SAT-based analysis of DNNs deployed in safety critical systems

Abdellah Harous

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

September 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Abdellah Harous**

Entitled: **SAT-based analysis of DNNs deployed in safety critical systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair/Internal Examiner
Dr. Abdelwahab Hamou-Lhadj

_____ External Examiner
Dr. Amr Youssef

_____ Supervisor
Dr. Otmane Ait Mohamed

Approved by _____
Yousef R.Shayan, Chair
Department of Electrical and Computer Engineering

_____ 2023

_____
Mourad Dabbabi, Dean
Faculty of Engineering and Computer Science

Abstract

SAT-based analysis of DNNs deployed in safety critical systems

Abdellah Harous

The analysis of Deep Neural Networks (DNNs) used in safety-critical systems using SAT techniques is covered in this thesis, which emphasizes how crucial it is to verify the networks' safety and reliability. Ensuring the proper behavior of these networks becomes essential for the safety of humans and the integrity of the systems involved as DNNs are increasingly integrated into safety-critical systems including autonomous vehicles, medical diagnosis, and aerospace systems.

The main objective of this research is to formally verify a deep neural network, namely the Vertical Collision Avoidance System (VCAS), deployed in safety-critical applications. The verification procedure is intended to ensure that the network complies with security requirements and performs dependably under different conditions. Through the process of verification, the DNN's predictable behavior is ensured, and the possibility of malfunctions that can result in risk or system failures is reduced. In order to get important insights into the behavior and vulnerabilities of DNNs used in safety-critical systems, this work investigates the application of SAT-based analytic methodologies.

The methodology employed in this research involves a comprehensive understanding of the DNN's structure, training process, and inference mechanisms. By comprehending the underlying principles and potential risks associated with DNNs, the verification process can be tailored to address specific safety concerns and requirements. Furthermore, fault injection techniques, including Single Event Upsets (SEUs) and Multiple Bit Upsets

(MBUs), are used to assess the network's resilience and their ability to recover from faults, contributing to a more comprehensive understanding of DNN robustness in safety-critical contexts. This research offers invaluable insights for experts by addressing the verification difficulties of DNNs in safety-critical systems. The findings contribute to the ongoing efforts to develop standardized verification methodologies and safety guidelines for DNN deployment, ensuring the reliability, trustworthiness, and safety of these networks in critical applications.

Acknowledgments

My sincere thanks goes out to Professor Otmane Ait Mohammed, who served as my thesis advisor, for his tremendous guidance, encouragement, and support throughout my research. His knowledge and perception have greatly influenced the course of my work and assisted me in overcoming obstacles. I am immensely grateful to have had him as my advisor during my time at Concordia University.

I also appreciate my HVG lab colleagues' cooperation and fruitful discussions, which have improved my knowledge of the subject and given me insightful criticism of my work. My stay at the lab has been productive and pleasurable thanks to their friendship and assistance.

Finally, I want to express my gratitude to my friends and family for their unfailing love, inspiration, and support during my academic career. I am eternally indebted to them for being in my life; their faith in me and my skills has been a constant source of inspiration and drive.

Contents

Bibliography **56**

List of Figures

List of Tables

Chapter 1

Introduction

Deep Neural Networks (DNNs) are becoming nowadays powerful tools to approximate complex functions (e.g. classification and regression tasks) via learning. And because of the advances in technology and the availability of powerful processing systems, DNNs are becoming deeper, more efficient, and used in an ever broader extent of domains including safety-critical systems, such as autonomous driving, intruder detection, and collision avoidance system in aircraft. In such systems, reliability concerns are raised and should comply with DO-178C/ED-12C functional safety standards for airborne systems, where the software standard released by the Radio Technical Commission for Aeronautics (RTCA) is DO-178C, while EUROCAE published ED-12C as its European equivalent. For instance, the evaluated FIT (Failures In Time) rates of hardware components must be 1 to 10 (meaning 1 to 10 failures per billion hours of operation) to pass the highest reliability level which requires diligent design in the case of avionics. These systems' complexity also makes it possible for faults and errors to seep in and cause inaccurate or unexpected behavior. Verification, the process of ensuring that a system works as planned and is error-free, is now necessary.

