robustness. In fact, for each of these properties, reducing the input region size L_i or R_i makes them more safe, and increasing it makes them less safe. Thus, by staying on the threshold between these two regimes, any minor change in the ANN implementation (e.g., the quantization level) can easily flip the verification outcome and yield the erratic results we observe.

Indeed, there is no prediction methodology or technique capable of indicating that such behavior will occur; however, the present work successfully reveals it and hints at how to devise a suitable scheme. For instance, a closed-loop approach could test a chosen set of properties against some possible quantization levels so that the realization with the smallest number of bits that still provide an output that is considered stable is chosen. This way, we could navigate through a response dependent on the target quantization level and, when a steady-state region is achieved, the narrowest format that allows this behavior is chosen. Currently, we can only validate or not a given quantized ANN; we leave the synthesis of neural net implementations as future work.

Effects of quantization on adversarial examples

Finally, let us comment on the effect of quantization on the counterexamples returned by our verification approach. As Tables 4.2 and 4.3 show, the verification outcome of the same

safety property changes depending on the chosen ANN representation. This is because different quantization granularities may either hide or reveal specific vulnerabilities in ANN computation. At the same time, even if the verification outcome is the same and the safety property is kept falsifiable (F), the counterexamples returned by the verification engine may be different.

Here, we present a qualitative comparison between counterexamples, which, in the context of machine learning research, are also known as *adversarial examples*. Specifically, we focus on our three levels of fixed-point quantization, namely 8, 16 and 32 bits, for which we present a selection of adversarial examples from the Vocalic benchmarks in Figs. 4.10a, 4.10b, and 4.10c respectively. Each figure contains pairs of images, where the center of the input region is on the left, and its corresponding adversarial example is on the right.

"A", $L_{40} \rightarrow$ "O". "E", $L_{40} \rightarrow$ "A". "I", $L_{40} \rightarrow$ "A". "O", $L_{40} \rightarrow$ "A". "U", $L_{40} \rightarrow$ "A".

(a)

"A", $L_{40} \rightarrow$ "U". "E", $L_{40} \rightarrow$ "A" "I", $L_{40} \rightarrow$ "E". "O", $L_{40} \rightarrow$ "I". "U", $L_{40} \rightarrow$ "I".

(b)

"A", $L_{40} \rightarrow$ "U". "E", $L_{40} \rightarrow$ "I". "I", $L_{40} \rightarrow$ "E". "O", $L_{40} \rightarrow$ "I". "U", $L_{40} \rightarrow$ "A".

(c)

Figure 4.10: Counterexamples for the Vocalic benchmarks with (a) 8, (b) 16, and (c) 32-bits fixed-point representations. For each safety property, we report the centroid and associated counterexample, the input region dimension L_i, and the incorrect output label that was generated.

As Figures 4.10a, 4.10b, and 4.10c show, the granularity of ANN quantization has a significant effect on the quality of the adversarial examples. On the one hand, coarser fixed-point representations, such as $\langle 4, 4 \rangle$, restrict the search space to fewer gray-scale levels, which is clearly seen in Fig. 4.10a, given the easily noticeable differences. On the other hand, finer quantizations, such as $\langle 16, 16 \rangle$, let the verification engine produce counterexamples with minimal noise spread across the whole image (see Fig. 4.10c). The latter is typical of floating-point ANNs. Besides, it is especially dangerous for this specific case of image classification, as such adversarial examples may go undetected even by a human observer [40]. It is important to notice that quantization levels also influence counterexamples. Moreover, they can be regarded as adapted to the contexts created by the latter.

These results successfully answer **RQ2 - Quantization effects**: we established that the verification time has some correlation with the number of bits of the ANN quantization. Moreover, we showed that the safety of an ANN is mostly stable across different quantization levels, which supports the use of aggressive quantization in machine learning practice as long as some verification is performed.

4.5 Comparison with state-of-the-art verification tools

This section compares our verification methodology with existing works in the literature. We note that the field is progressing very rapidly at the time of writing, and thus the present comparison is limited to what tools are currently available. Namely, the few existing approaches for verifying quantized ANNs (Giacobbe *et al.* [30], Baranowski *et al.* [29], Kai Jia *et al.* [93], Guy Amir *et al.* [94]) do not provide reliable source code to replicate their experiments. As such, we can only compare our methodology with earlier tools that verify the safety of ANNs as abstract mathematical models, i.e. in infinite precision. Among those, we choose the two most popular ones as follow:

- **Marabou** [14]. Based on the earlier tool Reluplex [18], Marabou uses a simplex-like algorithm to split the verification problem in smaller subproblems and invoke an SMT solver on each of them.

- **Neurify** [15]. It uses symbolic intervals to over-approximate the ReLU non-linearity of each neuron, and turn the verification process into finding the solution of a linear problem. These over-approximation are iteratively tightened by splitting each ReLU activation function in two independent linear problems.

