Automated Verification and Repair of Quantized Neural Networks

A thesis submitted to the University of Manchester for the degree of
Doctor of Philosophy
in the Faculty of Science and Engineering

2024

Xidan Song
Department of Computer Science

Contents

Word count: 33477

List of Figures

List of Tables

Abstract

Developing secure and bug-free AI systems is extraordinarily challenging. Detecting vulnerabilities in AI systems is particularly difficult due to the devastating effects that such vulnerabilities can have on financial security or an individual's well-being. The complexity of AI models and their increasing deployment in critical applications necessitate robust methods for ensuring their security and reliability.

To address these challenges, we present three significant novel contributions. Firstly, we introduce *QNNVerifier*, a state-of-the-art neural network verification method based on the SMT bounded model checker ESBMC. *QNNVerifier* is designed to identify vulnerabilities in quantized neural network models, which use reduced precision to improve computational efficiency and are susceptible to unique errors. It employs optimization algorithms to reduce verification complexity and thoroughly analyzes models to pinpoint potential security flaws. Secondly, we introduce *QNNRepair*, our innovative neural network repair method. Based on Mixed Integer Linear Programming and Gurobi as the backend, *QNNRepair* rectifies vulnerabilities and increases robustness in neural network models by applying targeted modifications in neurons. Thirdly, we introduce a comprehensive neural network repair platform, *AIRepair*, which integrates our methods and ensures compatibility with mainstream frameworks like TensorFlow and PyTorch.

We evaluate the effectiveness of *QNNVerifier* and *QNNRepair* within our *AIRepair* framework using state-of-the-art benchmarks. Our results show that *QNNVerifier* effectively detects vulnerabilities in quantized neural networks with high precision. Moreover, *QNNRepair* successfully addresses these issues, enhancing model robustness without compromising accuracy or performance. The *AIRepair* framework integrates these methods, streamlining the process of identifying and fixing vulnerabilities in neural network models.

In conclusion, our contributions provide a comprehensive solution for enhancing the security and reliability of AI systems. By combining advanced verification and repair techniques with a user-friendly platform, we offer a valuable toolset for developing secure and resilient AI systems, ultimately contributing to the safety and trustworthiness of AI technologies in critical applications.

Declaration of originality

I hereby confirm that no portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright statement

List of publications

Luiz Sena, **Xidan Song**, Erickson Alves, Iury Bessa, Edoardo Manino, and Lucas C. Cordeiro. "Verifying quantized neural networks using SMT-based model checking," *arXiv preprint arXiv:2106.05997*, 2021. Correspond to Chapter 4.

João Batista P. Matos, Eddie de Lima Filho, Iury Bessa, Edoardo Manino, **Xidan Song**, and Lucas C. Cordeiro, "Counterexample guided neural network quantization refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023. Correspond to Chapter 4.

Conference articles

Xidan Song, Youcheng Sun, Mustafa A. Mustafa, and Lucas C. Cordeiro, "QNNRepair: Quantized neural network repair," in *International Conference on Software Engineering and Formal Methods*, Springer, 2023, pp. 320–339. Correspond to Chapter 5.

Xidan Song, Youcheng Sun, Mustafa A. Mustafa, and Lucas C. Cordeiro, "AIREPAIR: A repair platform for neural networks," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2023, pp. 98–101. Correspond to Chapter 6.

Edoardo Manino, Danilo Carvalho, Yi Dong, Julia Rozanova, **Xidan Song**, Mustafa A. Mustafa, Andre Freitas, Gavin Brown, Mikel Lujan, Xiaowei Huang and Lucas Cordeiro, "Enncore: End-to-end conceptual guarding of neural architectures," 2022.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Lucas Cordeiro, for his unwavering support, insightful guidance, and constant encouragement throughout my research journey. Your expertise and dedication have been invaluable to the completion of this dissertation. Your patience, insights, and good humour were invaluable and very much appreciated. I would also like to thank my co-supervisors, Dr. Mustafa A. Mustafa and Dr. Youcheng Sun, for their guidance, assistance, and invaluable feedback.

I am also deeply grateful to my colleagues at the Systems and Software Security group, especially Edoardo Manino, Chenfeng Wei, and Tong Wu, for their support, camaraderie, and stimulating discussions that have enriched my research experience.

A special thanks to the administrative and technical staff at the Department of Computer Science, University of Manchester, for their assistance and for providing a conducive environment for my research.

Thanks to the University of Manchester. The research community at the University of Manchester helped make the PhD an enjoyable experience. The University of Manchester also offers me the chance to work as a Teaching and Research Assistant.

On a personal note, I would like to express my deepest appreciation to my family. To my parents, Mingrun and Qingzhi, for their unconditional love and support and for always believing in me. They also generously fund my educational journey, enabling me to pursue my dreams and acquire the knowledge and skills that have shaped my future.

Abbreviations

ACAS	Airborne Collision Avoidance Systems
ACSL	ANSI/ISO C Specification Language
ANN	Artificial Neural Network
BMC	Bounded Model Checking
CE	Counterexample
CEGAR	Counterexample-guided Abstraction Refinement
CIFAR	Canadian Institute For Advanced Research
COC	Clear-of-Conflict
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
DRP	Domain-wise Depair Problem
ESBMC	Efficient SMT-based Bounded Model Checker
EVA	Evolved Value Analysis
FFNN	Feed-Forward Neural Network
GTSRB	German Traffic Sign Recognition Benchmark
GLPK	GNU Linear Programming Kit
HDF5	Hierarchical Data Format version 5
LP	Linear Programming
MILP	Mixed Integer Linear Programming
MNIST	Modified National Institute of Standards and Technology database
NLP	Natural Language Processing

PSO	Particle Swarm Optimization
PWL	Piecewise Linear
QNN	Quantized Neural Network
ReLU	Rectified Linear Unit
ResNet	Residual Network
RNN	Recurrent Neural Network
rDLM	reduced Deep Learning Model
SAT	Satisfiability Solver
SMT	Satisfiability Modulo Theories
SRP	Sample-wise DNN Repair Problem
SSA	Static Single Assignment
TFLite	TensorFlow Lite
VGGNet	Visual Geometry Group Network

.

Chapter 1

Introduction

The widespread adoption of internet services and smart devices globally has driven significant advancements in labor automation, device interconnectivity, and human-machine interaction, leading to an exponential increase in available data compared to previous decades [1]. At the core of transforming abundant data and computational resources into practical applications is Machine Learning (ML), a field that intersects statistics and computer science [2]. ML's data-driven approach addresses complex problems that are difficult to model using rule-based systems founded on human expertise. Such issues include image classification, spam email filtering, and image segmentation [3].

Among ML, in particular, Deep Learning has demonstrated its strong performance and application potential in image classification [4], object detection [5], and natural language processing [6]. To elaborate, ML is a subset of Artificial Intelligence, whereas Deep Learning is a specific approach within ML. This hierarchical relationship helps us to understand how advancements in ML directly contribute to the broader field of Artificial Intelligence. For instance, in March 2016, AlphaGo [7], an artificial intelligence program developed by Google's DeepMind, made headlines worldwide by defeating Lee Sedol, one of the world's top Go players, in a five-game match. AlphaGo's success is a testament to how sophisticated ML algorithms, particularly those developed through Deep Learning techniques, can lead to groundbreaking achievements in Artificial Intelligence.

Neural networks are the core technology of deep learning, which mimic how the human brain works by processing complex data through many layers and nodes. Due to these capabilities, neural networks have also been widely used in safety-sensitive fields such as autonomous driving, facial recognition, and medical diagnosis [8]. For example, in autonomous driving, neural networks identify road signs and obstacles; in medical diagnosis, they can help recognize disease signs and perform pathology image analysis.

However, these advancements in AI introduce entirely new challenges, especially concerning the security and robustness of the models [9]. This lack of robustness becomes especially apparent in practical scenarios, where even minor modifications to road signs could mislead automated driving systems, leading to potentially hazardous outcomes.

Neural network verification [10] aims to validate the behavior of a neural network model for a specified range of inputs, either through complete or incomplete methods that approximate the verification problem. However, due to neural networks' intricate and opaque nature, deploying verification on large models and interpreting the results pose significant challenges. Researchers have explored various techniques to mitigate these challenges to simplify verification processes, including linear relaxation and abstract interpretation [11].

While neural network verification identifies issues with model robustness, complete retraining does not always resolve these issues and can be resource-intensive. Consequently, researchers have developed neural network repair methods [12]–[14] that address these vulnerabilities without requiring full retraining or access to original data. However, due to the complexity of neural network models, the repair process can be time-consuming and computationally demanding, prompting further research into optimizing repair procedures.

The exponential increase in data and computational resources has catalyzed significant advancements in ML, particularly in deep learning, which has demonstrated remarkable capabilities in various applications such as image classification [4], object detection [5], and natural language processing [6]. The success of neural networks in safety-sensitive fields, such as autonomous driving and medical diagnosis, highlights their potential and underscores the critical need for robust and secure models. Neural network verification and repair techniques have emerged to address these challenges, aiming to ensure model reliability without complete retraining. However, these processes are often complex and resource-intensive, necessitating ongoing research to enhance efficiency and effectiveness.

1.1 Research Motivation

Presently, a multitude of neural network training frameworks are prevalent, including TensorFlow [15], PyTorch [16], Caffe [17], among others. Each of these frameworks employs distinct formats for storing neural network models. However, existing neural network verification and repair tools are often limited to supporting only a single file format. Consequently, determining the optimal file format or category for repairing a neural network poses a challenge. Hence, there arises a necessity for a unified platform encompassing both verification and repair functionalities, facilitating format conversion and method comparison.

This challenge is further compounded by the inherent fragility and opacity of neural networks, often considered hard-to-understand black-box systems. For instance, a 2022 study by Australian researchers [18] analyzed three AI applications designed for mushroom identification and found that, on average, these apps correctly identified wild mushrooms only about 50% of the time. In some cases, the apps misidentified toxic mushrooms as edible, highlighting the potential dangers of overreliance on AI technology [19]. This is just one example of how the vulnerabilities of neural networks can lead to harmful outcomes.

Recent studies have revealed that Deep Neural Networks (DNNs) exhibit susceptibility to manipulation, wherein slight alterations in input data can result in incorrect predictions [20]–[22]. These alterations can take various forms, such as pixel-level perturbations [9], [20], [23], or through natural variations in the input, such as image rotation or changes in illumination [22], [24]. Natural variations pose a significant challenge as they can occur without malicious intent and have severe consequences [21], [25]. Testing [21] is a natural idea to determine whether a neural network is robust to such perturbations. Some scholars have developed the technique of neural network testing, where a neural network is given either normal or adversarial inputs against an attack and is allowed to make inferences [26]. The given neural network is judged to be robust based on whether the neural network output is normal but cannot guarantee the absence of failures.

In contrast to neural networks, there are already a variety of well-established methods and processes for verifying and validating the behavior of autonomous systems before deployment. This is particularly important for safety-critical systems, which can harm human life if not operated properly. Examples of such problems have been seen in the self-driving features of automobiles, resulting in civilian deaths [27]. Verification is a process used to ensure that the system of interest meets all requested requirements.

It is natural to think that we need neural network verification to find these defects in neural networks. Still, the difference between verifying neural network models and traditional software is safety properties and ideas. Traditional software verification [28] usually relies on explicit requirements specifications and logical tests to ensure the software functions as intended. This involves functional testing, performance testing, etc., to ensure that the stability and reliability of the software. Neural network verification [10], however, focuses on the representativeness of the data and the model's ability to generalize, often by splitting the data set (e.g., training, validation, and testing sets) to assess the model's ability to predict new data. The uncertainty and black-box nature of neural networks [29] also makes the verification process more focused on statistical metrics and error analysis. Therefore, due to the different verification goals, we are motivated to develop a new method for verifying quantized neural networks.

After identifying the vulnerabilities in neural network models, we must find a method to fix or improve them. In traditional software development, vulnerability remediation is usually a well-defined process that involves identifying bugs, analyzing the cause of the problem, writing the patch, testing the patch to ensure that no new problems are introduced, and ultimately deploying the patch [30]. This process relies on the understandability and operability of the code.

In neural network models, "repair" usually means modifying the network architecture, adjusting the hyperparameters, or re-selecting the training data to improve the model's performance or reduce errors [31]. This kind of repair is not done by directly modifying the code. Still, improving the learning algorithms and data inputs often requires a lot of experiments to find the optimal configuration. Thus, we also need a new method to repair and fix the vulnerabilities in neural network models.

Implemented in mobile devices, such as the mushroom recognition neural network, is a quantized neural network model converted from a floating point model to minimize computational requirements. Nonetheless, recent research on validating quantized neural networks is scarce, with no existing work on rectifying them. Hence, this thesis introduces the quantized neural network verification method *QNNVerifier*, the quantized neural network repair method *QNNRepair*, and the QNN repair platform *AIRepair*.

1.2 Research Questions

Today, neural network verification and repair tools are developed on various platforms. They cannot be uniformly compared and used, nor can it be determined which verification or repair method is best for a particular neural network. This type of repair does not require retraining or adding additional repair units. For some specific neural networks, such as quantized neural networks, there is still a lack of research on verifying and repairing such networks. More precisely, in this Ph.D. thesis, I ask the following fundamental research question:

Research Question. *How to verify, repair and evaluate quantized neural networks?*

Our starting point is to first use neural network verification to determine whether a given neural network model is robust. If a given neural network violates a safety property, then neural network repair methods should be utilized to bring the violating neural network into compliance with the safety statute. The verification uses an SMT-based approach to ensure completeness, with ESBMC as the backend, and the repair uses a linear programming approach to ensure efficiency, with Gurobi as the backend.

In addressing this question, we are confronted with many practical design challenges. Now, I divided this general research question into three sub-questions in this PhD thesis:

1.2.1 *RQ1: How to find vulnerabilities in Quantized Neural Network Models?*

Although there have been recent attempts to verify floating-point neural networks, converting fixed-point neural networks into a constraint-solving problem is an important step toward answering our research question. The question is, for a given trained model and a set of safety properties, we translate them to a set of SMT inequalities. Then, this is the input to the SMT-solver, which will give the result of neural network verification. During this procedure, the verification time is of the most concern. We want to generate the most efficient formulas free of search-space explosion and loops. This challenge and the following sub-challenges are tackled in Chapter 4.

1.2.2 *RQ2: How to Repair Quantized Neural Network Models?*

There are certain technical difficulties in quantized model repair. Suppose we want to repair quantized models by retraining or adding repair units. In that case, we need to find the original floating point model, complete the repair operation, and quantify it due to the limitations of deep learning platforms. For example, the TensorFlow framework's TFlite stores quantized neural network models [32]. Still, this format only supports inference, not retraining, as parameters such as loss are omitted from the quantization process. However, the original model is difficult to find and the cost required to retrain and re-quantize is prohibitive. This challenge is tackled in Chapter 5.

1.2.3 *RQ3: How to Reach Trade-off Between Repairing Correctness and Efficiency?*

The main disadvantage of using an integrated framework of different tools is that they all share the same computational resources. We must decide how many resources to allocate to each tool for a single verification procedure. Thus, we developed the *AIRepair* tool. Generally, this decision depends not only on the problem at hand but also on the partial results that we obtain from the tools in the cooperative framework. Specifically, in our cooperative framework, we combined the tools sequentially. We discuss a strategy to optimize our cooperative framework in Chapter 6.

1.3 Contributions

The main contribution of this Ph.D. thesis is the development, implementation, and evaluation of scalable methods to address vulnerabilities in quantized neural network (QNN) models, focusing on their verification and repair. Specifically, the thesis introduces the following three major novel contributions:

- *QNNVerifier*:
  - Introduced a new, fully open-source quantized neural network verification method, *QNNVerifier*, as detailed in [33], [34].
  - Designed and evaluated a novel symbolic verification framework leveraging software model checking (SMC) and satisfiability modulo theories (SMT) to detect vulnerabilities in artificial neural networks (ANNs).
  - Proposed several ANN-specific optimizations for SMC, including:
    - Invariant inference using interval analysis.
    - Program slicing and expression simplifications.
    - Discretization of non-piecewise-linear activation functions.
  - Portions of this work are published in [33].
- *QNNRepair*:
  - Presented *QNNRepair*, the first method in the literature dedicated to repairing quantized neural networks (QNNs).
  - Aimed at improving neural network performance post-quantization by addressing accuracy degradation.
  - Key features of *QNNRepair* include:
    - Identification of neurons causing performance degradation using software fault localization techniques.
    - Formulation of the repair problem as a mixed-integer linear programming (MILP) problem to adjust neuron weight parameters.
    - Ensured repaired QNNs improve performance on failing tests while preserving accuracy on passing tests.
  - Portions of this work are published in [35].
- *AIRepair*:
  - Proposed *AIRepair*, a comprehensive platform for neural network repair that builds upon *QNNRepair* and *QNNVerifier*.
  - Features integration of existing repair tools for fair comparison and evaluation of different repair techniques.

- Capabilities of *AIRepair* include:
    - ∗ Support for trained models and their training datasets (if specified in the configuration).
    - ∗ Pre-processing of benchmarks to ensure compatibility with various frameworks, such as TensorFlow [15] and PyTorch [16].
    - ∗ Isolation of different running environments for deep learning libraries.
    - ∗ Automated collection and analysis of repair results, including outputs, logs, and parameters.
    - ∗ Presentation of results to help users decide on the most suitable repair tool.
- Outputs include the repaired model along with comprehensive logs and parameter details.
- Portions of this work are published in [36].

1.4 Thesis Structure

The remainder of this thesis is arranged in the following structure; the thesis's visual structure is also illustrated in Figure 1.1.

Chapter 2 introduces the core concepts behind neural network verification and repair. First, it introduces the concepts, applications, and security issues of neural networks. Then, it introduces neural network vulnerabilities and the need for verification and repair. In addition, it presents related work on neural network verification, reachability analysis, testing, and repair.

Chapter 4 introduces *QNNVerifier*, a quantized neural network verification method based on SMT solving. This chapter describes the verification process of *QNNVerifier*, where we transform quantized neural networks into C files and use ESBMC as a backend for SMT verification. We also highlight the challenges in verifying quantized neural networks and how our approach handles and solves them.

Chapter 5 describes *QNNRepair*, a quantized neural network repair method. In this method, we first run neural network inference and then sort the neurons according to their importance. Afterward, using the repair set, the neural network repair problem is converted to a linear programming problem and solved using Gurobi as the backend.

Chapter 6 designs a platform *AIRepair* specialized for repairing neural networks.Firstly, it receives the models trained from different platforms and performs the model transformation. Then, it automatically selects the repair method according to the characteristics of these models and gives the repaired model with the repair report.

Chapter 7 summarizes the research contributions, highlighting the proposed framework's significance in verifying and repairing quantized neural network models, the evaluation, and future work.

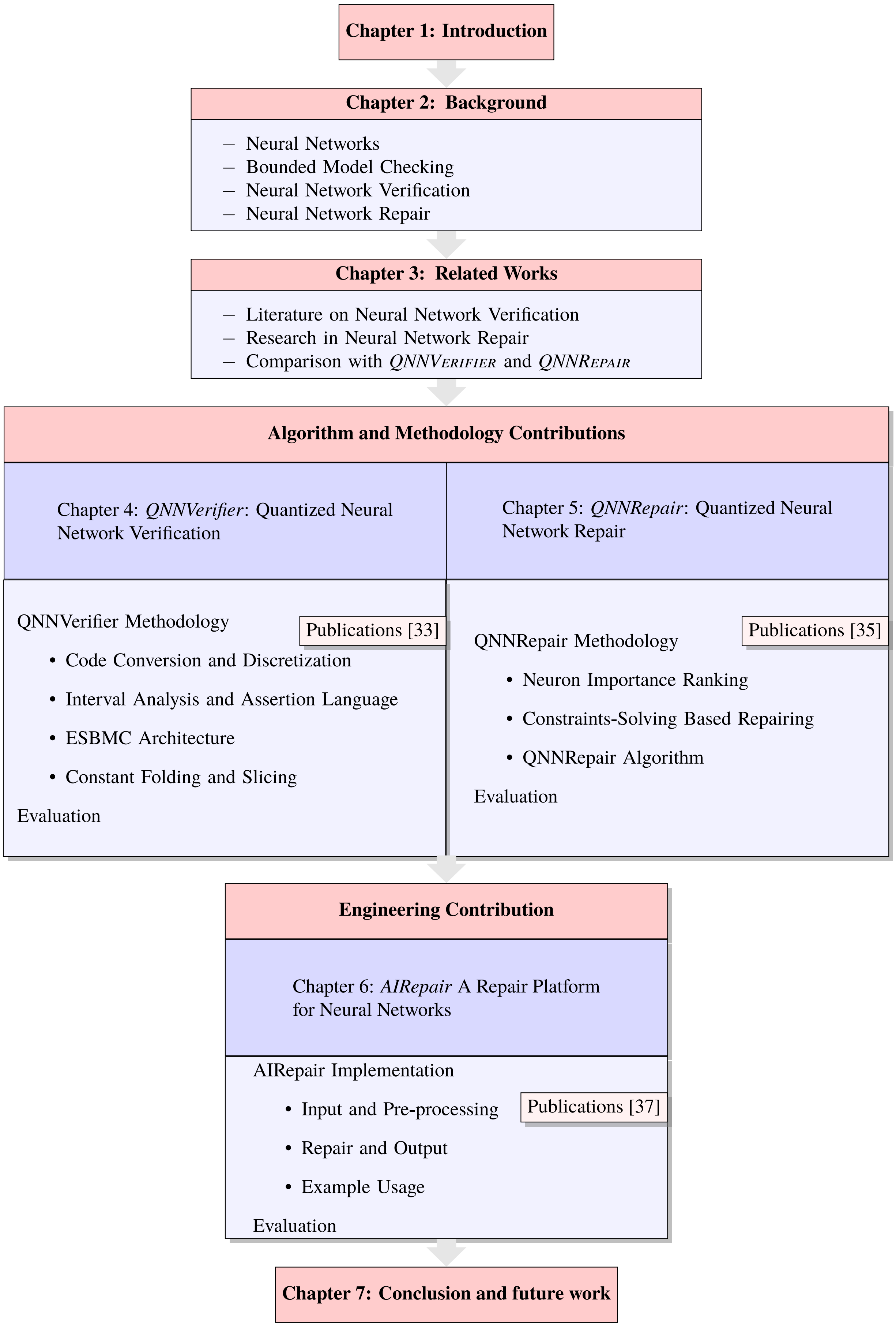

Chapter 1: Introduction
Chapter 2: Background
– Neural Networks
– Bounded Model Checking
– Neural Network Verification
– Neural Network Repair
Chapter 3: Related Works
– Literature on Neural Network Verification
– Research in Neural Network Repair
– Comparison with QNNVerifier and QNNRepair
Algorithm and Methodology Contributions
Chapter 4: QNNVerifier: Quantized Neural Network Verification
Chapter 5: QNNRepair: Quantized Neural Network Repair
QNNVerifier Methodology
Publications [33]
• Code Conversion and Discretization
• Interval Analysis and Assertion Language
• ESBMC Architecture
• Constant Folding and Slicing
Evaluation
QNNRepair Methodology
Publications [35]
• Neuron Importance Ranking
• Constraints-Solving Based Repairing
• QNNRepair Algorithm
Evaluation
Engineering Contribution
Chapter 6: AIRepair A Repair Platform for Neural Networks
AIRepair Implementation
Publications [37]
• Input and Pre-processing
• Repair and Output
• Example Usage
Evaluation
Chapter 7: Conclusion and future work

Figure 1.1. Thesis structure.

Chapter 2

Background

This chapter introduces the core concepts behind verifying and repairing neural networks. First, it presents neural networks, applications, and security concerns. Then, it includes neural network vulnerabilities and the necessity of neural network verification and repair, Bounded Model Checking (BMC), Abstract Interpretation, and Satisfiability Module Theory (SMT). It also presents related work in neural network verification, reachability analysis, testing, and repair.

2.1 Neural Networks (NNs)

A neural network consists of an input layer, an output layer, and one or more intermediate layers called hidden layers [38]. Each layer is a collection of nodes called neurons. The neuron accepts the signals entering it via the dendrites, performs a computation on those signals, and generates a signal on the axon. These input and output signals are referred to as activations. Each neuron is connected to other neurons by one or more directed edges.

Neural networks can be represented as graphs [39]. As Figure 2.1 shows, the edges (arrows) represent the weights and biases of linear transformations between the layers. The circles represent the nonlinear activation functions performed by the neurons or units. The interior (colored) layers are called hidden layers. Network architectures are described by their depth (number of layers) and layer widths (number of units).

Deep Neural Networks (DNNs) are a class of artificial neural networks with multiple layers between the input and output layers [40]. DNNs are a fundamental component of deep learning, a subset of machine learning focusing on models with many layers (hence the term "deep").

Let $f : \mathcal{I} \to \mathcal{O}$ be the neural network N with m layers. We use the most widely used neural network for image classification as an example. For a given input $x \in \mathcal{I}$, $f(x) \in \mathcal{O}$ calculates

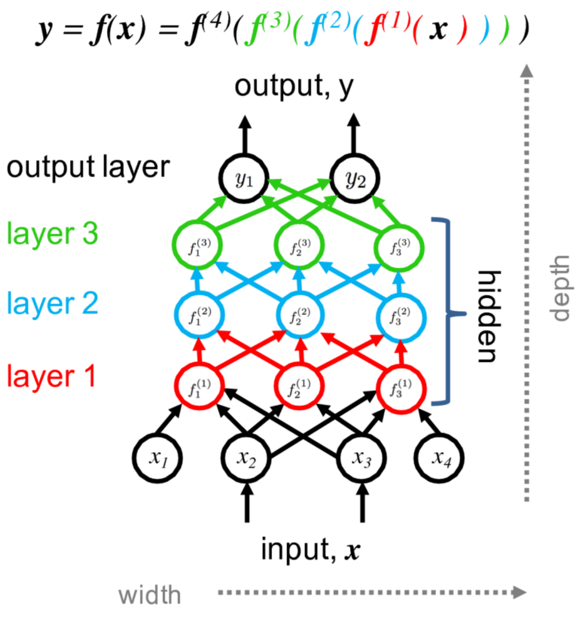


Figure 2.1. A typical neural network, with an input layer, three hidden layers, and an output layer, represented as a graph.

the output of the DNN, which is the classification label of the input image. Specifically, we have [40]:

$$f(x) = f_N\left(\ldots f_2\left(f_1\left(x; W_1, b_1\right); W_2, b_2\right)\ldots; W_N, b_N\right) \tag{2.1}$$

In this equation, W_i and b_i for $i = 1, 2, \ldots, N$ represent the weights and bias of the model, which are trainable parameters. $f_i\left(z_{i-1}; W_{i-1}, b_{i-1}\right)$ is the layer function that maps the output of layer $(i-1)$, i.e., z_{i-1}, to the input layer i, which are mathematical functions that mimic the behavior of biological neurons.

The most common activation functions include ReLU (Rectified Linear Unit) [41], Sigmoid [42], tanh [43], and others [44], whose expressions are shown in Table 2.1.

Table 2.1. Some of the most used fixed activation functions.

Name	**Expression**	**Range**
Identity	$\mathrm{id}(a) = a$	$(-\infty, +\infty)$
Step (Heavyside)	$H_{\geq 0}(a) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{otherwise} \end{cases}$	$\{0, 1\}$
Bipolar	$B(a) = \begin{cases} -1 & \text{if } a < 0 \\ +1 & \text{otherwise} \end{cases}$	$\{-1, 1\}$
Sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$(0, 1)$
Bipolar sigmoid	$\sigma_B(a) = \frac{1-e^{-a}}{1+e^{-a}}$	$(-1, 1)$
Hyperbolic tangent	$\tanh(a)$	$(-1, 1)$
Hard hyperbolic tangent	$\tanh_H(a) = \max(-1, \min(1, a))$	$[-1, 1]$
Absolute value	$\mathrm{abs}(a) = \lvert a \rvert$	$[0, +\infty)$
Cosine	$\cos(a)$	$[-1, 1]$

ReLU is the most widely used [45]. ReLU is a nonlinear activation function that allows neural networks to learn complex features and patterns. Compared to traditional activation functions (such as sigmoid and tanh), ReLU is faster to compute [46]. This is because ReLU takes a threshold on the input and does not involve complex mathematical operations. The existence of nonlinear functions allows neural networks to approximate arbitrary functions [47], which makes the neural network function more complex and, at the same time, increases the difficulty of verification. Because common activation functions are nonlinear, this kind of verification method does not scale in the case of large neural networks and suffers from the state-space explosion. For example, for the piece-wise linear activation function Relu, each Relu node has to be split into two linear constraints, i.e., if $y = \text{relu}(x)$, then $y = 0$ when $x < 0$ and $y = x$ when x is positive. Therefore, solving a verification problem of a network of n Relu nodes leads to solving 2^n linear sub-problems [48].

A Feed-Forward Neural Network (FFNN) is the first and simplest type of artificial neural network where connections between the nodes do not form a cycle. It consists of an input layer, one or more hidden layers, and an output layer. The term "Feed-Forward" refers to how the data moves through the model: it flows in one direction from the input layer, through the hidden layers, and finally to the output layer. Here are some key points about FFNNs:

An example FFNN can be described as follows (notice that FFNN in the real world usually contains thousands of neurons): The input layer receives the data, such as pixel values from an image or feature vectors from a text. These inputs are passed to the first hidden layer, where each node computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer. This process continues through each hidden layer until the output layer generates the final prediction, such as a class label or a numerical value. FFNNs are fundamental models in machine learning. Although they are simple compared to more advanced neural network architectures, understanding FFNNs is essential for grasping the basics of deep learning.

ACAS Xu [49] is a set of neural networks that consider the state of a drone aircraft and information about an "intruder" aircraft, which are determined as sensor measurements. Formally, for an FFNN N, we use n to denote the number of layers and s_i to denote the size of layer I (i.e., the number of its nodes). Layer 1 is the input layer, layer n is the output layer, and layers $2, \ldots, n-1$ are the hidden layers. The value of the $j-th$ node of layer i is denoted $v_{i,j}$ and the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$ is denoted V_i. Evaluating N entails calculating V_n for a given assignment V_1 of the input layer. This is performed by propagating the input values through the network using predefined weights and biases and applying the activation functions – ReLUs, in our case. Each layer $2 \leq i \leq n$ has a weight matrix W_i of size $s_i \times s_{i-1}$ and a bias vector B_i of size s_i, and its values are given by $V_i = \text{ReLU}\left(W_i V_{i-1} + B_i\right)$, with the ReLU function being applied element-wise. This rule is applied repeatedly for each layer until V_n is calculated. When the weight matrices $W_1, \ldots W_n$ do not have any zero entries, the network is said to be fully connected (see Fig. 2.1 for an illustration). An FFNN is said to be piecewise linear if it contains only Piece-Wised Linear (PWL) activation functions.

Definition 2.1.1 (Piece-Wised Linear (PWL) activation functions)**.** A PWL activation function typically has breakpoints or thresholds where the slope of the function changes abruptly. Each segment between two breakpoints is a linear function defined as

$$f(x) = m_i x + c_i \tag{2.2}$$

where m_i is the slope and c_i is the intercept of the i-th segment.

Before we give the key properties of neural networks, let us demonstrate the definition of continuous function [50]:

Definition 2.1.2 (Continuous Function)**.** The function f is continuous at some point c of its domain if the limit of $f(x)$, as x approaches c through the domain of f, exists and is equal to $f(c)$. In mathematical notation, this is written as

$$\lim_{x \to c} f(x) = f(c) \tag{2.3}$$

In detail, this means three conditions: first, f has to be defined at c (guaranteed by the requirement that c is in the domain of f). Second, the limit of that equation has to exist. Third, the value of this limit must equal $f(c)$. Here, we have assumed that the domain of f has no isolated points.

Baker demonstrated one of the key properties of neural networks demonstrated in [51] is their ability to approximate any given function. Loosely speaking, given a function g, we can construct a neural network f such that the distance between g and f is less than a given threshold. The Universal Approximation Theorem formally captures this property.

Definition 2.1.3 (Universal Approximation Theorem)**.** The Universal Approximation Theorem states that let $C\left(X, \mathbb{R}^m\right)$ denote the set of continuous functions from a subset X of a Euclidean $\mathbb{R}^n$ space to a Euclidean space $\mathbb{R}^m$. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$ be a continuous function. Note that $(\sigma \circ x)_i = \sigma\left(x_i\right)$, meaning $\sigma \circ x$ denotes σ applied to each component of x.

Then σ is not polynomial if and only if for every n $\in \mathbb{N}, m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C\left(K, \mathbb{R}^m\right), \varepsilon > 0$ there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon \tag{2.4}$$

where $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$ [52].

This theorem implies that neural networks can be used to approximate arbitrarily complex functions with arbitrary approximation accuracy. Even though it does not specify how to choose the neural network parameters (weights, number of neurons, number of neural layers, etc.) to achieve the desired approximation of the objective function. The theorem explains why neural networks seem to behave so intelligently, and understanding it is a critical step in developing a deep understanding of neural networks. Machine learning researchers decide, based on intuition and experience, how to construct a neural network architecture suitable for a given problem so that it approximates the multidimensional space well, knowing that such a network exists but also weighing computational performance.

For example, ACAS Xu was originally designed offline using dynamic programming and Markov decision processes (MDPs) [53]. This traditional approach involved creating a large rule table that defined the system's behavior in various scenarios.

Then, the ACAS Xu system was enhanced by leveraging neural networks. The large rule table was compressed by a factor of 1000 using a set of neural networks [54]. This transformation significantly improved the system's efficiency and scalability. By encoding the decision-making rules into neural networks, the system reduced its memory footprint and increased its processing speed. Neural networks offer superior generalization capabilities, allowing ACAS Xu to perform robustly in a wider range of scenarios with greater accuracy. Additionally, the neural network-based approach simplifies updates and maintenance, as new data can retrain the network, enabling the system to adapt and improve over time. This demonstrates a clear advantage of neural networks in enhancing the performance and practicality of complex decision-making systems like ACAS Xu.

2.1.1 NN Implementations

Neural networks, inspired by the structure and function of the human brain [55], consist of interconnected layers of nodes (neurons) that work together to process and learn from data. These networks can model complex patterns and relationships within data, making them powerful tools for a wide range of applications.

For example, Recurrent Neural Networks (RNNs) [56] excel in tasks requiring sequential data analysis, such as time series prediction and natural language processing [57], [58]. Another example is Generative Adversarial Networks (GANs), which generate synthetic data and images by pitting two neural networks against each other in a competitive framework [20].

Among the various types of neural networks, Convolutional Neural Networks (CNNs) have emerged as a particularly effective architecture for tasks involving spatial data [59]. A CNN is a specialized form of feed-forward neural network designed to learn features autonomously through optimized filters (kernels). It addresses challenges from earlier neural networks, such as vanishing or exploding gradients during back-propagation, by employing weight regularization across fewer connections [60]. Originally introduced by LeCun [61], CNNs have emerged as pivotal tools in computer vision, profoundly enhancing capabilities in tasks like image classification, object detection, and image segmentation. A CNN typically consists of multiple layers, each performing specific operations on the input data. These layers apply convolutional filters (kernels) to the input data.

Definition 2.1.4 (Convolutional Neural Network (CNN))**.** For an input **x** and a filter **w**, the convolution operation **y** is defined as:

$$\mathbf{y}(i,j) = (\mathbf{x} * \mathbf{w})(i,j) = \sum_m \sum_n \mathbf{x}(i+m, j+n) \cdot \mathbf{w}(m,n)$$

Here, (i,j) denotes the spatial location in the output feature map, and (m,n) indexes the positions within the filter.

After convolution, an activation function such as ReLU is applied to introduce non-linearity into the model. These layers perform down-sampling operations to reduce the spatial dimensions of the feature maps, thereby controlling overfitting and reducing computational load. The most common pooling operation is max pooling, defined as:

$$\mathbf{y}(i,j) = \max_{(m,n)\in\mathcal{P}} \mathbf{x}(i+m, j+n)$$

where $\mathcal{P}$ represents the pooling window.

These layers are typically used at the end of the network to perform high-level reasoning. They connect every neuron in one layer to every neuron in the next layer.

CNNs have revolutionized the field of computer vision and are widely used in various applications, including:

- **Image Classification**: Classifying an image into predefined categories.
- **Object Detection**: Identifying and localizing objects within an image.
- **Image Segmentation**: Partitioning an image into segments or objects.
- **Face Recognition**: Identifying and verifying individuals in images.

In summary, CNNs are a powerful and flexible architecture for image-related tasks, capable of learning complex patterns through the hierarchical extraction of features, making them a fundamental tool in modern deep learning implementations.

Building upon the success of CNNs, researchers have continued to innovate and develop new architectures to address the limitations of deep networks. One such notable advancement is ResNet, short for Residual Network. Introduced by He et al. [62], ResNet is a deep learning architecture primarily designed to facilitate the training of very deep neural networks by addressing the vanishing gradient problem.

A Residual Network (ResNet) is characterized by its use of residual blocks, which implement shortcut (or skip) connections that bypass one or more layers. Formally, a residual block can be described as follows:

Definition 2.1.5 (Residual Network)**.** Let $\mathbf{x}$ be the input to a residual block and $\mathcal{F}(\mathbf{x}, \{W_i\})$ represent the residual mapping to be learned, where $\{W_i\}$ are the weights of the layers within the block. The output of the residual block, $\mathbf{y}$, is given by:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

Here, $\mathbf{x}$ is directly added to the output of the residual function $\mathcal{F}(\mathbf{x}, \{W_i\})$. This addition is element-wise and helps preserve the network's gradient flow during back-propagation, effectively mitigating the vanishing gradient problem.

The main motivation behind ResNet is to make it feasible to train extremely deep neural networks by using residual learning. Traditional deep networks often suffer from the degradation problem, where adding more layers results in higher training errors, which is counter-intuitive. ResNet addresses this by reformulating the layers to learn residual functions concerning the layer inputs rather than unreferenced functions.