Any system's development process must include verification since it can help identify and fix faults early on before their deployment in the field. This is crucial in safety-critical

systems since even a small mistake can have disastrous results. Verification can also serve to increase a system's dependability and trustworthiness, which is crucial in applications where people's lives or significant sums of money are on the line. In the study of neural networks, verification is one area that is especially crucial. A class of machine learning models known as neural networks has been utilized to achieve cutting-edge performance in a variety of applications, including natural language processing and picture identification. These models can be challenging to comprehend and debug due to their high level of complexity, which includes thousands or even millions of parameters.

The significance of this topic cannot be emphasized, even if neural network verification is still an active area of research. It is crucial that we have trustworthy and dependable methods for evaluating neural networks' behavior because their use in applications ranging from medical treatment to financial trading is only going to increase. This will make it easier to guarantee that these models are secure, efficient, and advantageous for society as a whole.

1.1 Collision Avoidance Systems

Systems created to avoid collisions between moving objects are called collision avoidance systems (CAS), also known as collision avoidance technologies (CAT). Aircrafts with CAS are made to help pilots steer clear of collisions with other planes, the ground, or other objects. The Traffic Alert and Collision Avoidance System (TCAS) [1] and the Ground Proximity Warning System (GPWS) [2] are the two CAS types found in most modern aircrafts.

TCAS is a surveillance system that uses transponder signals from other aircraft to locate, track, and measure speed. In order to prevent potential collisions, TCAS gives pilots traffic advisories (TAs) and resolution advisories (RAs). RAs offer detailed instructions on how to avoid a potential collision, while TAs are sent out when another aircraft is nearby.

On the other hand, GPWS is a system that keeps track of an aircraft's height and alerts pilots when they are in risk of running into obstacles or the terrain. The system uses GPS and radar altimeters to determine the position and altitude of the aircraft in relation to the ground. The GPWS will alert the pilot if the aircraft is flying too low or is headed directly for terrain or other impediments. Controlled flight into terrain (CFIT) incidents and mid-air collisions have both been significantly decreased thanks to TCAS and GPWS.

While CAS can help pilots avoid crashes, it is crucial to remember that they cannot take the place of solid piloting abilities and situational awareness. Pilots need to be on the lookout and augment their own judgment and decision-making with CAS. In conclusion, collision avoidance technology is essential for maintaining aviation safety. Modern aircraft uses CAS systems like TCAS and GPWS, both of which have been effective in lowering the number of crashes and accidents. They do not, however, serve as a replacement for sound piloting techniques and situational awareness.

1.2 Faults in Neural Networks and Embedded Systems

Neural networks and embedded systems errors can have serious repercussions, especially in safety-critical applications. Since neural networks use probability distributions to process data, they can introduce unpredictability and make it difficult to guarantee precise and predictable system behaviour. When neural networks are included, incorporating fault tolerance into embedded system design becomes considerably more difficult. Many different things can go wrong, including malicious attacks, software bugs, and hardware malfunctions. By investigating and fortifying neural networks against probable flaws, as well as by using fault injection techniques to gauge their dependability in safety-critical situations, the impact of these flaws can be reduced. For embedded systems to remain secure and safe, neural network dependability must be guaranteed.

Single Event Upset (SEU) is a brief information loss in memory or logic components

brought on by powerful ionizing radiation. As ionized radiation particles enter the silicon substrate of a transistor, they may produce electron-hole pairs [3]. Following the generation of electron-hole pairs, the electric charge is transported to the transistor's drain region via processes like diffusion and drift. In memory components, the accumulated charge builds up until it eventually causes a glitch in the damaged transistor, disturbing the data that has been stored. Alpha particles from package material and neutrons from cosmic rays are the two types of particles that typically cause SEU in a terrestrial environment. Package materials with low alpha particle emission are utilized in security-sensitive applications to reduce alpha particle-induced SEU. SEU, however, poses a threat to terrestrial VLSIs and so calls for specialized safeguards because it is difficult to shield against the numerous neutrons in cosmic rays because they pass through earthen materials. MBU (Multiple Bit Upset) is another fault that may occur in a neural network when a single event upset causes several bits to flip [4].

1.3 Problem Statement

Incorporating neural networks into embedded systems, particularly in safety-critical applications is one of the most crucial challenges nowadays. These systems have strict requirements regarding safety, security, availability, and timing. Neural networks, while powerful for processing data, introduce unpredictability because they calculate output based on probability distributions. This unpredictability can make it difficult to ensure the system's behavior remains accurate and predictable, which is important for analyzing and identifying worst-case outcomes. Additionally, fault tolerance is crucial in embedded system design, and it becomes even more challenging when neural networks are involved. Nevertheless, researchers can investigate and harden neural networks against potential faults to mitigate their effects down to an acceptable threshold.