The goal of this comparison is showing that our quantized methodology is at least as efficient as these two state-of-the-art tools. At the same time, notice that our methodology provides more information on the safety of the actual ANN implementations than the abstract safety guarantees provided by tools like Marabou and Neurify.

To this end, we choose the AcasXu benchmark as our comparison suite (see Section 4.2). This benchmark has the advantage of being already implemented in both Marabou and Neurify,[4] [5] thus allowing us to run the authors' code for a fair comparison of their performance. Furthermore, the neural networks in the AcasXu benchmark contain ReLU activation functions exclusively, which makes them compatible with Neurify. In a similar vein, we focus our comparison on safety property 1 of the AcasXu benchmark, since it is the one that incurs the fewest time-outs with the aforementioned verification tools [14, 95]. Note that 45 different neural networks need to be verified for each safety property of AcasXu, thus giving us a larger enough sample size for a significant comparison. Regarding our verification methodology, we choose a 32 bit representation with 28 integer bits (including sign), which are needed to avoid overflows.

The summary of our results regarding verification time are shown in Figures 4.11 and 4.12. On the one hand, in Fig. 4.11, we compare our methodology with the SMT-based tool Marabou. Note how our verification methodology is considerably faster than

[4] https://github.com/NeuralNetworkVerification/Marabou
[5] https://github.com/tcwangshiqi-columbia/ReluVal

Marabou. We believe this is because our underlying model checker, ESBMC, is more efficient at producing optimized SMT formulae (see Section 3.8) than the custom simplex-like method employed by Marabou [14]. This also explains why the verification times of our methodology are almost constant across the whole comparison suite.

On the other hand, in Fig. 4.12, we compare our methodology with the symbolic interval tool Neurify. Note that this tool has been released by the authors as a multi-threaded software [15]. This is the version we compare to in Fig. 4.12a. However, for the sake of a fair comparison with our methodology, we also present a modified version of Neurify that uses only a single thread in Fig. 4.12b.[6] Note how the multi-threaded version of Neurify is faster than our methodology in a majority of cases. This is because, on our machine, the multi-threaded Neurify uses up to 22 processors in parallel, giving it an obvious advantage over our single-threaded methodology. At the same time, such advantage disappears for the single-threaded version: more specifically, for the latter our methodology is faster in verifying 24 out of the 45 neural networks.

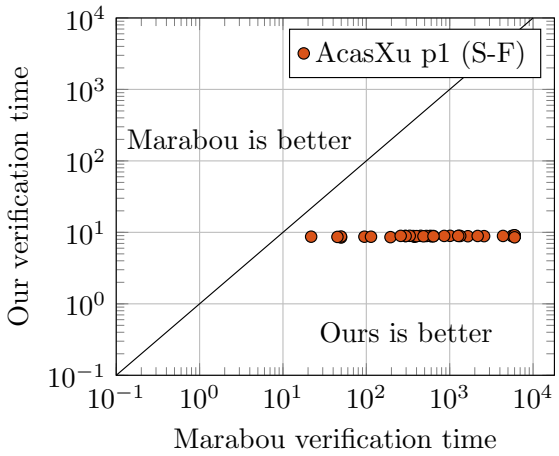
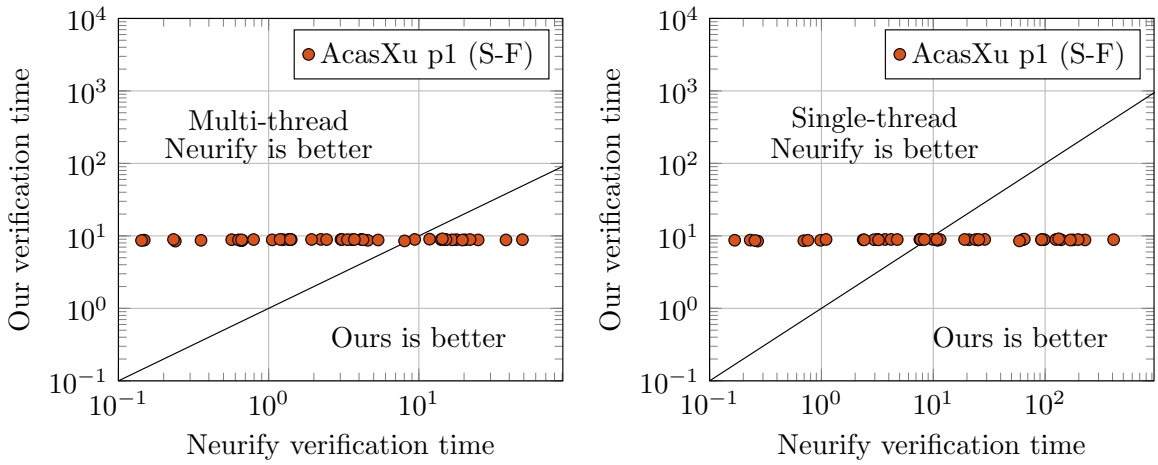


Figure 4.11: Comparison of the verification times of Marabou and our methodology on property 1 of the AcasXu benchmark.