The architecture of ResNet includes:

- **Residual Blocks**: These blocks contain a few convolutional layers with skip connections that add the input of the block to its output.
- **Bottleneck Architectures**: For deeper networks, ResNet uses bottleneck layers (e.g., ResNet-50, ResNet-101), which consist of three layers: 1x1, 3x3, and 1x1 convolutions, where the 1x1 layers are responsible for reducing and then restoring dimensions, making the network more efficient.
- **Identity Mapping**: In some versions, identity mappings are used to ensure that the shortcuts do not change the dimension of the input.

ResNet's introduction has enabled the development of models with hundreds or even thousands of layers, significantly advancing the state of the art in various computer vision tasks, including image classification, object detection, and segmentation. The success of ResNet has also influenced the design of numerous subsequent deep-learning architectures.

Building upon the advancements made by architectures like ResNet, which enable the training of very deep networks to achieve high accuracy in image classification and other tasks, MobileNet offers a different approach by focusing on efficiency and deployment on resource-constrained devices.

MobileNet [5] is a family of neural network architectures designed for efficient performance on mobile and embedded devices. Introduced by Google researchers in 2017, MobileNet models aim to balance high accuracy and low computational cost, making them suitable for applications where computational resources and power consumption are limited.

- **Depthwise Separable Convolutions**: MobileNet uses depth-wise separable convolutions, which factorize a standard convolution into a depth-wise convolution followed by a point-wise convolution. This reduces the number of parameters and computational complexity, significantly improving efficiency without compromising accuracy.
- **Width Multiplier**: This hyper-parameter allows the model to be scaled regarding the number of channels in each layer, trading off between model size and accuracy to fit specific resource constraints.
- **Resolution Multiplier**: This hyper-parameter allows the model to be scaled regarding input image resolution, providing another way to balance computational cost and accuracy.
- **Quantization**: MobileNet models support quantization, which reduces the precision of the numbers used to represent model parameters and activations from 32-bit floating point to 16-bit or 8-bit integers. This significantly reduces the model size and increases inference speed, particularly on hardware optimized for lower-precision arithmetic, without greatly sacrificing accuracy.

MobileNet has several versions, including MobileNetV2 [63] and MobileNetV3 [64], each improving on the previous by incorporating advances in neural network architecture design. These models have been widely adopted in applications such as image classification, object detection, and segmentation, particularly in scenario deployment on resource-constrained devices.

These advancements in deep learning models have been further propelled by the use of powerful hardware and efficient computational techniques. One of the key technologies that have driven this progress is CUDA (Compute Unified Device Architecture) [65]. CUDA is crucial in accelerating neural network computations, particularly in frameworks like TensorFlow and PyTorch [16]. These frameworks utilize CUDA to harness the massive parallel processing power of NVIDIA GPUs, enabling faster training and inference of deep learning models.

The Open Neural Network Exchange (ONNX) [66] format is an open-source format for representing machine learning models. It allows developers to transfer trained models between different learning frameworks with ease, making it possible to take advantage of the unique features and strengths of each framework.

2.1.2 NN Quantization

The benefits of quantization extend beyond MobileNet to a wide range of Deep Neural Network (DNN) models. Model quantization can reduce the size and inference time of DNN models and their application to most models and different hardware devices [67]. The model's storage requirements and computational complexity can be significantly optimized by reducing the number of bits per weight and activation. Typically, neural networks utilize floating-point numbers to represent weights and activations, demanding significant memory and computational resources. To address this issue, especially in embedded systems, practitioners often convert these floating-point numbers into fixed-point or integer representations, a process known as quantization. Fixed-point representation, in essence, is a specific form of quantization where the decimal point is fixed at a certain position. Hence, quantized neural networks essentially employ fixed-point numbers to represent the weights and activations of the network. Jacob et al. [68] report benchmark results on popular ARM CPUs for state-of-the-art MobileNet architectures and other tasks, showing significant improvements in the latency-vs-accuracy trade-offs. In the following formula, r is the true floating point value, q is the quantized fixed point value, Z is the quantized fixed point value corresponding to the 0 floating point value, and S is the smallest scale that can be represented after quantization of the fixed point. The formula for quantization from floating point to fixed point is as follows:

$$\begin{gathered} r = S(q - Z) \\ q = \text{round}\left(\tfrac{r}{S} + Z\right) \end{gathered} \tag{2.5}$$

Quantized Neural Networks (QNNs) have the potential to find a wide range of applications in safety-critical systems implemented on low-cost processors that support only integer arithmetic, as well as powerful gas pedals such as GPUs and FPGAs [69]. Fixed-point arithmetic systems are faster, consume less power and memory, and are less expensive because the computations can be performed on low-cost integer-only processors [70]. As noted in previous work, quantization is standard practice for deploying neural networks on real-time embedded devices. While it is well known that networks implemented using fixed-point algorithms have little to no impact on the accuracy of the network, they are not immune to malicious misclassification caused by adversarial attacks, and the verification of real-valued neural networks is insufficient to determine their correctness [71].

Currently, INT8 quantization is a widely used technique in neural networks to convert weights and activation values from 32-bit floating point (FP32) numbers to 8-bit integers (INT8) [67]. This process significantly reduces the model's storage and computational requirements. Specifically:

- Storage Space: Floating-point numbers require more memory compared to integers. An FP32 value takes 4 bytes, while an INT8 value only takes 1 byte. Quantization can reduce the model size by a factor of four.
- Computational Efficiency: Integer operations are typically faster and more energy-efficient than floating-point operations. Hence, quantized models can make more efficient use of hardware resources during inference.

Currently, Google's TensorFlow Lite [72] and nVIDIA's TensorRT [73] support the INT8 engine framework. QNNs decrease memory requirements and computational complexity, leading to faster inference times and lower power consumption while attempting to maintain model accuracy. Hence, they are applied in many areas:

1. **Mobile Computing:** QNNs are extensively used in mobile applications, allowing for real-time image and speech processing directly on smartphones and tablets without the need for constant cloud connectivity [74].
2. **Embedded Systems:** In embedded systems like drones, robotics, and surveillance cameras, QNNs enable sophisticated AI capabilities such as object detection, navigation, and autonomous decision-making with limited hardware resources [75].
3. **Internet of Things:** IoT devices benefit from QNNs through enhanced data analytics and intelligent decision-making capabilities at the edge, improving efficiency and responsiveness while reducing data transmission to the cloud. A study [76] explores the implementation and analysis of CNN-, QNN-, and BNN-based pattern recognition techniques on an FPGA, highlighting their significance for IoT applications. This research emphasizes the efficiency and lower power consumption achievable with QNNs in IoT devices, showcasing the potential transformation in industrial applications through reduced processing latency and power usage.
4. **Automotive:** In the automotive industry, QNNs are applied in advanced driver-assistance systems (ADAS) for features like pedestrian detection, lane tracking, and traffic sign recognition, enhancing safety while optimizing computational resources. As detailed in research that focuses on training quantized models to handle real-world radar data [77] effectively for noise reduction and signal clarity. This application is crucial for enhancing the performance of driver assistance systems and ensuring safer automotive operations.

5. **Wearable Devices:** Wearable health monitoring devices use QNNs to process physiological signals in real-time, providing immediate feedback on user health status and activity levels without heavy battery usage. Recently, a study on "ECG-based real-time arrhythmia monitoring using quantized deep neural networks" [78] showcases the application of quantized neural networks in wearable devices for health monitoring, particularly for ECG (electrocardiogram) analysis.

2.1.3 Adversarial Examples

Adversarial examples [23] represent a class of erroneous behavior. Suppose there is a neural network classifier on ImageNet [79]. We can add noise to a correctly classified panda image and then show the gibbon - at least according to the neural network. Humans still see the same picture of a panda because we can't detect the noise. The critical point is that adversarial aggression is a subjective concept in the following sense. This depends on the inability of humans to distinguish between the original image and the adversarial attack based on it. Thus, an adversarial attack is usually defined as a differently categorized object close to the original object. I give the formal definition of Adversarial example in the following:

Definition 2.1.6. (Adversarial Example) Given a (trained) deep neural network $f : \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$, a human decision oracle $\mathcal{H} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$ and a legitimate input $x \in \mathbb{R}^{s_1}$ with $\arg\max_j f_j(x) = \arg\max_j \mathcal{H}_j(x)$, an adversarial example of DNNs is defined as:

$$\begin{aligned} \exists \hat{x} : \arg\max_j \mathcal{H}_j(\hat{x}) &= \arg\max_j \mathcal{H}_j(x) \\ \wedge \|x - \hat{x}\|_p &\leq d \\ \wedge \arg\max_j f_j(\hat{x}) &\neq \arg\max_j f_j(x) \end{aligned} \tag{2.6}$$

where $p \in \mathbb{N}, p \geq 1, d \in \mathbb{R}$ and $d > 0$.

This proximity implicitly reflects the inability of humans to detect differences. However, adversarial attacks depend on humans consistently declaring that adversarial attacks are misclassified. One of the representative methods, Goodfellow et al. [80], chose a faster method - the Fast Gradient Symbology Method (FGSM) – to project the gradient onto the corners of a hypercube of radius.

2.2 Bounded Model Checking

Model checking is an automatic verification technique designed for large state transition systems, initially developed for finite-state concurrent systems. This method has been successfully applied to debugging complex computer hardware, communication protocols, and software [81]. In essence, Bounded Model Checking (BMC) symbolically executes a program up to a specified bound k and encodes all obtained traces C along with a given safety property P into an SAT/SMT formula, expressed as $C \wedge \neg P$ [82]. An automated theorem prover or solver, often used as a decision procedure, checks this generated formula for satisfiability. If the formula is satisfiable, it indicates a violation of the safety property and generates a witness (counterexample) [83]. Conversely, if the formula is unsatisfiable, it proves the program's safety within the provided bound k. Recently, model checking has also been applied to analyzing cyberphysical, biological, and financial systems [84].

In BMC, a program is modeled as a state Transition System (TS) derived from its control flow graph [85]. It is then transformed into a Static Single Assignment form (SSA). Each control graph node is transformed into an assignment, or a guard is created from a conditional expression. From a conditional expression. Each edge indicates a change in the control position of the program [86].

Kripke structure [87] is utilized as TS $M = (S, T, S_0)$ when modeling the program. A Kripke is an abstract machine consisting of a collection of states S, initial states $S_0 \subseteq S$, and transition relation $T \subseteq S \times S$. The collection of states is given as:

$$S = \{s_0, s_1, \ldots s_n\} : n \in \mathbb{N} \tag{2.7}$$

It includes all the states. Each state contains the values of all program variables and a program counter pc. Each transition is represented by $\gamma(si, si+1) \in T$, where it represents a logical formula encoding all the changes in variables and pc from s_i to s_{i+1}. Then, a Verification Condition (VC) denoted by Ψ is computed.

The major challenge for the technique is a phenomenon called the State Explosion Problem. The State Explosion Problem can consume much time and resources and detract from their primary responsibility [88].

To further conquer the State Explosion Problem, Biere et al. [89] proposed the Bounded Model Checking (BMC) using Boolean Satisfiability (SAT) solvers. The basic idea for BMC is relatively straightforward. Given a finite-state transition system, a temporal logic property, and a bound k (we assume k≥1), BMC generates a propositional logical formula whose satisfiability implies the existence of a counterexample of length k and then passes this formula to an SAT solver. This formula encodes the constraints on initial states, the transition relations for k steps, and the negation of the given property.

BMC helps debug as an efficient method of detecting subtle counterexamples. To prove correctness when a counterexample is not found using BMC, an upper bound on the number of steps to reach all reachable states needs to be determined. It has been shown that a state transfer system's diameter (i.e., the shortest path between any two states) can be used as an upper bound [89]. However, computing the diameter seems computationally difficult when the state transfer system is implicitly given. Other methods for accomplishing BMC are based on induction [90], cubic amplification [91], Craig interpolation [92], and circuit co-factorization [93]. This problem is still a topic of active research.

2.2.1 Bounded Model Checking Tools

Over the past years, Bounded Model Checking has been successfully applied to verify concurrent C programs. There are several state-of-the-art bounded model checkers available in the field, such as ESBMC [94], NuSMV [95], LLBMC [96], JBMC [97] and CBMC [98].

- **CBMC (C Bounded Model Checker):** CBMC is a popular tool for verifying C and C++ programs.
- **NuSMV (New Symbolic Model Verifier):** A symbolic model checker handles both bounded and unbounded model checking.
- **LLBMC (Low-Level Bounded Model Checker):** Focused on analyzing compiled code at a low level.
- **JBMC (Java Bounded Model Checker):** An extension of CBMC designed for Java applications.
- **ESBMC (Efficient SMT-based Context-Bounded Model Checker):** A mature, permissively licensed open-source context-bounded model checker for verifying single- and multithreaded C/C++, CUDA, CHERI, Kotlin, Python, and Solidity programs. It can automatically verify predefined safety properties (e.g., bounds check, pointer safety, overflow) and user-defined program assertions.

ESBMC also implements state-of-the-art incremental BMC and k-induction proof-rule algorithms based on Satisfiability Modulo Theories (SMT) and Constraint Programming (CP) solvers.

2.2.2 Satisfiability Modulo Theories Backends

Satisfiability modulo theory (SMT) [99] solving consists of deciding the satisfiability of a first-order formula with unknowns and relations lying in certain theories. For instance, the following formula has no solution $x, y \in \mathbb{R}$:

$$(x \leq 0 \vee x + y \leq 0) \wedge y \geq 1 \wedge x \geq 1. \tag{2.8}$$

The formula may contain negations ($\neg$), conjunctions ($\wedge$), disjunctions ($\vee$), and possibly quantifiers ($\exists, \forall$). An SMT-solver reports whether a formula is satisfiable and, if so, may provide a model of this satisfaction. For instance, if one omits $x \geq 1$ in the preceding formula, its solutions include $(x = 0, y = 1)$. Other possible features of SMT-solvers include the dynamic addition and retraction of constraints, the production of proofs and Craig interpolants [92], and optimization capabilities. SMT-solving has major applications in formally verifying hardware, software, and control systems.

Bounded Model Checking (BMC) relies on SMT solvers to handle the satisfiability problems generated during the verification process. The SMT solver's ability to deal with specific theories and efficiently solve complex formulas is essential for BMC's effectiveness. This synergy allows for the practical verification of systems, ensuring their correctness up to a specified bound. ESBMC supports a variety of SMT solvers as backends, including Boolector [100], Z3 [101], and Bitwuzla [102].

- Boolector [100] is an SMT (Satisfiability Modulo Theories) solver focused on the theories of bit-vectors, arrays, and uninterpreted functions. Boolector is known for its efficiency and performance in handling bit-precise problems, making it suitable for hardware and software verification tasks. It uses techniques, including pre-processing, rewriting, and bounded model checking, to solve complex SMT problems effectively.
- Z3 [101] is a high-performance SMT solver developed by Microsoft Research. It supports many theories, including bit-vectors, arrays, arithmetic, and datatypes. Z3 is designed to be highly versatile and is used in various domains such as formal verification, program analysis, and constraint solving. Its robust API and integration capabilities make it a popular choice for both academic research and industrial applications. Z3's development focuses on providing an efficient and scalable solution for a broad spectrum of SMT problems.
- Bitwuzla [102] is a modern SMT solver specifically tailored for bit-vectors and arrays. It is a successor to Boolector, developed by the same group led by Armin Biere. Bitwuzla aims to enhance the performance and capabilities of Boolector by incorporating advanced techniques such as SAT solving, bit-blasting, and optimization strategies. The solver is designed to handle complex bit-vector problems efficiently, making it suitable for formal methods, hardware verification, and software analysis tasks. Bitwuzla continues the legacy of Boolector while pushing the boundaries of what can be achieved in SMT solving for bit-precise domains.

This paper mainly uses the Boolector integrated in ESBMC as the SMT solver backend.

Still, we also compare the performance of different solvers for the same verification problem; see Section 4.3.3.

2.3 Neural Network Verification

Neural network verification is the process of formally ensuring that a neural network meets certain specifications or properties. The goal is to provide guarantees about the network's behavior under various conditions and to identify potential failure modes.

2.3.1 Verification Problem Statement

Neural network verification is inspired by formal methods developed for general-purpose software verification [103] and essentially aims at proving that a given neural network works as expected. This means not only defending against adversarial attacks but also things like denoising or correcting aircraft steering [49].

Moreover, at the technical level, it is not only robustness against adversarial attacks that can be verified. We note that adversarial attacks are caused by the very unstable behavior of neural networks. Therefore, the Lipschitz continuity [104] [49] [105] is explored and verified since it limits the amount by which the neural network output varies with the input. In addition, security properties can be encoded by so-called reachability analysis [106], which asks what the potential outputs are for a given set of inputs. Sometimes, verification procedures are embedded in other verification procedures. For example, one can use them to restrict the values of particularly important ReLU neurons, thus refining the analysis of the actual neural network. In addition, one can use smaller, more specialized neural networks within larger ones and validate the controllers, assuming that the smaller neural networks work properly [107].

A severe verification caveat for programs is that they lack the scalability of large-scale neural networks. Since the neural network verification problem is NP-Complete [108], it becomes computationally intractable. However, the need to validate neural networks is certainly not limited to small-scale neural networks. As a result, considerable effort has been put into speeding up the verification procedure, resulting in different branches of research. This has further contributed to the Verification of Neural Network Competition (VNN-COMP) [109] rewards the number of verification instances that can be solved correctly. This again highlights that scalability is seen as one of the main challenges in neural network verification. Below, we briefly discuss the main approaches currently used and provide examples for each idea to speed up the verification process

Formal verification proves or disproves that a system meets certain formal specifications and properties. A verification problem is defined as:

$$M \vDash P \,? \tag{2.9}$$

This is equivalent to answering the question: Does the system model M satisfy the property P? Depending on the verification technique, the system has to be modeled (e.g., a state transition model), and the specifications need to be expressed respecting some specific syntax (e.g., temporal logic).

A verification technique aims to prove that P holds on M or generate a counterexample witnessing the violation of P. Many verification techniques, such as model-checking, SAT/SMT, abstract interpretation, and theorem proving, have been broadly and successfully applied to verify software-intensive systems [110].

2.3.2 Verification of Safety Properties

Exploring new verification methods for DNNs highlights a broader trend in formal verification aimed at enhancing the reliability and safety of complex systems. As we transition from specific solutions like those for fixed-point DNNs to more general verification challenges, it becomes essential to focus on the foundational aspects of verification, such as safety properties. Safety properties ensure the systems behave within their expected parameters under all conditions, thus preventing any hazardous outcomes. Defining and verifying such properties becomes indispensable in the context of neural networks, particularly those utilized in safety-critical applications. This necessity brings us to the broader domain of software verification, where safety properties are tailored based on deep domain knowledge and the specific behaviors that need to be guaranteed safe.

In general, a safety property defines a condition that must not be violated as such an event will eventually lead to undesired behavior. In traditional software verification, such properties are usually defined by users according to their domain knowledge and actual requirements, which allows him/her to state which program behaviors are safe.

Assuming that the network only contains ReLU activations between each layer, the satisfiability problem to find a counterexample can be expressed as:

$$\mathbf{l}_0 \leq \mathbf{x}_0 \leq \mathbf{u}_0 \tag{2.10}$$

$$\hat{\mathbf{x}}_n \leq 0 \tag{2.11}$$

$$\hat{\mathbf{x}}_{i+1} = \mathbf{W}_{i+1}\mathbf{x}_i + \mathbf{b}_{i+1} \quad \forall i \in \{0, n-1\} \tag{2.12}$$

$$\mathbf{x}_i = \max(\hat{\mathbf{x}}_i, 0) \quad \forall i \in \{1, n-1\}. \tag{2.13}$$

Eq. (2.10) represents the constraints on the input, and Eq. (2.11) on the neural network output. Eq. (2.12) encodes the linear layers of the network and Eq. (2.13) the ReLU activation functions. If an assignment to all of the values can be found, this represents a counterexample. If this problem is unsatisfiable, no counterexample can exist, implying that the property is True. We emphasize that we must prove that no counter-examples can exist and not simply that none could be found [111].

Network Functionality. The ACAS Xu system maps input variables to action advisories. Each advisory is assigned a score, with the lowest score corresponding to the best action. The input state is composed of dimensions: ρ, $theta$, psi, v_{own} , v_{int} (shown in Figure 2.2), which represent information determined from sensor measurements (shown in Table 2.2) [53].

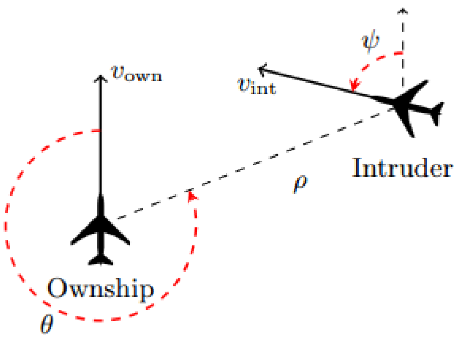


Figure 2.2. Geometry for ACAS Xu Horizontal Logic Table.

Table 2.2. Input state variables used in ACAS Xu.

Variable	Units	Description	Tables	NN
ρ	ft	distance between ownship and intruder	Y	Y
θ	rad	angle to intruder w.r.t ownship heading	Y	Y
ψ	rad	heading of intruder w.r.t. ownship heading	Y	Y
v_{own}	ft / s	velocity of ownship	Y	Y
v_{int}	ft / s	velocity of intruder	Y	Y
τ	s	time until loss of vertical separation	Y	N
a_{prev}	deg / s	previous advisory	Y	N

Each network is denoted $N_{\gamma,\beta}$, where γ corresponds to the index (1 to 5) of a specific value of previous advisory $a_{\text{prev}} \in \{COC, WL, WR, SL, SR\}$ and Developed in [54] and evaluated in [112], 45 separate neural networks were used to compress the lookup table. Each network is denoted $N_{\gamma,\beta}$, where γ corresponds to the index (1 to 5) of a specific value of previous advisory $a_{\text{prev}} \in \{\text{COC}, \text{WL}, \text{WR}, \text{SL}, \text{SR}\}$ and β corresponds to the index (1 to 9) of a specific value of time to loss of vertical separation $\tau \in \{0, 1, 5, 10, 20, 40, 60, 80, 100\}$ seconds. For example, $N_{2,3}$ corresponds to a neural network in which $a_{\text{prev}} = \text{WL}$ and $\tau = 5$. Then each of these networks receives inputs for the remaining five state variables (ρ, θ, ψ, v_{own}, and v_{int}) and outputs a value associated with each of the five output variables ($\{\text{COC}, \text{WL}, \text{WR}, \text{SL}, \text{SR}\}$), see Table 2.3. Several architectures and optimizers were considered and analyzed for the training process.

β corresponds to the index (1 to 9) of a specific value of time to loss of vertical separation $\tau \in 0, 1, 5, 10, 20, 40, 60, 80, 100\}$ seconds. For example, $N_{2,3}$ corresponds to a neural network in which $a_{\text{prev}} = WL$ and $\tau = 5$. Then each of these networks receives inputs for the remaining five state variables $(\rho, \theta, \psi, v_{own}$, and v_{int}) and outputs a value associated with each of the five output variables $(\{COC, WL, WR, SL, SR\})$. Several architectures and optimizers were considered and analyzed to train all 45 neural networks. AdaMax [113], a variant of Adam, was chosen as it proved to learn the fastest without getting stuck in local optima. As for the layered architecture, six hidden layers of 50 neurons each yielded the best results while maintaining an efficient computation time [54]. Hence, all the neural networks have five inputs, five outputs, and six hidden layers of 50 neurons each, as depicted in Figure 2.3.

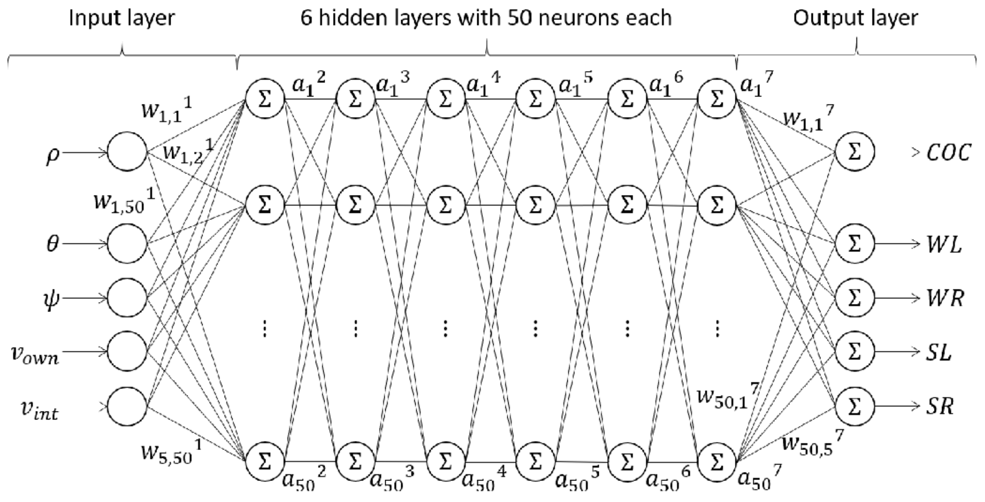


Figure 2.3. Depiction of the ACAS Xu neural network.

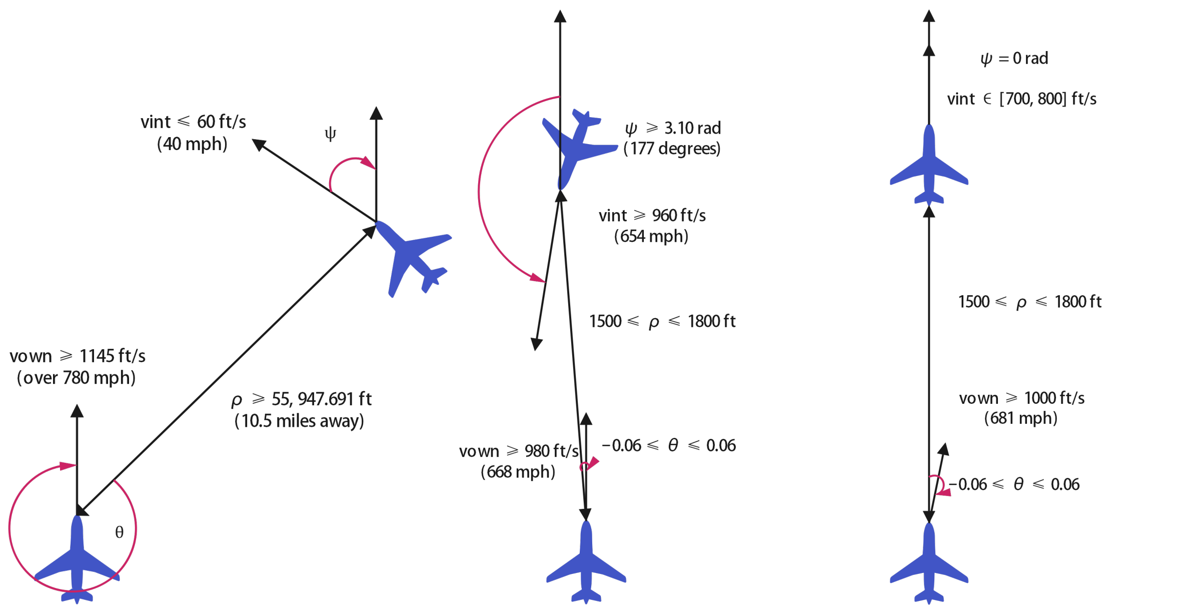
Table 2.3. ACAS Xu Actions (Horizontal Collision Avoidance).

Action	**Description**
SL	strong left at 3.0 deg/s
WL	weak left at 1.5 deg/s
COC	clear of conflict (do nothing)
WR	weak right turn at 1.5 deg/s
SR	strong right turn at 3.0 deg/s

We use $N_{x,y}$ to denote the network trained for the $x - th$ value of a_{prev} and $y - th$ value of τ. For example, $N_{2,3}$ is the network trained for the case where a_{prev} = weak left and $\tau = 5$. Using this notation, we now formally define each of the properties $\phi_1, \ldots, \phi_{10}$ that we tested. All 10 properties for ACAS Xu benchmarks are summarized in Table 2.4.

Table 2.4. ACAS Xu Benchmark Open Loop Properties.

ρ **(ft)**	θ **(rad)**	ψ **(rad)**	v_{own} **(ft/s)**	v_{int} **(ft/s)**	**Networks**	**Output**
φ_1	≥ 55948	$\times$	≥ 1145	≤ 60	All	COC $\leq$ 1500
φ_2	≥ 55948	$\times$	≥ 1145	≤ 60	$N_{x\geq 2,y}$	COC not max
φ_3	$\in [1500, 1800]$	$\in [-0.06, 0.06]$	≥ 3.10	≥ 980	$\neq N_{1,y\geq 7}$	COC not min
φ_4	$\in [1500, 1800]$	0	≥ 1000	$\in [700, 800]$	$\neq N_{1,y\geq 7}$	COC not min
φ_5	$\in [250, 500]$	$\in [0.2, 0.4]$	$\approx \pi$	$\in [100, 400]$	$N_{1,1}$	SR min
φ_6	$\in [12000, 62000]$	$\in [0.7, \pi] \vee \in [-\pi, -0.7]$	$\approx \pi$	$\in [100, 1200]$	$N_{1,1}$	COC min
φ_7	$\in [0, 60760]$	$\in [-\pi, \pi]$	$\in [100, 1200]$	$\in [0, 1200]$	$N_{1,9}$	SL, SR min
φ_8	$\in [0, 60760]$	$\in [-\pi, -0.75\pi]$	$\in [600, 1200]$	$\in [600, 1200]$	$N_{2,9}$	WL $\vee$ COC
φ_9	$\in [2000, 7000]$	$\in [-0.4, -0.14]$	$\approx -\pi$	$\in [100, 150]$	$N_{3,3}$	SR min
φ_{10}	$\in [36000, 60760]$	$\in [0.7, \pi]$	$\approx -\pi$	$\in [900, 1200]$	$N_{4,5}$	COC min



(a) Property φ_1 and φ_2 (b) Property φ_3 (head-on collision) (c) Property φ_4 (closing collision)

Figure 2.4. Illustration of properties φ_1, φ_3, and φ_4. Angles and distances are not drawn to true scale for clarity of the image.

In particular, as shown in Figure 2.4, properties 1-4 are described in detail, which are used in the later experimental chapters. Property φ_1 selects a distance ρ of approximately 10.5 miles (55,947 feet), exceeding the maximum turning radius of an aircraft flying at 1145 ft/s (43,736 feet). The maximum velocity of 1145 ft/s (approximately 780 mph) is a conservative upper bound that is significantly higher than typical commercial cruise speeds, which are closer to 678 mph at 30,000 feet altitude. Additionally, the intruder's velocity is set at 60 ft/s (40 mph), representing a conservative lower limit for fixed-wing aircraft. While these variables seem reasonable, φ_1 has limitations as it reflects the design constraints of the neural network rather than prioritizing the ranking of optimal actions to avoid conflicts. A better property would focus on ensuring that the network produces clear conflict-avoidance maneuvers. In contrast, φ_2 refines φ_1 by addressing a subset of its conditions and evaluating situations where avoiding conflict is not the last resort, making φ_2 more specific and potentially more effective in ensuring robust neural network performance. Together, these properties highlight the need for precise metrics to evaluate aircraft encounter scenarios effectively.

Properties φ_3 and φ_4 check that the neural network does not output clear of conflict in cases where the intruder is ahead of the ownship and a collision is imminent. Property φ_3 evaluates scenarios where the two aircraft are approaching a head-on collision within less than a second, assuming insufficient vertical separation, and requires the neural network to recommend immediate conflict-avoidance maneuvers to prevent the collision. For instance, if the relative velocity exceeds a certain threshold (e.g., 1000 ft/s), the network must trigger an action without delay. Property φ_4, on the other hand, examines situations where the ownship is directly behind the intruder and closing in at a relative speed of 200-300 ft/s, predicting a collision within 9 seconds unless evasive action is taken. This property ensures that the neural network accurately identifies the intruder's trajectory and provides optimal solutions, such as adjusting speed or trajectory, to avoid collision. Together, these properties assess the robustness of the neural network in handling critical and time-sensitive collision scenarios, emphasizing the importance of clear and immediate conflict resolution to maintain air traffic safety.

2.4 Neural Network Repair

Neural network repair is the process of modifying a neural network to correct undesirable behaviors or improve its performance on specific tasks without compromising its overall functionality. This is particularly important in applications where the neural network must meet strict safety, robustness, or fairness criteria, such as in autonomous driving, healthcare, and finance.

2.4.1 Repair Problem Statement

Similar to software testing, neural network testing evaluates the behavior of neural networks through a large number of test cases to determine the existence of misbehavior, and significant progress has been made in recent years in the areas of test coverage criteria [114] and use case generation [115] [116]. However, both neural network validation and neural network testing are post hoc analyses, i.e., they can only provide qualitative or quantitative conclusions on the presence or absence of erroneous behaviors. When a neural network does not satisfy a particular property, the results of neural network validation or testing cannot alleviate these violations. On the contrary, deep neural network training methods based on property satisfaction, which leave aside pre-trained models and focus on building and training a neural network from scratch that satisfies a specific property (the degree of satisfaction can be evaluated by neural network validation or testing), have gained a lot of attention and made significant breakthroughs, especially in deep neural network training methods based on robustness [117]. Whether it is neural network validation, neural network testing, or neural network training based on property satisfaction, all of them fail when faced with a neural network model with erroneous behaviors, i.e., they are unable to do anything about the erroneous behaviors that the neural network model has already exhibited. To solve this problem, Neural Network Repair (DNN repair) [118] has emerged, i.e., when a neural network is found to exhibit erroneous behaviors (achieved through neural network verification or testing techniques), the neural network repair aims to eliminate these erroneous behaviors, so that the repaired neural network meets the specific properties of the statute while minimizing the impact on the performance of the network model.

Definition 2.4.1 (Neural Network Repair)**.** Given a neural network f and a set of properties $\mathcal{P}$ that f should satisfy, the repair process starts by identifying instances where f violates $\mathcal{P}$. These instances are formally defined as a set of counterexamples $\mathcal{C}$:

$$\mathcal{C} = \{(x, y) \mid f(x) \notin \mathcal{P}(y)\}$$

where x is an input, y is the expected output, and $\mathcal{P}(y)$ represents the set of acceptable outputs for the input x.

Once the problematic instances are identified, the next step is to define the objectives of the repair process.

The objective of the repair process is to find a new neural network f' that minimally deviates from f but satisfies the properties $\mathcal{P}$ for all x:

$$\forall (x, y) \in \mathcal{C}, \quad f'(x) \in \mathcal{P}(y)$$

Additionally, f' should retain the performance of f on non-counterexample inputs as much as possible:

$$\forall (x, y) \notin \mathcal{C}, \quad f'(x) \approx f(x)$$

Several techniques can be employed to achieve these repair objectives. These techniques focus on different aspects of the network and the repair process, ensuring that the modified network meets the desired properties while maintaining its overall functionality.

- **Retraining**: Adjusting the network weights by fine-tuning on a modified dataset $\mathcal{D}_{\text{repair}}$ that includes counterexamples and possibly additional data to reinforce desired properties:

$$f' = \text{train}(f, \mathcal{D}_{\text{repair}})$$

- **Layer Adjustment**: Modifying the weights W and biases b of specific layers to correct the outputs for counterexamples:

$$W', b' = \text{adjust}(W, b, \mathcal{C})$$

- **Constraint-based Methods**: Introducing constraints $\mathcal{C}_{\text{constraints}}$ during the training process to enforce properties $\mathcal{P}$:

$$\text{minimize} \quad \text{loss}(f'(x), y) \quad \text{subject to} \quad \mathcal{C}_{\text{constraints}}$$

After applying these repair techniques, it is crucial to evaluate the repaired network to ensure that it meets the desired properties and performs well on a test set.

The repaired network f' must be evaluated to ensure it meets the desired properties $\mathcal{P}$ and performs well on a general test set $\mathcal{D}_{\text{test}}$. Formally, this involves:

- **Checking property satisfaction**:

$$\forall (x, y) \in \mathcal{D}_{\text{test}}, \quad f'(x) \in \mathcal{P}(y)$$

- **Assessing performance metrics** (e.g., accuracy, precision) to ensure f' performs adequately on $\mathcal{D}_{\text{test}}$:

$$\text{performance}(f', \mathcal{D}_{\text{test}}) \approx \text{performance}(f, \mathcal{D}_{\text{test}})$$

Through this systematic process, neural network repair aims to enhance neural networks' robustness and reliability, ensuring they perform as expected across a wide range of inputs.

Previous subsections have introduced the common definition of neural network repair. Specifically, because of the different nature of neural network protocols and their description types, more specific neural network repair problems have emerged in recent years. Regarding the nature of the network to be repaired, neural network repair problems can be categorized into sample-wise DNN repair problem (SRP) and Domain-Wise DNN Repair Problem (DWNRP). Sample-wise Repair Problem (SRP) and Domain-wise DNN Repair Problem (DRP) are defined as follows:

Definition 2.4.2 (The Sample-wise Repair Problem (SRP))**.** Given a neural network to be repaired and the set of properties $\{\mathcal{P}_i\}_{i=1}^p$ that it violates, i.e., $N_\theta \not\models \{(X_i; Y_i; \phi_i)\}_{i=1}^p$ the set of inputs X and outputs Y of each property $\mathcal{P}_i$ describes the set of sample points in the corresponding space. The neural network sample repair problem refers to the problem of modifying the parameter set θ to θ', by some repair means, to obtain a repaired neural network $N_{\theta'}$ that satisfies the set of constraint properties described in the form of sample points, i.e., $N_{\theta'} \models \{(X_i; Y_i; \phi_i)\}_{i=1}^p$.

Definition 2.4.3 (Domain-wise repair problem (DRP))**.** Given a neural network to be repaired and the set of normative properties, it violates $\{\mathcal{P}_i\}_{i=1}^p$, i.e., $N_\theta \not\models \{(X_i; Y_i; \phi_i)\}_{i=1}^p$ the set of inputs X and outputs Y. Each property $\mathcal{P}_i$ describes a region (e.g., interval, polyhedron, etc.) in the corresponding space. The neural network region repair problem refers to the problem of repairing a neural network by modifying the parameter set θ to θ' to obtain a neural network that satisfies these criteria. The repaired neural network satisfies the set of constraints described in the form of regions, i.e.,$N_{\theta'} \vDash \{(X_i; Y_i; \phi_i)\}_{i=1}^{p}$ [119].