Deep neural networks have proven to be a revolutionary and effective method for resolving difficult and complex real-world problems. However, one of the most challenging problems in implementing them in safety-critical systems is providing formal guarantees about their accuracy. As a result, network verification and validation tools and techniques are investigated by researchers in this field.

1.4 Thesis Contribution

The work presented in this thesis proposes a methodology to assess the dependability of neural networks in safety-critical scenarios, which involves utilizing specific characteristics and methods of introducing faults. The issues highlighted in the problem statement have resulted in the research conducted in this thesis, which has produced the following contributions:

- This thesis explores the challenge of providing formal guarantees about the behavior of deep neural networks in safety-critical systems. To address this challenge, network verification and validation tools and techniques are necessary. Marabou, an SMT-based framework, is presented as a tool for verifying deep neural networks. This framework can interpret network properties into constraint satisfaction issues, allowing it to answer questions about the network's behavior.

- The thesis also discusses the use of fault injection testing to verify neural networks. Fault injection testing involves intentionally introducing errors into a system to ensure that it can recover from those error conditions. The thesis utilizes two types of fault injections, SEU and MBU, to test a prototype called VCAS, which is a deep neural network implementation of the next-generation unmanned aircraft airborne collision avoidance system (ACAS Xu). The thesis concludes by proposing future research in this area.

1.5 Related Work

There are multiple research works involved in formally verifying neural networks such as a procedure for formally verifying the security of autonomous robots that process LiDAR pictures and make control decisions using Neural Network controllers [5]. The method entails creating a system abstraction with limited states and computing the set of safe initial states using reachability analysis. The authors offer the idea of imaging-adapted workspace partitions, where the imaging function is ensured to be affine, to mimic the imaging function. The Satisfiability Modulo Convex encoding is then used by the authors to list every conceivable assignment of ReLU nonlinearity in the previously learned NN controller. Another approach was a brand-new method for examining the novel behavioral properties of safety-critical neural networks they term ProVe (Property Verifier), which is based on interval algebra [6]. The authors show that the violation rate calculated by ProVe gives a good evaluation of the safety of trained models by applying it to several domains, such as mapless navigation for mobile robots, trajectory creation for manipulators, and the common ACAS benchmark.

1.6 Thesis Outline

The rest of this thesis is organized as follows:

- In chapter 2 we provide a brief overview of neural networks which are a type of machine learning algorithm and how they are generated. This section also explores the benefits of using neural networks, SMT, and MDPs in embedded systems, as well as the challenges associated with ensuring safety, reliability, and efficiency in these systems.

- In Chapter 3 we introduce Marabou which is an open-source SMT solver that is

designed to handle deep neural networks. Marabou's design and capabilities were explained as to why it is used for this thesis.

- In chapter 4 we explaine the proposed assessment methodology used which involves several steps to evaluate the reliability of neural networks in safety-critical applications by using certain properties and fault injection techniques.

- In Chapter 5 we present the experimental setup and the results of the different experiments done. We evaluate the impact of the SEU and MBU on the system's reliability.

- In Chapter 6 we present a summary of the work carried out throughout this thesis and provide a plan for future research.

Chapter 2

Theoretical Background

2.1 Deep Learning

A branch of machine learning called ”deep learning” involves building artificial neural networks with several layers [7]. The goal is to develop neural networks that can mimic the human brain's capacity for pattern recognition and decision-making. In numerous fields, including speech recognition, natural language processing, and image recognition, deep learning has been demonstrated to be quite effective. It has also been applied to projects like developing drugs, diagnosing illnesses, and self-driving cars. Deep learning's primary benefit is its capacity to automatically extract characteristics and representations from massive quantities of data, which can subsequently be applied to provide precise predictions or judgments.