[6]The modified code is available at https://github.com/ericksonalves/nn-verification-comparison



(a) (b)

Figure 4.12: Comparison of the verification times of Neurify and our methodology on property 1 of the AcasXu benchmark. On the left, (a) the original multi-threaded version of Neurify; on the right, (b) the modified single-threaded version.

For completeness, we also report the verification outcomes on all 45 benchmarks in Table 4.4. Note how both Neurify and our methodology are able to successfully verify all 45 neural networks, whereas Marabou incurs a time-out for 13 of them. These results confirm that our methodology offers a comparable performance to that of Neurify, and faster than the state-of-the-art tool Marabou. Note also that our methodology offers guarantees on the actual implementation of the ANNs, e.g. the ones that would be deployed on an autonomous aircraft in the AcasXu case, thus making it more attractive for practical scenarios where the safety of a deployed system is critical.

ACAS XU	Benchmark																						
Tool	1_1	1_2	1_3	1_4	1_5	1_6	1_7	1_8	1_9	2_1	2_2	2_3	2_4	2_5	2_6	2_7	2_8	2_9	3_1	3_2	3_3	3_4	3_5
Marabou	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	TO	TO	TO	S	S	S	S	S
Neurify	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
Ours <28, 4>	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

(a)

ACAS XU	Benchmark																						
Tool	3_6	3_7	3_8	3_9	4_1	4_2	4_3	4_4	4_5	4_6	4_7	4_8	4_9	5_1	5_2	5_3	5_4	5_5	5_6	5_7	5_8	5_9
Marabou	TO	TO	TO	TO	S	S	S	S	S	TO	TO	TO	TO	S	S	S	S	S	S	S	TO	TO
Neurify	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
Ours <28, 4>	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S

(b)

Table 4.4: Verification outcomes of Marabou, Neurify and our methodology on the AcasXu benchmarks for property 1. For reason of space, we split the 45 results in two subtables (a) and (b). Note that changing the parallelism of Neurify as in Figures 4.12a and 4.12b does not change its verification outcomes.

These results successfully answer **RQ3 - Comparison with SOTA**: We evaluated and compared our tool with other state-of-the-art tools, including SMT-based verification and symbolic intervals such as Marabou and Neurify, respectively. In terms of correctness, our approach can successfully verify all the benchmarks without timeout or crash. Furthermore, considering AcasXu property 1 benchmarks, our approach is significantly faster and solves more verification tasks than Marabou, a competitive opponent in SMT-based verification.

4.6 Limitations

We believe the work we present in this research is an essential milestone for verifying fixed- and floating-point ANNs with arbitrary activation functions. This way, it can be considered a unified quantization framework, with the potential of broad model exploration and verification regarding data representation. However, we still want to highlight a few limitations of our verification approach that need to be addressed in future work.

First, we handle non-linear activation functions by replacing them with lookup tables (see Section 3.3). This step is necessary for efficiency reasons but has a drawback: even with a proper resolution, a lookup table will always approximate an original function. Our experiments used lookup tables with a resolution of three decimal places and correctly validated all adversarial cases with MATLAB. However, we cannot exclude that our verification approach may produce incorrect adversarial examples or successful verification outcomes in other ANN verification scenarios, especially when a given lookup table's resolution does not match the adopted quantization granularity. This constitutes a potential threat to the validity of our method.

Second, the biggest challenge in ANN verification is scaling to large neural networks. In this regard, our Iris and Vocalic benchmarks are small to medium-sized regarding the number of neurons. Furthermore, the dataset themselves is small, which probably generated ANNs with low robustness to adversarial attacks. Both these factors contribute to keeping the dimensionality of resulting SMT formulae low and thus help our method achieve competitive verification times. However, a thorough investigation of which factors hamper verification performance and overcome them is still required.