This thesis mainly considers the sample-wise repair problem.

Definition 2.4.4 (DNN repair/patching problem)**.** Given an initial neural network N_θ to be repaired and the set of properties $\{\mathcal{P}_i\}_{i=1}^{p}$ that it violates, the following problem can be solved. The neural network repair problem refers to the problem of modifying the parameter set θ to θ', by some repair means, get the repaired neural network $N_{\theta'}$ to satisfy the set of constraint properties, i.e., $N_{\theta'} \vDash \{(X_i; Y_i; \phi_i)\}_{i=1}^{p}$

Many researchers have proposed their full-precision neural network repair techniques. These can be divided into three categories: Retraining, Direct Weight Modification, and Attaching Repair Units.

The requirements for neural network models can be summarized below:

- The repaired neural network can satisfy the properties of the statute violated by the original neural network N and correctly process the input samples or regions that cannot be correctly processed by N.
- For those input samples (regions) that are different from those to be repaired, the repair strategy can not change the behavior of the original neural network on these samples (regions) as much as possible, i.e., the repair neural network can perform as close as the original neural network.
- The repaired neural network N has to maintain the performance of the original neural network N as much as possible. When a performance loss is incurred, the repair strategy must minimize this performance degradation. Of course, if the repair results in an increase in the performance of the neural network, that is even better! Whether the repair strategy can support higher dimensional descriptions of the samples or regions to be repaired, e.g., the dimensionality of polyhedra, etc. Whether the repair strategy can support different neural network units, e.g., convolutional neural network, RNN, etc.
- The repair strategy should be computationally efficient, as evidenced by the fact that the repair strategy can accomplish the given neural network repair task in a determined amount of time and can run on hardware at the current state of the art.

2.4.2 Mixed Integer Linear Optimization

The repair strategies for neural networks, as discussed, need to be computationally efficient to be feasible on current hardware. Mixed Integer Linear Programming (MILP), with its ability to handle discrete decision variables and complex constraints in a linear framework, offers a robust methodology for achieving such efficiency. By formulating the neural network repair problem as a MILP, it becomes possible to efficiently explore and optimize the vast solution space, even under stringent operational constraints. This connection highlights the practical utility of MILP in ensuring that neural network repair strategies not only meet theoretical performance benchmarks but are also implementable with the computational resources available today. This sets the stage for a deeper examination of MILP's features and application in the subsequent section.

MILP (Mixed Integer Linear Programming) extends linear programming in which some or all decision variables are restricted to integers. In this type of problem, the objective function and all constraints are linear, but the solution space becomes discrete due to integer constraints, making the problem more complex and challenging.

Definition 2.4.5 (MILP (Mixed Integer Linear Programming))**.** Minimize (or Maximize) the objective function:

$$c^T x + d^T y \tag{2.14}$$

Subject to linear constraints:

$$\begin{aligned} &Ax + By \leq b \\ &x \geq 0, y \geq 0 \end{aligned} \tag{2.15}$$

Here, x represents a vector of decision variables that must take integer values;y represents a vector of decision variables that can take real values; and A, B, c, d, and b are matrices and vectors of problem parameters.

The difficulty in solving MILP problems primarily stems from the integer constraints, which make the feasible solution space highly complex.

All state-of-the-art solvers for MILP employ one of many existing variants of the well-known branch-and-bound algorithm of [120]. This class of algorithm searches a dynamically constructed tree (known as the search tree). The state-of-the-art MILP solvers include Gurobi [121], a commercial solver widely used for linear programming, integer programming, and mixed integer linear programming. According to B. Meindl and M. Templ's [122] Analysis of commercial and free and open source solvers for linear optimization problems, Gurobi is the fastest solver and can solve most problems. Another reason for choosing Gurobi was primarily in the area of neural network robustness, and other approaches, such as alpha-beta-crown in the area of neural network verification, use Gurobi as their backend. Hence, we use Gurobi as the backend to solve the neural network repair problem.

Other MILP solver include: CPLEX [123], GLPK (GNU Linear Programming Kit) [124]. Python external library Scipy [125] also provides some functions for MILP.

- **CPLEX:** CPLEX is a high-performance mathematical programming solver developed by IBM. It is widely used for solving large-scale Linear Programming (LP), Mixed-Integer Programming (MIP), and Quadratic Programming (QP) problems.
- **GLPK (GNU Linear Programming Kit):** GLPK is an open-source solver for Linear Programming (LP) and Mixed-Integer Programming (MIP) problems. It is part of the GNU Project and is known for its flexibility and ease of use.
- **Scipy:** Scipy is a Python library for scientific and technical computing. It builds on NumPy and provides a wide range of functions for optimization, integration, interpolation, eigenvalue problems, and other scientific computations.

QNNRepair is a neural network repair tool with a Gurobi backend that converts neural network repair problems into linear programming problems.

2.5 Chapter Summary

This chapter delves into the foundational elements of neural networks, highlighting their practical implementations and addressing the critical areas of neural network verification and repair. We provide a comprehensive overview of cutting-edge techniques in the field while also drawing parallels with existing concepts.

This chapter begins by elucidating neural network fundamentals. Recognizing the significant deployment of neural networks within the embedded systems domain, we discuss quantized neural networks, which serve as a cornerstone for our subsequent analyses of verification and repair processes. This is complemented by examining defense strategies against emerging threats that compromise neural network robustness, setting the stage for exploring verification methodologies with an emphasis on bounded model checking.

Subsequent sections present a thorough review of contemporary neural network verification strategies, categorically divided into four main approaches: Exact Verification, Approximate Verification, Verification through Satisfaction Theory, and Verification via mixed-integer linear programming. Our discussion extends to the verification of neural network reachability and quantization effects, spotlighting areas of conceptual overlap.

The concluding segments of this chapter shift focus to the realm of neural network repair. We navigate through prevalent neural network repair methodologies, such as retraining, direct adjustment of weights, integration of repair units, and counterexample-guided repair mechanisms, with a particular consideration for quantization nuances.

Through this literature review, we gain a holistic understanding of the current landscape in neural network research, encompassing both theoretical underpinnings and practical applications. This exploration enriches our knowledge base and illuminates the inter-connectedness of verification and repair strategies, underscoring the importance of continuous innovation and adaptation in the face of evolving technological challenges and threats.

Chapter 3

Related Works

In this chapter, we will provide an overview of related work in neural network verification and repair. For neural network verification, we classify existing methods into two categories based on two criteria: whether approximation techniques are employed to reduce verification complexity and the type of problem-solving approach used (e.g., SMT-based methods or MILP-based methods). For neural network repair, we categorize current approaches into four main groups: Retraining, Adjusting Weight, Attaching Repair Units, and Counter-example guided repairing. Additionally, we explore the impact of quantization operations on both verification and repair processes. Through this discussion, we aim to highlight the state-of-the-art methodologies and their respective strengths and limitations in neural network verification and repair, providing a foundation for the subsequent contributions of this thesis.

3.1 Verification Methods

Verification methods for neural networks can be categorized into full neural network verification and approximate neural network verification based on completeness. Depending on the back-end of the problem solving, they can be categorized as SMT-based using an SMT solver such as Z3 and LP-based and MILP-based using a linear programming solver such as Gurobi, discussed below.

3.1.1 Exact Verification

Most neural network verification problems can be solved by reducing them to constraint-solving problems in first-order logic. Formulas in first-order logic consist of constants, variables, function and predicate symbols, logical connectives, and quantifiers. This thesis needs only one particular type of first-order logic, namely the quantifier-free fragment of Linear Rational Arithmetic (LRA) [126].

Definition 3.1.1 (Linear Rational Arithmetic)**.** The syntax of linear rational arithmetic (LRA) is as follows. The set of terms is defined by the following grammar

$$s, t \in \text{ Term } ::= c|x|s + t \mid c \cdot t \tag{3.1}$$

where x is a variable symbol and c is a rational number.

For example, first, let $\mathcal{X} = \{x_0, x_1, \ldots\}$ be a set of variables which range over values in $\mathbb{R}$. Then, we define terms as follows: a term is either a constant $c \in \mathbb{R}$, a variable $x \in \mathcal{X}$, or a function application $t_1 \circ t_2$, where $\circ \in \{+, \cdot\}$ and t_1, t_2 are two terms. For instance, 5, x , and $3 \cdot x + 2 \cdot y$ are terms. To reflect the usual notation, we often drop the multiplication sign.

- **SMT Solvers:** Two SMT solvers, Reluplex [112] and Planet [127], have been proposed for verifying properties of DNNs expressible with SMT constraints. SMT solvers typically perform well on problems expressed as Boolean combinations of constraints on other variable types. SMT solvers typically combine SAT solvers with specialized decision procedures for other theories.
- **SAT Method:** Narodytska [128], [129] propose to verify the properties of a class of neural networks (i.e., binary neural networks where both weights and activations are binary) by reducing to the well-known Boolean satisfiability. Using this Boolean encoding, they utilize the power of modern SAT solvers and the proposed counterexample-guided search procedure to verify various properties of these networks. One of the focuses is on the robustness against adversarial perturbations. Experimental results show this approach applies to medium-sized deep neural networks used in image classification tasks.

The advantages of exact neural networks are that they are complete, verifiable, and give reliable counterexamples. The disadvantages are that they take a long time and use many computation resources. They are especially CPU intensive and cannot be accelerated using GPUs, which are widely used in approximate verification. And they are not scalable for large networks.

3.1.2 Approximate Verification

Initially, the robustness problem involved small perturbations in the data to deal with measurement errors and other factors contributing to noise in the input data [127]. It quickly became a security issue as more and more networks began to be deployed in safety-critical applications.

Recent advances in formal verification methods such as Satisfiability Modulo Theory (SMT) [130] and Mixed Integer Linear Programming (MILP) [131] have made it possible to use general-purpose solvers applied to neural network verification problems. However, these solvers are too slow to verify any real network [132] [11] [133]. Therefore, additional DNN-level inference is needed to make any verification procedure scale.

Approximate Verification uses a variety of approximation or abstract interpretation methods to simplify the verification process of neural networks; the most common simplification method is to abstract the activation function in the neural network, which includes segmented linear approximation, Zonotope, etc., which is described below.

The representative segmented linear approximation is β-CROWN developed by Wang et al., [134], a new bound propagation-based method that can fully encode neuron splits via optimizable parameters constructed from either primal or dual space. It solves an optimization problem equivalent to the expensive LP-based methods with neuron split constraints while still enjoying the efficiency of bound propagation methods.

As described by [135], a Zonotope is a specific kind of convex polytope. What makes it unique is its ability to be represented as the sum of line segments via a mathematical operation called Minkowski sum. This characteristic makes Zonotopes useful for expressing affine transformations and approximating sets in mathematical contexts.

When we use a Zonotope to approximate a function such as ReLU (Rectified Linear Unit), we aim to encompass all the potential values the function might output within a defined range of inputs. In simpler terms, we're trying to create a shape that covers all the possible outcomes of the function within a specific input range.

The most famous use of Zonotope and achievement on VNN-COMP [109] is AI^2, an approximate verification technique proposed in [11] for authenticating security and robustness properties using abstract interpretation.

Abstract interpretation is a static program analysis technique that can infer the state a program may reach during execution. In the context of neural networks, abstract interpretation is often used to estimate the range of outputs or the behavior of a neural network given inputs.

Due to the use of abstractions, it is a sound but incomplete approach, as verification queries are performed on an abstraction of the neural network, and the abstraction may contain points that do not appear in any concrete implementation. The concept was introduced by the seminal work of Cousot et al. [136], who articulated the theoretical foundations of abstract interpretation in their paper. Their work outlined how abstract interpretation could be used to perform static program analysis to infer program properties. Since then, this theoretical framework has been widely used in program analysis and verification, including verification of neural networks.

Tran et al. [137] discovered that in terms of prediction, the approximate star method is more robust than the abstract domain and Zonotope. Interestingly, the approximate star method is much faster than the abstract domain method for LeakyReLU networks.

A notable contribution to the field is POPQORN [138]. It finds robustness certificates for RNN-based networks that utilize the 2D plane to limit cross-nonlinearities in Long Short-Term Memory (LSTM) networks so that certificates within the L^p sphere can be found if the lower bound on the true value. The L^p sphere is a concept in mathematics, particularly in functional analysis and geometry. It is defined in the context of L^p spaces, which are function spaces characterized by the integrability of the p-th power of the absolute value of a function.

Definition 3.1.2 (L^p sphere)**.** The L^p sphere consists of all elements in an L^p space that have a norm equal to one. Mathematically, for a given $p \geq 1$, the L^p sphere in an L^p space $L^p(X, \mu)$ is the set

$$\{f \in L^p(X, \mu) : \|f\|_p = 1\},$$

where $\|f\|_p$ denotes the L^p norm of f. The concept generalizes the idea of the unit sphere in Euclidean space to more abstract function spaces.

The labeled output cell is greater than the upper bound of all other output cells. Later, Cert-RNN [139] introduced a powerful RNN authentication framework that overcomes the limitations of POPQORN [138]. This framework maintains the correlation between variables and accelerates the evaluation of RNN nonlinearities for practical use. This work utilizes Zonotopes [140] to encapsulate input perturbations. The Cert-RNN verifies the properties of the output Zonotope to determine provable robustness.

However, these approaches may fail to provide any conclusion on the original network when the property is violated on the abstract model. This is, in fact, due to spurious counterexamples. Namely, when the property does not hold, a Counter-Example (CE) on the abstract model is generated. Still, due to the over-approximation of the abstract model, this CE might not correspond to any real behavior in the original model (i.e., spurious counterexample) [48].

Sometimes, the abstraction of neural networks is too coarse to prove the desired properties. Researchers apply the Counterexample-Guided Abstraction Refinement (CEGAR) principle [141]. The CEGAR loop is guaranteed to terminate as it makes at least one refinement per iteration, and there are only finitely many (though exponential) exact polytopes. This ensures that the procedure eventually converges to a solution, verifying the property or providing a counterexample. The iterative refinement process helps in systematically narrowing down the ambiguous regions, improving the accuracy of the verification and reducing the computational complexity in each iteration. The effectiveness of the CEGAR approach makes it a powerful tool in the formal verification of neural networks, especially in ensuring their robustness and reliability in critical applications.

Another technique to avoid an explosion in the number of polytopes is to find polytopes that represent a safe inductive invariant. Some work considers improving incomplete verification methods by introducing a reinforcement phase to exclude as many spurious CEs as possible. In other words, the verification method iteratively controls the abstract model until we can prove that the properties hold or that the generated CEs exhibit realistic behavior on the original model [127] [142] [143] [144] [145].

The advantage of approximate verification is that it is fast, has scalability, supports multiple models, and supports GPU operations. However, its disadvantages are that it is incomplete and not guaranteed to satisfy networks.

3.1.3 Verification by Satisfiability Modulo Theories

Verification by Satisfiability Modulo Theories (SMT) is a formal verification method used to determine the satisfiability of logical formulas with respect to certain theories, such as arithmetic, bit-vectors, arrays, and others. SMT solvers extend the capabilities of traditional Boolean Satisfiability (SAT) solvers by handling more complex mathematical constructs and providing a more expressive framework for verification.

In recent years, SMT solvers have been adapted to address the verification of neural networks, a growing area of interest due to the increasing reliance on neural networks in critical applications. One notable advancement in this field is Reluplex, proposed by Katz et al. in their paper [112].

Reluplex is an algorithm and toolkit for verifying the properties of FFNNs. It addresses the activation function problem head-on by extending the simplex algorithm, a standard algorithm used to solve LP instances, to support ReLU constraints. This is done by exploiting the piecewise linear nature of ReLU and attempting to progressively satisfy the constraints imposed by the algorithms as they search for feasible solutions. They called this algorithm Reluplex, meaning ”ReLU with Simplex.” They also presented the first proof of NP-completeness for the neural network verification problem.

Marabou is an extension of Reluplex developed by G Katz et al. in their paper [146]. It builds on the original Reluplex algorithm and improves upon it significantly. Moreover, unlike Reluplex, they built their own Simplex kernel instead of relying on GLPK [124]. Marabou's enhancements allow it to handle a broader range of neural network architectures and configurations, offering more robust verification capabilities across different application scenarios.

Expanding further into formal verification, model checking offers another powerful technique. Model checking is wholly based on temporal logic, which provides a framework for reasoning about the behavior of systems over time. Temporal logic [147] is instrumental in situations where the truth value of statements can vary with different states or time points within a system. This is particularly pertinent to systems with dynamic behaviors where states evolve, such as digital circuits or software programs. An example of a propositional temporal logic is Computational Tree Logic (CTL). CTL can specify that when some initial condition is satisfied (e.g., all program variables are positive or no cars on a highway straddle two lanes), then all possible executions of a program avoid some undesirable condition (e.g., dividing a number by zero or two cars colliding on a highway). In this example, the safety property could be verified by a model checker that explores all possible transitions out of program states satisfying the initial condition and ensures that all such executions satisfy the property.

Definition 3.1.3 (Computational Tree Logic)**.** The following grammar generates the language of well-formed formulas for CTL:

$$\begin{aligned} \phi ::= & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \\ \mid & \ \mathrm{AX}\,\phi \mid \mathrm{EX}\,\phi \mid \mathrm{AF}\,\phi \mid \mathrm{EF}\,\phi \mid \mathrm{AG}\,\phi \mid \mathrm{EG}\,\phi \mid \mathrm{A}[\phi \mathrm{U} \phi] \mid \mathrm{E}[\phi \mathrm{U} \phi] \end{aligned} \tag{3.2}$$

Where p ranges over a set of atomic formulas. It is not necessary to use all connectives - for example, $\{\neg, \wedge, \mathrm{AX}, \mathrm{AU}, \mathrm{EU}\}$ comprises a complete set of connectives, and the others can be defined using them. – A means 'along All paths' (inevitably) – E means 'along at least (there Exists) one path' (possibly)

For example, the following is a well-formed CTL formula:

$$\mathrm{EF}(\mathrm{EG}p \Rightarrow \mathrm{AF}r\ ) \tag{3.3}$$

The following is not a well-formed CTL formula:

$$\mathrm{EF}(r\mathrm{U}q) \tag{3.4}$$

The problem with this string is that U can occur only when paired with an A or an E. CTL uses atomic propositions as its building blocks to make statements about a system's states. These propositions are then combined into formulas using logical and temporal operators.

To implement model checking, we represent the model in a program written in a relevant programming language. However, a significant limitation of this method is the state explosion problem, which occurs when performing explicit CTL model checking. Specifically, the model size grows exponentially with the number of variables used in the program's description, rendering the technique computationally infeasible for large systems.

Despite these challenges, exact verification methods remain crucial as they are both sound and complete. These methods will either return proof of the property under scrutiny or provide a counterexample that violates it. This thoroughness ensures that the verification is reliable, albeit at the cost of higher computational demands.

The exactness of the aforementioned methods comes at the price of high computational complexity, making them suitable only for small networks. Therefore, approximate methods have been developed to address the limitations of exact methods. These methods sacrifice completeness for speed, allowing for faster verification processes. Many verification problems, for instance, focus on determining which outputs a neural network can produce, given a specific set of inputs.

In exploring these alternative verification methods, one area that has garnered significant attention is the use of optimization techniques. This leads us to consider an auspicious approach known as Mixed-Integer Linear Programming (MILP).

3.1.4 Verification by Mixed-Integer Linear Programming

To address these challenges and enhance the efficiency of verification methods, researchers have explored various alternative approaches. One promising approach involves the use of optimization techniques to frame verification problems, allowing for a balance between precision and computational feasibility. Among these techniques, Mixed-Integer Linear Programming (MILP) has emerged as a notable method.

MILP (Mixed Integer Linear Programming) extends linear programming in which some or all decision variables are restricted to integers. In this type of problem, the objective function and all constraints are linear, but the solution space becomes discrete due to integer constraints, making the problem more complex and challenging.

Definition 3.1.4 (MILP)**.** A mixed-integer linear program is an optimization problem where the objective and constraints are linear, and the variables may be either real-valued or binary. They are often written as:

$$W_i f_{i-1}\left(x_{i-1}\right) + b_i \tag{3.5}$$

where c, h, A, G, and b are constants, in this case, coming from the network's weights and biases. The binary variables d represent the state of each ReLU in the network. The MILP solver searches through the ReLU states using built-in methods to reason about binary variables.

Many tools are available to analyze and solve mixed-integer linear programs, one of which is Gurobi [148]. This includes GLPK [124], Gurobi [148], CPLEX [149] and others. The main differences between them are that they are open-source to use and which are commercial, requiring a license.

Recently, the verification problem for DNNs implemented as fixed-point networks has been solved [71], [150] by reducing the problem to an SMT solving problem involving the theory of infinitesimal word bit-vector arithmetic. However, the poor scalability of this approach motivates us to explore alternative verification methods. We introduce a method for verification of quantized neural networks that reduces the problem to a decision problem using mixed integer linear programming (MILP) [131]. MILP solver cannot be directly used to verify quantized neural networks because they do not support shift and round operations, the most basic primitives needed to perform arithmetic operations using fixed-point numbers. To address this limitation, one effective approach is to divide nonlinear quantization operations into several linear segments [151]. This method simplifies the problem by breaking the nonlinear region into multiple linear regions, making it suitable for MILP solvers.

3.1.5 Quantization Aspect

Naively, one could independently compute the output intervals of QNNs and DNNs using existing neural network verification tools in the literature and then directly compute their output differences by interval subtraction. However, this approach is ineffective due to significant accuracy loss.

Some researchers have proposed quantized neural network verification based on the Mixed-Integer Linear Programming (MILP) problem. Yedi et al. developed QEBVerif [152], a quantization error-bound verification method. QEBVerif encodes the verification problem into an equivalent MILP problem, which off-the-shelf solvers can solve if the Differential Reachability Analysis (DRA) fails to prove the error bound. Yedi et al. also developed QVIP [153], which reduces the verification problem of QNNs into solving integer linear constraints.

At the same time, Henzinger et al. [150] proposed the quantized neural network verification method based on Satisfiability Modulo Theories (SMT). This paper proved that the verification problem for quantized neural networks with bit-vector specifications is PSPACE-hard. If a problem is PSPACE-hard, it is at least as hard as any NP-hard problem, given that NP is a subset of PSPACE [154]. Proving a problem is PSPACE-hard often indicates a higher level of computational difficulty compared to proving it is NP-hard, as it must account for the broader range of problems solvable with polynomial space. They ran some evaluations on ACAS Xu and Modified National Institute of Standards and Technology (MNIST) [155] benchmarks, and it requires a lot of time to prove some simple properties.

Jia et al, [156] went further, they proposed a method, Efficient Exact Verification (EEV), for verifying Binarized Neural Networks (BNN) - neural networks with binary weights and activations at run-time which consists of two parts: one is a novel SAT solver that speeds up BNN verification, and the other one is strategies to train solver-friendly robust BNNs.

Huang et al., [157] present a framework for formally verifying the properties of quantized neural networks. Their baseline technique is based on integer linear programming, guaranteeing rationality and completeness. They then show how to utilize gradient-based heuristic search methods, boundary advancement Search methods, and boundary propagation techniques to improve efficiency. They evaluate their approach to perceptual networks quantized with PyTorch.

3.1.6 Comparison with *QNNVerifier*

Table 3.1 provides a comprehensive comparison of various neural network verification tools, highlighting their approximation methods, verification techniques, supported neuron types, backend environments, and supported datasets. Each tool listed in the table is designed with specific capabilities tailored to meet distinct verification needs. For instance, alpha-Beta-CROWN employs linear programming relaxation and branch-and-bound methods for verification and supports both FFNN and CNN architectures, making it versatile for tasks involving MNIST and CIFAR datasets. Similarly, Marabou (Reluplex) combines abstract interpretation with simplex and MILP methods, catering to both FFNN and quantized FFNNs. This diversity in approaches reflects the adaptability of these tools to various neural network structures and problem domains.

Table 3.1. Comparison between *QNNVerifier* and SOTA, for the perspective of approximation, verification method, supported neuron type, datasets, and running environment.

Name	**Approximation**	**Verification Method**	**Neuron Type**	**Backend & Environment**	**Supported Datasets**
alpha-Beta-CROWN [134]	Linear programming relaxation, Branch and Bound	Linear programming	FFNN, CNN	Gurobi, PyTorch	MNIST, CIFAR
AI2 [11]	Abstract Interpretation, Zonotope	Linear programming	FFNN, CNN	Gurobi, Tensorflow	MNIST, CIFAR
EEV [156]	BNN-specific encodings	SAT-based Method	Binarized Neural Network	MiniSAT, Z3, PyTorch	MNIST, CIFAR
EQV [157]	Interval Analysis, Heuristic search	Integer Linear Programming	FFNN, CNN, Quantized FFNN	Gurobi, PyTorch	MNIST
Henzinger et al. [150]	Abstract Interpretation, Minimum Bit Allocation	SMT-based Method	Quantized FFNN	Boolector, Tensorflow	MNIST, Fashion MNIST
Marabou (Reluplex) [146] [158]	Abstract Interpretation, Simplex	SMT-based Method, MILP	FFNN, Quantized FFNN	Z3, Gurobi	ACAS Xu
NeuronSAT [159]	Abstract Interpretation	SMT-based Method	FFNN, CNN	Gurobi, PyTorch	ACAS Xu, MNIST, CIFAR
QEBVerif [152]	Abstract Domain, Differential Reachability Analysis	MILP	Quantized FFNN	Gurobi, Tensorflow+Keras	ACAS Xu, MNIST
QVIP [153]	Constraint Simplification	Integer Linear Programming	Quantized FFNN	Gurobi, Tensorflow	MNIST, Fashion-MNIST
QNNVerifier	Search Table, Quantization	SMT-based Method	FFNN, CNN	ESBMC, Tensorflow, Keras, PyTorch	CIFAR, MNIST, ACAS Xu

As can be seen from the table, the SOTA methods use different approaches to approximate and simplify neural network validation, and the validation methods are divided into two main categories: SMT/SAT-based Methods and Linear Programming. The supported neural networks are mainly FFNNs, and some of them can be validated as CNNs or quantized FFNNs. The backends are mainly Gurobi or Z3, and the mainstream datasets used are MNIST and CIFAR. This detailed comparison is a valuable guide for this thesis in conducting experiments in Chapter 4.

3.2 Repair Methods

Neural network repair methods can be categorized into four groups: retraining, adjusting weights, attaching repair units, and counterexample-guided repair methods. In this section, we introduce SOTA repair methods in four categories, then we consider neural network quantization with neural network repair, and finally, we compare these SOTA with *QNNRepair*.

3.2.1 Repair by Retraining

In the first category of repair methods, the idea is to retrain or fine-tune the model for the corrected output with the identified misclassified input.

Repair by Retraining is a neural network repair technique that involves fine-tuning the model by retraining it on a modified or augmented dataset. This method aims to correct specific errors or misclassifications while improving the overall performance of the network.

- **Targeted Dataset Modification:** The primary step involves identifying problematic regions or instances where the neural network fails to meet desired performance. This can be achieved through error analysis, adversarial attacks, or domain-specific criteria. The dataset is then modified or augmented to include these problematic instances, ensuring that the network learns to handle these specific cases correctly during retraining.
- **Fine-Tuning:** Instead of training the network from scratch, repair by retraining focuses on fine-tuning the existing model. This approach leverages the knowledge already captured by the network while making targeted adjustments to improve performance. Fine-tuning involves training the network on the augmented dataset for a few epochs with a lower learning rate to avoid overfitting and to ensure smooth updates to the weights.
- **Regularization Techniques:** Regularization techniques such as dropout, weight decay, and data augmentation are employed to prevent overfitting on the augmented dataset. These techniques help maintain the network's generalization capabilities.

Yu et al. [12] propose a style-guided data augmentation method DeepRepair for repairing DNN in the operational environment, which learns and introduces the unknown failure patterns within the failure samples into the training data via the style transfer. Then, they conducted training on the augmented datasets.

The Apricot repair framework proposed by Zhang et al. [118] aims to repair neural network models iteratively by weight adaptation. Its core idea is that if a deep learning neural network model is trained on many different subsets of the original training dataset, the weights in the resulting reduced Deep Learning Model (rDLM) can be used to repair the neural network model. rDLM can provide insights into the direction and size of the original model's weights to deal with the original model's misclassification test cases. Test cases for categorizing errors in the original network model.

In most of the current neural networks, the prediction of a single input sample depends on all of the model parameters, so it is difficult to correct the behavior of the neural network for a specific input without affecting the model's performance on other inputs. For this reason, Sinitsin et al. [160] proposed editable neural networks and the corresponding editable training (editable) algorithm. The editable training uses a new objective function.

$$\text{Obj}\left(\theta, l_e\right)=\mathcal{L}_{\text{base}}\left(\theta\right)+c_{\text{edit}}\cdot\mathcal{L}_{\text{edit}}\left(\theta\right)+c_{loc}\cdot\mathcal{L}_{\text{loc}}\left(\theta\right) \tag{3.6}$$

The training process is a multi-objective optimization process, where the base objective function $\mathcal{L}_{\text{base}}$ ensures that the initial model is trained for θ; the edit training objective function.

$$\mathcal{L}_{\text{edit}}\left(\theta\right)=\max\left(0, l_e\left(Edit_{\alpha}^{k}\left(\theta, l_e\right)\right)\right) \tag{3.7}$$

Where Edit $\left(\theta, l_e\right)$ is referred to as the edit function, and the goal is to adjust the parameter set θ during training. The final term $\mathcal{L}_{\text{loc}}\left(\theta\right)$ is responsible for locality by minimizing the KL divergence between the predictions of original and edited models. KL divergence (Kullback-Leibler divergence) [161] is a way to measure how different one probability distribution P is from another distribution Q, often thought of as how much information is lost when Q is used to approximate P. The idea of reducing it is: If P represents true data distribution and Q is approximating it. Having more representative data can help Q learn a closer approximation to P.

Repair by Retraining is a powerful and flexible approach to improving neural network performance by addressing specific errors through targeted dataset modifications and fine-tuning.

3.2.2 Repair by Adjusting Weights

The second category uses solvers to get the corrected weights and modify the weights in the trained model directly. These types of methods, including [162] and [163], used SMT solvers for solving the weight modification needed at the output layer for the neural network to meet specific requirements without any retraining. The details of SMT solvers can be found in Section 2.2.2.

Repair by Adjusting Weights is a technique for correcting a neural network's behavior by directly modifying its weight parameters. This method involves identifying and updating specific weights within the network to address errors or misclassifications, thereby improving the network's performance without the need for extensive retraining.

Several key techniques are employed to adjust weights in neural networks. These techniques focus on making precise adjustments to the network's weights, utilizing gradient-based methods for optimization, and applying constraints to ensure beneficial outcomes. The following methods illustrate these approaches:

- **Targeted Weight Adjustment:** The primary focus is on identifying the weights contributing to the network's incorrect outputs or undesirable behavior. The network's performance can be corrected by making precise adjustments to these weights. Local Repair of Neural Networks (LRNN) [164] expresses the nature of the statute as a set of predicates that impose constraints on the output of the neural network over the repair region and defines the neural network repair problem as mixed-integer quadratic programming to adjust the weights of the single-layer neural network according to the given predicates.
- **Gradient-Based Optimization:** Gradient-based methods [163] [165] are often used to determine the necessary weight adjustments. This can involve using back-propagation to compute the gradients of the loss function for the weights and updating the weights accordingly.
- **Constraint-Based Methods:** In some cases, constraints can be applied to the weight adjustments to ensure that the modifications lead to desirable outcomes. For example, constraints can ensure that the adjustments do not degrade the network's performance on correctly classified instances [163] [166].

In addition to the above three common methods, other researchers have proposed more distinctive and effective methods, such as:

Fu et al. [167] proposed REASSURE, which differs from normal neural network weight modification methods. Two ReLU neural network repair techniques are proposed: patch function and support network. The patch function is to find a function $p_A = cx + d$ for a linear region

A to be repaired such that for any sample x in the polyhedral region $A = a_i x \leqslant b_i, i \in I$, $f(x) + p_A(x)$ satisfies the property, where is the output of the original neural network.

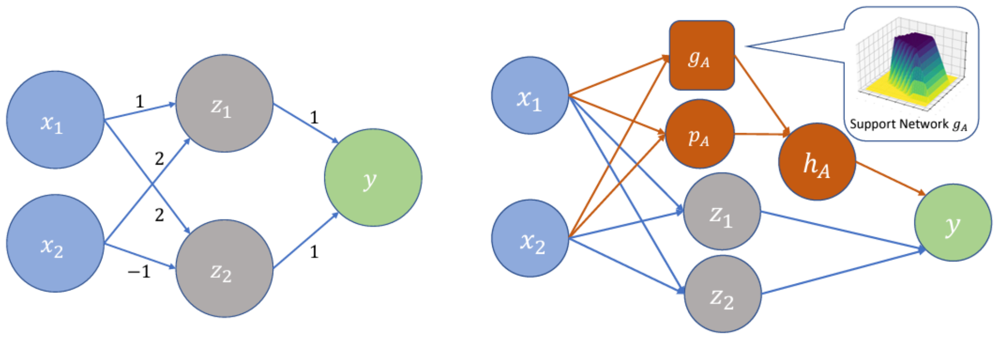


Figure 3.1. The neural network model before and after repair. Left: the target DNN with buggy inputs. Right: The REASSURE-repaired DNN with the patch network is shown in red.

Figure 3.1 illustrates the target Deep Neural Network (DNN) and its repaired version. The left part shows the original network with defective inputs, while the right part demonstrates the network repaired by introducing a patch network. The patch network consists of two subnetworks: the support network g_A and the affine patch function network p_A. The support network g_A is designed to approximate the characteristic function on the linear region $\mathcal{A}$, ensuring the locality of the patch network. The affine patch function p_A guarantees that the region $\mathcal{A}$ satisfies the target specification Φ. This design enables the repaired network to maintain local features while addressing input defects, thereby improving overall robustness and performance.

Usman et al. [163] proposed using the neuron activation pattern [168] as a guide for middle-layer repair. Since it is very difficult to repair all output classes simultaneously, NNRepair solves a set of expert networks, one for each target class, and combines these to obtain the final repaired classifier. NNRepair defines suspicious neuron, consider a mis-classified input, X_f with ideal label C. Let σ_C be the *correct-label* pattern with highest support for C. Let L be the layer for this pattern, and let N denote a neuron at layer L. Then the set of suspicious neurons $\mathcal{N}_{\text{sus}}$ can be defined as follows:

$$N \in \mathcal{N}_{\text{sus}} \iff \big(N \in \text{on}(\sigma_C) \wedge N(X_f) \leq 0\big) \vee \big(N \in \text{off}(\sigma_C) \wedge N(X_f) > 0\big)$$

The NNRepair repair strategy consists of the following steps: (1) Error localization: The goal of this step is to identify a set of suspicious neurons and the suspicious weights of the input edges. (2) Symbolic execution. (3) Concolic execution [169]: Add values to the weights of the suspicious edges and then use the network to execute along the positive and negative samples in parallel to collect the values of the symbolic expressions of the suspicious neurons. Then, the network is executed along the positive and negative samples in parallel to collect the symbolic expression values of the suspicious neurons. (4) Constraint solving: A symbolic expression is formed by a set of repair constraints and solved by an existing solver.

Sun et al. [170] proposed a particle swarm optimization (PSO) algorithm [171] for neuron parameter updating. The particle swarm optimization algorithm simulates the intelligent collective behavior of animals, such as schools of fish and flocks of birds, and is a widely used continuous space optimization algorithm. The PSO algorithm provides the weights for the parameters of the neurons identified in the localization session. The PSO algorithm searches for minor adjustments to the weight parameters of the identified neurons in the localization session to meet the specified properties and maintain the performance of the original network as much as possible to complete the network repair.

3.2.3 Repair by Attaching Repair Units

Repair by Attaching Repair Units is a neural network repair technique that introduces additional units, or small auxiliary networks, to modify and correct the behavior of the original neural network without altering its core structure. This method addresses specific errors or misclassifications by providing targeted adjustments through these repair units.

- **Repair Units:** These are small auxiliary networks or functions that are attached to the original neural network. They are designed to correct specific outputs or regions of the input space where the original network fails to meet desired properties or performance criteria.
- **Non-Intrusive Modification:** Instead of altering the weights and architecture of the original network, repair units are added externally. This non-intrusive approach preserves the integrity and learned features of the original network while providing a mechanism for targeted corrections.
- **Patch Functions:** A common form of repair units is patch functions, which are linear or non-linear functions added to the network's output in specific regions. For example, a patch function $p_A(x) = cx + d$ can be added to the output of a region A to correct errors in that region.
- **Support Networks:** Another form of repair unit is a support network, which is a small neural network designed to correct the original network's outputs. These support networks are explicitly trained to handle problematic regions or scenarios identified in the original network's performance.

Sotoudeh et al. [13] proposed PRDNN (Provable Repair of DNNs), a neural network sample and region (polytope-based) repair technique. PRDNN places no restrictions on the activation functions used by neural networks. The key insight is the introduction of a new architecture called decoupled deep neural networks (DDNNs), in which every deep neural network has an equivalent decoupled counterpart. The repair of any single layer in the decoupled deep neural network can be reduced to a linear programming problem.

Definition 3.2.1 (Decoupled DNN)**.** A Decoupled DNN (DDNN) having layers of size $s_0, \ldots, s_n$ is a list of triples $\left(W^{(a,1)}, W^{(v,1)}, \sigma^{(1)}\right), \ldots, \left(W^{(a,n)}, W^{(v,n)}, \sigma^{(n)}\right)$, where $W^{(a,i)}$ and $W^{(v,i)}$ are $s_i \times s_{i-1}$ matrices and $\sigma^{(i)} : \mathbb{R}^{s_i} \to \mathbb{R}^{s_i}$ is some activation function.

DDNNs can be extended to non-differentiable activation functions as discussed in Sotoudeh and Thakur [13]. Moving to another innovative approach, DeepCorrect [172] corrects the worst distortion-affected filter activations by appending correction units. They apply small stacks of convolutional layers with residual connections at the output of these ranked filters. They train them to correct the worst distortion-affected filter activations while leaving the rest of the pre-trained filter outputs in the network unchanged.