Deep learning can be characterized as a sub-field of machine learning where the emphasis is on learning and extracting characteristics that are on higher abstraction levels at each layer utilizing a series of subsequent information processing layers [8]. Characteristics are the pieces of information in data that need to be interpreted by the intelligent agent, for example, the ANN (Artificial Neural Network). ANNs are a popular deep learning model that were inspired by the biological neurons in our brain [9]. Artificial neurons arranged

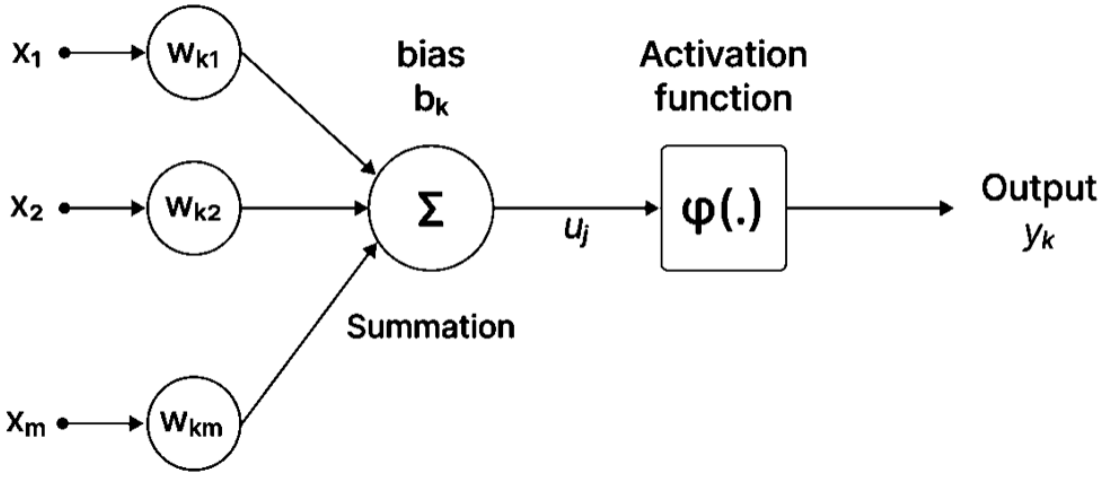

Figure 2.1: Artificial neuron

in layers make up an ANN. The most basic mathematical representation of a biological neuron is an artificial neuron. It typically consists of three components: a set of weighted inputs, an adder function that adds up the weighted inputs, and a function called activation that determines whether the neuron's output should be active or not. This activation function could simply be a threshold or it could be a more complicated non-linear function. Figure 2.1 is a representation of an artificial neuron.

2.2 Neural Networks

In recent years, there has been a significant shift in the way of constructing and developing complex systems. The majority of engineers now prefer to use deep neural networks (DNNs) [10] [11] instead of spending many hours hand-crafting complex and sophisticated software. DNNs are machine learning models that are generated using training algorithms that generalize from a finite group of samples to previously unknown inputs. In fields such as game playing [12], speech recognition [13], and image classification [11], their performance often outperforms that of manually produced software.

Regardless of their all-inclusive success, DNNs' opacity is a source of concern, and certification techniques that can provide meticulous guarantees regarding network behavior are urgently needed. The first steps in this direction were taken by the formal methods community by developing neural network verification tools and algorithms [14, 15] [16] [17] [18] [19] [20] [21] [22]. Two main parts make up the DDN verification inquiry: a neural network and a property to be examined. The output of a DNN verification query is either an official guarantee that the network meets the condition or a counterexample which is a specific input for which the property has been violated. A verification query can be used to express the fact that a network is resilient to minor adversarial perturbations in its input, for example.

A neural network is made up of neurons that are arranged in layers. The network is analyzed by values being assigned to the neurons in the input layer and then iteratively calculates the assignments of neurons in each subsequent layer using these values. Finally, the network's output is generated using the values of neurons in the last layer. The assignment of a neuron is evaluated by computing a weighted sum of the neurons assigned in the previous layer and then a non-linear activation function is applied to the result.

Hence, a network can be considered as a collection of linear constraints and non-linear constraints (weighted sums and activation functions, respectively). A verification query, in addition to a neural network, comprises a property to be tested on the inputs and outputs of the neural network, this property is expressed as constraints whether they were linear or non-linear. As a result, the verification problem is reduced to either discovering assigned values for the neurons that fulfill all the criteria at the same time or confirming that no such assignment is possible.