Finally, quantized frameworks do not usually publish the code of their methods, which compromises any direct comparison attempt. Even so, the research presented here is itself SOTA and can pave the way for further research towards ANN deployment in restricted systems based on formal guarantees.

```
1  float potential_8(float *w,
2                    float *x,
3                    float b){
4    float result = 0;
5
6    for (int i=0; i<8; ++i) {
7      result += w[i] * x[i];
8    }
9
10   result += b;
11
12   return result;
13 }
```

(a)

```
1  float potential_8(float *w,
2                    float *x,
3                    float b) {
4    float tmp_01 = w[0]*x[0]+w[1]*x
       [1];
5    float tmp_23 = w[2]*x[2]+w[3]*x
       [3];
6    float tmp_45 = w[4]*x[4]+w[5]*x
       [5];
7    float tmp_67 = w[6]*x[6]+w[7]*x
       [7];
8    float tmp_0123 = tmp_01 + tmp_23
       ;
9    float tmp_4567 = tmp_45 + tmp_67
       ;
10   float result = tmp_0123 +
       tmp_4567;
11   result += b;
12   return result;
13 }
```

(b)

Figure 4.6: An example of balancing the order of MAC operations when computing activation potentials with eight inputs: (a) sequential version with a loop and (b) balanced version with a divide-and-conquer pattern.

Chapter 5

Related Work

This work's main contribution is providing a sound verification approach for checking the safety of MLPs with arbitrary activation functions and taking into account FWL effects in computations (weights, bias, and operations) due to fixed-point implementation, in addition to activation function discretization. SMT-based approaches [28, 17, 18, 14, 50] have been used for safety verification of ANNs. Besides, the main advantage of those techniques lies in SMT solvers' soundness; however, there is an important drawback: the scalability is limited since they are sensitive to the ANN complexity. For this reason, most of them are unable to deal with large ANNs.

Wang *et al.* [15] propose an efficient approach for checking different safety properties of large neural networks, aiming at finding adversarial cases. Their approach is based on two main ideas. First, symbolic linear relaxation combines symbolic interval analysis and linear relaxation to create an efficient propagation method for tighter estimations. Second, directed-constraint refinement, which identifies nodes whose output is overestimated and iteratively refines their output ranges. Those two techniques are implemented in a tool called Neurify that was validated against multiple ANN architectures. Furthermore, to scale up their verification framework, they have implemented their code using multi-threaded programming techniques. However, as the previous tools [28, 17, 18], Neurify only supports ReLU activation functions. Katz *et al.* [14] present Marabou that extends the Reluplex approach and uses lazy search to deal with nonlinearities of activation functions, allowing verification of ANNs with any piecewise-linear activation functions.

Recently, set-theoretic methods for reachability-based verification have been proposed for verifying ANN-controlled closed-loop systems. In particular, Tran *et al.* propose the NNV tool [12], which over-approximates the exact reachable set by approximating the exact reachable set after applying an activation function. It allows support to hyperbolic tangent and sigmoid activation functions. Other approaches [10, 13] also employ set-theoretic methods and polynomial approximation of hyperbolic tangent and sigmoid, using Taylor's [13] or Bernstein's [10] polynomials. Our approach also allows verifying ANNs with non-linear activation functions. This approximation is based on lookup tables created with a suitable number of intervals (i.e., expected error) to avoid use of non-linear operators' in SMT solvers. This approach allows support to any piecewise continuous activation function.

Robustness is the ability to ensure safe outputs under the presence of disturbances and uncertainties, such as input noises and implementation issues [30]. In this sense, Dey *et al.* [96] provide a parametric regularization methodology to improve the robustness of ANNs concerning additive noise. However, sensitivity to FWL effects is not considered in

that approach. ANNs are usually designed to work in real arithmetic; however, it is already shown that safety violations may occur due to the floating- [97] and fixed-point [30] implementations. In particular, Baranowski *et al.* presented a practical SMT-based approach for verifying neural networks' properties considering fixed-point arithmetic. Their approach employs a realistic model of FWL effects that includes different rounding and overflow models. However, as shown by Henziger *et al.* [19], the scalability of this kind of approach is compromised due to the hardness of the verification of fixed-point implementations of ANNs. Therefore, a new method for verifying fixed-point implementations based on abstract interpretation is proposed in [19] to reduce complexity and increase scalability. However, that method can only verify ANNs with piecewise linear activation functions since it does not consider the propagation of FWL effects through generic non-linear functions. Our approach also considers FWL effects of fixed-point implementations of ANNs based on an efficient FWL implementation model that reduces complexity when verifying those ANNs. It is also important to mention that our approach deals with real C code implementations in floating and fixed-point representations of ANNs. Our experiments and previous work on verification of fixed-point digital controllers [56] indicated that scalability is not compromised by the use of this FWL implememntation model.

Our approach implemented on top of ESBMC has some similarities with other techniques described here, e.g., regarding the covering methods proposed by Sun *et al.* [98], model checking to obtain adversarial cases proposed by Huang *et al.*[17], and incremental verification of ANNs implemented in CUDA by Sena *et al.* [50]. However, the main contribution concerns our requirements and how we handle, with invariant inference, actual implementations of ANNs with non-linear activation functions, also considering FWL effects. Moreover, the latter results in promptly deployable ANNs, which could be integrated into a unified design framework. Only ANNs' weights, bias descriptors, and desired input regarding a dataset are required to run our proposed safety verification. For tools such as DeepConcolic [98] and DLV [17], obtaining adversarial cases or safety guarantees in customized ANNs depends on the intrinsic characteristics of models. For instance, in their implementations, they do not support complex non-linear activation functions. Moreover, Sena *et al.* [50] do not exploit invariant inference to prune the state space exploration, which is done in our proposed approach.