Adding repair units provides better repair results compared to retraining and better generalization ability. Compared to directly modifying the weights, there is no need to perform SMT or Gurobi solving, which greatly saves repair time. However, the repair method of adding repair units still changes the structure of the original neural network, which may affect the scalability and localization of the neural network.

In addition to these methods, *AIRepair* [37] aims to integrate multiple existing repair techniques into the same platform. However, it is important to note that these methods only support the full-precision models and cannot apply to quantized models.

3.2.4 Repair by Counterexamples

Some researchers have proposed the counterexample-guided repair methods; David Boetius et al., [14] showed that counterexample-guided repair can be considered a robust optimization algorithm. While termination guarantees for neural network repair remain beyond our reach, they prove termination for more restrained machine learning models and disprove termination in general.

Counterexample-guided neural network repair methods are a class of techniques that iteratively refine a neural network by addressing specific errors or failures identified through counterexamples. These methods leverage counterexamples – specific inputs that cause the network to produce incorrect or undesired outputs – to guide the repair process. The goal is to improve the network's performance and reliability by systematically correcting these errors.

1. **Counterexample Generation:** Generate or identify counterexamples that cause the net-

work to fail. This can be done using various techniques, such as adversarial attacks, property-based testing, or formal verification methods.

2. **Error Localization:** Analyze the network's behavior on the counterexamples to pinpoint the components responsible for the errors. Techniques such as gradient analysis, sensitivity analysis, or symbolic execution can be used to localize errors.

3. **Repair Strategy:** Apply targeted fixes to the localized errors. This can involve adjusting weights, modifying activation functions, or adding auxiliary components to the network. The repair strategy should be designed to correct the specific errors identified while minimally affecting the overall network performance.

4. **Verification and Iteration:** Verify the repaired network on a verification set to ensure that the counterexamples are addressed and that no new errors are introduced. If necessary, generate new counterexamples and repeat the process until the network meets the desired performance criteria.

Fu et al. [173] developed REGLO (repair technique with provable guarantees on satisfying global robustness properties). It first identifies the violating regions where the counterexamples are located, then uses the validated robustness bounds on these regions to formulate a robust optimization problem to compute the minimum weight change in the network to make modifications to the weights in the neural network.

SpecRepair proposed by Bauer-Marquart et al. [174] is another representative work of the retraining repair paradigm, whose core idea lies in converting the safety statute property into an objective function that is negative for all network inputs that violate the property and then detecting the counterexamples using a global optimization method. These detected counterexamples are then used to make the neural network safe by penalized retraining.

Dong et al. [175] proposed NREPAIR. It divides the input region into two non-overlapping regions. It restarts the validation process using these two new input regions until the input region cannot be divided. The validator generates a counterexample to be provided to the repair process. In the repair process, given the violated property and the counterexample sample, NREPAIR defines a loss function and then minimizes the counterexample's loss value by modifying the neuron's output. Next, the gradient of each neuron is computed based on the values of the provided counterexamples and loss function, and each neuron node is sorted in descending order of the gradient to find the neuron with the largest gradient and with no more than the allowed number of modifications. After identifying these suspicious neurons, NREPAIR modifies its output to its original output minus the product of the neuron's gradient and a predetermined step size. Finally, a new neural network is run through the validator again to check that the repair is complete, and if the network is repaired, it is returned to the new network; otherwise, the process is repeated until a solution is found or a timeout occurs.

3.2.5 Quantization Aspect

One option often explored to fix quantized neural networks is retraining. However, this method has its challenges. Retraining necessitates the use of the original dataset. Plus, quantized neural networks cannot undergo direct training. Instead, one typically retrains the corresponding floating-point neural network first and then applies quantization. Yet, even after this process, the quantized version might still encounter robustness problems.

Then, let us consider adding repair units. However, it would also require a quantization operation after operating on a floating-point neural network, which would also suffer from robustness issues. Therefore, none of the latest repair methods consider the repair of quantized neural networks.

Therefore, we developed a quantized neural network repair method, as detailed in Chapter 5, which directly modifies weights based on constraint solving. It has the advantage that it does not need to retrain the floating-point neural network and avoids the new robustness problem introduced in quantized floating-point neural networks.

3.2.6 Comparison with *QNNRepair*

Table 3.2. Comparison between *QNNRepair* and SOTA, for the perspective of fault localization, repair method, measurement, neuron type, and running environment.

Name	**Fault Localization**	**Repairing Method**	**Measurement**	**Neuron Type**	**Environment**	**Supported Datasets**
Apricot [118]	-	Training and adapting	Accuracy	CNN, ResNet	Keras, Tensorflow	CIFAR
Arachne [165]	Gradient Loss (GL) based, Random Selection	Patch generation	Fairness	CNN, LSTM	TensorFlow, PyTorch	CIFAR, GTSRB, F-MNIST
DeepCorrect [172]	Computing Correction Prioritization	Adding correction units	Accuracy	CNN with correction units	Theano, Keras	CIFAR, Imagenet
DeepRepair [12]	-	Data augmentation, Fine-Tuning	Accuracy	Densenet, ResNet, All-ConvNet, Renet, LSTM	PyTorch	CIFAR
DeepFault [176]	Tarantula, Ochiai, D*	Suspiciousness-guided input synthesis	Loss, Accuracy, Suspicious neurons distribution, Similarity	CNN	Keras, TensorFlow	MNIST, CIFAR
PRDNN [13]	Randomly-selected or CNN layer	Polytope Repair, Architecture extension	E: Efficacy (%), D: Drawdown (%), G: Generalization (%), T: Time	SqueezeNet, FFNN	PyTorch, Gurobi	ImageNet, ACAS Xu, CIFAR
QNNRepair	Tarantula, Ochiai, D*, Euclid, Ample, Jaccard, Wong3	SMT-based, Weight Modification	Accuracy, Robustness	CNN, ResNet	Keras, Tensorflow, PyTorch	CIFAR, GTSRB, ImageNet
Socrates [177]	Causal attribution	Particle Swarm Optimization (PSO) algorithm	Fairness, Backdoor Safety, Accuracy	CNN, FFNN	PyTorch, Gurobi	GTSRB, MNIST, F-MNIST
SpecRepair [174]	Counterexample generated by Verification	Quadratic programming	Accuracy	CNN, FFNN	PyTorch	ACAS Xu, MNIST

Table 3.2 provides a comparison of *QNNRepair* with other state-of-the-art (SOTA) tools in the field of neural network fault repair. The table outlines key features such as fault localization techniques, repair methods, evaluation metrics, supported neuron types, and the runtime environment for each tool. For example, Arachne employs Gradient Loss-based and Random

Selection methods for fault localization and utilizes patch generation as its repair technique. It is evaluated based on fairness metrics and supports CNN and LSTM neuron types, with compatibility for TensorFlow and PyTorch environments. On the other hand, DeepFault integrates traditional fault localization approaches such as Tarantula and Ochiai, combining them with suspiciousness-guided input synthesis to improve accuracy and neuron distribution similarity, specifically for CNNs.

3.3 Chapter Summary

This chapter provides an in-depth discussion of the related works on neural network verification and repair, summarizing the advancements and challenges in the field. Firstly, using the classic ACAS Xu system as an example, it elaborates on the safety properties addressed in neural network verification, including robustness, monotonicity, output bounds, and fairness. These properties serve as critical metrics for evaluating whether neural networks behave as expected.

Next, the chapter categorizes and introduces the various methods for neural network verification as follows:

1. **Exact Verification**: This approach employs exhaustive search or symbolic reasoning to precisely verify the safety properties of neural networks. While it offers high accuracy, it is computationally expensive and generally applicable to smaller-scale networks.
2. **Approximate Verification**: Techniques like abstract interpretation or relaxed constraints are used to rapidly assess network safety. Though some precision may be sacrificed, this method is suitable for larger-scale networks.
3. **Verification by Satisfiability Modulo Theories**: By formulating the verification problem as satisfiability modulo theories (SMT), this approach achieves theoretical completeness, though the efficiency can be limited by the scale of the constraints.
4. **Verification by Mixed-Integer Linear Programming**: Utilizing mixed-integer linear programming, this method is particularly effective for networks with linear activation functions, striking a balance between efficiency and precision.

The strengths and limitations of these methods are analyzed to highlight their applicable scenarios and constraints.

Subsequently, the chapter delves into the works on neural network repair, classifying them into the following categories:

1. **Repair by Retraining**: Errors are corrected by retraining the network. While this method is broadly applicable, it often requires significant amounts of data and computational resources.

2. **Repair by Adjusting Weights**: Network weights are directly adjusted to address specific issues. This method is efficient but may introduce new problems.

3. **Repair by Attaching Repair Units**: Additional network structures or repair modules are incorporated to limit erroneous behavior under specific inputs, enhancing the flexibility of the network.

4. **Repair by Counterexamples**: This approach leverages counterexamples generated during verification to guide targeted optimization, demonstrating high specificity and effectiveness.

Lastly, the chapter also considers the impact of network quantization on verification and repair methods, analyzing its influence on verification accuracy and repair outcomes to ensure that theoretical research can be effectively applied to practical scenarios.

Chapter 4

QNNVerifier: Quantized Neural Network Verification

This chapter focuses on our neural network verification tool *QNNVerifier*. *QNNVerifier* has a set of processing procedures for input neural network models that are described in detail in this chapter, including neural network code conversion, discretization of non-linear activation function, interval analysis, assertion language, constant folding and slicing. This chapter also introduces *QNNVerifier*'s backend ESBMC. We then conducted an experimental evaluation of the *QNNVerifier*, benchmarking it on the ACAS Xu, GTSTB, and CIFAR datasets and comparing it with the experimental results of SOTA, which illustrates the advantages of the *QNNVerifier* over SOTA in validating neural network models.

4.1 Chapter Introduction

To fix the vulnerabilities, we need first to detect them. Unlike traditional software vulnerabilities, which are often related to specific lines or blocks of code, the vulnerabilities in neural networks are embedded in the weights of the trained models. Traditional software vulnerabilities can be detected and precisely located within the programming code, but DNNs remain largely black boxes. Moreover, traditional vulnerability detection tools cannot read and understand the DNNs' weights and, therefore, cannot detect errors within these DNNs.

This chapter mainly introduces the *QNNVerifier*, a specialized tool designed to detect errors in DNNs and provide counterexamples that exploit the erroneous models. Unlike conventional methods that might directly attempt to verify neural network models, *QNNVerifier* circumvents the complexity associated with different model formats trained on various frameworks. Instead, *QNNVerifier* converts these models into an intermediate C code representation. This intermediate step allows the insertion of safety properties and the abstraction of activation functions, facilitating a more straightforward verification process.

The core of *QNNVerifier*'s verification process is supported by ESBMC (please see Section 2.2.1), a mature and permissively licensed open-source context-bounded model checker. ESBMC is adept at verifying single- and multi-threaded C programs for code safety violations. By leveraging ESBMC as the backend, *QNNVerifier* can effectively check the converted C code for errors, ensuring a thorough verification process.

After detailing the algorithm and implementation of *QNNVerifier*, a series of evaluations were conducted to validate its effectiveness. This includes an ablation study to identify the optimal parameter combinations for *QNNVerifier*, ensuring the best performance. Additionally, *QNNVerifier* was compared with state-of-the-art (SOTA) neural network verification tools, highlighting its advantages and demonstrating its efficacy in various scenarios.

4.2 *QNNVerifier* Framework Overview

As shown in Figure 4.1, the *QNNVerifier* framework consists of the following components: code generation, representation, abstraction of activation functions, generation of safety properties, Invariant generation, model checking, deployment, and retraining.

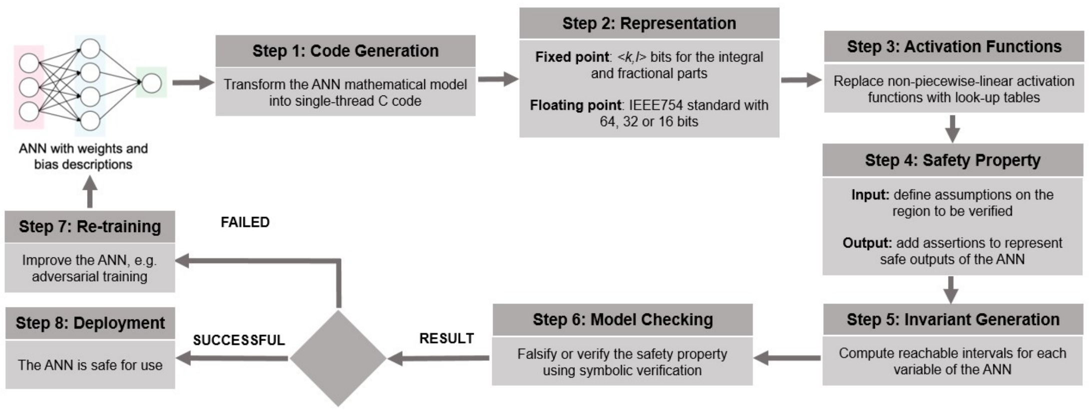

ANN with weights and bias descriptions
Step 1: Code Generation
Transform the ANN mathematical model into single-thread C code
Step 2: Representation
Fixed point: <k,l> bits for the integral and fractional parts
Floating point: IEEE754 standard with 64, 32 or 16 bits
Step 3: Activation Functions
Replace non-piecewise-linear activation functions with look-up tables
Step 4: Safety Property
Input: define assumptions on the region to be verified
Output: add assertions to represent safe outputs of the ANN
Step 5: Invariant Generation
Compute reachable intervals for each variable of the ANN
Step 6: Model Checking
Falsify or verify the safety property using symbolic verification
RESULT
FAILED
SUCCESSFUL
Step 7: Re-training
Improve the ANN, e.g. adversarial training
Step 8: Deployment
The ANN is safe for use

Figure 4.1. The proposed verification workflow for fixed- and floating-point DNNs. The inputs are neural network models, and the outputs are models that can be used safely.

1. **C Code Generation:** Given a neural network model to be validated and a set of safety properties as inputs, the code generation module will produce a C-file that is consistent with the behavior of the original neural network model, and the user can compile this C-file to perform inference operations. *QNNVerifier* uses this C-file as input for subsequent verification.

2. **Representation of Quantized Behaviour:** In representation, *QNNVerifier* converts trained models and neural network operations, e.g., realization, from floating-point representation into fixed-point, followed by a check of the desired properties.

3. **Discretization of non-linear activation functions:** In the abstraction of activation functions, *QNNVerifier* presents an approach to convert such non-linear functions into look-up tables, thus significantly speeding up verification processes.

4. **Introducing safety properties:** Verifying a neural network model means proving that a given safety property holds. Such a safety property is a falsifiable mathematical relation defined by the values of a DNN's variables. Since we are considering software implementation of DNNs in generating safety properties, we then show how to annotate DNN code and specify a desired safety property.

5. **Invariant Generation:** Given the sequential nature of inference in neural networks, the set of $\mathcal{H}$ values allowed by the safety property premise also limits the scope of subsequent intermediate computational steps. Thus, if we can explicitly derive these additional constraints and unfold them onto intermediate variables to propagate them and benefit from them in subsequent operations, we can more succinctly tell the model checker where to find counterexamples. On the more practical side, many tools exist to perform interval analysis of C code. In our experiments, we have used the Evolved Value Analysis (EVA) plugin of the open-source tool Frama-C [178].

6. **Model Checking:** Given the annotated C code containing safety properties from the previous module, we are now tasked with answering the following verification question: does the *all* input that satisfies the precondition `assume` also satisfy the associated `assert` post-condition in a particular neural network implementation? In other words, can we find at least `one` particular input that violates the safety property of a quantized neural network implementation with a specific accuracy? We will introduce the use of state-of-the-art model-checking techniques to answer this question.

7. **Deployment or Re-training:** After completing model checking, *QNNVerifier* determines whether the model can be deployed or needs to be retrained based on ESBMC verification result (Verification Successful or Verification Failed) of the given safety properties.

4.2.1 Neural Network Code Conversion

QNNVerifier uses ESBMC as the backend, a C program verification tool, to perform bounded model checking on the converted C files. To utilize this tool, *QNNVerifier* first needs to convert the neural network model into the corresponding C program and ensure that the converted model's behavior is consistent with that of the original model.

Converting neural network models to C programs requires understanding the different file formats used by various neural network frameworks. Ensuring compatibility and accurate translation from these formats to C code is crucial for the verification process. Hence, we need to validate the converted code. During the validation step, we use an original model and its implementation in C code to generate and store outputs based on the mentioned datasets. Then, we take the absolute value of the difference between these two sets, i.e., outputs from the original and C-based elements, to show that they are within a range given by ϵ, which is arbitrarily small. Indeed, our goal is to show that their outputs are similar enough to reason about their equality so that the associated implementation is regarded as valid, keeping the original intended behavior. This way, we ensure that our transformation strategy is reliable and does not affect the obtained inference results.

The current neural network implementation uses popular training and development platforms such as TensorFlow [72], PyTorch [16], and Caffe [17]. These platforms define specific file formats to store the neural network's architecture and trained weights. For example, PyTorch's .pth [179] file format is typically used to save a model's weights or the entire model. Although this is not an open-standard file format, it is based on PyTorch's serialization mechanism. On the other hand, TensorFlow Keras uses the HDF5 file format to preserve the structure and weights of the model. HDF5, known as Hierarchical Data Format version 5 [180], is a file format designed for storing and organizing large amounts of data. This format is well-suited for processing and storing complex data, such as scientific and large datasets.

Before the code conversion, *QNNVerifier* implemented the functions of each basic building block in the neural network, such as the fully connected unit, the convolutional neural network unit, and the activation function, in C language. Then, the neural network model is read, and the weights in the model are imported into the C file according to the corresponding format.

In the following, an example in Figure 4.2 is given to illustrate conversion from a neural network model to a C file:

On the left side, the code snippet defines a function resnet_block that builds a single block of the ResNet architecture using the Keras library. The function takes an input tensor x and applies a series of transformations to it:

- A 1x1 convolution that adjusts the number of filters. Batch normalization followed by ReLU activation (lines 3-6).
- A 3x3 convolution with "SAME" padding to maintain the spatial dimensions (lines 7-8).
- Another round of batch normalization and ReLU activation (lines 9-10).
- A final 1x1 convolution to adjust the channel dimensions, followed by batch normalization (lines 11-12).

```
1  def resnet_block(x, filters ,
2  kernel_size=3, stride=1):
3   x= layers.Conv2D(filters ,1 ,
4  strides=stride)(x)
5   x= layers.BatchNormalization()(x)
6   x= layers.Activation(”relu”)(x)
7   x= layers.Conv2D(filters ,
8  kernel_size , padding=”SAME”)(x)
9   x= layers.BatchNormalization()(x)
10  x= layers.Activation(”relu”)(x)
11  x= layers.Conv2D(4∗ filters , 1)(x)
12  x= layers.BatchNormalization()(x)
13  x= layers.Add()([shortcut ,x])
14  x= layers.Activation(”relu”)(x)
15 return x
```

```
1  size_t conv2d_1_stride[2]={1,1} ;
2  size_t conv2d_1_dilation[2]={1,1} ;
3  k2c_tensor conv2d_1_output = [
4  conv2d_1_output_array , 3, 65536,{16,16,256 ,
5  1. 1}}:
6  k.2c_tensor conv2d_1_kernel = {
7  conv2d_1_kerne1_array , 4,16384,{ 1, 1,
8   64,256,1}};
9  k2c_tensor conv2d_1_bias = {conv2d_1_bias_array
10 ,1,256,{256,1,1,1,1}} ;
11 size_t conv2d_4_stride[2] = {1,1};
12 size_t conv2d_4_dilation[2] = {1,1};
13 k2c_tensor conv2d_4_output= {
14 conv2d_4_output_array,3,65536,{16,16,256,
15 1,1}};
16 k2c_tensor conv2d_4_kernel = {
17 conv2d_4_kernel_array , 4,16384,{1,1,
18 64,256,1}};
19 k2c_tensor conv2d_4_bias = {conv2d_4_bias_array
20 1,256,{256,1,1,1,1}};
21 k2c_tensor add_output = {add_output_array
22 ,3,65536,{16,16,256,1,1}};
23 size_t add_num_tensors0=2 ;
```

Figure 4.2. Conversion from the neural network structure defined by Keras to the neural network structure defined by C. This is a code snippet of resnet_block, the basic unit in the ResNet (see Definition 2.1.5) model mentioned in Section 2.1.1 that will be tested in the experimental section, which contains three convolutional layers as well as an added layer, with the activation function chosen to be ReLU.

- An addition operation that combines the original input (assumed to be a shortcut) with the transformed input, followed by another ReLU activation (lines 13-14).

The right side shows the translated C code. The variables and structures suggest the definition of convolutional layers and parameters such as:

- Definitions of tensor structures for convolutional layers' outputs, kernels, and biases (lines 1-10).
- Settings for strides and dilations in convolutional operations (lines 11-12).
- The allocation of tensors for outputs and kernel weights, alongside their dimensions (lines 13-20).
- The setup suggests configurations for multiple convolution layers, likely corresponding to the ones defined in the Python version (lines 21-23).

4.2.2 Discretization of Non-linear Activation Functions

Specific activation functions can have a considerable impact on verification times. Piece-wised linear functions can be readily represented with if-then-else instructions, but non-linear activation ones require careful adjustments to avoid severe performance degradation. This section presents a method to convert such non-linear functions into look-up tables, thus speeding up the verification process.

Firstly, let us introduce the concept of Lipschitz continuous function in calculus:

Definition 4.2.1 (Lipschitz continuous function)**.** A function $f : X \to Y$ is called Lipschitz continuous with constant **C** if, for each $x_1, x_2 \in X$ one has

$$d(f(x_1), f(x_2)) \leq \mathbf{C} \cdot d(x_1, x_2), \tag{4.1}$$

where d stands for the distance.

Assume that the non-linear activation function $\mathcal{N} : \mathcal{U} \mapsto \mathbb{R}$ is a piecewise Lipschitz continuous function [181], 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

$$\|\mathcal{N}_i(u_1) - \mathcal{N}_i(u_2)\| \leq \lambda_i \|u_1 - u_2\|, \quad \forall u_1, u_2 \in \mathcal{U}_i, \tag{4.2}$$

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

QNNVerifier applies the following discretization approach to each subset $\mathcal{U}_i$. First, it discretizes each $\mathcal{U}_i$ with a countable set $\tilde{\mathcal{U}}_i \subset \mathcal{U}_i$. Then, it builds a lookup table for rounding the evaluation of $\mathcal{N}_i(u)$ to $\tilde{\mathcal{N}}_i(u) : \mathcal{U}_i \mapsto \mathcal{R}$, thus 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 $\mathcal{U}_i$ to ensure the accuracy $\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \epsilon$. More specifically, let L_i be the length of the interval $\mathcal{U}_i$, i.e.,

$$L_i \triangleq \sup_{u \in \mathcal{U}_i} u - \inf_{u \in \mathcal{U}_i} u. \tag{4.3}$$

Given the length L_i and the desired accuracy ϵ, N_i can be chosen according to Theorem 1:

Theorem 1. Let the non-linear activation 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 discrete 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

$$\|\tilde{\mathcal{N}}(u) - \mathcal{N}(u)\| \leq \epsilon, \tag{4.4}$$

for a given ϵ, if the following inequality holds:

$$N_i \geq 1 + \frac{L_i \lambda_i}{\epsilon}, \forall i \in \mathbb{N}_{\leq a}. \tag{4.5}$$

Proof. Given that the length of each interval $\mathcal{U}_i$ is L_i (cf. (4.3)), the length of each sub-interval, obtained by uniformly dividing $\mathcal{U}_i$ into N_i samples, is $\frac{L_i}{N_i - 1}$. Considering the Lipschitz continuity in (4.2), the rounding error for $\tilde{\mathcal{N}}_i(u)$ is bounded as

$$\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \frac{L_i}{N_i - 1}\lambda_i. \tag{4.6}$$

If (4.4) holds for all $i \in \mathbb{N}_{\leq a}$, the inequality

$$\frac{L_i}{N_i - 1}\lambda_i \leq \epsilon \tag{4.7}$$

and (4.5) also hold. Moreover, from (4.6) and (4.7), $\|\tilde{\mathcal{N}}_i(u) - \mathcal{N}_i(u)\| \leq \epsilon$ for all $i \in \mathbb{N}_{\leq a}$. $\square$

Without loss of generality, let us rewrite the discretization $\tilde{\mathcal{N}}_i(u)$ as follows:

$$\tilde{\mathcal{N}}_i(u) = \mathcal{N}_i\left(\mathcal{A}_i(u)\right), \tag{4.8}$$

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 the approximation $\tilde{\mathcal{N}}_{\text{Sigm}}$ of the sigmoid activation function $\mathcal{N}_{\text{Sigm}}$, 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 $\sup_{u\in\mathbb{U}} |\mathcal{N}_{\text{Sigm}}(u)| = 0.25$ and $\mathcal{U} = \mathbb{R}$. Moreover, let us choose the following three intervals to define the approximation $\tilde{\mathcal{N}}_{\text{Sigm}}(u)$:

$$\mathcal{U}_1 = (-\infty, -20]\,, \mathcal{U}_2 = (-20, 20)\,, \text{and}\mathcal{U}_3 = [20, \infty)\,, \tag{4.9}$$

Since the derivative of $\mathcal{N}_{\text{Sigm}}(u)$ is negligible for $u \in \mathcal{U}_1 \cup \mathcal{U}_3$, we have $\lambda_1 \approx 0$ and $\lambda_3 \approx 0$, whereas λ_2 is equivalent to the global Lipschitz constant, i.e. $\lambda_2 = \lambda_{\text{Sigm}} = 0.25$. Now, we can use (4.5) to compute the number of samples per interval to ensure the desired accuracy $\epsilon = 0.01$. Accordingly, we have $N_1 = N_3 = 1$ and $N_2 = 1001$ since $L_2 = 40$ (cf. (4.3)), and $\mathcal{A}_i$ can be arbitrarily chosen. For this example, we suggest 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 DNNs.

Figure 4.3 illustrates the discretization effect on the Sigmoid function. The approximation fits well for $\epsilon = 0.01$, but it becomes poor when ϵ increases. It is worth mentioning that a lookup table is a trade-off between speed and memory. If the latter is not a restriction, ϵ should be as small as possible and much lower than the quantization step incurred by a fixed-point format. In this way, the correctness of the verification process will not be compromised.

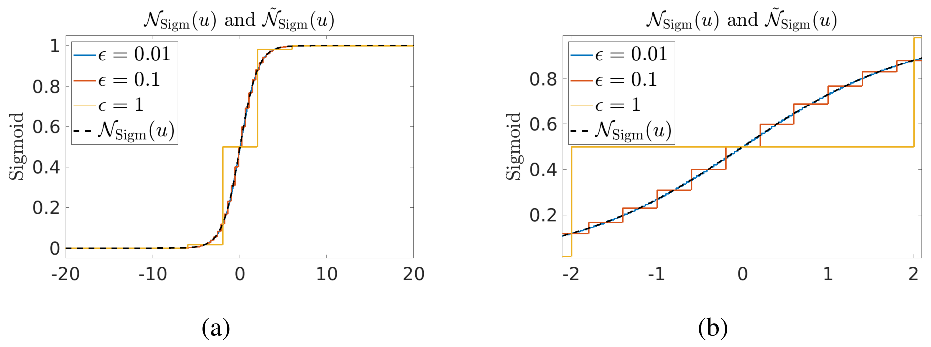


Figure 4.3. Comparison between the real sigmoid $\mathcal{N}_{\text{Sigm}}$ and its discretizations $\tilde{\mathcal{N}}_{\text{Sigm}}$ for $\epsilon = 0.01$ ($N_2 = 1001$), $\epsilon = 0.1$ ($N_2 = 101$), $\epsilon = 1$ ($N_2 = 11$): (a) sigmoid activation function together with its approximations within the range $[-20, 20]$ and (b) a zoom in to show the interval $[-2, 2]$.

4.2.3 Interval Analysis via Frama-C

Once the safety property has been specified, as explained in Section 2.3.1, *QNNVerifier* can inject further `assume` instructions into the code and reduce the model checker's search space. Indeed, given the sequential nature of DNN computation, the set $\mathcal{H}$ of values allowed by the premise of a safety property also constraint the range of the following intermediate computation steps.

To effectively utilize these constraints, it is crucial to propagate the initial set $\mathcal{H}$ through the network accurately. This propagation allows us to refine the bounds on the network's intermediate and output values, thereby tightening the overall verification process. One common method to achieve this is through invariant analysis.

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 variable) by propagating the initial set $\mathcal{H}$ through a DNN with interval arithmetic rules.

QNNVerifier uses the EVA plugin of Frama-C as a tool for interval analysis. The Evolved Value Analysis plug-in [182] computes variable variation domains. Although the user may guide the analysis in places, it is quite automatic. It handles a wide spectrum of C constructs. This plug-in uses abstract interpretation techniques.

ACSL (ANSI/ISO C Specification Language) [183] is a specification language for describing the behavior of C programs that can be used with Frama-C [184] to verify that C code conforms to its specification.

Within EVA, the expression evaluator interleaves forward and backward evaluation steps. A forward evaluation is a bottom-up propagation of value abstractions from the values and constants to the root of an expression. It queries the state abstractions to extract a value for variables and relies otherwise on the value semantics of the C operators.

Example 1. Consider two memory domains, I and C, storing information about the possible values of integer variables. Domain I uses intervals, and domain C uses congruences for value abstractions. Assume the condition $x > 3$, where I provides the interval abstraction $[0, 12]$ and C provides the congruence $x \equiv 0 \pmod 3$ for x. The values for x are reduced to $[4, 12]$ in I and remain $x \equiv 0 \pmod 3$ in C when backward-evaluating the condition. The inter-reduction between these values further narrows down the interval in I to $[6, 12]$. This refinement enables learning more precise information for x as it abstracts the effects of the entire condition.

QNNVerifier then injects intervals into converted C-code as additional pre-condition instructions `assume` on intermediate variables, thus covering the entire processing chain. Finally, *QNNVerifier* is compared with the native interval analysis support provided by the state-of-the-art verification tool ESBMC [94] and found that combining them (both enabled) yields the best results.

4.2.4 Assertion Language in ESBMC

To further enhance the verification process, it is essential to leverage a robust assertion language that can accurately specify the properties of neural networks within the verification framework. By defining clear and precise assertions, the verification tool can effectively verify the correctness and safety of the neural network model.

In traditional software programs, many assertion languages have been developed over the years. These range from simple state-based assertions to more complex forms of logic, such as Hoare logic [185] and temporal logic [186]. However, the existing engines that specify neural network properties often do so in an ad hoc manner, lacking a standardized approach. This ad hoc specification poses a challenge as neural networks, being a new programming paradigm, are used in a wide array of applications, each with distinct requirements. Consequently, it is crucial to establish a common language that can uniformly specify the required properties across different neural networks.

One key aspect of this common language is the reachability attribute. This attribute specifies that the neural network output must satisfy a specific constraint (e.g., within a specific range) if the input satisfies a specific constraint (e.g., within a specific range). This form of constraint is already supported by existing tools such as Reluplex [112], which demonstrates the feasibility and importance of such specifications. Specifying these constraints in our proposed language is straightforward. For instance, Figure 4.4 illustrates Property 1 from [112] using our assertion language, showcasing how easily these properties can be defined and understood.

By integrating such a robust assertion language, we can significantly improve the verification process, making it more comprehensive and reliable. This ensures that neural networks can be effectively validated, leading to safer and more dependable AI systems.

```
(assert (<= X_0 0.679857769))
(assert (>= X_0 0.6))

(assert (<= X_1 0.5))
(assert (>= X_1 -0.5))

(assert (<= X_2 0.5))
(assert (>= X_2 -0.5))

(assert (<= X_3 0.5))
(assert (>= X_3 0.45))

(assert (<= X_4 -0.45))
(assert (>= X_4 -0.5))
; Unsafe if COC >= 1500. Output scaling is 373.94992 with a bias of 7.518884: (1500 - 7.518884) / 373.94992 = 3.991125
(assert (>= Y_0 3.991125645861615))
```

```
__ESBMC_assume(test1_dense_42_input_input.array[0] <= 0.679857769 && test1_dense_42_input_input.array[0] >= 0.6);
__ESBMC_assume(test1_dense_42_input_input.array[1] <= 0.5 && test1_dense_42_input_input.array[1] >= -0.5);
__ESBMC_assume(test1_dense_42_input_input.array[2] <= 0.5 && test1_dense_42_input_input.array[2] >= -0.5);
__ESBMC_assume(test1_dense_42_input_input.array[3] <= 0.5 && test1_dense_42_input_input.array[3] >= 0.45);
__ESBMC_assume(test1_dense_42_input_input.array[4] <= -0.45 && test1_dense_42_input_input.array[4] >= -0.5);

__ESBMC_assert(c_activation_48_test1.array[0] >= 3.991125645861615, "Classification is not safe.");
```

Figure 4.4. The conversion from safety properties defined by VNN-LIB to safety properties defined by ESBMC is an example of the conversion.

The code block on the left shows the safety properties defined in VNN-LIB [109]. It contains a series of assertion statements that define constraints on parameters such as relative distances, angles, speeds, and other critical metrics between the vehicle and the intruder. The purpose of the VNN-LIB standard is to provide a unified format for the description of NNs for verification. To support the widest range of network architectures and related properties, the standard builds on the ONNX (see Section 2.1.1) format to represent the network models and on the SMT-LIB language for property specification.

For example, the assertion statement

```
__ESBMC_assert(c_activation_48_test1.array[0] >= 3.991125645861615, "Classification␣is␣not␣safe.");
```

checks a specific safety condition. This statement asserts that `c_activation_48_test1.array[0]` (which corresponds to `Y_0` in the VNN-LIB format) must be greater than or equal to 3.99. If this condition is unmet, ESBMC will trigger an error with the message "Classification is not safe." This mechanism ensures that the system adheres to predefined safety standards, preventing potential hazards by verifying that critical thresholds are maintained.

The code block on the right shows how these safety properties can be translated into a form that ESBMC can understand. Here, __ESBMC_assume assumes constraints on the input values, and __ESBMC_assert is used to assert safety properties under specific conditions. This translation allows ESBMC to verify the system's behavior against the defined safety properties, ensuring compliance with the required safety standards.

4.2.5 ESBMC Architecture

Figure 4.5 shows the current architecture of ESBMC; white rectangles represent input and output, while grey rectangles represent the verification steps. The tool is composed of several modules, each with a specific goal.

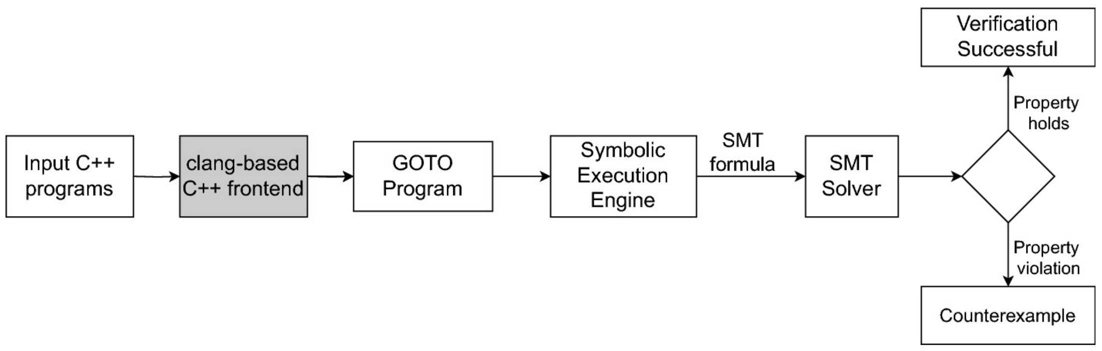


Figure 4.5. ESBMC architecture.

Frontend. This converts the program into an Abstract Syntax Tree (AST). ESBMC has three frontends: one clang-based C frontend, one CBMC-based C frontend, and one CBMC-based C++ frontend.

GOTO converter. This module converts the AST generated by the front end into a state transition system called the GOTO program. The GOTO program can be changed to add property checks and k-induction-specific instructions. Converts the GOTO program into a sequence of static single assignments (SSA). This module unwinds the loops of the GOTO program, propagating constants to generate a minimal set of SSAs. This module can also add property checks, most of which are related to dynamically allocated memory.

SMT encoding. Converts the set of SSAs into SMT and checks for satisfiability. If the formula is satisfiable, then the SMT solver is queried for relevant information in order to build the counterexample. ESBMC 7.6.1, which *QNNVerifier* uses as the bounded model checker in this thesis, currently supports five solvers: Z3, Bitwuzla, Boolector, Yices [187], and CVC5.

4.2.6 Constant Folding and Slicing

Constant Folding and slicing are pivotal optimization techniques in compiler design, aiming to enhance the efficiency and performance of programs by reducing computation overhead and eliminating redundant code. As detailed by Muchnick et al., [85], Constant Folding involves pre-calculating expressions during compile time rather than at runtime. This optimization can be applied to various constant expressions, including arithmetic operations and string literals, effectively reducing the runtime computation requirements.

Constant Slicing, while not as frequently discussed as constant folding, is crucial in optimizing compiler performance. According to Wegman et al., [188], constant slicing involves the removal of parts of the program that are deemed unnecessary for the execution outcome. This process often works hand-in-hand with constant propagation, where the compiler replaces expressions with known constant values during the compilation process, streamlining the code and improving execution efficiency.

In particular, *QNNVerifier* exploits the constant propagation technique to reduce the number of expressions associated with specific neuron computation procedures and activation functions. By simplifying the SSA (Static Single Assignment) representation, *QNNVerifier* can use local and recursive transformations to remove functionally redundant expressions (for neuron computation procedures and activation functions) and redundant literals (for safety properties). This involves identifying expressions whose values can be predetermined at compile time and propagating these values throughout the program to simplify further expressions and eliminate unnecessary calculations.

For example, if a neuron's activation function consistently produces a specific value for given inputs, this value can be propagated, eliminating the need to recompute the function repeatedly. Similarly, if certain safety properties involve constants that can be determined at compile time, these can be embedded directly into the assertions, reducing the overhead during runtime verification.