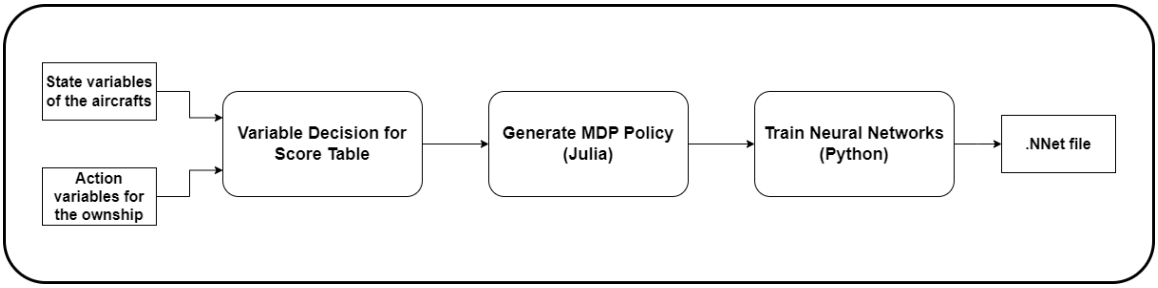

Figure 2.2: Neural Network Generation Process

2.3 Neural Network Generation

2.3.1 Variable decision

In a TCAS (Traffic Collision Avoidance System) neural network, the decision-making process [23] is based on input data such as the location, altitude, and velocity of nearby aircraft. The network uses this information to predict potential collisions and generates a set of possible actions for the pilot to take, such as changing altitude or heading. The specific action that is taken is determined by a set of decision-making rules or algorithms that are programmed into the network. These rules take into account factors such as the relative positions and speeds of the aircraft, as well as the pilot's own preferences and constraints. The decision-making process is typically designed to be as quick and efficient as possible, in order to minimize the risk of a collision.

Therefore, before generating the score table that will be used later to generate and train the neural networks, the variables of the collision avoidance system should be determined. The score table basically assigns values to each combination of the variables. In this system, the variables are divided into action and state variables. The resolution advisories given to the ownship are depicted by the actions in the score table. The advisories may differ from one system to another, some examples of the possible actions that can be generated and sent to the ownship are Clear-of-conflict, turn left, turn right, descend, climb, and many more which are explained in detail in the Experimental Setup and Results chapter.

An action is chosen from these many advisories based on the aircraft encounter and this encounter can be defined by the state variables. The state variables can vary, it could be the distance from ownship to the intruder, speeds of ownship and intruder, angles that can determine the perspectives of both ships, etc.

2.3.2 MDP policy generation

Queuing and inventory control are two areas where Markov Decision Process systems (MDPs) and stochastic sequential decision systems have been applied. MDPs are used to simulate dynamic systems that are subject to decision-making, and they can be categorized according to how frequently decisions are made, how long the decision-making horizon is, the state and action spaces, and the optimality criteria. This chapter focuses on issues when the state and action sets are either finite, countable, compact, and decisions are taken regularly at discrete time intervals. The major goals of MDP analysis are to characterize the form of an optimal policy, to provide an optimality equation that describes the ultimate value of the objective function, and to develop effective computational methods for identifying optimal policies. The optimality equation, also known as the Bellman equation, is the basis of MDP theory, and most findings are derived from a study of it [24].

MDPs are appealing planning and decision-making models. MDPs enhance conventional artificial intelligence planning models because of their state-transition and more comprehensive incentive structures. Interesting environments can be expressed using MDPs, and the most effective algorithms are employed to compute the best decision-making processes. Using current computer technology, such methods scale to MDPs with millions of states.

To accurately model scores for all advisories across all discrete encounter states, one technique for generating the logic for collision avoidance systems uses dynamic programming. The table tends to grow exponentially as the number of state variables, making

storage in certified avionic hardware difficult [25]. A neural network may be trained to estimate the logic table, requiring significantly less storage space.

By modeling the collision avoidance issue as a Markov decision process (MDP) [26] [23], the collision avoidance score table is generated. An MDP is made up of states $s \in S$, actions $a \in A$, a reward function *R(s,a)*, and state transition probabilities $T(s'|s,a)$, which define the probability of arriving at state *s'* by performing action *a* in state *s*. A policy $\pi(s)$ that maps states to actions in order to optimize the accumulation of reward over time is the solution to an MDP. Dynamic programming can be used to compute the policy $\pi(s)$ by defining state-action values *Q(s.a)*, which are initially all zeros, and iteratively updating *Q* by using Bellman equation (1), where γ is a discount factor to guarantee convergence. The policy π can be deduced from this equation (2) after computing *Q(s,a)*.

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s \in S} T(s'|s,a) max_{a' \in A} Q(s',a') \tag{1}$$

$$\pi(s) = argmax_{a' \in A} Q(s,a) \tag{2}$$

The logic table to be scaled down with a neural network is formed by the state-action values *Q*. Even though the network could directly approximate $\pi(s)$, as opposed to all state-action values, systems such as ACAS X, use the values to approximate the best action in multi-intruder encounters, so this work approximates the state-action values instead of just the policy [27] [28].