Chapter 6

Conclusion

Verification of ANNs has recently attracted considerable attention, with notable approaches using optimization, reachability, and satisfiability methods. While the former two promise to scaling to large neural networks, they achieve such a goal by relaxing and approximating the verification problem. In contrast, satisfiability methods are exact by construction but are confronted with the full complexity of the original verification problem.

In this work, we propose a satisfiability modulo theory (SMT) approach to address ANN verification. More specifically, we view the ANN not as an abstract mathematical model but as a concrete piece of software (i.e., source code), which performs a sequence of fixed- or floating-point arithmetic operations. We can borrow several techniques from software verification and seamlessly apply them to ANN verification with this view. In this regard, we center our verification framework around software model checking (SMC) and empirically show the importance of interval analysis, constant folding, tree balancing, and slicing in reducing the total verification time. Furthermore, we propose a tailored discretization technique for non-linear activation functions that allow us to verify ANNs beyond the piecewise-linear assumptions that many state-of-the-art methods are restricted to.

Besides, in our experimental evaluation, we covered an important relationship between the granularity of ANN quantization and verification time and the correctness of its properties. The more granular the quantization, the more significant the search space and thus the more prolonged the verification time. This is contrary to the main existing theoretical result in the literature, which states that verifying quantized ANNs is computationally harder than verifying real-valued ones. However, further research is needed to shed more light on this phenomenon. Regarding correctness, we verified that narrower bit widths can be used but must be verified before deployment to achieve the minimum format that still provides broadly correct results. However, when that minimum representation is obtained, more comprehensive formats will usually provide correct results, as the stationary response of a curve relating bit width and verification result.

Besides, regarding the minimum compression that an ANN requires, it was possible to obtain the minimum compression by running all the benchmarks in all word lengths inside 32 bits. Here we considered the minimum compression the word length capable of keeping the dynamic range and also the verification outcomes of all the safety properties. In Vocalic, e.g., 22 bits were the minimum word length that resulted on the same verification outcomes that 32 bits provided.

We have also evaluated and compared our tool with Marabou and Neurify. Consider-

ing the ACASXu property one benchmarks [80], we have observed that our approach is significantly faster and solves more verification tasks than Marabou [14], a competitive opponent in SMT-based verification. However, in many cases, Neurify [15] is faster than our tool since it deploys a multi-threaded algorithm to solve the verification tasks. However, note that neither Marabou nor Neurify can verify quantized neural networks as in our approach.

6.1 Future Works

Finally, we believe that the problem of verifying ANNs is still open. More specifically, it is unclear which set of techniques yields the best performance when scaling to large networks. In this regard, our future work includes comparing our verification approach to other existing techniques and optimizing our verification performance even further. In addition, the results of our work can be regarded as the first steps towards an approach capable of revealing the most aggressive ANN representation that still provides correct operation, which aims at achieving maximum compression for a particular model.

We believe that symbolic execution and iterative refinement [99] will improve our performance and scale our approach to even bigger ANNs. Since these techniques could be executed before the verification step and trigger the verification machine to check for counterexamples under only the refined intervals.

Another promising approach is improving the decision procedure that picks the most relevant input elements w.r.t the questioned safety property. Then ESBMC could be used to efficiently refine the most relevant intervals and then perform the verification process under the refined intervals.

There are also some techniques presented in Neurify [15] and Reluplex [100] can be integrated before and during the verification process. Lp solvers are more suitable to the safety properties solving problem, due its efficiency when dealing with piecewise-linear systems as Relu activation functions.

Bibliography

[1] C. Bishop, Pattern Recognition and Machine Learning. Springer, 2006.

[2] M. Nour, Z. Cömert, and K. Polat, "A novel medical diagnosis model for COVID-19 infection detection based on deep features and bayesian optimization," Appl. Soft Comput., vol. 97, p. 106580, 2020.

[3] H. Wu, D. Lv, T. Cui, G. Hou, M. Watanabe, and W. Kong, "SDLV: Verification of steering angle safety for self-driving cars," Form. Asp. Comput., 2021.

[4] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in Conf. on Computer Vision and Pattern Recognition, 2018, pp. 1625–1634.

[5] S. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in Advances in Neural Information Processing Systems 30, I. Guyon, U. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774.

[6] M. Fazlyab, M. Morari, and G. Pappas, "Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming," IEEE Trans. Autom. Control, pp. 1–15, 2020.