Overall, these optimization techniques not only enhance the performance and efficiency of the compiled code but also contribute to more reliable and maintainable software by ensuring that only the necessary computations are performed, and all redundant code is removed.

Thus, *QNNVerifier* simplifies 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

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

We apply such simplifications to reduce the resulting formula's size and then achieve simplification within each time step and across time steps during the encoding procedure of a DNN.

QNNVerifier's approach to constant folding and slicing is intricately linked to the workings of the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). ESBMC, as introduced by Gadelha, Morse, and Cordeiro in their discussion on handling the complexities of C programs [189], utilizes these optimizations as part of its process to convert C programs into GOTO programs. This conversion simplifies the program's structure by replacing all control structures with conditional and unconditional jumps, making it more amenable to analysis and optimization. The GOTO program representation is further processed through symbolic execution, transforming the program into a Static Single Assignment (SSA) [190] form. This transformation facilitates the unrolling of loops and recursive functions dynamically, incorporating unwinding assertions to check the adequacy of the unrolling bounds.

Slicing removes expressions that do not contribute to the checking procedure of a given safety property. The verification engine combines two slicing strategies: it removes all instructions after the last assert in the SSA set. It collects all symbols in assertions and removes instructions that do not contribute to them. Both slicing strategies ensure that unnecessary instructions are ignored during SMT encoding. As an example, using Figure 4.6, if we are interested in checking that DNN's output only, we rewrite the final assert statement, in line 12, as $f <= 4$. It indicates that everything not involving f does not impact the conclusion of the intended safety property, and the resulting SSA for that code 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.

```
1   int main() {
2     _Bool x, y;
3     int a, b, f;
4     x = nondet_bool();
5     y = nondet_bool();
6     a = ((2*x) - (3*y));
7     a = a < 0 ? 0 : a;
8     b = (x + (4*y));
9     b = b < 0 ? 0 : b;
10    f = ((3*x) + y);
11    f = f < 0 ? 0 : f;
12    assert(a <= 2 && b <= 5 && f <= 4);
13    return 0;
14  }
```

Simple neural network

```
1   x1 = nondet_symbol(nondet0);
2   y1 = nondet_symbol(nondet1);
3   a1 = 2 * (int)x1 - 3 * (int)y1;
4   a2 = (a1 < 0 ? 0 : a1);
5   b1 = (int)x1 + 4 * (int)y1;
6   b2 = (b1 < 0 ? 0 : b1);
7   f1 = 3 * (int)x1 + (int)y1;
8   f2 = (f1 < 0 ? 0 : f1);
9   (assert) a2 <= 2;
10  (assert) b2 <= 5;
11  (assert) f2 <= 4;
```

Neural Network in SSA form

Figure 4.6. (a) A simple NN 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 NN C program was converted into SSA form.

4.3 Evaluation

4.3.1 Description of the *QNNVerifier* Benchmarks

In our evaluation, we consider DNNs trained on four datasets: the ACAS Xu dataset [54], the GTSRB dataset [191], the CIFAR-10 dataset and the TinyImageNet dataset:

ACAS Xu:

- The ACAS Xu benchmark, derived from avionics research focused on Airborne Collision Avoidance Systems (ACAS) for Unmanned Aircraft (Xu), employs a large state-action table that dictates appropriate piloting decisions and compresses them into 45 DNNs. Each neural network's input scale is 1×5, and the output scale is 1×5. The details about the definition can be found in Section 2.3.2.

- Due to using a large lookup table in the design, a neural network compression of the policy was proposed.
- Analysis of this system has spurred a significant body of research on neural network verification in the formal methods community.

German Traffic Sign Recognition Benchmark (GTSRB):

- GTSRB (German Traffic Sign Recognition Benchmark) [191] is employed for machine learning and computer vision research, particularly in the realm of traffic sign recognition, featuring a wide array of real traffic signs in varied conditions in Germany.
- The GTSRB contains 43 classes of traffic signs, divided into 39,209 training images and 12,630 test images.
- The images have varying light conditions and rich backgrounds.

CIFAR-10 Dataset:

- Krizhevsky et al. [192] assembled a CIFAR-10 dataset consisting of ten classes with 6,000 images per class. These classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The standard train/test split is class-balanced, containing 50,000 training and 10,000 test images. The classes are designed to be completely mutually exclusive. For example, neither automobile nor truck contains images of pickup trucks.

Tiny ImageNet Dataset:

- The Tiny ImageNet dataset is a scaled-down version of the ImageNet dataset designed for academic and research purposes. It contains 200 classes, each with 500 training images, 50 validation images, and 50 test images, making it more manageable than the full ImageNet dataset. Each image is resized to 64×64 pixels, significantly smaller than the standard ImageNet images. Tiny ImageNet is often used for benchmarking and developing new machine-learning models due to its balance of complexity and size. It provides a good challenge for evaluating the performance of algorithms and architectures in image classification tasks.

At the same time, Table 4.1 describes the network structure of the different benchmarks and the number of neurons in columns so that the reader can grasp the difficulty of the verification problem. The output log of ESBMC also contains how many assignments the transformed SSA contains for an input C file, which can be seen as the complexity of a verification problem. As shown in Table 4.1, we have similarly tabulated the number of assignments for each verification property. The Iris benchmarks contain fewer assignments and require less time, while ACAS Xu contains more properties and assignments. Hence, it needs much more time to verify. We also give the average and maximum time to validate a property in this table.

Table 4.1. The baseline models. Parameters include the trainable and non-trainable parameters in the models; the unit is kilo (K). The two parameters are for the converted C file model, including the capacity of the C file and the number of assignments in the ESBMC procedure.

Model	Dataset	#Layers	#Params	#Benchmarks	Converted C file	
					Average C file capacity	ESBMC assignments
ACAS Xu	ACAS Xu	6	-	135	230KB	$\sim$ 7500
CIFAR-10-max-pooling	CIFAR-10	10	113K	24	580KB	$\sim$ 37900
GTSRB_small	GTSRB	8	320K	15	520KB	$\sim$ 13100
GTSRB_medium	GTSRB	8	844K	15	1300KB	$\sim$ 25800
GTSRB_big	GTSRB	8	1630K	15	2300KB	$\sim$ 41900

4.3.2 Experiments Setup

In this regard, we are mainly interested in the following experimental goals:

EG1 - Ablation study - Is it possible to establish the role of each employed enhancement technique and also define an optimal setup, both regarding verification time and performance?

EG2 - Quantization effects - How does a quantization choice influence our verification process and the safety of a neural network?

EG3 - Comparison with SOTA techniques - What is the performance of our verification approach when compared to the existing literature?

The experimental evaluation was conducted on an AMD EPYC 7T83 2.5 GHz 64-core Processor with 90 GB of RAM and Linux OS. We present the CPU execution times as measured with the `times` system call [193]. ESBMC v7.3.0. was configured to run without time or memory limits. Thus, timeouts are all due to exceedingly high memory consumption [1] The following command is used to run ESBMC unless specifically noted: `esbmc <file.c> -I <path-to-OM> --force-malloc-success --no-div-by-zero-check --no-pointer-check --boolector --k-induction-parallel --no-bounds-check --fixedbv`. The execution was run without time or memory limits.

In the above command, several options were used to configure the ESBMC execution:

Table 4.2. Command Line Options of ESBMC and Their Descriptions.

Option	**Description**
`--force-malloc-success`	do not check for malloc/new failure
`--no-div-by-zero-check`	do not check divide by zero
`--no-pointer-check`	do not do pointer check
`--k-induction-parallel`	prove by k-induction, running each step on a separate process
`--no-bounds-check`	do not do array bounds check
`--fixedbv`	encode floating-point as fixed bit-vectors

These options were chosen to disable certain runtime checks and to employ specific encoding and verification techniques. The aim was to streamline the verification process, reduce overhead,

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

Table 4.3. The time consumption (in seconds) in different phases of Bounded Model Checking (BMC) tasks using different solvers

	Symex	Encoding	Solver	Overall BMC time
Z3	1007	266	10	1283
Boolector	1012	275	4	1291
Bitwuzla	1011	286	20	1317
CVC5	1042	273	16743	18057

and leverage parallel processing capabilities where possible. By doing so, ESBMC can be ensured to perform efficiently and effectively under the given constraints.

4.3.3 SMT Solvers Comparison

As mentioned in Section 4.2, our approach relies on model checking to reason about the satisfiability of a safety property concerning a DNN implementation. For our experiments, we have chosen ESBMC as a verification engine, as it is an extensively evaluated tool with SOTA results [194]. It converts input C code into SMT formulae and then calls an external SMT solver. Currently, ESBMC supports three solvers by default: Boolector, Z3, and Bitwuzla, which has been introduced in Section 2.2 yield different verification results, both in terms of counterexample (if any) and verification time.

Here, we compare the performance of such solvers in verifying DNN implementations. We run them on our ACAS Xu benchmarks, with bit widths of 8. This way, we cover the most used quantization lengths and analyze the behavior of our methodology on a varied test suite. We use these experimental settings throughout our ablation study.

The runtime of ESBMC consists of several components: symbolic execution, encoding, and bounded model checking. These components are detailed in Section 4.2.5. Our examination of the ESBMC verification logs for the ACAS Xu benchmarks revealed interesting patterns in the time distribution across different solvers.

To provide a clearer picture, we calculated the average time each solver spent on symbolic execution, encoding, and bounded model checking. The results, summarized in Table 4.3, show distinct performance characteristics for each solver across the different processes.

The phases include Symbolic Execution (Symex), Encoding, Solving, and the Overall BMC time. The times for the Symex phase are quite close across different solvers. Z3 takes 1007 seconds, Boolector takes 1012 seconds, Bitwuzla takes 1010 seconds, and CVC5 takes 922 seconds. The Encoding phase times are also relatively close. Z3 takes 266 seconds, Boolector takes 275 seconds, Bitwuzla takes 286 seconds, and CVC5 takes 294 seconds. The Encoding phase times are also relatively close. Z3 takes 266 seconds, Boolector takes 275 seconds, Bitwuzla takes 286 seconds, and CVC5 takes 294 seconds.

It is evident that CVC5 spends a substantial amount of time in the Solver phase (16743 seconds), which greatly increases the overall BMC time. This suggests that CVC5 might be less efficient in handling certain types of problems or that these problems are particularly complex for CVC5. The other three solvers (Z3, Boolector, and Bitwuzla) have relatively short solving times, with Boolector being the fastest at just 4 seconds. This indicates that Boolector has a significant advantage in solving efficiency.

The main differences between the solvers are seen in the Solver phase efficiency, which is likely influenced by the solver's algorithms, implementation details, and optimizations for specific types of problems. CVC5 performs significantly worse in this phase, leading to a much higher overall BMC time compared to the other solvers.

The results of our comparison between Z3, Boolector and Bitwuzla are summarized in Figure 4.7. In this graph, I compared their performance on ACAS Xu Property 1,3 and 4, due to the same verification results of these three properties.

In comparison between Bitwuzla and Z3, for Property 1, most data points are below the diagonal line, indicating that Bitwuzla's verification time is generally shorter than Z3's. For property 3, while some points are above the diagonal line, the majority are still below it, indicating that Bitwuzla's verification time is slightly shorter than Z3's. For Property 4, the data points are clearly below the diagonal line, showing that Bitwuzla's verification time is significantly shorter than Z3's.

In comparison between Bitwuzla and Boolector, for Property 1, Bitwuzla's verification time is generally shorter than Boolector's, with data points close to the diagonal but slightly skewed towards the horizontal axis. For Property 3, Bitwuzla's verification time is slightly shorter than Boolector's, with a relatively even distribution. For Property 4, Bitwuzla's verification time is significantly shorter than Boolector's.

Noticed that L_s represents the amount of non-determinism in the source code, i.e., the set of possible values that program constraints can assume in the state space search. Thus, as the size of the L_s region grows, so does the state space search, leading to more complex formulae to be solved. This increase in complexity affects the performance of the SMT solvers, making the verification process more time-consuming and challenging.

Given the results in Figure 4.7, we chose Bitwuzla as our SMT solver for the rest of this experimental section. While it is impossible to know exactly why Bitwuzla is the best-performing solver on our test suite, we speculate it happens because 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, Bitwuzla represents

[2]`https://smtlib.cs.uiowa.edu/logics.shtml`

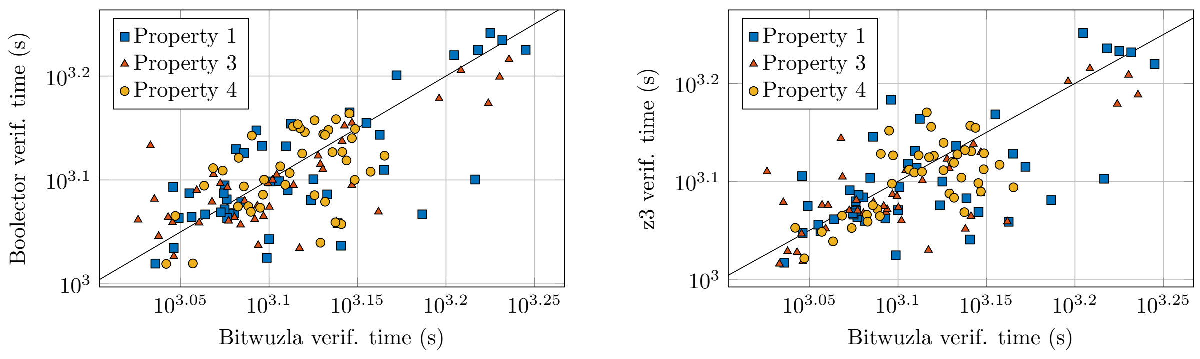


Figure 4.7. Comparison of verification times with different SMT solvers for the fixed-point ACAS Xu benchmarks: The unit of the verification time is seconds(s).

the SOTA SMT solver. [3]

The results presented here successfully answer **EG1**. We identified the best configuration of ESBMC within our framework, which consists of all the above techniques with solver Bitwuzla.

4.3.3.1 Effects of Quantization on Verification Time and Memory

First, the relation between DNN quantization and verification time will be tackled. Indeed, verifying quantized neural networks is PSPACE-hard [195]. However, this is theoretical, and empirical results provided by Giacobbe *et al.* [71] show a positive correlation between the number of bits and verification time. In this study, we conducted a comprehensive evaluation of the GTSRB benchmark by performing experiments using quantized bit-widths ranging from 2 bits to 32 bits, focusing specifically on even bit-width values (e.g., 2, 4, 6, ..., 32). The goal of this evaluation was to assess how different quantization bit-widths influence execution time and memory usage during the neural network's verification and inference stages. For each quantization setting, we measured and recorded the computational performance, allowing us to observe trends and trade-offs as the bit-width increases.

In addition to quantized models, we also ran the GTSRB floating-point model as a baseline for comparison. The floating-point model represents the standard full-precision setting, which allows us to directly compare the performance of quantized models against the full-precision counterpart. By comparing these results, we aim to determine whether lower-bit quantized models can offer significant computational advantages, such as reduced memory usage and faster execution, while maintaining acceptable accuracy and model behavior, or if higher precision (e.g., floating-point) is necessary for more robust results. Because of the long verification time required for the floating-point model, we set the time limit for verifying the floating-point model

[3] `https://smt-comp.github.io/2020/results/qf-aufbv-single-query`

to 36,000s and do not set a memory cap (the physical memory limit is 90GB) The physical memory is 90GB and the verification task for a single floating-point model takes up all the CPU time.

Fig. 4.8 illustrates the performance differences between quantized neural networks with varying bit widths and floating-point neural networks. Specifically, `time_30`, `time_48`, and `time_64` represent the verification times corresponding to resolutions of 30x30, 48x48, and 64x64 on the GTSRB benchmark, respectively. Similarly, `max_mem_res30`, `max_mem_res48`, and `max_mem_res64` indicate the maximum memory usage for the same resolutions on the GTSRB benchmark.

As shown in Fig. 4.8 (a), in terms of memory usage, the floating-point model consistently consumes slightly more memory compared to both the maximum and average values of the quantized models. This discrepancy becomes particularly noticeable when considering the floating-point model at a resolution of 64x64, where its memory footprint significantly exceeds the average memory usage of the quantized model. Moreover, in the worst-case scenario, the memory consumption of the floating-point model is even larger than its own typical memory footprint, making it noticeably more resource-intensive than the quantized versions, especially at higher resolutions.

Turning to time usage, as highlighted in Fig. 4.8 (b), the floating-point model requires significantly more validation time compared to the quantized models across all resolutions. This time difference is evident for `time_30`, `time_48`, and `time_64`, where the floating-point model consistently takes much longer to validate than even the maximum time required by the quantized models. At each resolution level, the floating-point model's validation time far exceeds that of the quantized models, underscoring its higher computational demands.

Furthermore, Fig. 4.8 (c) demonstrates a gradual yet steady increase in memory usage as the digit size grows for most memory results, particularly for `res30` and `res48`. However, `res64` stands out by maintaining a nearly constant high value across all digit sizes, indicating that the memory requirement for `res64` is significantly higher and less sensitive to changes in digit size compared to the other resolutions.

Lastly, Fig. 4.8 (d) reveals that while processing times remain relatively stable for each resolution, there are clear distinctions in the time required for `time_30`, `time_48`, and `time_64`. Notably, `time_64` takes considerably longer than the other two, with the gap becoming increasingly pronounced as the resolution increases. This suggests that higher resolutions impose a much greater computational load, particularly for `time_64`, further emphasizing the computational advantages of using quantized models at lower resolutions.

In the experimental results of the fully quantized neural network, both the maximum memory usage and verification time show only a gradual increase as the network scales. Therefore, for

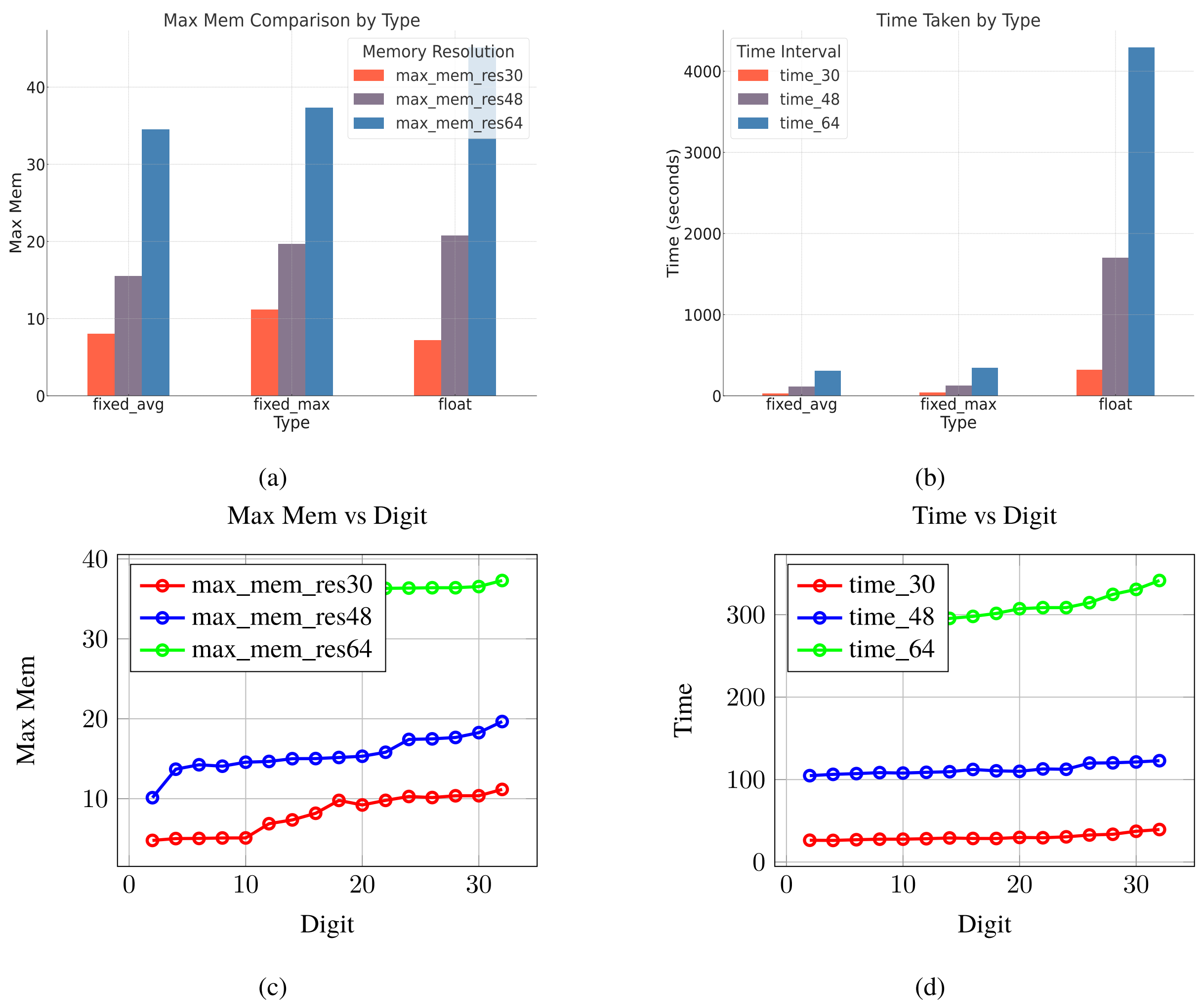


Figure 4.8. Comparison of average case, worst case time spent and maximum memory usage of quantized networks with floating point networks, and verification time vs max memory usage for different quantization bit-widths: from 2-bit to 32-bit.

our practical verification, we opted to use the int8 data type for the quantized neural network. This choice aligns with common industry practices, as int8 is the most widely used bit width in real-world applications due to its balance between performance and resource efficiency.

Notice that the security property of robustness of the validation neural network in the face of small perturbations of the inputs is fundamentally different from the ACAS Xu benchmark, which is usually performed for small perturbations in the input space. Whereas input perturbations are usually very localized, i.e., restricted to a small region (e.g., by setting an epsilon value to represent the range of input perturbations), ACAS Xu is a much more complex system containing multiple inputs, outputs, and rules. These systems are designed for real-time decision-making to avoid aircraft collisions. Their safety validation usually involves complex multidimensional spatial analysis and needs to consider the dynamic behavior of the system (e.g., relative speed, position, and orientation of multiple aircraft, etc.), which significantly increases the size and complexity of the problem, and therefore it requires a shorter validation time than ACAS Xu and can give a counterexample very quickly.

These results answer **EG2**: the verification time and memory have some correlation with the number of bits used for DNN quantization and the scale of DNN. Moreover, the safety of a DNN 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.

4.3.4 Comparison with State-of-the-art Verification Tools

This section compares *QNNVerifier* with existing studies. Related work is progressing rapidly, but only a few available tools exist. Besides, the approaches for verifying QNNs [71], [196]–[198]) do not always provide means for replicating their experiments or performing the comparisons. As such, *QNNVerifier* can only be compared with earlier tools that verify DNN safety as abstract mathematical models in infinite precision. Among those, the three most popular ones were chosen:

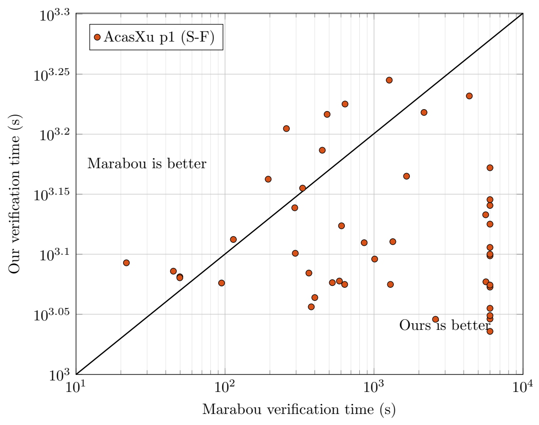
- **Marabou** [146]. Based on the Reluplex [112], Marabou uses a simplex-like algorithm to split the verification problem into smaller subproblems and invoke an SMT solver on each of them. It also has an option to use LP relaxation for bound tightening, which uses Gurobi as an LP solver.
- **ERAN** [199]. It combines abstract domains with custom multi-neuron relaxations to support fully connected, convolutional, and residual networks with ReLU, Sigmoid, Tanh, and Maxpool activations.
- **Alpha-Beta-CROWN** [134]. It uses a new bound propagation-based method that can fully encode neuron splits via optimizable parameters constructed from either primal or dual space.
- **NeuralSAT** [159]. NeuralSAT is a deep neural network (DNN) verification tool. It integrates the DPLL(T) [200] approach commonly used in SMT solving with a theory solver specialized for DNN reasoning. NeuralSAT exploits multicores and GPU for efficiency and can scale to networks with millions of parameters. It also supports a wide range of neural networks and activation functions.

The goal is to show that our quantized methodology is at least as efficient as these SOTA tools. Besides, *QNNVerifier* provides more information on the safety of actual DNN implementations than the abstract safety guarantees provided by Marabou and Alpha-Beta-Crown.

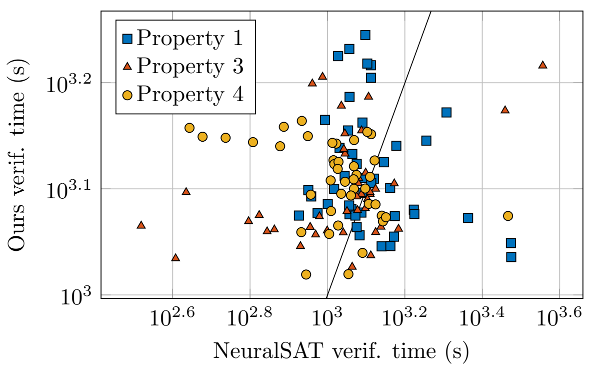
The ACAS Xu Property 1,3,4 benchmark was chosen as a comparison suite (see Section 4.3.1) for property verification, which has the advantage of being already implemented in Marabou, NeuralSAT, and Alpha-beta-crown,[4] [5] [6]thus allowing a fair performance comparison.

The results regarding verification time are shown in Figure 4.9. *QNNVerifier* was compared to our methodology with the SMT-based tool Marabou and the DPLL(T)-based SAT backends tool NeuralSAT. We believe this is because the ESBMC is more efficient at producing optimized SMT formulae (see Section 4.2.6) than the custom simplex-like method employed by Marabou [146]. This also explains why the verification times of our methodology are almost constant across the whole comparison suite.

In the comparison, since NeuralSAT runs authentication using all cores of the CPU by default, NeuralSAT was modified by limiting the number of CPU cores to force it to run authentication using a single core to ensure that it is comparable to ESBMC running on a single core.



Comparison with Marabou



Comparison with NeuralSAT

Figure 4.9. Comparison of verification time among our methodology and SOTA tools on Property 1 of ACAS Xu. The unit of the verification time is seconds (s).

In Figure 4.9, the left plot is the comparison of the verification time between *QNNVerifier* and Marabou. Many of the cases that validate Marabou are timeouts (we set the timeout to 3600 seconds), so we place these points on the far right of the x-axis. Most points lie below the diagonal, indicating that our method generally performs faster than Marabou for most test cases. This demonstrates the efficiency of our verification technique in comparison with the existing tool. A few points are above the diagonal line, indicating that Marabou is faster in some cases. The spread of points below the diagonal suggests significant efficiency gains in our method. For instance, in several cases, the verification time is reduced by an order of magnitude or more. As the verification time increases (towards the right side of the plot), our method shows consistent performance improvements, highlighting its scalability for more complex verification tasks. The comparison clearly illustrates the advantages of *QNNVerifier* in terms of speed and efficiency.

The right plot provides a detailed comparison of the verification times between our method

[4]`https://github.com/NeuralNetworkVerification/Marabou`
[5]`https://github.com/dynaroars/neuralsat`
[6]`https://github.com/Verified-Intelligence/alpha-beta-CROWN`

and NeuralSAT across different properties of the ACAS Xu model. The points are spread around the diagonal line, suggesting that for many test cases, both our method and NeuralSAT have similar verification times. However, there are notable differences for specific properties and test cases. In Property 1, Many blue squares lie close to or just above the diagonal line, indicating that our method performs comparably to NeuralSAT for Property 1. Since our advantage over NeuralSAT on Property 1 is not really significant, we added a comparison experiment for Properties 2 and 3. A few points below the diagonal showcase where our method is faster. Brown triangles are more dispersed, with several points below the diagonal line, indicating that our method is generally faster for Property 3. There are fewer points above the diagonal, suggesting that in most cases, our method outperforms NeuralSAT for this property. Yellow circles also show a spread around the diagonal line, with a notable number of points below the diagonal. This indicates that for Property 4, our method tends to be faster than NeuralSAT in many test cases. Some points lie close to the diagonal, suggesting similar performance in those cases.

Our method exhibits competitive verification times across different properties compared to NeuralSAT. For Property 3 and Property 4, our method generally performs better, showing faster verification times in many test cases. Property 1 shows comparable performance, with our method being slightly faster in several cases.

We also compared the counterexample generated by our tool with the counterexample generated by alpha-beta-crown, as shown in Figure 4.10. Since Marabou is not scalable for GTSRB benchmarks and NeuralSAT does not give the counterexample when verification failed. So, for counterexample generation, we chose the GTSRB dataset and alpha-beta-crown as the comparison suite. The original figure, the counterexample generated by our tool as well as the one generated by alpha-beta-crown can be recognized by humans as a turnpike sign, but the counterexample generated by our tool possesses less noise and the image is clearer.

In the top row, we use Property from the GTSRB dataset, 08258.jpg, which is the image with the blue circular road sign with the white arrow pointing to the left. The original is clear and crisp, and the white arrow on the blue background is very visible. The image in the middle is a counterexample created by our tool, which decreases in clarity and the arrow becomes more distorted and less recognizable. But when we look at the third image, the counterexample created by alpha-beta-crown, the arrow appears fragmented and there is a lot of noise in this image.

The center row has three circular road signs indicating a speed limit of “40” from 07040.jpg in the GTSRB dataset. The original image on the left shows a clearly legible white “40” placed within a black circle with a white border. The “40” on the center image is slightly blurred, a counterexample created by our method. And the image on the far right, from alpha-beta-crown, is so badly distorted that the numbers are barely legible.

The bottom row shows the “100” speed limit sign, which comes from 11985.jpg in the GTSRB dataset, and the middle-most image in the row, from a counterexample created by our

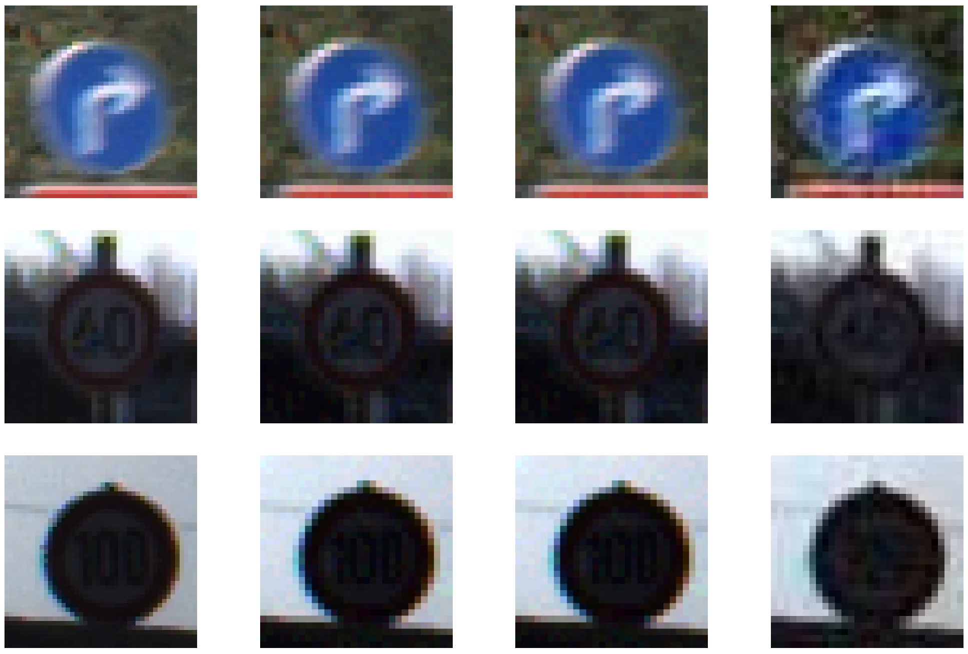


Figure 4.10. Enlarged counterexamples for the GTSRB benchmarks 30 $\times$ 30 pixels image resolutions generated by our method (with and without approximation) and alpha-beta-crown, compared with the original image.

method, blurs and distorts the sign and makes the background color lighter. The third image, from alpha-beta-crown, is the blurriest version of the number, with the 100-speed limit sign barely recognizable and with a lot more noise in the background.

These results successfully answer **EG3** and **Research Question** 1: We evaluated and compared our tool with SOTA ones, including Marabou (SMT-based), Alpha-beta-CROWN (based on Linear Programming and Branch-and-Bound), and NeuralSAT (based on DPLL(T)). Our approach can successfully verify all the benchmarks without timeout or crash. Furthermore, considering the Property 1 of ACAS Xu, our approach is significantly faster and solves more verification tasks than Marabou; considering the Property 2 and 3, our approach is slightly better than NeuralSAT.

4.4 Chapter Summary

Through these efforts, we aim to provide a robust, scalable, and reliable framework for neural network verification that can be adapted to the evolving landscape of deep learning technologies.

Verification of DNNs has recently attracted considerable attention, with notable approaches using optimization, reachability, and satisfiability methods. While the former two promise 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.

We propose an SMT approach to address DNN verification. More specifically, we view the DNN 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. With this view, we can borrow several software verification techniques and seamlessly apply them to DNN verification. In this regard, we center our verification framework around Software Model Checking (SMC) and empirically show the importance of interval analysis, constant folding, slicing, and expression simplifications in reducing the total verification time. Furthermore, we propose a tailored discretization technique for non-piecewise-linear activation functions that allow us to verify DNNs beyond the piecewise-linear assumptions that many state-of-the-art methods are restricted to.

Finally, the problem of verifying DNNs 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 approach to other existing techniques and optimizing our verification performance even further. In addition, our work can be regarded as the first step towards an approach capable of revealing the most aggressive DNN representation that still provides correct operation, aiming to achieve maximum compression for a particular model.

In our future work, we aim to significantly enhance the scalability of our technology, addressing both the computational efficiency and the broader applicability of our methods. Here is an expanded view of our planned initiatives:

1. **Engineering improvements:** We intend to refine the current *QNNVerifier* implementation by optimizing algorithms and streamlining data structures to handle larger datasets and more complex neural network models. This involves reevaluating the computational bottlenecks and memory usage to ensure efficient resource management and faster processing times. Especially during the code conversion procedures, the C file converter may generate redundant code, and the C code needs to be optimized.
2. **Soundness guarantees:** Enhancing soundness guarantees is crucial for the reliability of *QNNVerifier*. We plan to develop mechanisms that provide stronger soundness assurances without adversely affecting performance, including generating checkable proof objects that can be independently verified, offering an extra layer of verification for the correctness of analyses.
3. **Proof of correctness:** We aim to generate externally checkable proofs of correctness that can be verified by third-party tools or manual inspection. This will increase the verification process's trustworthiness and foster transparency and reproducibility in neural network verification research.
4. **Handling diverse DNN architectures:** Finally, We plan to extend our approach to accommodate DNNs with various layers, such as convolutional, recurrent, and residual layers.

This extension will involve developing new abstraction and verification techniques suitable for these layers' unique characteristics, thereby broadening *QNNVerifier* to more complex and state-of-the-art neural network architectures.

Chapter 5

QNNRepair: Quantized Neural Network Repair

This chapter introduces the *QNNRepair* implementation and the evaluation of *QNNRepair*. Firstly, it presents the concept, application, and security concerns of neural networks. Then, it includes the *QNNRepair* core methodology: neural importance ranking, constrains-solving based repairing. After that, the *QNNRepair* algorithm is described, and *QNNRepair* was tested on popular benchmarks and compared *QNNRepair* with the state-of-the-art methods.

5.1 Chapter Introduction

Chapter 4 introduces the neural network verification tool *QNNVerifier*, which aims to identify vulnerabilities in trained models. While this tool is effective at uncovering weaknesses, the question remains: how do we address and rectify these vulnerabilities once they are found? This chapter introduces the neural network model repairing tool, *QNNRepair*, explicitly designed to address this challenge.

QNNRepair provides a systematic approach to fixing identified issues in neural networks. By integrating *QNNVerifier* and *QNNRepair*, we create a comprehensive framework that detects and remedies faults in neural networks, ensuring improved performance and reliability. The repair tool analyzes the detected vulnerabilities and applies targeted modifications to the model parameters or architecture to mitigate these issues.

To make the repair process effective, *QNNRepair* employs techniques such as fine-tuning, retraining specific layers, or altering the network structure where necessary. These methods are chosen based on the type and severity of the vulnerabilities identified by *QNNVerifier*. For example, if a vulnerability is due to an overfitting issue, *QNNRepair* might implement regularization techniques, including dropout, which randomly disables neurons during training to reduce co-adaptation or adjust the training data to enhance the model's generalizability.

Additionally, *QNNRepair* is designed with usability in mind, featuring an intuitive interface that guides users through the repair process. This interface provides clear instructions and visualizations to help users understand the changes being made to the model. The goal is to make the tool accessible to both experts and non-specialists, allowing a wider audience to benefit from its capabilities.

Another critical aspect addressed by *QNNRepair* is compatibility with models trained on different platforms and stored in various formats. To handle this, the tool includes a robust model conversion mechanism that supports popular frameworks such as TensorFlow, PyTorch, ONNX, and Keras. This ensures that models can be seamlessly imported into *QNNRepair*, processed, and then exported back to their original or preferred format.

After introducing *QNNVerifier*, this chapter describes a series of evaluations to validate its effectiveness. This includes an ablation study to identify the optimal parameter combinations for *QNNRepair*, ensuring the best performance. The ablation study systematically examines the impact of different parameters on the tool's performance, helping to fine-tune *QNNRepair* for various types of neural network models and use cases. By identifying the best configurations, *QNNRepair* can maximize the accuracy and efficiency of the repair process.