2.3.3 Neural Network training

Following the computation of the state-action value table $Q(s,a)$, a neural network representation $\bar{Q}(s,a)$ is used to approximate the table. Previous studies have demonstrated that a neural network representation retains accuracy while minimizing representation size,

outperforming numerous different compression methods such as symmetry analysis or decision trees [28] [29].

The input layer of a neural network contains the encounter geometric features, the output layer comprises the action scores, and the middle layers are known as hidden layers. Before applying a nonlinear activation function, each hidden layer is calculated as an affine transformation of the previous layer. This work employs networks with rectified linear unit activations (ReLU) [30], which are defined as $relu(x) = max(0, x)$. Using x_0 as the input to the network with L hidden layers, the hidden layers can be calculated using the formula below equation (3), in which W_i and b_i are trainable network parameters. The network output $\bar{Q}(s, .)$ is calculated with no activation function, so $\bar{Q}(s, .) = W_L x_L + b_L$.

$$x_{i+1} = relu(W_i x_i + b_i) \forall i \in 0, 1, ..., L - 1 \tag{3}$$

The network parameters are randomly initialized and updated with gradient descent methods to reduce a loss function representing the error of the network. Pretty standard regression concerns employ a mean squared error loss function; however, when approximating $Q(s, a)$, it is also important to maintain $argmax_{a' \in A} Q(s, a)$ so that $\pi(s)$ is precisely approximated. Nominal mean squared error equally weights under and over-approximation errors, which may lead to modifications to $\pi(s)$ if the value of the best action is undervalued whereas another action is overvalued.

This study employs an asymmetric mean squared error function, which severely penalizes undervaluing the best action while overvaluing the others. In addition, when the error is small, a linear term is also added to the loss function to account for undervaluing the best action or overvaluing the other actions, which strengthens the gradient. In supervised learning, state-action values from the table are randomly divided into smaller batches of 512 for each epoch. The neural network parameters are then modified using Adam's adaptive method of gradient descent. The neural networks have been trained for various epochs

until their performance plateaus [25].

2.4 Satisfiability Modulo Theories

The use of satisfiability checking in computer science is very important, particularly in formal verification of hardware and software. Many problems can be reduced to checking the satisfiability of a formula in some logic, and SAT solvers can efficiently solve problems formulated in propositional logic. However, some problems are better formulated in classical logics like first-order or higher-order logics, which have a more expressive language that includes non-Boolean variables, function and predicate symbols, and quantifiers. To make the satisfiability problem decidable, restrictions can be placed on the logic, either syntactically or semantically. These restrictions lead to fragments of first-order logic, and the resulting satisfiability problem is called Satisfiability Modulo Theories (SMT). The progress in SMT research has been driven by several factors, including a focus on background theories and classes of problems that occur in practice, innovations in core algorithms and data structures, and attention to implementation details. There are now several powerful and sophisticated SMT solvers that are being used in a rapidly expanding set of applications, including processor verification, static analysis, planning, and optimization. The SMT-LIB initiative, which standardizes the input/output format for SMT solvers, has played a significant role in this progress, along with the SMT workshop and SMT-COMP international competition for SMT solvers [31].

Thanks to the amazing expansion in the capabilities and performance of SMT solvers over the past ten years, model checking has been widely used SMT. Model checking for infinite-state transition systems and software model checking, in general, are now frequently supported by SMT solutions. SMT is most commonly focused on the T-satisfiability of quantifier-free formulas and decidable theories T. To decide the T-satisfiability problem, a procedure for deciding the T-satisfiability of constraints (conjunctions of literals)

is required. However, converting any quantifier-free formula to Disjunctive Normal Form (DNF) leads to an impractical solution due to the frequent exponential blow-up in the size of the resulting formula. The T-satisfiability of quantifier-free formulas is NP-hard, and the blow-up cannot be eliminated in general, except for degenerate and uninteresting examples of theories. Most modern SMT solvers take the lazy route in order to avoid the drawbacks of DNF conversions. This method entails building and verifying a DNF for the input formula gradually and as required. In order to efficiently reason about the propositional connectives, it combines specific constraint satisfiability techniques or theory solvers with a conflict-driven clause-learning (CDCL) SAT solver. There are various variations of this strategy, which vary in how sophisticated the SAT engine and theory solvers interact.