[7] A. Rössig and M. Petkovic, "Advances in verification of ReLU neural networks," J. Glob. Optim., 2020.

[8] A. Venzke and S. Chatzivasileiadis, "Verification of neural network behaviour: Formal guarantees for power system applications," IEEE Trans. Smart Grid, vol. 12, no. 1, pp. 383–397, 2021.

[9] C. Huang, J. Fan, X. Chen, W. Li, and Q. Zhu, "Divide and slide: Layer-wise refinement for output range analysis of deep neural networks," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pp. 3323–3335, 2020.

[10] C. Huang, J. Fan, W. Li, X. Chen, and Q. Zhu, "ReachNN: Reachability analysis of neural-network controlled systems," ACM Trans. Embed. Comput. Syst., vol. 18, no. 5s, pp. 1–22, 2019.

[11] W. Xiang, H.-D. Tran, and T. T. Johnson, "Output reachable set estimation and verification for multilayer neural networks," IEEE Trans. Neural Netw. Learn. Syst., vol. 29, no. 11, pp. 5777–5783, 2018.

[12] 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 Computer Aided Verification. Springer, 2020, pp. 3–17.

[13] R. Ivanov, T. Carpenter, J. Weimer, R. Alur, G. Pappas, and I. Lee, "Verifying the safety of autonomous systems with neural network controllers," ACM Trans. Embed. Comput. Syst., vol. 20, no. 1, pp. 1–26, 2021.

[14] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. L. Dill, M. J. Kochenderfer, and C. Barrett, "The marabou framework for verification and analysis of deep neural networks," in Computer Aided Verification. Springer, 2019, pp. 443–452.

[15] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient Formal Safety Analysis of Neural Networks," in 32nd Int. Conf. on Neural Information Processing Systems. Curran Associates Inc., 2018, p. 6369–6379.

[16] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," 2018.

[17] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in Computer Aided Verification. Springer, 2017, pp. 3–29.

[18] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in Computer Aided Verification. Springer, 2017, pp. 97–117.

[19] T. A. Henzinger, M. Lechner, and D. Žikelić, "Scalable verification of quantized neural networks (technical report)," 2020.

[20] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in Int. Conf. on Learning Representations, 2019.

[21] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener, "Efficient verification of ReLU-based neural networks via dependency analysis," Conf. on Artificial Intelligence, vol. 34, no. 04, pp. 3291–3299, 2020.

[22] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli, "A dual approach to scalable verification of deep networks," in 34th Conf. on Uncertainty in Artificial Intelligence, vol. 2, 2018, pp. 550–559.

[23] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in Computer Vision – ECCV, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 525–542.

[24] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in Advances in Neural Information Processing Systems, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016.

[25] C.-H. Cheng, G. Nührenberg, C.-H. Huang, and H. Ruess, "Verification of binarized neural networks via inter-neuron factoring," in Verified Software. Theories, Tools, and Experiments, R. Piskac and P. Rümmer, Eds. Cham: Springer International Publishing, 2018, pp. 279–290.

[26] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in 4th Int. Conf. on Learning Representations, Y. Bengio and Y. LeCun, Eds., 2016.

[27] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in 33rd Int. Conf. on Machine Learning, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. PMLR, 2016, pp. 2849–2858.

[28] L. Pulina and A. Tacchella, "Challenging smt solvers to verify neural networks," Ai Communications, vol. 25, no. 2, pp. 117–135, 2012.

[29] M. Baranowski, S. He, M. Lechner, T. S. Nguyen, and Z. Rakamarić, "An smt theory of fixed-point arithmetic," in Automated Reasoning, N. Peltier and V. Sofronie-Stokkermans, Eds. Cham: Springer International Publishing, 2020, pp. 13–31.

[30] M. Giacobbe, T. A. Henzinger, and M. Lechner, "How many bits does it take to quantize your neural network?" in Tools and Algorithms for the Construction and Analysis of Systems, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 79–97.

[31] G. Parascandolo, H. Huttunen, and T. Virtanen, "Taming the waves: sine as activation function in deep neural networks," 2016.

[32] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," Advances in Neural Information Processing Systems, vol. 33, 2020.

[33] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[34] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2020.

[35] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in Advances in Neural Information Processing Systems, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016.

[36] Y. Guo, "A survey on methods and theories of quantized neural networks," 2018.

[37] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," Distributed computing, vol. 2, no. 3, pp. 117–126, 1987.

[38] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," Comput. Sci. Rev., vol. 37, p. 100270, 2020.

[39] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," Foundations and Trends in Optimization, vol. 4, no. 3-4, pp. 244–404, 2021.

[40] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in Int. Conf. on Learning Representations, 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[41] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev, "Fast and effective robustness certification." NeurIPS, vol. 1, no. 4, p. 6, 2018.