Additionally, this chapter compares *QNNRepair* with state-of-the-art (SOTA) neural network verification tools. This comparison involves rigorous benchmarking against existing tools to highlight *QNNRepair*'s advantages. Key metrics such as accuracy, speed, scalability, and robustness are evaluated to provide a comprehensive assessment of *QNNRepair*'s capabilities. The results of these comparisons demonstrate the efficacy of *QNNRepair* in various scenarios, showcasing its ability to not only detect but also effectively repair vulnerabilities in neural network models.

These evaluations underscore the practical benefits of *QNNRepair*, illustrating how it can be integrated into existing workflows to enhance the reliability and security of neural networks. By providing concrete evidence of its performance and advantages, these studies build a strong case for the adoption of *QNNRepair* in both academic research and industry applications.

5.2 *QNNRepair* Methodology

The overall workflow of *QNNRepair* is illustrated in Figure 5.1. It takes two neural networks, a floating-point model and its quantized version for repair, as inputs. There is also a repair dataset

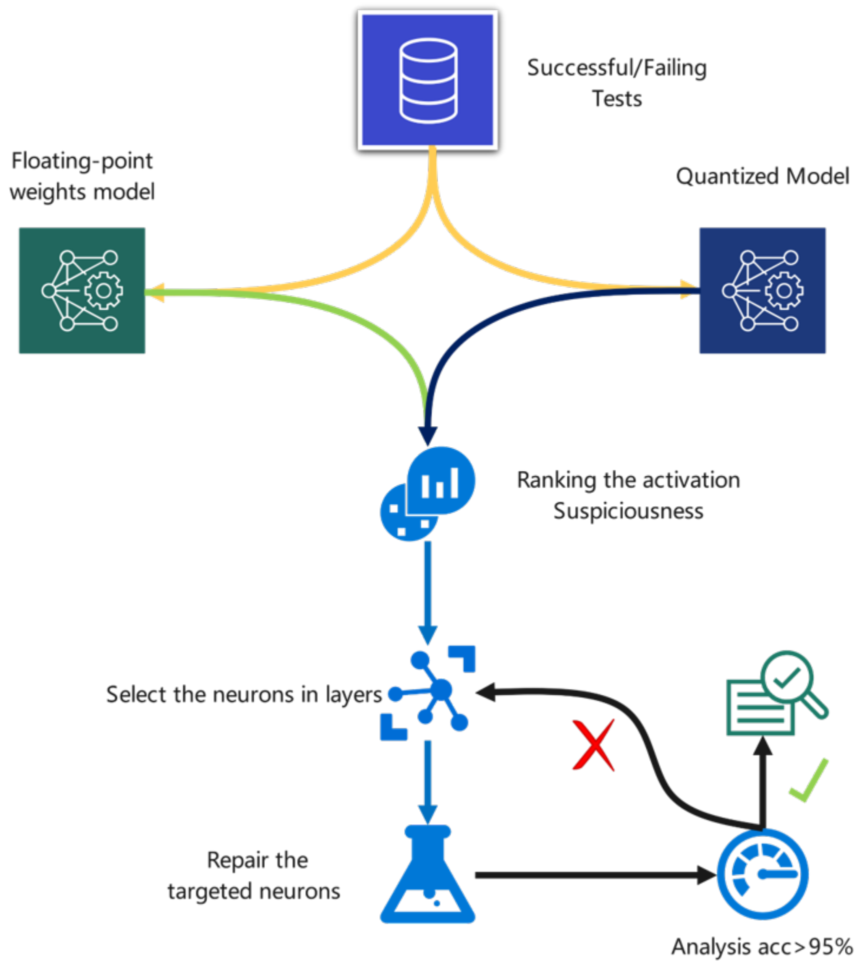

Successful/Failing Tests
Floating-point weights model
Quantized Model
Ranking the activation Suspiciousness
Select the neurons in layers
Repair the targeted neurons
Analysis acc>95%

Figure 5.1. The *QNNRepair* Architecture. In order to obtain the information that the quantized neural network is missing after quantization we need the original floating point neural network, as well as to localize the faults by Successful/Failing the dataset. The output of this method is the quantized neural network with high accuracy.

of successful (passing) and failing tests, signifying whether the two models would produce the same classification outcome when given the same test input.

Next, this chapter thoroughly describes these procedures in the above graph. The process begins with the initial inputs, comprising the floating-point and quantized neural networks, alongside a repair dataset that categorizes tests into successful (passing) and failing based on the models' classification outcomes. This dataset is crucial as it helps identify discrepancies in the outputs of the two models, serving as the basis for the repair process.

5.2.1 Neuron Importance Ranking

QNNRepair starts with evaluating the importance of the neurons in the neural network for causing the output difference between the quantized model and the floating point one. When conducting an inference procedure on an image, the intermediate layer in the model has a series of outputs as the inputs for the next layer. The outputs go through activation functions, and we assume that they are ReLU functions. For the output, if it is positive, we place it at one. If not, we place it at zero and name it the activation output. Let f_i and q_i represent the activation output of a single neuron in full-precision and quantized models separately. If there is a testing image that makes (f_i, q_i) not equal, we consider the neuron as "activated", and we set $v_{mn} = 1$, otherwise $v_{mn} = 0$. Then we define the activation function matrix to assemble the activation status of all neurons for the floating-point model:

$$\begin{pmatrix} f_{11} & \cdots & f_{1n} \\ \vdots & \ddots & \vdots \\ f_{m1} & \cdots & f_{mn} \end{pmatrix} = f_i \text{ and } q_i \text{ for the quantized model.}$$

QNNRepair defines the activation differential matrix to evaluate the activation difference between the floating point and the quantized model. Given an input image i, *QNNRepair* calculates $\mathrm{diff}_i = f_i - q_i$ between the two models. *QNNRepair* forms a large matrix of these $diff_i$ regarding the image i. The element in this matrix should be 0 or 1, representing whether the floating and quantized neural networks' activation status is the same.

QNNRepair borrows concepts from traditional software engineering, replacing the statements in traditional software with neurons in neural network models. In this context, passing tests are defined as images in the repair set where the floating-point and quantized models produce the same classification output while failing tests are those where their classification results differ. For a set of repair images, we define $< C_n^{\mathrm{af}}, C_n^{\mathrm{nf}}, C_n^{\mathrm{as}}, C_n^{\mathrm{ns}} >$ as follows:

- C_n^{af} is the number of "activated" neurons for failing tests.
- C_n^{nf} is the number of "not activated" neurons for failing tests.
- C_n^{as} is the number of "activated" neurons for passing tests.
- C_n^{ns} is the number of "not activated" neurons for passing tests.

QNNRepair also draws from traditional software fault localization methods, such as Tarantula [201], Ochiai [202], DStar [203], Jaccard [204], Ample [205], Euclid [206], and Wong3 [207], to define indicators of neuronal suspicion. These indicators are summarized in Table 5.1. Note that in DStar, * represents the nth power of C_n^{af}.

QNNRepair then ranks the quantitative metrics of these neurons from largest to smallest based on certain weights, with higher metrics indicating more suspicious neurons and the ones *QNNRepair* needs to target for repair. This ranking helps us prioritize which neurons to address first, ensuring that the most likely sources of errors are tackled early in the repair process.

Table 5.1. Importance (i.e., fault localization) metrics used in experiments.

Tarantula:	$\frac{C_n^{\text{af}}/\left(C_n^{\text{af}}+C_n^{\text{nf}}\right)}{C_n^{\text{af}}/(C_n^{\text{af}}+C_n^{\text{nf}})+C_n^{\text{as}}/(C_n^{\text{as}}+C_n^{\text{ns}})}$	Euclid:	$\sqrt{C_n^{\text{af}}+C_n^{\text{ns}}}$
Ochiai:	$\frac{C_n^{\text{af}}}{\sqrt{(C_n^{\text{af}}+C_n^{\text{as}})(C_n^{\text{af}}+C_n^{\text{nf}})}}$	DStar:	$\frac{C_n^{\text{af}^*}}{C_n^{\text{as}}+C_n^{\text{nf}}}$
Ample:	$\left\lvert\frac{C_n^{\text{af}}}{C_n^{\text{af}}+C_n^{\text{nf}}}-\frac{C_n^{\text{as}}}{C_n^{\text{as}}+C_n^{\text{ns}}}\right\rvert$	Jaccard:	$\frac{C_n^{af}}{C_n^{af}+C_n^{nf}+C_n^{as}}$
Wong3:	$C_n^{\text{af}}-h \quad h=\begin{cases} C_n^{as} & \text{if } C_n^{as}\le 2 \\ 2+0.1\,(C_n^{as}-2) & \text{if } 2<C_n^{as}\le 10 \\ 2.8+0.01\,(C_n^{as}-10) & \text{if } C_n^{as}>10 \end{cases}$		

5.2.2 Constraints-Solving Based Repairing

After the neuron importance evaluation, *QNNRepair* obtains a vector of neuron importance for each layer and ranks this importance vector. The neuron with the highest importance is our target for repair, as it could have the most significant impact on the corrected error outcome. This step is crucial because it allows us to focus our repair efforts on the most influential neurons, thereby increasing the efficiency and effectiveness of the repair process.

The optimization problem for a single neuron can be described as follows:

Definition 5.2.1 (Optimization Problem for Neural Network Repair)**.**

Minimize: M

Subject to:

$$M \geq 0$$

$$\delta_i \in [-M, M] \quad \forall i \in \{1, 2, \ldots, n\}$$

If floating model gives the result 1 and quantized model gives 0: (5.1)

$$\forall x_i \text{ in TestSet } X : \sum_{i=1}^{m} w_i x_i < 0 \text{ and } \sum_{i=1}^{m} (w_i + \delta_i) x_i > 0$$

If floating model gives the result 0 and quantized model gives 1:

$$\forall x_i \text{ in TestSet } X : \sum_{i=1}^{m} w_i x_i > 0 \text{ and } \sum_{i=1}^{m} (w_i + \delta_i) x_i < 0$$

In the formula, m represents the number of neurons connected to the previous layer of the selected neuron, and we number them from 1 to m. We add incremental δ to the weights to indicate the weights that need to be modified. The solutions for the symbolic δ's obtained from the solver guarantee that all the inputs (both passing and failing) satisfy the pattern and are thus likely to be classified as C by the network. These solutions are then used to update the weights of the network. M is used to make $\delta_1...\delta_i$ sufficiently small. The value $\delta_1...\delta_i$ are encoded as the non-deterministic variables, and our task is to use Gurobi to solve these non-deterministic based on the given constraints.

QNNRepair assumes that in the full-precision neural network, this neuron's activation function gives the result 1, and the quantized gives 0. The corrected neuron in the quantized model result needs to be greater than 0 for the output of the activation function to be 1. If in the full-precision neural network, this neuron's activation function gives the result 0, and the quantized gives 1. The corrected neuron in the quantized model result needs to be smaller than 0 for the output of the activation function to be 0. In this case, we make the distance of the repaired quantized neural network as close as possible to that of the original quantized neural network.

QNNRepair's inputs are a quantized neural network Q that needs to be repaired, a set of data sets X for testing, and the full-precision neuron network model F to be repaired. *QNNRepair* uses Gurobi [121] as the constraint solver to solve the constraint and then replace the original weights with the result obtained as the new weights.

5.2.3 *QNNRepair* Algorothm

QNNRepair is formulated in Algorithm 1. The input to the algorithm is the full-precision model F, the quantized model Q. The repair set X, the validation set V, and the number of neurons that need to be repaired N (Line 1). Firstly, *QNNRepair* initializes arrays to store the activation states of the floating and quantized model, the values of the neuron importance, and four arrays $C^{as}[]$, $C^{af}[]$, $C^{ns}[]$ and $C^{nf}[]$ mentioned in Section 5.2.1. For these six arrays, *QNNRepair* sets all elements to 0.

Next, in lines 3-4, for each input in the test set $x \in \mathcal{X}_n$, *QNNRepair* performs the inference process once it obtains the neurons' activation states in the corresponding model layers and stores them in the activation states of the floating and quantized model. In line 5, if $x[i]$ is a failing test, then we add the difference of activation status between the float model and quantized model to $C^{as}[i]$, and vice versa. In line 11 and 12, we calculate $C^{ns}[]$ and $C^{nf}[]$ according to the definition in Section 5.2.1. *QNNRepair* calculates the importance (this section uses DStar as an example) for each neuron regarding seven importance metrics and sort them in descending order, then store them in set $I_n[]$ in line 14.

Then, *QNNRepair* picks the neuron in $I_n[]$, according to the neuron's weights and the corresponding inputs from the previous layer, we create and solve the LP problem we discussed in Section 5.2.2, get the correction of each neuron, and update their weights. When it arrives at the maximum number of neurons to repair, the loop breaks and we correct all the neurons. These are implemented at lines 17-24 in Algorithm 1.

Algorithm 1: Repair algorithm

```
Input: Floating-point model $F$, Quantized model $Q$, Repair set $X$, Validation set $V$, Number
       of neurons to be repaired $N$
Output: Repaired model $Q'$, Repaired model's accuracy $Acc$
1  Initialize $F_a[][], Q_a[][], I_n[], C^{as}[], C^{af}[], C^{ns}[], C^{nf}[]$
2  foreach $X$ do
3      $F_a[][i] = \text{getActStatus}(F, x_i)$
4      $Q_a[][i] = \text{getActStatus}(Q, x_i)$
5      if $x[i]$ is a failing test then
6          $C^{af}[i] = C^{af}[i] + |F_a[][i] - Q_a[][i]|$
7      else
8          $C^{as}[i] = C^{as}[i] + |F_a[][i] - Q_a[][i]|$
9      end
10 end
11 $C^{nf}[] = C^{nf}[] - C^{af}[]$
12 $C^{ns}[] = C^{ns}[] - C^{as}[]$
13 $I_n[] = \text{DStar}(C^{as}_n[], C^{as}_n[], C^{as}_n[], C^{as}_n[])$
14 $I_n[] = \text{sort}(I_n[])$ // In descending order
15 Initialize weight of neurons $w[][]$ and the increment $\delta[][]$
16 foreach $neuron[i] \in I_n[]$ do
17     foreach $edge[j][i] \in neuron[i]$ do
18         $w[j][i] = \text{getWeight}(edge[j][i])$
19     end
20     $\delta[][i] = \text{solve}(X, w[][i])$ // Solve LP problem 5.1
21     foreach $edge[j][i] \in neuron[i]$ do
22         $edge[j][i] = \text{setWeight}(w[j][i] + \delta[j][i])$
23         $Q' = \text{update}(Q, edge[j][i])$
24     end
25     if $i >= N$ then
26         break
27     end
28 end
29 $Acc = \text{calculateAcc}(Q', V)$
30 return $Q'$
```

5.3 Evaluation

Finally, this chapter evaluates the classification accuracy of the corrected quantized model. If it satisfies the requirements, then the model is repaired. Otherwise, try other combinations of parameters like important metrics or the maximum number of neurons needed to repair and repeat the LP solving and correction process. The output for this algorithm is the repaired model with updated weight.

5.3.1 Description of the *QNNRepair* Benchmarks

In evaluation, the full-precision MobileNetV2 benchmarks are directly obtained from the Keras library, whereas we trained the VGGNet [4] and ResNet-18 models on the CIFAR-10 dataset.

Table 5.2. The baseline models. Parameters include the model's trainable and non-trainable parameters; the unit is million (M). The two accuracy values are for the original floating point model and its quantized version, respectively.

Model	Dataset	#Layers	#Params	Accuracy	
				floating point	quantized
Conv3	CIFAR-10	6	1.0M	66.48%	66.20%
Conv5	CIFAR-10	12	2.6M	72.90%	72.64%
VGGNet	CIFAR-10	45	9.0M	78.67%	78.57%
ResNet-18	CIFAR-10	69	11.2M	79.32%	79.16%
MobileNetV2	ImageNet	156	3.5M	71.80%	65.86%

We also defined and trained two smaller convolutional neural networks on CIFAR-10 for comparison: Conv3, which contains three convolutional layers, and Conv5, which contains five convolutional layers. Both models have two dense layers at the end. The quantized models are generated by using TensorFlow Lite (TFlite) [208] from the floating point models. In TFLite, we chose dynamic range quantization, and the weights are quantized as 8-bit integers. The quantized convolution operation is optimized for performance, and the calculations are done in the fixed-point arithmetic domain to avoid the overhead of de-quantizing and re-quantizing tensors.

A subset of ImageNet called ImageNet-mini [209] is also used for evaluation, which contains 38,668 images in 1,000 classes for repairs of the quantized model's performance. The dataset is divided into the repair set and the validation set. The repair set contains 34,745 images, and the validation set contains 3,923 images. The CIFAR-10 dataset contains 60,000 images in 10 classes in total. 50,000 of them are training images, and 10,000 of them are test sets. 1,000 images are used as the repair set. The repair set are used to identify suspicious neurons, generate LP constraints, apply corrections to the identified neurons, and use the validation set to evaluate the accuracy of the models. The same experiment was repeated ten times for random neuron selection and to get the average to eliminate randomness in repair methods.

5.3.2 Experiments Setup

We are mainly interested in the following experimental goals:

EG1 - Fault Localization - How does a fault localization choice influence our repair process and performance?

EG2 - Repair Efficiency - What is the performance when repairing a model?

EG3 - Comparison with the SOTA - What is the performance of our repair approach when compared to the existing literature?

The experiments are conducted on a machine with Ubuntu 18.04.6 LTS OS Intel(R) Xeon(R) Gold 5217 CPU @ 3.00GHz and two Nvidia Quadro RTX 6000 GPUs. The experiments are

Table 5.3. *QNNRepair* results on CIFAR-10 models. The best repair outcome for each model, w.r.t. the dense layer in that row, is in **bold**. We further highlight the best result in $\boxed{blue}$ if the repair result is even better than the floating point model and in red if the repair result is worse than the original quantized model. Random means that *QNNRepair* randomly selects neurons at the corresponding dense layer for the repair, whereas Fault Localization refers to the selection of neurons based on important metrics in *QNNRepair*. In All cases, all neurons in that layer are used for repair. 'n/a' happens when the number of neurons in the repair is less than 100, and '-' is for repairing the last dense layer of 10 neurons, and the result is the same as the All case.

	Random				Fault Localization				-
#Neurons repaired	1	5	10	100	1	5	10	100	All
Conv3_dense-2	63.43%	64.74%	38.90%	n/a	66.26%	**66.36**%	62.35%	n/a	57.00%
Conv3_dense-1	65.23%	66.31%	-	n/a	66.10%	66.39%	-	n/a	**66.46**%
Conv5_dense-2	72.49%	72.55%	72.52%	72.52%	**72.56**%	72.56%	72.56%	72.56%	72.54%
Conv5_dense-1	72.51%	72.52%	-	n/a	**72.58**%	72.56%	-	n/a	72.56%
VGGNet_dense-3	78.13%	78.44%	78.20%	78.38%	$\boxed{\textbf{78.83}\%}$	78.82%	78.78%	78.66%	78.60%
VGGNet_dense-2	78.36%	78.59%	78.44%	78.22%	78.55%	$\boxed{\textbf{78.83}\%}$	78.83%	78.83%	78.83%
VGGNet_dense-1	78.94%	67.75%	-	n/a	$\boxed{\textbf{79.29}\%}$	69.04%	-	n/a	74.49%
ResNet_dense_1	78.90%	78.92%	-	n/a	79.08%	**79.20**%	-	n/a	78.17%

run with TensorFlow2 + nVidia CUDA platform. We use the Gurobi [121] as the linear program solver and enable multi-thread solving (up to 16 cores). *QNNRepair* was applied to repair a benchmark of five quantized neural network models, including MobileNetV2 [63] on ImageNet datasets [79], and ResNet-18 [62], VGGNet [4] and two simple convolutional models trained on CIFAR-10 dataset [192].

5.3.3 Repair Results on Baselines

In this part, *QNNRepair* was applied to these baseline quantized models, except for MobileNetV2, in Table 5.3. In our experiments, MobileNetV2 is trained on ImageNet while other models are trained on CIFAR-10, and it contains more layers. The results for MobileNetV2 are reported in Section 5.3.5.

For each model, *QNNRepair* performs a layer-by-layer repair of its last dense layers. These dense layers are named dense-3 (the third last layer), dense-2 (the second last layer), and dense-1 (the output layer).

The *QNNRepair* results are reported in Table 5.3. *QNNRepair* ranked the neurons using important metrics and chose the best results among the seven metrics. The evaluation also ran randomly picked repairing as a comparison. *QNNRepair* has chosen Top-1, Top-5, Top-10, Top-100, and all neurons as the repairing targets. For most models, the repair procedure improved the accuracy of the quantized networks and, in some cases, even higher than the accuracy of the floating-point model.

The dense-2 layer only contains 64 neurons in the Conv3 model. Hence, *QNNRepair* selected 30 neurons as the repair targets. In the dense-1 layer of Conv3, the effect of repairing individual neurons is not ideal, but as the number of repaired neurons gradually increases, the correct information the Conv3 quantization model obtains from the floating-point model. Hence, the

Table 5.4. *QNNRepair* results on ImageNet model.

	Random		Fault Localization		-
#Neurons repaired	10	100	10	100	All
MobileNetV2_dense-1	70.75%	70.46%	**70.77**%	70.00%	68.98%

accuracy gradually improves until it reaches 66.46% (which does not exceed the accuracy of the floating-point Conv3 neural network, but it gets very close to it: 66.48%, see Table 5.3). This is because all the repair information in the last layer comes from the original floating-point neural network. Note that because of the simple structure of the Conv3 neural network, the floating-point version of Conv3 itself is inaccurate, and the quantized and repaired neural network does not exceed the accuracy. In the dense-2 layer of Conv5, applying importance metrics to repair this layer is slightly better than random selection, only 0.01% regarding randomly selecting five neurons compared with using fault localization to select Top-5 neurons. Compared to the quantized model before repair, whose accuracy is 72.64%, the repairing only gets an accuracy of 72.56%, which does not improve the model's accuracy. In the dense-1 layer of Conv5, the best result is using fault localization to pick the Top-1 neuron and repair at 72.58% accuracy, and this is not better than the quantized model before repair.

For VGGNet and ResNet-18 neural networks, the dense-1 layer is a good comparison. Both VGGNet and ResNet-18 have relatively complex network structures, and the accuracy of the original floating-point model is close to 80%. In the dense-1 layer of ResNet-18, only some of the neurons were repaired with accuracy close to their original quantized version, but all of them did not exceed the exact value of the floating-point neural network after the repair. However, unlike ResNet-18, correcting a single neuron randomly in the dense-1 layer of VGGNet makes it more accurate than the quantized version of VGGNet. Using the importance metric and correcting a single neuron make the accuracy even higher than the floating-point version of VGGNet. However, repairing dense-1 of VGGNet was unsatisfactory, especially when 5 neurons were selected for repair; it suffered a significant loss of accuracy, even below 70%, which was regained if all ten neurons in the last layer were repaired. In the dense-2 layer of VGGNet, the overall accuracy is higher than 78%. When the importance metric is applied, the accuracy reaches 78.83%, noting that this accuracy is also achieved if all neurons in this layer are repaired. For the dense-3 layer of VGGNet, repairing 5 or 10 neurons using importance metrics will achieve the highest accuracy at 78.83%, the same as repairing the dense-2 layer.

ImageNet The evaluation also conducted repair on the last layer for MobileNetV2 trained on the ImageNet dataset of high-resolution images. Using Euclid as the importance metric and picking 10 neurons as the correct targets achieve the best results, at 70.77%, improving the accuracy of the quantized model.

5.3.4 Effects of Passing and Failing Tests in *QNNRepair*

In Section 5.2.1, we discussed that *QNNRepair* uses the passing and failing tests to locate the fault neurons. For comparison purposes, we chose four neural networks trained on the CIFAR dataset: Conv3, Conv5, VGGNet, and ResNet, which also have experimental results in Section 5.3.3 baselines.

Table 5.5. Effects of Failing Tests on Model Repair. FP model error means Failing tests in which the floating point model is misclassified but the quantitative model is classified correctly, and vice versa for QNN error.FP accuracy is the accuracy of the floating point model on the test set, and the QNN accuracy we divide into before fixing and after fixing are given respectively.

Model	Failing tests	FP model error	QNN error	FP accuracy	QNN accuracy	QNN accuracy (after repair)	Improvement(by %)
Conv3	204	58	71	66.48%	66.20%	66.46%	**0.26**%
Conv5	300	93	119	72.90%	72.64%	72.58%	**-0.08**%
ResNet	183	50	79	79.32%	79.16%	79.20%	**0.04**%
VGGNet	139	44	53	78.67%	78.57%	79.29%	**0.72**%

Table 5.5 reflects the effect of failing and passing tests on neural network model repair. For models with higher accuracy than the quantized neural network after repair, we mark the improved accuracy in blue; otherwise, it is in red. FP model errors occur when the FP model misclassifies while the QNN model correctly classifies, and vice versa for QNN errors. The results show that the repair process yields varying improvements across different models. For Conv3 and ResNet, the improvements are marginal (0.26% and 0.04%, respectively), indicating limited repair effectiveness. Conversely, Conv5 exhibits a slight accuracy decline (-0.08%), suggesting that the repair process may not always generalize well.

Also, there is a trend in the table that the fewer the failing tests, the better the repair results, e.g., for VGGNet, 139 Failing tests are the fewest among the four models, but the best repair results are achieved, and the accuracy of the repair is even higher than that of the original floating-point model. However, for Conv5, with 300 failing tests, the repair effect is not so good, and the accuracy even slightly decreases (-0.08%) compared with the quantized model before repair. One speculation is that the more failing tests, the more modifications are made to the original quantization model during the repair process, which in turn destroys the integrity of the original model and decreases the accuracy.

5.3.5 Fault Localization Metrics in *QNNRepair*

Let the model and the layer stay the same, and MobileNetV2's last layer is the target. Seven important metrics mentioned in Section 5.3.5 were compared. In these experiments, we used 1,000, 500, 100, and 10 jpeg images as the repair sets to assess the performance of different importance assessment methods.

Firstly, *QNNRepair* ranked the neurons in the last layer using seven different representative

Table 5.6. The results regarding importance metrics, including 7 fault localization metrics and 1 random baseline. The number of images indicates how many inputs are in the repair set.

Model+Repair Layer	#Images	Tarantula	Ochiai	DStar	Jaccard	Ample	Euclid	Wong3	Random
MobileNetV2_dense-1	1000	70.61%	69.76%	69.73%	69.73%	69.72%	**70.70**%	69.73%	69.56%
	500	68.99%	69.01%	69.05%	69.05%	68.99%	**69.46**%	69.06%	69.00%
	100	69.50%	69.42%	69.46%	69.46%	69.53%	69.98%	69.46%	**70.12**%
	10	70.62%	70.15%	70.12%	70.12%	70.17%	**70.73**%	70.12%	70.18%
VGGNet_dense-3	1000	78.64%	78.64%	78.64%	78.64%	78.65%	**78.66**%	**78.66**%	78.22%
VGGNet_dense-2	1000	**78.83**%	**78.83**%	**78.83**%	**78.83**%	**78.83**%	**78.83**%	**78.83**%	78.38%
Conv3_dense	1000	**59.50**%	**59.50**%	**59.50**%	**59.50**%	59.27%	59.27%	59.27%	32.42%

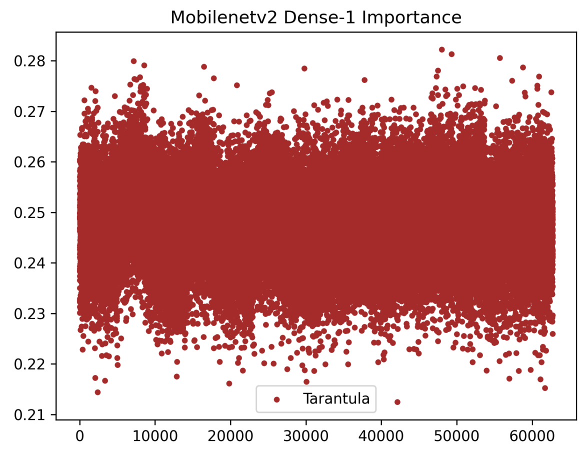
important metrics, which are Tarantula [201], Ochiai [202], DStar [203], Jaccard [204], Ample [205], Euclid [206] and Wong3 [207]. As shown in Figure 5.2, for the last fully connected layer of MobileNetV2, the important neurons are mainly concentrated at the two ends, those neurons with the first and last numbers. The evaluation metrics results are relatively similar for different neurons.

100 neurons (for Conv3, it is 30 neurons) were selected with the highest importance and could be solved by MILP solvers according to different importance measures. The deltas were obtained according to Equation 5.1, and *QNNRepair* applied the deltas to the quantized model. After that, *QNNRepair* used the validation sets from ImageNet, which contains 50,000 JPEG images, to test the MobileNetV2 model. The validation sets from CIFAR-10, which contain 10,000 PNG image files, were used to test VGGNet and Conv5 after the repair. As a comparison, 100 neurons were also randomly picked to apply to repair, and their accuracy was tested. The results of the top 100 important neurons after selection and repair are shown in Table 5.6.

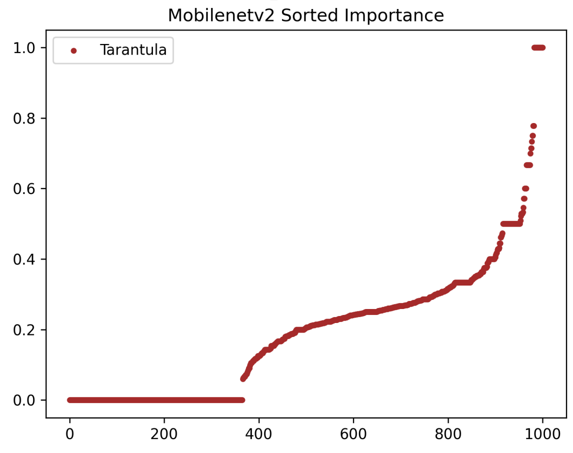
Tarantula was picked, and scatter plots based on the importance distribution of the different neurons were plotted. The importance of those neurons was ranked, and line plots were drawn as illustrated in Figure 5.2.

The figures give scatter plots of neuron importance and ranked line plots for the last dense layer of MobileNetV2. The horizontal coordinates of these plots are the serial numbers of the neurons. For the last layer in the MobileNetV2 model, few neurons have the highest importance. More than 300 neurons had an importance measurement of 0, and another large proportion had an importance of 0.5 or less. Based on the ranking of the importance of neurons, all the evaluation metrics except Tarantula and Euclid considered 108, 984, 612, 972 to be the four most important neurons in this layer, and among the 5th-10th most important neurons, 550, 974, 816, 795, and 702, just in a different order. This is reflected in the importance distribution graphs as spikes at the ends and as spikes at the ends of the graphs. Hence, Ochiai, Dstar, Jaccard, Ample, and Wong3 have similar performance regarding the accuracy evaluation, and Euclid and Tarantula achieve better accuracy on ImageNet validation sets.

Table 5.6 shows that the Euclid importance assessment method is highly effective, achieving relatively good results from restoration with 500 images to restoration with ten images and achieving only weaker accuracy than the Tarantula method in a restoration scenario with 1,000 images. A random selection of neurons can achieve good restoration results, especially when 100 images are selected as restoration images, achieving a validation accuracy higher than 70%. Also, in Table 5.6, *QNNRepair* works well with models containing fewer neurons. In experiments with Conv3_dense, the approach achieves more than 20% higher accuracy than random selection. When it comes to large models, although the improvement is not as pronounced as with smaller models, higher accuracy than random selection is still achieved in most cases, even if random selection is better than importance ranking by only a small margin (0.14%). Considering the successful and failing tests used for repair, i.e., the repair images, in the experiments, the repair results using 10 repair images were slightly better than using 1,000 repair images. For the Euclid method that produces the best repair results, the accuracy using 10 images is 0.03% higher than using 1,000 repairing images.



MobileNetV2 Importance Distribution



MobileNetV2 Sorted Importance Distribution

Figure 5.2. Importance distribution regarding certain importance metrics on MobileNetV2.

For the VGGNet model, for the same reason as MobileNetV2 regarding the neuron importance ranking, the Tarantula, Ochiai, DStar, Jaccard, Euclid, and Wong3 give the same results when selecting 100 top important neurons to repair. As a comparison, the accuracy of random selection in dense-2 layer and dense-3 has a slight drop, at 78.38% and 78.22%. For the Conv3 model, the seven importance metrics give the same results, and randomly selected 30 neurons

Table 5.7. The Gurobi solving time for constraints of each neuron in the dense-2 layer of the VGGNet model. There are 512 neurons in total.

Duration	<=5mins	5-10mins	10-30mins	30mins-1h	No solution
Percentage	75%	8.98%	5.27%	1.76%	8.98%

suffered a great accuracy loss, at 32.42%. But compared to the results in Table 5.3, repairing 30 top neurons also suffered accuracy drops. For the dense layer of conv3, the best repair is still to select one neuron for repair based on Tarantula sorting at 66.10%, and if random selection is taken into account, then selecting five neurons for repair would give the best result at 64.74%.

A side-by-side comparison of the number of images required for the repair on MobileNetV2 was also conducted. It shows that the best results are obtained using 1,000 images for repair and 10 images for the repair, but given the amount of time required to generate constraints for the repair using 1,000 images and to solve the constraints using Gurobi, we recommend using a smaller set of repair images for the model.

Euclid demonstrates that it has the highest accuracy most of the time, and repairing with importance evaluation is more accurate than repairing randomly selected neurons.

These results successfully answer **EG1** and **Research Question** 2: *QNNVerifier* was proposed based on the MILP(see definition in Definition 2.4.5) algorithm to verify the quantized neural network model, which converted from floating-point models. *QNNVerifier* uses the Neuron Importance Ranking method to optimize the repairing algorithm to answer **Research Question** 3. *QNNVerifier* was evaluated and compared with baselines and it can successfully repair all the benchmarks.

5.3.6 Repair Efficiency

The constraints-solving part contributes to the major computation cost in *QNNRepair*. Other operations, such as importance evaluation, weight modification, model formatting, etc., take only a few minutes to complete. Thereby, Table 5.7 measures the runtime cost when using the Gurobi to solve the values of the new weights for a neuron for our experiments on the VGGNet model. It is shown in Table 5.7 that 75% of the solutions were completed within 5 minutes, and less than 9% of the neurons could not be solved, resulting in a total solution time of 9 hours for a layer of 512 neurons.

These results answer **EG2**: We evaluated our tool on VGGNet model. Our approach can successfully repair most of the benchmarks without a timeout or crash. *QNNRepair* finished 75% of the benchmarks in 5 minutes.

Table 5.8. *QNNRepair* vs SQuant

	MobileNetV2		ResNet-18	
	Accuracy	Time	Accuracy	Time
SQuant [210]	46.09%	1635.37ms	70.70%	708.16ms
QNNRepair	**70.77%**	∼15h	**79.20%**	∼9h

5.3.7 Comparison with Data-free Quantization

SQuant [210] was tested in this section, which is a fast and accurate data-free quantization framework for convolutional neural networks, employing the Constrained Absolute Sum of Error (CASE) of weights as the rounding metric. SQuant was tested on two quantized models, the same as our approach: MobileNetV2 trained on ImageNet and ResNet-18 on CIFAR-10. This thesis modified the original code to support MobileNetV2, which is not reported in their experiments.

In contrast, to complete data-free quantization, our constraint solver-based quantization does not require a complete dataset but only some input images for repair. Despite taking much more time than SQuant because it uses Gurobi and a constrained solution approach, MobileNetV2 – a complex model trained on ImageNet – *QNNRepair* achieves much higher accuracy.

These results answer **EG3**: We evaluated and compared our tool with the SQuant. Our approach can successfully repair all the benchmarks without a timeout or crash.

5.4 Chapter Summary

This chapter presented *QNNRepair*, a novel method for repairing quantized neural networks. It is inspired by traditional software statistical fault localization. This chapter evaluated the importance of the neural network models and used Gurobi to get the correction for these neurons. According to the experimental results, after correcting the model, accuracy increased compared with the quantized model. *QNNRepair* was also compared with state-of-the-art techniques; the experiment results show that our method can achieve much higher accuracy when repair models are trained on large datasets.

According to Nemhauser et al. [211], the MILP problem is NP-Hard. There is no known polynomial time algorithm that can solve all MILP instances. Therefore, for very large or structurally complex problems, the solver may take a long time to find the optimal or acceptable approximate solution. Hence, selecting more repairing images for correction will have a greater likelihood of Gurobi being unable to solve the MILP problem, which is reflected in the limitation of improving accuracy.

As the future works, *QNNRepair* will move forward to larger datasets; currently, *QNNRepair* supports MobileNetV2 trained on ImageNet. In the future, we will test our tool and make it scalable for larger models and not limited to classification tasks. These include generative tasks like text generation (e.g., GPT) and image synthesis (e.g., Stable Diffusion) [212], where models produce new content rather than assigning predefined labels. It also includes sequence-to-sequence tasks like translation and summarization, as well as reinforcement learning tasks such as decision-making in dynamic environments (e.g., robotics or gaming). Due to the complexity of the model itself, repairing these large networks will require a lot of computational resources, and we will find a balance between improving accuracy and computing time.

For some of the repairing problems, Gurobi was not able to solve them in the given time limit, so in the future, the target of *QNNRepair* is to optimize the encoding of the neural network repair problem to increase the speed of the repair solution and to solve some of the repair problems that were not previously solved. More problem solvers, such as SMT solvers, will also be integrated in the future to solve these problems that Gurobi cannot solve.

Chapter 6

AIRepair: A Repair Platform for Neural Networks

This chapter introduces *AIRepair*, the neural network repair platform. Firstly, it introduces the overall framework. Then, it describes the usages of *AIRepair* and gives a simple example of using *AIRepair*. After that, several experiments were conducted on popular benchmarks, and the evaluation results were reported.

6.1 Chapter Introduction

In Chapter 4 and 5, the neural network verification method *QNNVerifier* and the neural network repair method *QNNRepair* were introduced. But the question is, how do you make this workflow fluid and make it quickly accessible to non-specialists?