2.5 Embedded Systems

Systems that handle data inside of a bigger system are called embedded systems [32]. Examples might involve various car parts, phones, or surveillance devices. Embedded systems typically rely on constrained computational, energy, and data storage resources. On the other hand, due to the sensitivity of their applications, embedded systems are frequently made to satisfy soft or hard time constraints or meet a certain degree of safety standards.

2.5.1 Safety Critical Systems and Fault Tolerance in Embedded Sytems

Applications that are considered to be safety-critical are those whose malfunction could cause disasters or harm to people [33]. The ability of a system to continue performing its intended job in the face of hardware or software flaws, or faults, is known as fault-tolerance [34]. Based on where they infiltrate the system, different types of faults can be identified. They may be hardware, design, or standard errors. Hardware faults are of importance in this thesis, as they can be caused by flaws in individual hardware parts or by

external, non-system variables.

Permanent and transient faults are two main categories of hardware errors [33]. A short circuit or a grounded wire are examples of persistent physical hardware defects that result in permanent faults. Radiation and electrostatic discharge are the main causes of transient faults, the most prevalent kind of faults in memory devices. Single Event Upsets (SEU) as mentioned earlier in Chapter I are a type of transient fault or soft error [35]. A transient perturbation that flips a bit in a storage element to a different state results in an SEU [36].

The stuck-at-fault model is a typical technique for simulating faults in digital circuitry. The output of a gate is presumptively stuck at logical 1, 0, or even an unidentified state X in this framework. The circuit with one or more stuck-at faults can have input vectors applied to it, and the output can then be assessed and compared to the ground-truth output. Test engineers can identify which gates, nodes, or signals can obfuscate the output of the circuit when a fault strikes them by performing this iteratively and for all potential stuck-at faults. Then, designers can apply this knowledge to create a circuit that is more resilient.

The random bit flip is another prevalent error model. Storage element SEUs and transient defects can be modelled using the random bit flip model [37]. A system's fault tolerance can be assessed at different levels of complexity, from the system and application levels all the way down to the gate and netlist levels. Either hardware or software can be used to evaluate fault tolerance. It can be evaluated using a variety of techniques, including injecting faults into the object or system being tested by means of fault injection circuits and into a simulated system via fault injection software.

2.5.2 Embedded Systems and Neural Networks relations

Three main fields of research are involved in implementing deep learning in embedded systems: first, designing, optimizing, and customizing deep neural networks to be compatible with embedded systems. Deep neural networks typically require a lot of processing

power. It is extremely difficult to deploy them on embedded systems in their current state. Designing hardware platforms that are better suited for successfully running Neural Network (NN) building blocks for inference is the second field of study. These platforms cannot and need not handle both inference and training. The development of tools to connect the two environments is yet another field of study.

Deep neural networks (DNNs) are optimized for devices with limited resources using a variety of methods. DNNs are difficult to execute on embedded systems because they frequently have a large number of parameters and demand a lot of computational power such as VGG-16 [38] [39]. A frequent method for lowering a network's computational load and memory requirements is quantization. It entails mapping network parameters to a condensed set of permitted values, which lowers the amount of storage space needed and permits more effective mathematical processes. Quantization could, however, reduce precision. Another method is pruning which is a method for making DNNs more compatible with the hardware [40]. Pruning entails removing connections from the network whose weights are below a predetermined threshold. This method can lower the number of network operations, but retraining is required to offset the accuracy loss [41]. By factorizing the convolution operation into two stages, depth-wise separable convolutions are a method used to decrease the number of operations in the network [42]. This method can decrease computation complexity with little loss in precision.

For deep neural network (DNN) inference, which entails processing a lot of data, hardware designers have tried to optimize platforms. The dataflow in processing devices is frequently optimized in order to take advantage of data reuse. Based on the stationary operand or intermediate operand and the degree of the memory hierarchy, data reuse is categorized. In order to maximize reuse and increase energy efficiency in convolutional layers for batch sizes larger than 16, Chen et al. [43] proposed a row stationary dataflow for DNN processing that keeps a row of weights in a convolutional kernel stationary at a

time and multiple rows of input activations at the register file level. Several architectures, including Eyeriss [44], NVDLA [45], and Edge TPU [46], which are optimized for various application types and have various capabilities and configurations, were created especially to speed up deep learning.