[42] R. Jia, A. Raghunathan, K. Göksel, and P. Liang, "Certified robustness to adversarial word substitutions," in Conf. on Empirical Methods in Natural Language Processing and the 9th Int. Joint Conf. on Natural Language Processing. Hong Kong, China: Association for Computational Linguistics, 2019, pp. 4129–4142.

[43] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in Handbook of Model Checking. Springer, 2018, pp. 305–343.

[44] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," Proceedings of the IEEE, vol. 103, no. 11, pp. 2021–2035, 2015.

[45] C. Barrett, A. Stump, C. Tinelli et al., "The smt-lib standard: Version 2.0," in 8th Int. workshop on satisfiability modulo theories, vol. 13, 2010, p. 14.

[46] R. de Salvo Braz, C. O'Reilly, V. Gogate, and R. Dechter, "Probabilistic inference modulo theories," in IJCAI, 2016.

[47] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[48] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in Computer Aided Verification. Springer, 2011, pp. 171–177.

[49] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2009, pp. 174–177.

[50] L. Sena, I. Bessa, M. Ramalho, L. Cordeiro, and E. Mota, "Incremental bounded model checking of artificial neural networks in CUDA," in IX Brazilian Symp. on Computing Systems Engineering, 2019.

[51] K.-S. Oh and K. Jung, "Gpu implementation of neural networks," Pattern Recognition, vol. 37, no. 6, pp. 1311–1314, 2004.

[52] J. Zhu and P. Sutton, "Fpga implementations of neural networks – a survey of a decade of progress," in Field Programmable Logic and Application, P. Y. K. Cheung and G. A. Constantinides, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1062–1066.

[53] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," ACM Comput. Surv., vol. 52, no. 2, 2019.

[54] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in 43rd Int. Symp. on Computer Architecture, 2016, pp. 267–278.

[55] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," J. Mach. Learn. Res., vol. 18, no. 1, p. 6869–6898, 2017.

[56] L. Chaves, H. Ismail, I. Bessa, L. Cordeiro, and E. B. de Lima Filho, "Verifying fragility in digital systems with uncertainties using DSVerifier v2.0," J Syst Softw, vol. 153, no. 2019, pp. 22–43, 2019.

[57] D. Manzanas Lopez, T. Johnson, H.-D. Tran, S. Bak, X. Chen, and K. L. Hobbs, "Verification of neural network compression of acas xu lookup tables with star set reachability," in AIAA Scitech 2021 Forum, 2021, p. 0995.

[58] M. O. Searcóid, Metric Spaces. Springer-Verlag, 2006.

[59] M. Gadelha, F. Monteiro, J. Morse, L. Cordeiro, B. Fischer, and D. Nicole, "ESBMC 5.0: an industrial-strength C model checker," in 33rd Int. Conf. on Automated Software Engineering, 2018, pp. 888–891.

[60] M. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "ESBMC v6.0: Verifying C programs using k-Induction and invariant inference - (competition contribution)," in 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2019, pp. 209–213.

[61] H. F. Albuquerque, R. F. Araújo, I. Bessa, L. Cordeiro, and E. B. de Lima Filho, "Optce: A counterexample-guided inductive optimization solver," in Formal Methods: Foundations and Application, 2017.

[62] W. Rocha, H. Rocha, H. Ismail, L. Cordeiro, and B. Fischer, "DepthK: A k-Induction verifier based on invariant inference for C programs - (competition contribution)," in 26th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2017, pp. 360–364.

[63] R. E. Moore, R. B. Kearfott, and M. J. Cloud, Introduction to Interval Analysis. USA: Society for Industrial and Applied Mathematics, 2009.

[64] A. Blanchard, N. Kosmatov, and F. Loulergue, "A lesson on verification of iot software with frama-c," in Int. Conf. on High Performance Computing & Simulation, 2018.

[65] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer, "Esbmc 1.22," in 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2014, pp. 405–407.

[66] E. M. Clarke, T. A. Henzinger, and H. Veith, Introduction to Model Checking. Cham: Springer International Publishing, 2018, pp. 1–26.

[67] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in Computer Aided Verification, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190.

[68] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," in Tools and Algorithms for the Construction and Analysis of Systems, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.

[69] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, vol. 13, pp. 451–490, 1991.

[70] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[71] H. Günther and G. Weissenbacher, "Incremental bounded software model checking," in 21st Int. SPIN Symp. on Model Checking of Software, 2014, pp. 40–47.

[72] B. Dutertre, "Yices 2.2," in Computer Aided Verification. Springer, 2014, pp. 737–744.

[73] L. Cordeiro, "Smt-based bounded model checking of multi-threaded software in embedded systems," Ph.D. dissertation, University of Southampton, UK, 2011. [Online]. Available: http://eprints.soton.ac.uk/186011/

[74] J. Morse, "Expressive and efficient bounded model checking of concurrent software," Ph.D. dissertation, University of Southampton, UK, 2015. [Online]. Available: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.658818

[75] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," arXiv preprint arXiv:1912.01703, 2019.