To achieve this, it is crucial to develop an intuitive user interface that abstracts the complexity of the underlying processes. This interface should provide clear, step-by-step guidance to users, enabling them to run verification and repair tasks without requiring deep technical knowledge. Additionally, comprehensive documentation and user support resources, such as tutorials and FAQs, can significantly enhance the accessibility of these tools.

At the same time, neural network models supplied by users are often trained on various platforms and frameworks and stored in different formats. Addressing this heterogeneity is another significant challenge. To handle these diverse models in a unified way, it is essential to implement a standardized model import and export functionality. This could involve developing or integrating a conversion library that supports a wide range of formats, such as TensorFlow, PyTorch, ONNX (see Section 2.1.1), and Keras. Converting these models into a common intermediate representation can ensure compatibility with the *QNNVerifier* and *QNNRepair*.

Moreover, automated scripts or pipelines can be provided to facilitate the conversion process, reducing the manual effort required from users. These scripts can be designed to detect the format of the input model and perform the necessary transformations automatically. In addition, providing a robust API can allow advanced users to integrate these tools into their existing workflows more seamlessly.

Hence, the neural network repair platform *AIRepair* was proposed, which integrates various repair tools and can accept different neural network model formats. *AIRepair* is designed to provide a comprehensive and flexible solution for repair neural networks, accommodating the diverse needs and formats commonly encountered in the field.

In the front end of *AIRepair*, the platform automatically identifies the neural network model type and converts it into a format that the specified repair tool can accept. This automatic identification and conversion process eliminates the need for manual intervention, making it user-friendly and efficient. *AIRepair* supports a wide range of formats, including TensorFlow, PyTorch, ONNX, and Keras, ensuring broad compatibility with models from different frameworks.

During the repair process, *AIRepair* monitors system resource usage, including CPU, memory, and GPU utilization. This monitoring ensures that the repair process is efficient and does not overburden the system, providing real-time feedback and alerts if resource usage exceeds predefined thresholds. This feature is particularly important for large-scale models that require substantial computational power.

After completing the repair process, *AIRepair* generates a detailed repair report. This report includes information on the original vulnerabilities detected, the specific repairs made, and the overall impact on model performance. Additionally, *AIRepair* compares the effectiveness of different repair methods used, providing insights into which method is more suitable for the input model. This comparative analysis helps users understand the strengths and weaknesses of various repair approaches, enabling them to make informed decisions for future repairs.

Overall, *AIRepair* offers a robust and user-friendly platform for neural network repair, integrating advanced tools and automated processes to enhance the reliability and performance of neural networks. By providing comprehensive monitoring, detailed reporting, and comparative analysis, *AIRepair* stands out as a valuable resource for both researchers and practitioners in the field of neural network security and maintenance.

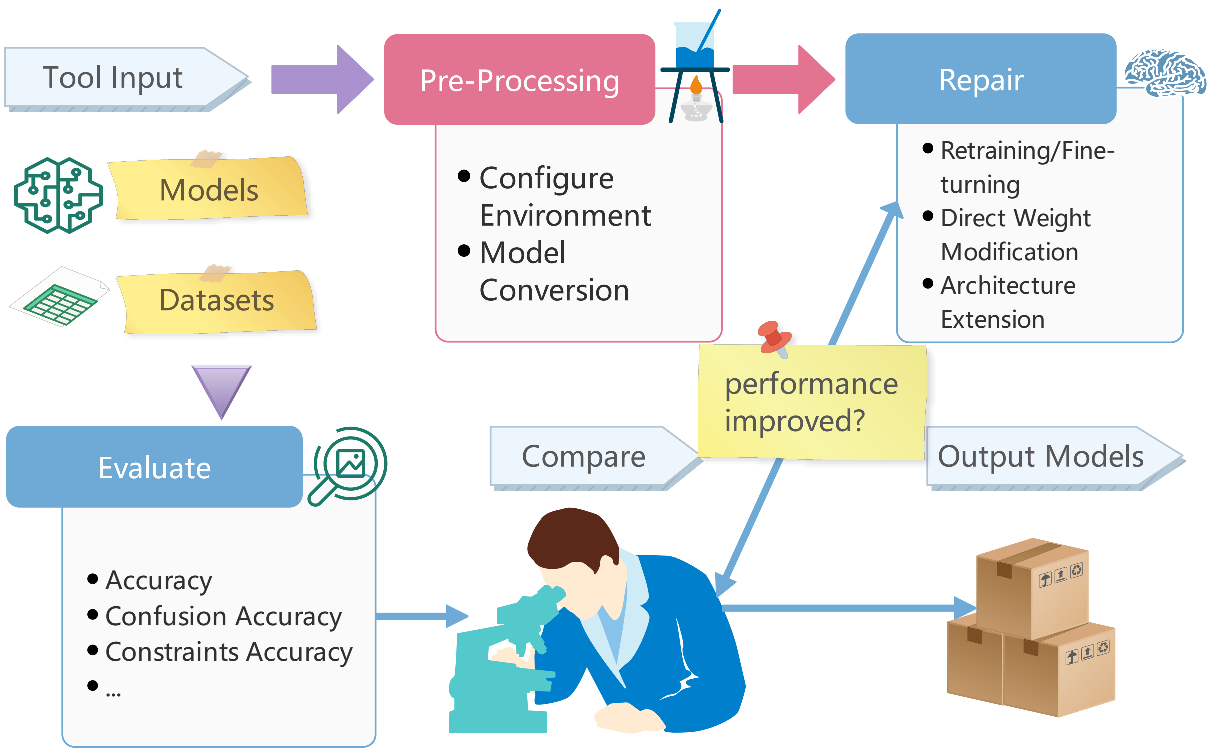


Figure 6.1. The *AIRepair* Architecture. The input of the framework is the neural network to be repaired, and the dataset used for training, *AIRepair*, automatically selects the repair method as well as evaluates the repair effect, and the output is the repaired neural network and the repair report.

6.2 *AIRepair* Framework

The primary motivation behind the development of *AIRepair* is to create a comprehensive platform dedicated to testing and evaluating various repair methods in a standardized and compatible manner. This platform addresses the need for a unified system where different repair techniques can be assessed on a level playing field, ensuring consistency and comparability in their evaluation. By providing a standardized environment, *AIRepair* facilitates the benchmarking of repair methods, allowing researchers and practitioners to identify the most effective approaches. Additionally, this platform is designed to foster innovation and collaboration in the field of repair technologies, enabling the sharing of insights and best practices among the community. Through *AIRepair*, the goal is to advance the understanding and improvement of repair methods, ultimately leading to more reliable and efficient solutions in various applications.

As illustrated in Figure 6.1, *AIRepair* accepts the trained models and the training datasets if specified in the configuration. It performs pre-processing on different benchmarks to make them capable of other frameworks. Pre-processing isolates different running environments for deep learning libraries, e.g., TensorFlow [15] or PyTorch [16]. After the repair, *AIRepair* collects the results and analyses them automatically, which is done by examining the outputs and experimental logs. Finally, it presents the results so the user can decide which repair tool suits their models. The output from *AIRepair* includes the repaired model, logs, and parameters.

In the following sections, this chapter details each component in *AIRepair*: 1) input, 2) pre-processing, 3) repair, and 4) evaluation.

6.2.1 Input

The input to *AIRepair* is trained neural network models and testing or training datasets depending on the repair method configured. *AIRepair* accepts fully connected feed-forward neural networks and convolutional neural networks in popular deep learning model formats such as .pth and .pt from PyTorch, and .pb, and .h5 from TensorFlow + Keras [213]. *AIRepair* has been tested on standard datasets like MNIST, Fashion-MNIST, CIFAR-10, and CIFAR-100.

MNIST is a widely used benchmark dataset consisting of 70,000 grayscale images of handwritten digits (0-9), each of size 28x28 pixels. It serves as a standard for evaluating image classification algorithms and models in ML and computer vision.

Fashion-MNIST is a dataset similar in format to MNIST but contains 70,000 grayscale images of 10 different types of clothing items, such as T-shirts, trousers, and dresses. Each image is also 28x28 pixels. Fashion-MNIST was created as a more challenging replacement for MNIST, aimed at testing more advanced machine learning algorithms.

For **CIFAR-10** and **CIFAR-100**, please check Section 4.3.1.

JavaScript Object Notation (JSON) [214] is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is a text format that is completely language-independent but uses conventions that are familiar to programmers of the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

Existing efforts on neural network analyzing techniques have resulted in multiple impressive tools, including verifiers we mentioned in Chapter 2.4. These tools, however, have their input format. Thus, *AIRepair* accepts JSON format and supports a variety of neural network models and an assertion language to provide a common ground for different neural network repair techniques.

The JSON file provided to *AIRepair* consists of a series of keys specifying details of the network model and attributes. Table 6.1 shows the keys used to define models. At the top level, the model is determined using the key model. The values of the model are then defined as JSON objects using a tuple consisting of the key shape, boundary, and layer. The key shape is used to define the shape of the input sample, a tuple containing multiple integer values. The first value represents the length of the sample. Note that for acyclic neural networks, the first value is always 1; for recurrent neural networks, the first value is always n, where $n \geq 1$. The remaining numbers represent the shape of each element in the sample. For example, the tuple $(5;80)$ indicates that the input sample is a sequence of length 5, and each sequence component is a vector of size 80.

The key layer specifies the details of the network layer by layer. Its value is an array. Each element in the array is a JSON object representing a layer of the network with a specific type. Depending on each type, a layer can contain multiple keys to specify the parameters of that layer. In general, each layer can be thought of as a function. The original sample of the network is the input of the first layer, the output of the nth layer is the input of the $n+1$ th layer, and the output of the last layer is the output of the entire network.

Each type is associated with a set of parameter values specified with one or more predefined keys. For example, the keys' weight, bias, and function define the weight matrix, bias vector, and activation function, respectively.

Depending on the type of the layer, the key filters, stride, padding, h_0, and c_0, are used to define the values of the filter matrix, stride value, padding value, h_0 vector, and c_0 vector, respectively. If the layer has more than one parameter of the same type, indexes are used to distinguish between them. For example, two filter matrices in the ResNet-21 layer can be defined using two keys, filter1, and filter2, and ResNet-21 means that the ResNet mentioned in Section 2.1.1 model consists of 21 layers.

Table 6.1. Definitions of keys used in the neural network model.

Key	**Definition**
model	The details of the network model
shape	The shape of input sample
bounds	The bounds of input sample
layers	The details of model layers
type	The type of a layer
weights	The weight matrix of a layer
biases	The bias vector of a layer
func	The activation/transformation function of a layer
filters	The filter matrix of a convolutional layer
padding	The padding value of a convolutional layer
stride	The stride value of a convolutional layer
h_0/c_0	The h_0/c_0 vector of a recurrent layer

The *AIRepair* platform is meticulously engineered to support a variety of neural network architectures, including fully connected feed-forward networks and convolutional neural networks, which are prevalent in the deep learning landscape. This platform is compatible with widely-used deep learning model formats such as .pth and .pt for PyTorch implementations, alongside .pb and .h5 formats from TensorFlow + Keras frameworks. The versatility in model format compatibility is inspired by the extensive work in neural network development and application, as highlighted by Chollet in his seminal work on Keras [213].

To ensure robustness and applicability, *AIRepair* platform has been rigorously tested on standard machine learning datasets, including MNIST [59], Fashion-MNIST [215], CIFAR-10, and

CIFAR-100 [216]. These datasets are acknowledged benchmarks within the machine learning community, providing a solid foundation for evaluating neural network performance. By aligning *AIRepair* 's testing protocols with these established datasets, the platform guarantees a consistent and reliable framework for model assessment. Specifically, the platform's design allows for automatically recognizing the dataset associated with a user's pre-trained model. This ensures that models are evaluated under conditions similar to their training, facilitating a direct and relevant analysis of performance and robustness [40].

Upon integration of a neural network model and its corresponding dataset, *AIRepair* embarks on a detailed mutation testing regimen. This process involves the deliberate alteration of input data to uncover potential vulnerabilities in the network's structure and function, drawing on strategies from both traditional software engineering and machine learning fields [24]. Mutation tests are integral to diagnosing the network's resilience to input perturbations, which is a crucial aspect of the evaluation of machine learning models.

In addition to mutation testing, *AIRepair* applies dataset augmentation techniques to probe the neural network's robustness further. Dataset augmentation, a method well-documented by Shorten and Khoshgoftaar [217], involves the application of various transformations to the original data, such as image rotation, scaling, and cropping. These augmented datasets present alternative scenarios to test the network's ability to maintain performance under varied input conditions, thereby offering insights into the model's generalization capabilities.

In conclusion, the *AIRepair* platform provides a comprehensive and multifaceted approach to evaluating and enhancing the robustness of neural network models. The platform stands as a testament to the current state of deep learning research and application through the integration of versatile model formats, alignment with standard testing datasets, and the implementation of mutation and augmentation testing methodologies.

6.2.2 Pre-processing

Pre-processing is designed to streamline the workflow of neural network diagnostics and repair by facilitating model conversions between different frameworks, automating the setup of diverse runtime environments, and conducting thorough pre-repair evaluations. The versatility of neural network modeling tools has led to diverse formats and frameworks used in model development, such as PyTorch and TensorFlow. These frameworks differ in syntax, design philosophy, computational paradigms, and support for different hardware accelerators.

To address these disparities and ensure seamless interoperability within the *AIRepair* ecosystem, the platform incorporates a model conversion utility based on MMdnn. This comprehensive, cross-framework solution supports conversion among various neural network formats. Introduced by Liu et al. [218], MMdnn acts as a translational bridge, preserving model integrity and performance across conversions. To ensure the model integrity, MMdnn introduces the concept of faithful model conversion:

Definition 6.2.1 (Faithful Model Conversion)**.** $\mathcal{A}$ is a faithful model conversion algorithm if the following two conditions are satisfied:

- **Syntactic Legality.** Given an arbitrary source model $\mathcal{M}_1 = \langle V_1, E_1, P_1 \rangle$, a legitimate target model $\mathcal{M}_2 = \langle V_2, E_2, P_2 \rangle$ should be produced:

$$\mathcal{M}_1 \vDash_{\mathcal{A}} \mathcal{M}_2 \wedge (P_2 = P_1)$$

- **Semantic Equivalence.** Given an arbitrary valid input, the source and target models should always return the same result:

$$Z_{tp} = \mathcal{F}_{\mathcal{M}_1}(X_{tp}) \vDash_{\mathcal{A}} Z_{tp} = \mathcal{F}_{\mathcal{M}_2}(X_{tp})$$

Z_{tp}, X_{tp} are the output and input tensor tuples, respectively.

Ideally, MMdnn performs the *isomorphic graph transformation* or graph rewriting [219] on $\mathcal{M}_1$: for each source node u_i (e.g., a TensorFlow Conv2D node), a new node v_i with the same u_i.op operator (still Conv2D for the case) is generated in $\mathcal{M}_2$; if u_i has an edge pointing from its k-th output tensor to the l-th input tensor of u_j (i.e., $(u_i, u_j) \in \mathcal{M}_1$), a corresponding edge from v_i's k-th output to v_j's l-th input is added. The faithfulness clearly holds.

This capability is crucial as it allows the *AIRepair* platform to cater to a wide array of neural network models, ensuring broad applicability and user convenience.

Furthermore, recognizing the diverse and often complex dependencies required by different neural network repair tools, *AIRepair* leverages Anaconda, an open-source package and environment management system, to streamline the configuration process [220]. Before initiating any repair procedure, the platform automatically configures the necessary runtime environments specific to each tool. This process involves the creation of isolated environments, ensuring that each repair tool operates under optimal conditions with the correct versions of libraries and dependencies. This meticulous preparation mitigates compatibility issues and enhances the reproducibility of repair outcomes.

By automating these preparatory steps, *AIRepair* significantly reduces the barrier to entry for users seeking to analyze and enhance the robustness of their neural network models. The platform's integrated approach, combining model conversion with automatic environment configuration, establishes a cohesive and user-friendly workflow, enabling practitioners to focus more on interpreting results and less on navigating the technical intricacies of model repair and evaluation.

6.2.3 Repair

The repair component within the *AIRepair* platform stands as a critical mechanism meticulously designed to address and rectify issues within neural network models. Upon receiving input models and subsequent comprehensive pre-processing, this component intelligently tailors and initiates repair methodologies specifically suited to the diagnosed issues within these models. This customization involves intricate parameter setting and conditions to ensure precise and impactful repair actions, aligning with practices outlined in contemporary neural network repair research, such as those described by B Yu et al. [12] in their work on DeepRepair.

During the intricate repair phase, the *AIRepair* system prioritizes efficient resource management and transparent communication with the user. It continuously monitors hardware resource consumption, including CPU and GPU utilization, alongside memory and disk usage. This vigilant oversight ensures that the repair processes are conducted without overwhelming the system, reflecting strategies akin to those found in system monitoring standards described in the Anaconda documentation [220]. This approach allows for real-time adjustment of the repair workload, safeguarding both the efficiency of the repair process and the overall system performance.

Moreover, the *AIRepair* platform commits to maintaining open lines of communication with the user throughout the repair procedure. It provides ongoing updates on the repair status, including progress, warnings, and errors, thereby keeping users informed and engaged. This aspect of user interaction is critical, as highlighted by Goodfellow et al. [40] in their discussions on the importance of transparency and communication in automated systems. In addition to real-time updates, *AIRepair* meticulously logs every detail of the repair process, capturing vital information such as parameter adjustments, method outputs, and resource consumption. These logs are indispensable for post-repair evaluation, allowing users to scrutinize the repair actions and understand the decision-making processes behind the implemented repair strategies.

By integrating model conversion capabilities, system resource monitoring, and user communication, the repair component of the *AIRepair* platform ensures not only the effective repair of neural network models but also a user-friendly and transparent experience. This comprehensive approach enables users to focus on the substantive aspects of model repair and improvement, supported by a robust framework that adheres to established best practices in the field.

6.2.4 Evaluation and Output

This component measures the performance of a model before and after repair. There are several metrics for characterizing a model's performance from complementary perspectives, including the model's *(classification) accuracy*, *constraint accuracy* [221], and *confusion accuracy*. The constraint accuracy describes the percentage of predictions given by the model that satisfies the constraint associated with the problem, which requires that the probabilities of groups of classes have either a very high or a very low probability.

For example, we can declaratively express constraints over quantities that are not explicitly computed by the network, such as

$$p_\theta(\boldsymbol{x})_{\text{people}} < \epsilon \vee p_\theta(\boldsymbol{x})_{\text{people}} > 1 - \epsilon. \tag{6.1}$$

This constraint on the output activations of a CIFAR-100 classifier p_θ says that for a network input $\boldsymbol{x}$, the probability of people, denoted $p_\theta(\boldsymbol{x})_{\text{people}}$, is either very small or very large. As CIFAR-100 does not have a class of people, we define it as a function of other output activations:

$$p_\theta(\boldsymbol{x})_{\text{people}} = p_\theta(\boldsymbol{x})_{\text{baby}} + p_\theta(\boldsymbol{x})_{\text{boy}} + p_\theta(\boldsymbol{x})_{\text{woman}} + \cdots \tag{6.2}$$

The confusion accuracy is defined as $P = \frac{TP}{TP+FP}$, where TP and FP are True Positive and False Positive classifications. These two metrics evaluate the model's robustness. Data augmentation of different kinds of blurs (glass, motion, and zoom) [222] can be applied to the input dataset when collecting these metrics.

- To understand confusion accuracy, let us consider two types of images, cats, and dogs, in the CIFAR dataset and use the confusion matrix to evaluate the model's accuracy in classifying cats and dogs. For example, if the prediction is a dog and the actual label of the picture is a dog, we consider it as True Positive (TP); If the actual label is a cat, we consider it as False Positive (FP). The confusion accuracy is defined as $P = \frac{TP}{TP+FP}$, where TP is True Positive and FP is False Positive.

Typically, existing repair tools take some misclassified inputs, and the repair goal is to correct those erroneous nodes. Each repair method often focuses on improving performance according to one type of evaluation metric. The *AIRepair* tool integrates different repair methods and performance metrics to give a comprehensive view when repair a neural network model.

6.2.5 *AIRepair* Implementation

The *AIRepair* platform employs a sophisticated approach to facilitate the repair of neural network models across different frameworks and architectures by leveraging environment isolation, JSON parsing, and command concatenation techniques. Initially, it ensures that each repair tool operates within an isolated runtime environment. This isolation, typically achieved through technologies like Docker or virtual environments in Python, prevents conflicts between the dependencies required by different tools, thereby ensuring consistency and reproducibility. Environment isolation aligns with principles similar to those documented for Anaconda, a widely recognized platform for managing project-specific environments and dependencies [220].

In addition to environment isolation, *AIRepair* enhances its functionality through effective user input handling and configuration management. To achieve this, *AIRepair* utilizes JSON parsing to interpret user inputs and configurations effectively. JSON, a lightweight data interchange format, is widely used for its simplicity and ease of integration with various programming languages. In the context of *AIRepair*, JSON parsing allows for flexible and dynamic interpretation of repair parameters and settings, as outlined by Crockford in the documentation of JSON standards [223].

Additionally, the platform automates the process of command concatenation based on user inputs and the parsed JSON data. This process involves assembling the command-line arguments and parameters required to invoke the repair tools tailored to the specific needs of the model under repair. Using command-line interfaces for automation and scripting is well-established in software engineering, reflecting practices documented in Neal's comprehensive exploration of command-line utilities [224].

Integrating these three core components –environment isolation, JSON parsing, and command concatenation – the *AIRepair* platform streamlines the model repair process, enhancing user experience and ensuring effective repair outcomes.

6.2.6 Example Usage

In the first step, it is recommended to train the baseline model using the script ('train_baseline.py') provided. Subsequently, one can configure and run different network repair tools with *AIRepair*:

```
python AIRepair.py [-h] [--all]
[--net_arch NETARCH] [--dataset DATASET]
[--pretrained PATH_AND_FILENAME]
[--depth DEPTH]
[--method METHOD] [--auto]
[--additional_param PARAM]
[--input_logs INPUT_LOGS]
[--testonly]
```

For example, the setup below configures and runs *AIRepair* with three repair methods, Apricot, DeepRepair, and DL2, as discussed in Section 2.4. They are applied to repair a model named 'cifar10_resnet34' with the CIFAR-10 dataset. When *AIRepair* finishes executing this command, it will save the running log and print the comparison results.

```
python AIRepair.py --method apricot deeprepair dl2 --pretrained
    cifar10_resnet34_baseline.pt --dataset cifar10 --net_arch resnet --depth 34
```

In particular, users need to specify the model's architecture and depth when using '–pre-trained' to specify the path of a trained neural network model to repair. PyTorch has two methods to save the trained model: the entire model, the state_dict, or the checkpoint. When loading the neural network model, *AIRepair* needs to know its structure for the second method. It has the built-in structure definition for ResNet families and several convolutional neural networks for MNIST and Fashion-MNIST. Hence, users only need to specify the net architecture and depth when loading the state_dict. For the architectures that do not belong to these three models, users either provide the entire model or customize *AIRepair*'s pre-processing module. Currently, *AIRepair* can process both feed-forward and convolutional neural networks.

The parameter '–net_arch' specifies the architecture of models, and '–dataset' selects the corresponding dataset (that are needed for retraining/refining or attaching correction units). These are as discussed in Section 6.2.1.

To further specify the repair process, the specific repair method (tool) can be indicated using '–method'. In addition, the '–auto' option will automatically invoke the repair process using the default parameters for the selected repair methods. For example, it sets the following configurations for the DeepRepair method to repair a ResNet-34 model trained on the CIFAR-10 dataset:

```
--batch_size 128 --lr 0.1 --lam 0
--extra 128 --epoch 60 --beta 1.0
--cutmix_prob 0 --ratio 0.9
```

However, if users need to set them manually, add the desired parameters after –additional param. There is no set required to run the tool for running Apricot, and the parameter '–-additional_param' is no longer helpful for this tool.

For those who wish to evaluate a model's performance without undergoing any repair procedure, the '—testonly' option is available. This can be used before or after the repair process to check the model's performance, providing valuable insights into the effectiveness of the repair. The evaluation metrics include accuracy, confusion, and constraint accuracy, offering a comprehensive assessment of the model.

To streamline the repair process, users can employ the 'python AIRepair.py –all' command, which runs all the repair methods on all available models automatically with default parameters. This comprehensive approach ensures thorough testing and repair across multiple models and methods. However, it is important to note that this option requires substantial computing power, as it encompasses a wide range of operations and evaluations. For more details, refer to the documentation available at `https://zenodo.org/record/7627801#.Y-X6g3bP3tU`.

6.3 Evaluation

6.3.1 Description of the AIRepair Benchmarks

Three repair methods, Apricot [118], DeepRepair [12], and DL2 [221], are chosen as baselines for this study. The details of these tools or methods can be found in Section 3.2. Each of these methods brings a unique approach to the task of repair neural networks, addressing various aspects of model reliability and robustness.

To begin with, **Apricot** [118] is a tool designed for repair neural networks by leveraging gradient-based optimization techniques to identify and mitigate erroneous behaviors in trained models. It aims to improve model robustness by adjusting the weights to correct specific errors.

Similarly, **DeepRepair** [12] focuses on enhancing the robustness and reliability of deep learning models by utilizing training data augmentation and specialized loss functions to address and fix faults identified during model evaluation.

Meanwhile, **DL2** [221], which stands for Deep Learning with Differentiable Logic, is a framework that integrates logical constraints into the training process of neural networks. This ensures that the models adhere to specified properties, thereby improving their correctness and safety.

In addition to these methods, **DeepState** [225] is a testing framework for deep learning systems that combines symbolic execution with state space exploration. This approach helps identify and repair faulty behaviors in neural network models, ensuring they perform reliably under various conditions.

Lastly, **RNNRepair** [226] is a repair method specifically aimed at recurrent neural networks (RNNs). It addresses issues related to sequence learning and temporal dependencies by employing specialized techniques to fix errors and improve the overall performance of RNN models.

These repair methods are applied to a benchmark of 11 neural network models, including ResNet models [62], using four datasets commonly used in image classification: MNIST [155], Fashion-MNIST [215], CIFAR-10, and CIFAR-100 [216].

To utilize these repair methods, users can invoke them by specifying '–auto', as discussed in Section 6.2.6. We conducted comparisons and reported the best results for each repair method from different settings, subsequently setting the corresponding optimal settings as defaults for each specific tool.

Table 6.2. A summary table for all benchmarks used in the experiments.

Dataset Name	Models	Type	Scale
CIFAR-10/100	ResNet18	CNN	32*32 inputs, 18 layers deep, 10/100 outputs
	ResNet34	CNN	32*32 inputs, 34 layers deep, 10/100 outputs
	ResNet50	CNN	32*32 inputs, 50 layers deep, 10/100 outputs
MNIST/F-MNIST	CNN	CNN	28*28 input, 6 conv layers, 10 output
	FFNN	FFNN	28*28 input, 6 layers, 784 per layer, 10 output

Recognizing that repair tools often exhibit randomness, we repeated the same experiment three times to eliminate this variability and ensure the reliability of the results. This thorough approach helps to provide a clear and accurate assessment of each repair method's effectiveness.

6.3.2 Experiments Setup

We are mainly interested in the following experimental goal:

EG - Efficacy - Can *AIRepair* run repair on different models using appropriate repair methods?

The experiments were conducted on a machine with Ubuntu 18.04.6 LTS OS Intel(R) Xeon(R) Gold 5217 CPU @ 3.00GHz and two Nvidia Quadro RTX 6000 GPUs. The experiments are run with TensorFlow2 + nVidia CUDA platform. The Gurobi [121] was used as the linear program solver and enable multi-thread solving (up to 16 cores). *QNNRepair* was applied to repair a benchmark of five quantized neural network models, including MobileNetV2 [63] on ImageNet datasets [79], and ResNet-18 [62], VGGNet [4] and two simple convolutional models trained on CIFAR-10 dataset [192].

6.3.3 Train Baseline Models

DL2 was used to train the baseline models. We trained three Convolutional Neural Network models with different depths, ResNet18, ResNet34, and ResNet50 [62], on the CIFAR-10, CIFAR-100 datasets separately. For the definition and details of these models, please check Section 2.1.1. We also trained two simple convolutional Neural Network Models on the MNIST and Fashion-MNIST datasets. The weights of the DL2 abstraction layer were set to 0 in order to serve as the baseline network, and all subsequent neural network repair was performed on the baseline neural network models.

Table 6.3. *AIRepair* results: 'running' means the experiment is still running. '–' means that the tool does not apply to the model. The best accuracy (Acc.) and constraints accuracy (Const.) improvement for each model are highlighted in % and % separately.

Datasets		**CIFAR-10**			**CIFAR-100**			**MNIST**	**F-MNIST**
Models		ResNet18	ResNet34	ResNet50	ResNet18	ResNet34	ResNet50	MNIST	F-MNIST
Baselines	Acc.	92.05%	91.34%	94.42%	46.84%	44.16%	47.36%	99.45%	92.20%
	Const.	90.51%	90.27%	90.66%	86.62%	85.95%	85.21%	99.96%	100%
Apricot	Acc.	-2.65%	-0.38%	-3.4%	+9.02%	+13.74%	+11.15%	+0.06%	+0.61%
DeepRepair	Acc.	+0.5%	-1.27%	-4.14%	+10.91%	+21.42%	+20.32%	+0.17%	+0.47%
	Const.	-9.46%	-8.82%	-12.77%	-37.62%	-34.95%	-29.71%	-0.43%	-4.10%
DL2	Acc.	-2.16%	+0.23%	-1.95%	+0.87%	+1.17%	-1.16%	+0.08%	+0.28%
	Const.	+9.3%	+9.61%	+5.4%	-0.49%	-0.89%	-0.4%	+2.55%	+6.27%

6.3.4 Compare Different Repair Tools on the Same Benchmark

Table 6.3 shows the complete *AIRepair* results of 3 repair methods on 8 neural networks (columns) from 4 datasets. The performance was measured by accuracy and constraint accuracy. The baseline represents each model's actual performance, and for each repair method, we report the absolute performance increment or decrement after repair. Apricot repairs a neural network by directly modifying its weight parameter values. DeepRepair and DL2 belong to the category of retraining and architecture extension, respectively, for repair neural networks. The three baselines are selected to represent all 5 categories of neural network repair methods, as discussed in Section 2.

For the models trained on CIFAR-10, DL2 brings the greatest improvement in constraint accuracy (Const.). At the same time, this often comes with slight drops in plain accuracy (Acc.), which is specially designed to repair the constraints' accuracy and can effectively improve it. When repairing ResNets on CIFAR-10 datasets, there were a few accuracy drops, but the constraints' accuracy increased significantly. The adversarial robustness and the plain accuracy are two contradicting goals [227]. Besides, the performance of the CIFAR-10 models drops to different extents by Apricot and DeepRepair. Apricot is not designed to improve the constrains accuracy of models. We do not evaluate the constraints accuracy metrics on models repaired by Apricot.

At the same time, Apricot and DeepRepair have better improvements on CIFAR-100 neural network models. This indicates that when the original model's performance is relatively low, the Apricot and DeepRepair are more effective. Moreover, as the ResNet models become deeper, from 18 layers to 50 layers, DeepRepair can improve accuracy by sacrificing constraint accuracy. For the models trained on CIFAR-100, Apricot is an outstanding tool for improving the accuracy of the models. Because they are more complex than the models trained on CIFAR-10, and DL2 uses semi-supervised learning tactics while handling the CIFAR-100 dataset, DL2 has no significant improvement in performance this time of these models. On the other hand, DeepRepair considerably improves accuracy while sacrificing constraints accuracy. It is based on data augmentation, enabling a neural network to process more input images efficiently. As the ResNet models become deeper, from 18 layers to 50 layers, DeepRepair can perform better and improve accuracy. Although originally Apricot, DeepRepair, and DL2 were designed to run on CIFAR-10 and CIFAR-100 datasets, we compared the experimental results presented in their paper and found similar results.

The original Apricot, DeepRepair, and DL2 tools do not support them. So, in this section, patches were created for these tools to enable them to run these models under the *AIRepair* framework. The patches do not change the algorithms and the functions inside their tool. They only make the repair tool accept MNIST and Fashion-MNIST datasets and define the model structure trained on these two datasets. Therefore, we supposed it would not affect the repair performance of these tools. As shown in Table 6.3 (last two columns), all three repair methods result in noticeable performance improvements of different levels for MNIST and Fashion-MNIST models, even though the original networks' performance is already high (99.45% and 92.20% respectively).

In summary, there are some general observations from the experiments. There is no single best repair method on all benchmarks and on all evaluation metrics. Different repair methods seem to complement each other highly, and this would motivate future "combined repair" of neural network models, for which *AIRepair* can serve as the test bed. Mostly, as expected, less complex neural networks (MNIST/Fashion-MNIST in Table 6.3) and lower performance models (CIFAR-100 examples in Table 6.3) are easier to repair. We still consider them valuable observations, as they confirm that *AIRepair* is a valid platform for benchmarking different repair methods.

We also tested *AIRepair* on three RNN-based architectures. We apply DeepState and RNNRepair to repair LSTM, BLSTM, and GRU models trained on MNIST. The baseline model classification accuracy is 98.6%, 98.54%, and 98.93%. DeepState slightly improves the accuracy of these models by 0.05%, 0.19%, and 0.18%, whereas the model accuracy drops by 0.09%, 0.06%, and 0.33% when using RNNRepair. Moreover, when it comes to mixed testing datasets, we can see significant improvements in the models repaired by DeepState. RNNRepair has a slight drop in accuracy after repair. Although it is not as high as DeepState, RNNRepair improves after repair when testing it on mixed datasets. DeepState performs better than RNNRepair for LSTM, BLSTM, and GRU neural network models trained on the MNIST dataset.

These results successfully answer **Research Question** 3 and **EG**: *AIRepair* can convert the models between different formats and apply them with appropriate repair methods. *AIRepair* supports JSON format input to describe the input model and attributes. *AIRepair* also implements environment isolation to ensure the comparison of different repair methods is sound.

6.4 Chapter Summary

This chapter introduces *AIRepair*, a comprehensive platform for repair neural networks, and it can test and compare different neural network repair methods. This paper gives the results of five existing neural network repair tools integrated into *AIRepair*. Although *AIRepair* is an early prototype, it shows promising results. *AIRepair* will support and test more neural network repair methods and propose a unified interface for developers to test and benchmark their repair methods.

Based on *AIRepair* experiments, a prioritized strategy has been designed for addressing performance anomalies in neural network models. The recommended repair methods are as follows:

1. **Direct Weight Modification:** This method is most effective for simpler neural network architectures, such as those trained on datasets like MNIST and Fashion-MNIST. Direct weight modification is quick and efficient as it does not require access to the original training or testing datasets and can be performed on machines without specialized GPU hardware. However, it may not be sufficient for models with higher complexity or performance requirements, as it might lead to suboptimal solutions or fail to address deep-rooted issues within the network structure.

2. **Fine-Tuning/Retraining:** Retraining or fine-tuning becomes necessary for more complex models, especially those like ResNets with millions of parameters. Direct weight modification in such complex networks often leads to the search space explosion problem, making it impractical. Retraining allows for comprehensive adjustment of the model's parameters in response to the identified anomalies, often yielding more robust and stable solutions. This method, however, may require substantial computational resources, including powerful GPUs, to process the large datasets and extensive parameter space efficiently.

3. **Attaching a New Repair Structure:** In cases where retraining leads to overfitting or when the model needs to enhance its performance on specific tasks or improve constraint accuracies, introducing new repair units or structures to the existing network might be beneficial. These units are designed to correct specific deficiencies or enhance certain aspects of the

model's performance. This approach is particularly relevant when the existing model structure is insufficient to achieve the desired level of accuracy or functionality.

4. **Model and Scenario Specific Considerations:** The choice of repair method should also consider the neural network's specific application scenario. For instance, models trained on the MNIST dataset are often used for handwriting recognition, which may have different performance requirements and constraints compared to models trained on more complex datasets like ImageNet, which is used for detailed image recognition and classification tasks. Furthermore, models deployed in resource-constrained environments, such as mobile or embedded devices, may benefit from specialized repair methods that consider the limitations of these platforms, such as quantized neural network models.

Each repair method has unique advantages and is suitable for different neural network models and application scenarios. Therefore, it is crucial to carefully evaluate the nature of the performance anomaly, the complexity of the neural network model, and the specific application requirements before selecting the most appropriate repair strategy.

Chapter 7

Conclusions and Future Works

This thesis first describes the actual examples and the safety problems in neural network model implementations. Therefore, we need a way to ensure the security of neural networks. Unlike traditional software verification and vulnerability patching, verification and remediation of neural networks remains a challenging problem, as evidenced by the fact that such remediation is not accomplished by directly modifying the code but by improving the learning algorithms and data inputs, which often requires extensive experimentation to find the optimal configuration. This paper then investigates current neural network verification and repair efforts and identifies confusing concepts.

In this thesis, we have presented three contributions to quantized neural network robustness. Firstly, we introduced our new quantized neural network verification method *QNNVerifier* in Chapter 4. It is based on Abstract Interpretation and SMT-based solver ESBMC. We evaluated *QNNVerifier* on two popular datasets, ACAS_Xu and GTSRB, and proved that *QNNRepair* is able to find the vulnerabilities in neural network models effectively. This contribution answered Research Question 1 in Section 1.2.1.

Secondly, we described our new quantized neural network repair method *QNNRepair* in Chapter 5. It converts the neural network repairing a problem into a Mixed Integer Linear Programming (MILP) problem and solves it by using Gurobi as the backend. We also evaluated the *QNNRepair* on CIFAR-10 and MobileNetV2 datasets and verified the effectiveness of the methods. At last, we compared *QNNRepair* with the SOTA data-free quantization method and proved our advantages over it. This contribution answered Research Question 2 in Section 1.2.2.

Finally, we presented our neural network repair platform *AIRepair* in Chapter 6. It provides isolations between different running environments and converts the neural network models from different frameworks like TensorFlow and PyTorch. *AIRepair* also can generate the comparison results of different repairing methods on the same neural network models and give recommendations. We also tested *AIRepair* on the SOTA neural network repairing tools and proved the effectiveness. This contribution answered Research Question 3 in Section 1.2.3.

In the future, we will continue to optimize the tools based on the algorithms in the three chapters above.