[76] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in 12th USENIX Symp. on operating systems design and implementation, 2016, pp. 265–283.

[77] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[78] A. Gulli and S. Pal, Deep learning with Keras. Packt Publishing Ltd, 2017.

[79] L. Cordovil, P. Coutinho, I. Bessa, M. F. D'Angelo, and R. Palhares, "Uncertain data modeling based on evolving ellipsoidal fuzzy information granules," IEEE Trans. Fuzzy Syst., vol. 28, no. 10, pp. 2427–2436, 2020.

[80] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in 35th Digital Avionics Systems Conf. IEEE, 2016, pp. 1–10.

[81] F. Monteiro, E. Alves, I. Silva, H. Ismail, L. Cordeiro, and E. de Lima-Filho, "ESBMC-GPU a context-bounded model checking tool to verify cuda programs," Sci. Comput. Program., vol. 152, pp. 63–69, 2018.

[82] D. Beyer, "Software verification: 10th comparative evaluation (sv-comp 2021)," Tools and Algorithms for the Construction and Analysis of Systems, vol. 12652, p. 401, 2021.

[83] J. Morse, M. Ramalho, L. Cordeiro, D. A. Nicole, and B. Fischer, "ESBMC 1.22 - (competition contribution)," in 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, ser. LNCS, vol. 8413, 2014, pp. 405–407.

[84] M. Gadelha, R. Menezes, F. R. Monteiro, L. Cordeiro, and D. A. Nicole, "ESBMC: scalable and precise test generation based on the floating-point theory - (competition contribution)," in 23rd Int. Conf. Fundamental Approaches to Software Engineering, ser. LNCS, vol. 12076, 2020, pp. 525–529.

[85] A. De Lucia, "Program slicing: Methods and applications," in 1st Int. Workshop on Source Code Analysis and Manipulation. IEEE, 2001, pp. 142–149.

[86] D. Karmon, D. Zoran, and Y. Goldberg, "LaVAN: Localized and visible adversarial noise," in 35th Int. Conf. on Machine Learning, J. Dy and A. Krause, Eds., vol. 80. PMLR, 2018, pp. 2507–2515.

[87] M. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "Esbmc v6.0: Verifying c programs using k-induction and invariant inference," in 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2019.

[88] D. Bühler, "Eva, an evolved value analysis for frama-c : structuring an abstract interpreter through value and state abstractions," Ph.D. dissertation, 2017, thèse de doctorat dirigée par Blazy, Sandrine et Yakobowski, Boris Informatique Rennes 1 2017. [Online]. Available: http://www.theses.fr/2017REN1S016/document

[89] J. Zory and F. Coelho, "Using algebraic transformations to optimize expression evaluation in scientific code," in Int. Conf. on Parallel Architectures and Compilation Techniques, 1998, pp. 376–384.

[90] E. Dekel, S. Ntafos, and S.-T. Peng, "Parallel tree techniques and code optimization," in VLSI Algorithms and Architectures, F. Makedon, K. Mehlhorn, T. Papatheodorou, and P. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 205–216.

[91] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal, "Verification of loop and arithmetic transformations of array-intensive behaviors," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 32, no. 11, pp. 1787–1800, 2013.

[92] L. Chaves, I. Bessa, H. Ismail, A. B. Frutuoso, L. Cordeiro, and E. B. de Lima Filho, "DSVerifier-Aided verification applied to attitude control software in unmanned aerial vehicles," IEEE Trans. Reliab., vol. 67, no. 4, pp. 1420–1441, 2018.

[93] K. Jia and M. Rinard, "Verifying low-dimensional input neural networks via input quantization," arXiv preprint arXiv:2108.07961, 2021.

[94] G. Amir, H. Wu, C. Barrett, and G. Katz, "An smt-based approach for verifying binarized neural networks," in Int. Conf.on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2021, pp. 203–222.

[95] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in 27th Conf. on Security Symp. USENIX Association, 2018, p. 1599–1614.

[96] P. Dey, K. Nag, T. Pal, and N. R. Pal, "Regularizing multilayer perceptron for robustness," IEEE Trans. Syst., Man, Cybern. Syst., vol. 48, no. 8, pp. 1255–1266, 2018.

[97] K. Jia and M. Rinard, "Exploiting verified neural networks via floating point numerical error," 2020.

[98] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Structural test coverage criteria for deep neural networks," ACM Trans. Embed. Comput. Syst., vol. 18, no. 5s, pp. 1–23, 2019.

[99] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in 27th USENIX Security Symp., 2018, pp. 1599–1614.

[100] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," 02 2017.