For *QNNVerifier*, we aim to significantly enhance the scalability, addressing both the computational efficiency and the broader applicability of *QNNVerifier*. First, we will optimize the conversion process from the neural network model to the C-file, which will effectively reduce the size of the C-file and reduce the time it takes ESBMC to validate a single file. Secondly, we will optimize the approximation mechanism to provide stronger soundness assurances and to minimize performance overhead. Finally, we plan to extend our approach to accommodate DNNs with various layers, such as convolutional, recurrent, and residual layers.

For *QNNRepair*, due to the limitation of LP solver Gurobi, some of the repair problems could not be solved. So, in the future, we intend to improve the encoding from the neural network model and safety properties to LP problems. We will also try other problem solvers and try to encode the repair problems under these solvers, like SMT-based solvers. Also, due to the complexity of neural network models, repairing these large networks will require a lot of computational resources, and we will continuously find a balance between improving accuracy and computing time.

For *AIRepair*, in the future, we aim to support more neural network frameworks and repair tools. Also, we will improve the mechanism of selecting tools when performing repairs on neural network models. Besides, the interface of *AIRepair* will be improved in order to make researchers use this tool easily.

References

[1] M. Krenn, L. Buffoni, B. Coutinho, *et al.*, "Forecasting the future of artificial intelligence with machine learning-based link prediction in an exponentially growing knowledge network," *Nature Machine Intelligence*, vol. 5, no. 11, pp. 1326–1335, 2023 (cited on p. 18).

[2] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015 (cited on p. 18).

[3] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 7, pp. 3523–3542, 2021 (cited on p. 18).

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014 (cited on pp. 18, 19, 104, 106, 126).

[5] A. G. Howard, M. Zhu, B. Chen, *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017 (cited on pp. 18, 19, 34).

[6] J. Guo, H. He, T. He, *et al.*, "Gluoncv and gluonnlp: Deep learning in computer vision and natural language processing," *Journal of Machine Learning Research*, vol. 21, no. 23, pp. 1–7, 2020 (cited on pp. 18, 19).

[7] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016 (cited on p. 18).

[8] S. G. Finlayson, H. W. Chung, I. S. Kohane, and A. L. Beam, "Adversarial attacks against medical deep learning systems," *arXiv preprint arXiv:1804.05296*, 2018 (cited on p. 18).

[9] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*, Ieee, 2017, pp. 39–57 (cited on pp. 19, 20).

[10] X. Huang, D. Kroening, W. Ruan, *et al.*, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100 270, 2020 (cited on pp. 19, 20).

[11] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE symposium on security and privacy (SP)*, IEEE, 2018, pp. 3–18 (cited on pp. 19, 55, 61).

[12] e. a. Yu Bing, "Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment," *IEEE Transactions on Reliability*, 2021 (cited on pp. 19, 62, 69, 121, 125).

[13] M. Sotoudeh and A. V. Thakur, "Provable repair of deep neural networks," in *PLDI*, 2021 (cited on pp. 19, 66, 67, 69).

[14] D. Boetius, S. Leue, and T. Sutter, "A robust optimisation perspective on counterexample-guided repair of neural networks," *arXiv preprint arXiv:2301.11342*, 2023 (cited on pp. 19, 67).

[15] M. Abadi, P. Barham, Chen, *et al.*, "{Tensorflow}: A system for {large-scale} machine learning," in *USENIX OSDI*, 2016 (cited on pp. 19, 24, 116).

[16] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019 (cited on pp. 19, 24, 34, 75, 116).

[17] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678 (cited on pp. 19, 75).

[18] S. E. Hodgson, C. McKenzie, T. W. May, and S. L. Greene, "A comparison of the accuracy of mushroom identification applications using digital photographs," *Clinical Toxicology*, vol. 61, no. 3, pp. 166–172, 2023 (cited on p. 20).

[19] F. Menczer, D. Crandall, Y.-Y. Ahn, and A. Kapadia, "Addressing the harms of ai-generated inauthentic content," *Nature Machine Intelligence*, vol. 5, no. 7, pp. 679–680, 2023 (cited on p. 20).

[20] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014 (cited on pp. 20, 31).

[21] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars,” in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314 (cited on p. 20).

[22] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, “Exploring the landscape of spatial robustness,” in *International conference on machine learning*, PMLR, 2019, pp. 1802–1811 (cited on p. 20).

[23] C. Szegedy, W. Zaremba, I. Sutskever, *et al.*, “Intriguing properties of neural networks,” *arXiv preprint arXiv:1312.6199*, 2013 (cited on pp. 20, 37).

[24] K. Pei, Y. Cao, J. Yang, and S. Jana, “Deepxplore: Automated whitebox testing of deep learning systems,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17, ACM, 2017 (cited on pp. 20, 119).

[25] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, “Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 132–142 (cited on p. 20).

[26] J. Zhang and J. Li, “Testing and verification of neural-network-based safety-critical control software: A systematic literature review,” *Information and Software Technology*, vol. 123, p. 106 296, 2020 (cited on p. 20).

[27] M. Wicki, “How do familiarity and fatal accidents affect acceptance of self-driving vehicles?” *Transportation research part F: traffic psychology and behaviour*, vol. 83, pp. 401–423, 2021 (cited on p. 20).

[28] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, “Survey of approaches and techniques for security verification of computer systems,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 19, no. 1, pp. 1–34, 2023 (cited on p. 20).

[29] D. Castelvecchi, “Can we open the black box of ai?” *Nature News*, vol. 538, no. 7623, p. 20, 2016 (cited on p. 20).

[30] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1219–1219 (cited on p. 21).

[31] D. L. Calsi, M. Duran, X.-Y. Zhang, P. Arcaini, and F. Ishikawa, “Distributed repair of deep neural networks,” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2023, pp. 83–94 (cited on p. 21).

[32] J. Lee, N. Chirkov, E. Ignasheva, *et al.*, "On-device neural net inference with mobile gpus," *arXiv preprint arXiv:1907.01989*, 2019 (cited on p. 22).

[33] X. Song, E. Manino, L. Sena, *et al.*, "Qnnverifier: A tool for verifying neural networks using smt-based model checking," *arXiv preprint arXiv:2111.13110*, 2021 (cited on pp. 23, 26).

[34] L. Sena, X. Song, E. Alves, I. Bessa, E. Manino, L. Cordeiro, *et al.*, "Verifying quantized neural networks using smt-based model checking," *arXiv preprint arXiv:2106.05997*, 2021 (cited on p. 23).

[35] X. Song, Y. Sun, M. A. Mustafa, and L. C. Cordeiro, "Qnnrepair: Quantized neural network repair," in *International Conference on Software Engineering and Formal Methods*, Springer, 2023, pp. 320–339 (cited on pp. 23, 26).

[36] X. Song, Y. Sun, M. A. Mustafa, and L. C. Cordeiro, "Airepair: A repair platform for neural networks," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2023, pp. 98–101 (cited on p. 24).

[37] X. Song, Y. Sun, M. A. Mustafa, and L. Cordeiro, "Airepair: A repair platform for neural networks," in *ICSE-Companion*, IEEE/ACM, 2022 (cited on pp. 26, 67).

[38] M. Uzair and N. Jamil, "Effects of hidden layers on the efficiency of neural networks," in *2020 IEEE 23rd international multitopic conference (INMIC)*, IEEE, 2020, pp. 1–6 (cited on p. 27).

[39] B. K. Spears, "Contemporary machine learning: A guide for practitioners in the physical sciences," *arXiv preprint arXiv:1712.08523*, 2017 (cited on p. 27).

[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016 (cited on pp. 27, 28, 119, 121).

[41] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018 (cited on p. 28).

[42] J. Han and C. Moraga, "The influence of the sigmoid function parameters on the speed of backpropagation learning," in *International workshop on artificial neural networks*, Springer, 1995, pp. 195–201 (cited on p. 28).

[43] B. L. Kalman and S. C. Kwasny, "Why tanh: Choosing a sigmoidal function," in *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, IEEE, vol. 4, 1992, pp. 578–581 (cited on p. 28).

[44] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete, "A survey on modern trainable activation functions," *Neural Networks*, vol. 138, pp. 14–32, 2021 (cited on p. 28).

[45] G. Zhao, Z. Zhang, H. Guan, P. Tang, and J. Wang, "Rethinking relu to train better cnns," in *2018 24th International conference on pattern recognition (ICPR)*, IEEE, 2018, pp. 603–608 (cited on p. 28).

[46] S. Hayou, A. Doucet, and J. Rousseau, "On the impact of the activation function on deep neural networks training," in *International conference on machine learning*, PMLR, 2019, pp. 2672–2680 (cited on p. 29).

[47] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006 (cited on p. 29).

[48] F. Boudardara, A. Boussif, P.-J. Meyer, and M. Ghazel, "A review of abstraction methods toward verifying neural networks," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 4, pp. 1–19, 2024 (cited on pp. 29, 56).

[49] M. P. Owen, A. Panken, R. Moss, L. Alvarez, and C. Leeper, "Acas xu: Integrated collision avoidance and detect and avoid capability for uas," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, IEEE, 2019, pp. 1–10 (cited on pp. 29, 41).

[50] S. Lang, *Undergraduate analysis*. Springer Science & Business Media, 2013 (cited on p. 30).

[51] M. R. Baker and R. B. Patil, "Universal approximation theorem for interval neural networks," *Reliable Computing*, vol. 4, pp. 235–239, 1998 (cited on p. 30).

[52] Y. Lu and J. Lu, "A universal approximation theorem of deep neural networks for expressing probability distributions," *Advances in neural information processing systems*, vol. 33, pp. 3094–3105, 2020 (cited on p. 30).

[53] M. J. Kochenderfer and J. Chryssanthacopoulos, "Robust airborne collision avoidance through dynamic programming," *Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-371*, vol. 130, 2011 (cited on pp. 31, 43).

[54] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, IEEE, 2016, pp. 1–10 (cited on pp. 31, 43, 44, 84).

[55] S. Shanmuganathan, *Artificial neural network modelling: An introduction*. Springer, 2016 (cited on p. 31).

[56] L. R. Medsker, L. Jain, *et al.*, “Recurrent neural networks,” *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001 (cited on p. 31).

[57] H. Hewamalage, C. Bergmeir, and K. Bandara, “Recurrent neural networks for time series forecasting: Current status and future directions,” *International Journal of Forecasting*, vol. 37, no. 1, pp. 388–427, 2021 (cited on p. 31).

[58] P. Liu, X. Qiu, and X. Huang, “Recurrent neural network for text classification with multi-task learning,” *arXiv preprint arXiv:1605.05101*, 2016 (cited on p. 31).

[59] Y. LeCun, C. Cortes, and C. Burges. “The mnist database of handwritten digits.” (1998), [Online]. Available: `http://yann.lecun.com/exdb/mnist/` (cited on pp. 31, 118).

[60] R. Venkatesan and B. Li, *Convolutional neural networks in visual computing: a concise guide*. CRC Press, 2017 (cited on p. 32).

[61] Y. LeCun, Y. Bengio, *et al.*, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995 (cited on p. 32).

[62] e. a. He Kaiming, “Deep residual learning for image recognition,” *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016 (cited on pp. 33, 106, 125, 126).

[63] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520 (cited on pp. 34, 106, 126).

[64] A. Howard, M. Sandler, G. Chu, *et al.*, “Searching for mobilenetv3,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324 (cited on p. 34).

[65] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010 (cited on p. 34).

[66] J. Bai, F. Lu, K. Zhang, *et al.*, *Onnx: Open neural network exchange*, `https://github.com/onnx/onnx`, 2019 (cited on p. 35).

[67] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, “A white paper on neural network quantization,” *arXiv preprint arXiv:2106.08295*, 2021 (cited on pp. 35, 36).

[68] "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713 (cited on p. 35).

[69] Q. Ducasse, P. Cotret, L. Lagadec, and R. Stewart, "Benchmarking quantized neural networks on fpgas with finn," *arXiv preprint arXiv:2102.01341*, 2021 (cited on p. 35).

[70] H. Benmaghnia, M. Martel, and Y. Seladji, "Code generation for neural networks based on fixed-point arithmetic," *ACM Transactions on Embedded Computing Systems (TECS)*, 2022 (cited on p. 35).

[71] 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: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II 26*, Springer, 2020, pp. 79–97 (cited on pp. 35, 60, 89, 92).

[72] R. David, Duke, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, 2021 (cited on pp. 36, 75).

[73] H. Vanholder, "Efficient inference with tensorrt," in *GPU Technology Conference*, vol. 1, 2016, p. 2 (cited on p. 36).

[74] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4820–4828 (cited on p. 36).

[75] W. Roth, G. Schindler, B. Klein, *et al.*, "Resource-efficient neural networks for embedded systems," *arXiv preprint arXiv:2001.03048*, 2020 (cited on p. 36).

[76] M. R. Biswal, T. S. Delwar, A. Siddique, P. Behera, Y. Choi, and J.-Y. Ryu, "Pattern classification using quantized neural networks for fpga-based low-power iot devices," *Sensors*, vol. 22, no. 22, p. 8694, 2022 (cited on p. 36).

[77] J. Rock, W. Roth, P. Meissner, and F. Pernkopf, "Quantized neural networks for radar interference mitigation," *arXiv preprint arXiv:2011.12706*, 2020 (cited on p. 36).

[78] H. D. M. Ribeiro, A. Arnold, J. P. Howard, *et al.*, "Ecg-based real-time arrhythmia monitoring using quantized deep neural networks: A feasibility study," *Computers in Biology and Medicine*, vol. 143, p. 105 249, 2022 (cited on p. 37).

[79] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255 (cited on pp. 37, 106, 126).

[80] A. Kurakin, I. Goodfellow, and S. Bengio, “Adversarial machine learning at scale,” *arXiv preprint arXiv:1611.01236*, 2016 (cited on p. 37).

[81] E. M. Clarke and Q. Wang, “25 years of model checking.,” in *Ershov Memorial Conference*, 2014, pp. 26–40 (cited on p. 38).

[82] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t),” *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006 (cited on p. 38).

[83] L. Chaves, I. Bessa, L. Cordeiro, and D. Kroening, “Dsvalidator: An automated counterexample reproducibility tool for digital systems,” in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, 2018, pp. 253–258 (cited on p. 38).

[84] Q. Wang, “Model checking for biological systems: Languages, algorithms, and applications,” Ph.D. dissertation, Ph. D. thesis, Carnegie Mellon University, 2016 (cited on p. 38).

[85] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997 (cited on pp. 38, 82).

[86] L. Cordeiro, B. Fischer, and J. Marques-Silva, “Smt-based bounded model checking for embedded ansi-c software,” *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011 (cited on p. 38).

[87] S. A. Kripke, “Semantical analysis of modal logic i normal modal propositional calculi,” *Mathematical Logic Quarterly*, vol. 9, no. 5-6, pp. 67–96, 1963 (cited on p. 38).

[88] H. Garavel, R. Mateescu, and I. Smarandache, “Parallel state space construction for model-checking,” in *Model Checking Software: 8th International SPIN Workshop Toronto, Canada, May 19–20, 2001 Proceedings 8*, Springer, 2001, pp. 217–234 (cited on p. 38).

[89] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” in *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS’99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS’99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*, Springer, 1999, pp. 193–207 (cited on pp. 38, 39).

[90] M. Sheeran, S. Singh, and G. Stålmarck, “Checking safety properties using induction and a sat-solver,” in *International conference on formal methods in computer-aided design*, Springer, 2000, pp. 127–144 (cited on p. 39).

[91] K. L. McMillan, “Applying sat methods in unbounded symbolic model checking,” in *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, Springer, 2002, pp. 250–264 (cited on p. 39).

[92] K. L. McMillan, “Interpolation and sat-based model checking,” in *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*, Springer, 2003, pp. 1–13 (cited on pp. 39, 40).

[93] M. K. Ganai, A. Gupta, and P. Ashar, “Efficient sat-based unbounded symbolic model checking using circuit cofactoring,” in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, IEEE, 2004, pp. 510–517 (cited on p. 39).

[94] R. Menezes, M. Aldughaim, B. Farias, *et al.*, “Esbmc v7. 4: Harnessing the power of intervals,” *arXiv preprint arXiv:2312.14746*, 2023 (cited on pp. 39, 80).

[95] A. Cimatti, E. Clarke, E. Giunchiglia, *et al.*, “Nusmv 2: An opensource tool for symbolic model checking,” in *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, Springer, 2002, pp. 359–364 (cited on p. 39).

[96] F. Merz, S. Falke, and C. Sinz, “Llbmc: Bounded model checking of c and c++ programs using a compiler ir,” in *International Conference on Verified Software: Tools, Theories, Experiments*, Springer, 2012, pp. 146–161 (cited on p. 39).

[97] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, “Jbmc: A bounded model checking tool for verifying java bytecode,” in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 183–190 (cited on p. 39).

[98] D. Kroening and M. Tautschnig, “Cbmc–c bounded model checker: (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, Springer, 2014, pp. 389–391 (cited on p. 39).

[99] D. Monniaux, “A survey of satisfiability modulo theory,” in *Computer Algebra in Scientific Computing: 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings 18*, Springer, 2016, pp. 401–425 (cited on p. 39).

[100] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 53–58, 2014 (cited on p. 40).

[101] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340 (cited on p. 40).

[102] A. Niemetz and M. Preiner, “Bitwuzla at the smt-comp 2020,” *arXiv preprint arXiv:2006.01621*, 2020 (cited on p. 40).

[103] A. R. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007 (cited on p. 41).

[104] M. O’Searcoid, *Metric spaces*. Springer Science & Business Media, 2006 (cited on p. 41).

[105] S. J. Oh, B. Schiele, and M. Fritz, “Towards reverse-engineering black-box neural networks,” *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 121–144, 2019 (cited on p. 41).

[106] W. Xiang, H.-D. Tran, and T. T. Johnson, “Output reachable set estimation and verification for multilayer neural networks,” *IEEE transactions on neural networks and learning systems*, vol. 29, no. 11, pp. 5777–5783, 2018 (cited on p. 41).

[107] H. Zhang, P. Zhang, and C.-J. Hsieh, “Recurjac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5757–5764 (cited on p. 41).

[108] L. Weng, H. Zhang, H. Chen, *et al.*, “Towards fast computation of certified robustness for relu networks,” in *International Conference on Machine Learning*, PMLR, 2018, pp. 5276–5285 (cited on p. 41).

[109] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, “The third international verification of neural networks competition (vnn-comp 2022): Summary and results,” *arXiv preprint arXiv:2212.10376*, 2022 (cited on pp. 41, 55, 81).

[110] D. Beyer and A. Podelski, “Software model checking: 20 years and beyond,” in *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, Springer, 2022, pp. 554–582 (cited on p. 42).

[111] R. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. K. Mudigonda, “A unified view of piecewise linear neural network verification,” *Advances in Neural Information Processing Systems*, vol. 31, 2018 (cited on p. 43).

[112] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, “Reluplex: An efficient smt solver for verifying deep neural networks,” in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, Springer, 2017, pp. 97–117 (cited on pp. 43, 54, 57, 80, 92).

[113] S. Vani and T. M. Rao, “An experimental approach towards the performance assessment of various optimizers on convolutional neural network,” in *2019 3rd international conference on trends in electronics and informatics (ICOEI)*, IEEE, 2019, pp. 331–336 (cited on p. 44).

[114] X. Xie, L. Ma, F. Juefei-Xu, *et al.*, “Deephunter: A coverage-guided fuzz testing framework for deep neural networks,” in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 146–157 (cited on p. 46).

[115] X. Zhang, X. Xie, L. Ma, *et al.*, “Towards characterizing adversarial defects of deep learning software from the lens of uncertainty,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 739–751 (cited on p. 46).

[116] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, “A comprehensive study on challenges in deploying deep learning based software,” in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 750–762 (cited on p. 46).

[117] S. H. Silva and P. Najafirad, “Opportunities and challenges in deep learning adversarial robustness: A survey,” *arXiv preprint arXiv:2007.00753*, 2020 (cited on p. 46).

[118] H. Zhang and W. Chan, “Apricot: A weight-adaptation approach to fixing deep learning models,” in *ASE*, IEEE, 2019 (cited on pp. 47, 63, 69, 125).

[119] Z. Liang, T. Wu, C. Zhao, *et al.*, “Repairing deep neural networks based on behavior imitation,” *arXiv preprint arXiv:2305.03365*, 2023 (cited on p. 49).

[120] A. H. Land and A. G. Doig, *An automatic method for solving discrete programming problems*. Springer, 2010 (cited on p. 50).

[121] G. Optimization, *Inc.. gurobi optimizer reference manual, version 5.0*, 2012 (cited on pp. 50, 103, 106, 126).

[122] B. Meindl and M. Templ, "Analysis of commercial and free and open source solvers for linear optimization problems," *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, vol. 20, 2012 (cited on p. 50).

[123] S. Nickel, C. Steinhardt, H. Schlenker, and W. Burkart, "Ibm ilog cplex optimization studio—a primer," in *Decision Optimization with IBM ILOG CPLEX Optimization Studio: A Hands-On Introduction to Modeling with the Optimization Programming Language (OPL)*, Springer, 2022, pp. 9–21 (cited on p. 51).

[124] A. Makhorin, "Glpk (gnu linear programming kit)," *http://www. gnu. org/s/glpk/glpk. html*, 2008 (cited on pp. 51, 57, 60).

[125] E. Bressert, "Scipy and numpy: An overview for developers," 2012 (cited on p. 51).

[126] S. M. Kolb, S. Teso, A. Passerini, L. De Raedt, *et al.*, "Learning smt (lra) constraints using smt solvers," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence ((IJCAI-18)*, IJCAI, 2018, pp. 2333–2340 (cited on p. 54).

[127] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*, Springer, 2017, pp. 269–286 (cited on pp. 54, 57).

[128] N. Narodytska, "Formal analysis of deep binarized neural networks.," in *IJCAI*, 2018, pp. 5692–5696 (cited on p. 54).

[129] N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018 (cited on p. 54).

[130] Y. Y. Elboher, J. Gottschlich, and G. Katz, "An abstraction-based framework for neural network verification," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, Springer, 2020, pp. 43–65 (cited on p. 55).

[131] G. Sierksma and Y. Zwols, *Linear and integer optimization: theory and practice*. CRC Press, 2015 (cited on pp. 55, 60).

[132] M. Fowler, *Refactoring*. Addison-Wesley Professional, 2018 (cited on p. 55).

[133] K. Ghorbal, E. Goubault, and S. Putot, “The zonotope abstract domain taylor1+,” in *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, Springer, 2009, pp. 627–633 (cited on p. 55).

[134] S. Wang, H. Zhang, K. Xu, *et al.*, “Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 29 909–29 921, 2021 (cited on pp. 55, 61, 92).

[135] T. Ladner and M. Althoff, “Exponent relaxation of polynomial zonotopes and its applications in formal neural network verification,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 21 304–21 311 (cited on p. 55).

[136] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252 (cited on p. 55).

[137] H.-D. Tran, N. Pal, D. M. Lopez, *et al.*, “Verification of piecewise deep neural networks: A star set approach with zonotope pre-filter,” *Formal aspects of computing*, vol. 33, pp. 519–545, 2021 (cited on p. 56).

[138] C.-Y. Ko, Z. Lyu, L. Weng, L. Daniel, N. Wong, and D. Lin, “Popqorn: Quantifying robustness of recurrent neural networks,” in *International Conference on Machine Learning*, PMLR, 2019, pp. 3468–3477 (cited on p. 56).

[139] T. Du, S. Ji, L. Shen, *et al.*, “Cert-rnn: Towards certifying the robustness of recurrent neural networks.,” *CCS*, vol. 21, no. 2021, pp. 15–19, 2021 (cited on p. 56).

[140] D. Eppstein, “Zonohedra and zonotopes,” 1995 (cited on p. 56).

[141] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*, Springer, 2000, pp. 154–169 (cited on p. 56).

[142] H.-D. Tran, S. Bak, W. Xiang, and T. T. Johnson, “Verification of deep convolutional neural networks using imagestars,” in *International conference on computer aided verification*, Springer, 2020, pp. 18–42 (cited on p. 57).

[143] H.-D. Tran, D. Manzanas Lopez, P. Musau, *et al.*, “Star-based reachability analysis of deep neural networks,” in *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings 3*, Springer, 2019, pp. 670–686 (cited on p. 57).

[144] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, “Efficient formal safety analysis of neural networks,” *Advances in neural information processing systems*, vol. 31, 2018 (cited on p. 57).

[145] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, “Formal security analysis of neural networks using symbolic intervals,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1599–1614 (cited on p. 57).

[146] G. Katz, D. A. Huang, D. Ibeling, *et al.*, “The marabou framework for verification and analysis of deep neural networks,” in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, Springer, 2019, pp. 443–452 (cited on pp. 57, 61, 92, 93).

[147] M. Fisher, *An introduction to practical formal methods using temporal logic*. John Wiley & Sons, 2011 (cited on p. 58).

[148] G. Optimization *et al.*, *Gurobi optimizer reference manual*, 2020 (cited on p. 60).

[149] C. Bliek1ú, P. Bonami, and A. Lodi, “Solving mixed-integer quadratic programming problems with ibm-cplex: A progress report,” in *Proceedings of the twenty-sixth RAMP symposium*, 2014, pp. 16–17 (cited on p. 60).

[150] T. A. Henzinger, M. Lechner, and Đ. Žikelić, “Scalable verification of quantized neural networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 3787–3795 (cited on pp. 60, 61).

[151] V. Tjeng, K. Xiao, and R. Tedrake, “Evaluating robustness of neural networks with mixed integer programming,” *arXiv preprint arXiv:1711.07356*, 2017 (cited on p. 60).

[152] Y. Zhang, F. Song, and J. Sun, “Qebverif: Quantization error bound verification of neural networks,” in *International Conference on Computer Aided Verification*, Springer, 2023, pp. 413–437 (cited on pp. 60, 61).

[153] Y. Zhang, Z. Zhao, G. Chen, *et al.*, “Qvip: An ilp-based formal verification approach for quantized neural networks,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13 (cited on pp. 60, 61).

[154] M. V. Marathe, H. B. Hunt III, R. E. Stearns, and V. Radhakrishnan, “Approximation algorithms for pspace-hard hierarchically and periodically specified problems,” *SIAM Journal on Computing*, vol. 27, no. 5, pp. 1237–1261, 1998 (cited on p. 60).

[155] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, 2012 (cited on pp. 60, 125).

[156] K. Jia and M. Rinard, “Efficient exact verification of binarized neural networks,” *Advances in neural information processing systems*, vol. 33, pp. 1782–1795, 2020 (cited on p. 61).

[157] P. Huang, H. Wu, Y. Yang, *et al.*, “Towards efficient verification of quantized neural networks,” *arXiv preprint arXiv:2312.12679*, 2023 (cited on p. 61).

[158] H. Wu, O. Isac, A. Zeljić, *et al.*, “Marabou 2.0: A versatile formal analyzer of neural networks,” in *International Conference on Computer Aided Verification*, Springer, 2024, pp. 249–264 (cited on p. 61).

[159] H. Duong, D. Xu, T. Nguyen, and M. B. Dwyer, “Harnessing neuron stability to improve dnn verification,” *arXiv preprint arXiv:2401.14412*, 2024 (cited on pp. 61, 92).

[160] A. Sinitsin, V. Plokhotnyuk, D. Pyrkin, S. Popov, and A. Babenko, “Editable neural networks,” *arXiv preprint arXiv:2004.00345*, 2020 (cited on p. 63).

[161] J. R. Hershey and P. A. Olsen, “Approximating the kullback leibler divergence between gaussian mixture models,” in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP’07*, IEEE, vol. 4, 2007, pp. IV–317 (cited on p. 63).

[162] e. a. Goldberger Ben, “Minimal modifications of deep neural networks using verification.,” in *LPAR*, 2020, 23rd (cited on p. 64).

[163] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu, “Nn repair: Constraint-based repair of neural network classifiers,” in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, Springer, 2021, pp. 3–25 (cited on pp. 64, 65).

[164] K. Majd, S. Zhou, H. B. Amor, G. Fainekos, and S. Sankaranarayanan, “Local repair of neural networks using optimization,” *arXiv preprint arXiv:2109.14041*, 2021 (cited on p. 64).

[165] J. Sohn, S. Kang, and S. Yoo, “Arachne: Search-based repair of deep neural networks,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–26, 2023 (cited on pp. 64, 69).

[166] Z. Tao, S. Nawas, J. Mitchell, and A. V. Thakur, "Architecture-preserving provable repair of deep neural networks," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 443–467, 2023 (cited on p. 64).

[167] F. Fu and W. Li, "Sound and complete neural network repair with minimality and locality guarantees," *arXiv preprint arXiv:2110.07682*, 2021 (cited on p. 64).

[168] D. Gopinath, H. Converse, C. Pasareanu, and A. Taly, "Property inference for deep neural networks," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 797–809 (cited on p. 65).

[169] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005 (cited on p. 65).

[170] B. Sun, J. Sun, L. H. Pham, and J. Shi, "Causality-based neural network repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 338–349 (cited on p. 66).

[171] T. M. Blackwell, J. Kennedy, and R. Poli, "Particle swarm optimization," *Swarm Intelligence*, vol. 1, no. 1, pp. 33–57, 2007 (cited on p. 66).

[172] T. S. Borkar and L. J. Karam, "Deepcorrect: Correcting dnn models against image distortions," *IEEE Transactions on Image Processing*, vol. 28, no. 12, pp. 6022–6034, 2019 (cited on pp. 67, 69).

[173] F. Fu, Z. Wang, J. Fan, *et al.*, "Reglo: Provable neural network repair for global robustness properties," in *Workshop on Trustworthy and Socially Responsible Machine Learning, NeurIPS 2022*, 2022 (cited on p. 68).

[174] F. Bauer-Marquart, D. Boetius, S. Leue, and C. Schilling, "Specrepair: Counter-example guided safety repair of deep neural networks," in *International Symposium on Model Checking Software*, Springer, 2022, pp. 79–96 (cited on pp. 68, 69).

[175] G. Dong, J. Sun, J. Wang, X. Wang, and T. Dai, "Towards repairing neural networks correctly," *arXiv preprint arXiv:2012.01872*, 2020 (cited on p. 68).

[176] H. F. Eniser, S. Gerasimou, and A. Sen, "Deepfault: Fault localization for deep neural networks," in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2019, pp. 171–191 (cited on p. 69).

[177] L. H. Pham, J. Li, and J. Sun, "Socrates: Towards a unified platform for neural network analysis," *arXiv preprint arXiv:2007.11206*, 2020 (cited on p. 69).

[178] 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 (cited on p. 74).

[179] E. Stevens, L. Antiga, and T. Viehmann, *Deep learning with PyTorch*. Manning Publications, 2020 (cited on p. 75).

[180] S. Koranne and S. Koranne, “Hierarchical data format 5: Hdf5,” *Handbook of open source tools*, pp. 191–200, 2011 (cited on p. 75).

[181] M. Ó. Searcóid, *Metric Spaces*. Springer-Verlag, 2006 (cited on p. 77).

[182] D. Bühler, “Eva, an evolved value analysis for frama-c: Structuring an abstract interpreter through value and state abstractions,” Ph.D. dissertation, Rennes 1, 2017 (cited on p. 79).

[183] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, “Acsl: Ansi c specification language,” *CEA-LIST, Saclay, France, Tech. Rep. v1*, vol. 2, 2008 (cited on p. 79).

[184] N. Kosmatov and J. Signoles, “Frama-c, a collaborative framework for c code verification: Tutorial synopsis,” in *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings 7*, Springer, 2016, pp. 92–115 (cited on p. 79).

[185] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969 (cited on p. 80).

[186] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, ieee, 1977, pp. 46–57 (cited on p. 80).

[187] B. Dutertre, “Yices 2.2,” in *International Conference on Computer Aided Verification*, Springer, 2014, pp. 737–744 (cited on p. 82).

[188] M. N. Wegman and F. K. Zadeck, “Constant propagation with conditional branches,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991 (cited on p. 82).

[189] M. Y. Gadelha, L. C. Cordeiro, and D. A. Nicole, “Encoding floating-point numbers using the smt theory in esbmc: An empirical evaluation over the sv-comp benchmarks,” in *Formal Methods: Foundations and Applications: 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29—December 1, 2017, Proceedings 20*, Springer, 2017, pp. 91–106 (cited on p. 83).

[190] A. Stoutchinin and F. De Ferriere, “Efficient static single assignment form for predication,” in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, IEEE, 2001, pp. 172–181 (cited on p. 83).

[191] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, “The german traffic sign recognition benchmark: A multi-class classification competition,” in *The 2011 international joint conference on neural networks*, IEEE, 2011, pp. 1453–1460 (cited on pp. 84, 85).

[192] Krizhevsky, Alex and Hinton, Geoffrey, “CIFAR-10 (canadian institute for advanced research),” University of Toronto, Tech. Rep., 2009. [Online]. Available: `https://www.cs.toronto.edu/~kriz/cifar.html` (cited on pp. 85, 106, 126).

[193] 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 (cited on p. 86).

[194] 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 (cited on p. 87).

[195] T. A. Henzinger, M. Lechner, and Đ. Žikelić, *Scalable verification of quantized neural networks (technical report)*, 2020. eprint: `2012.08185` (cited on p. 89).

[196] 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 (cited on p. 92).

[197] K. Jia and M. Rinard, “Verifying low-dimensional input neural networks via input quantization,” *arXiv preprint arXiv:2108.07961*, 2021 (cited on p. 92).

[198] 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 (cited on p. 92).

[199] M. Christakis, H. F. Eniser, H. Hermanns, *et al.*, “Automated safety verification of programs invoking neural networks,” in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, Springer, 2021, pp. 201–224 (cited on p. 92).

[200] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Dpll (t): Fast decision procedures,” in *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, Springer, 2004, pp. 175–188 (cited on p. 92).

[201] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282 (cited on pp. 101, 109).

[202] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “On the accuracy of spectrum-based fault localization,” in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, IEEE, 2007 (cited on pp. 101, 109).

[203] W. E. Wong, V. Debroy, R. Gao, and Y. Li, “The dstar method for effective software fault localization,” *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013 (cited on pp. 101, 109).

[204] P. Agarwal and A. P. Agrawal, “Fault-localization techniques for software systems: A literature review,” *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, 2014 (cited on pp. 101, 109).

[205] V. Dallmeier, C. Lindig, and A. Zeller, “Lightweight bug localization with ample,” in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005, pp. 99–104 (cited on pp. 101, 109).

[206] Z. Galijasevic and A. Abur, “Fault location using voltage measurements,” *IEEE Transactions on Power Delivery*, vol. 17, no. 2, pp. 441–445, 2002 (cited on pp. 101, 109).

[207] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, “Effective fault localization using code coverage,” in *COMPSAC*, IEEE, vol. 1, 2007, pp. 449–456 (cited on pp. 101, 109).

[208] *TensorFlow Lite*. [Online]. Available: `https://www.tensorflow.org/lite` (cited on p. 105).

[209] ifigotin, *Imagenetmini-1000*, `https://www.kaggle.com/datasets/ifigotin/imagenetmini-1000`, Accessed: April 4, 2023, 2021 (cited on p. 105).

[210] e. a. Guo Cong, “Squant: On-the-fly data-free quantization via diagonal hessian approximation,” *arXiv preprint arXiv:2202.07471*, 2022 (cited on p. 112).

[211] G. L. Nemhauser and L. A. Wolsey, “Integer and combinatorial optimization john wiley & sons,” *New York*, vol. 118, 1988 (cited on p. 112).

[212] S. Bengesi, H. El-Sayed, M. K. Sarker, Y. Houkpati, J. Irungu, and T. Oladunni, “Advancements in generative ai: A comprehensive review of gans, gpt, autoencoders, diffusion model, and transformers.,” *IEEE Access*, 2024 (cited on p. 113).

[213] F. Chollet, *Keras*, https://keras.io, 2015 (cited on pp. 117, 118).

[214] C. Severance, “Discovering javascript object notation,” *Computer*, vol. 45, no. 4, pp. 6–8, 2012 (cited on p. 117).

[215] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms,” *arXiv:1708.07747*, 2017 (cited on pp. 118, 125).

[216] A. Krizhevsky, “Learning multiple layers of features from tiny images,” University of Toronto, Tech. Rep., 2009 (cited on pp. 119, 125).

[217] C. Shorten and T. M. Khoshgoftaar, “A survey on image data augmentation for deep learning,” *Journal of Big Data*, vol. 6, no. 1, p. 60, 2019 (cited on p. 119).

[218] Y. Liu, C. Chen, R. Zhang, *et al.*, “Enhancing the interoperability between deep learning frameworks by model conversion,” in *ESEC/FSE*, 2020, pp. 1320–1330 (cited on p. 119).

[219] G. Rozenberg, *Handbook of graph grammars and computing by graph transformation*. World scientific, 1997, vol. 1 (cited on p. 120).

[220] I. Anaconda, *Anaconda documentation*, `https://docs.anaconda.com/`, 2020 (cited on pp. 120, 121, 123).

[221] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, and M. Vechev, “DL2: Training and querying neural networks with logic,” in *ICML*, 2019 (cited on pp. 121, 125).

[222] A. Laugros, A. Caplier, and M. Ospici, “Are adversarial robustness and common perturbation robustness independant attributes?” In *ICCV*, 2019 (cited on p. 122).

[223] D. Crockford, *The application/json media type for javascript object notation (json)*, `https://www.ietf.org/rfc/rfc4627.txt`, 2006 (cited on p. 123).

[224] B. Neal, *Effective Command-Line Interface*. Pragmatic Bookshelf, 2018 (cited on p. 123).

[225] Z. Liu, Y. Feng, Y. Yin, and Z. Chen, “DeepState: Selecting test suites to enhance the robustness of recurrent neural networks,” in *ICSE*, 2022 (cited on p. 125).

[226] X. Xie, W. Guo, L. Ma, *et al.*, “Rnnrepair: Automatic rnn repair via model-based analysis,” in *ICML*, 2021 (cited on p. 125).

[227] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "There is no free lunch in adversarial robustness (but there are unexpected benefits)," *arXiv:1805.12152*, vol. 2, no. 3, 2018 (cited on p. 127).