Deploying models based on deep learning on embedded systems, which can be divided into CPUs, GPUs, FPGAs, and ASICs, is fraught with difficulties. It takes architecture-specific frameworks and tools to optimize, compile, and map neural networks on hardware because there are so many different embedded systems available. For embedded platforms, a number of deep learning tools have been created, such as TensorFlow Lite (TFLite) [47] and TFLite Micro [48], which concentrate on compact and effective neural network models. Python, C, C++, and Java are just a few of the computer languages that support the TFLite API. With the development of frameworks and libraries like ARMNN and CMSIS-NN [49], a provider ARM of embedded computing systems, has added support for handling neural networks on their hardware products.

2.6 IEEE Number Formats

Computer systems can represent and store digital signals that reflect real and integer numbers in either fixed-point or floating-point formats. As the name implies, the fixed-point format has a set number of digits following the decimal point. We are accustomed to calculating and working with numbers, so fixed-point representations of data and processing are comparable. The dynamic range, which is dependent on the number of bits accessible, is evenly distributed among the numbers that can be expressed in the fixed-point format. The following equations show how floating points numbers can be presented in DNNs:

$$x = (-1)^{Sign} * (1 + Mantissa)^{(Exponent - Bias)} \tag{4}$$

$$\pm mantissa * base^{exponent} \tag{5}$$

On the other hand, a number is stated using four terms in the floating-point format: a sign, a mantissa/fraction, an exponent, and a base. Therefore, a sign, a mantissa, and an exponent are required to represent a floating-point integer in a computer system. Different standards exist for the floating-point format. For floating-point numbers, the technical standard IEEE754 specifies the arithmetic, storage, encoding, interchange format, and rounding up guidelines [50]. Single precision floating-point format, as used in this paper, is described by IEEE 754 as shown in Figure 2.3.

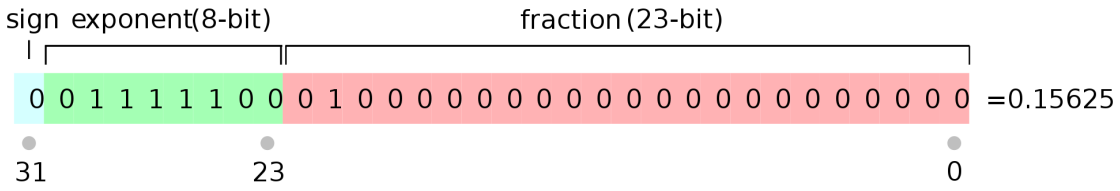

Figure 2.3: IEEE754 Single precision floating-point format

The aforementioned standard has 1, 23, and 8 bits for sign, mantissa, and exponent, correspondingly. However, because of the leading bit convention rule, there is an implicit 1, which acts as the number's integer portion, making the mantissa a total of 24 bits with 23 bits of stored fractional information. Additionally, excess-127 coding is used to encode the exponent part, which merely adds 127 to the exponent before storing it.

Chapter 3

Marabou tool

The Reluplex [16] [51] [52] [53] [54] [55] project concentrated on using SMT-based techniques to verify DNNs, and the Marabou project [17] builds on that work. Marabou follows in the footsteps of Reluplex by employing a lazy search technique based on SMT: it iteratively seeks an assignment that gratifies all given limitations while treating non-linear limitations lazily in the hopes that several of them will turn out to be unrelated to the property in question and will not need to be handled. In addition to searching, Marabou uses deduction to acquire new facts about non-linear constraints in order to make them easier to understand.

3.1 Marabou Design

Marabou [17] treats every node in the network as a constant and looks for a variable assignment that meets the query's linear and non-linear constraints at the same time. Marabou keeps track of the current variable assignment, the list of current constraints, and both the lower and upper bounds for each variable at all times. The variable assignment is then changed for each iteration to rectify a violated linear or non-linear constraint. The Marabou

verification technique is valid and comprehensive, in the sense that the aforesaid loop ultimately comes to an end. This may be demonstrated using a simple expansion of Reluplex's completeness and soundness proof. Nevertheless, Marabou permits piecewise-linear activation functions in order to ensure termination. The ReLU and Max functions are already supported by the tool, and it is flexible in the sense that many more piecewise-linear functions can be easily added. The deduction, or the derivation of tighter lower and upper variable bounds, is another crucial part of Marabou's verification technique. The motivation is that by constraining piecewise-linear constraints to one of their linear segments, such bounds can transform them into linear constraints. Marabou accomplishes this by examining linear and nonlinear constraints frequently, as well as executing network-level reasoning, with the goal of establishing tighter variable bounds. The Marabou tool is able to do the verification procedure using the main components shown in Fig 3.1 that will be described in the next sections.

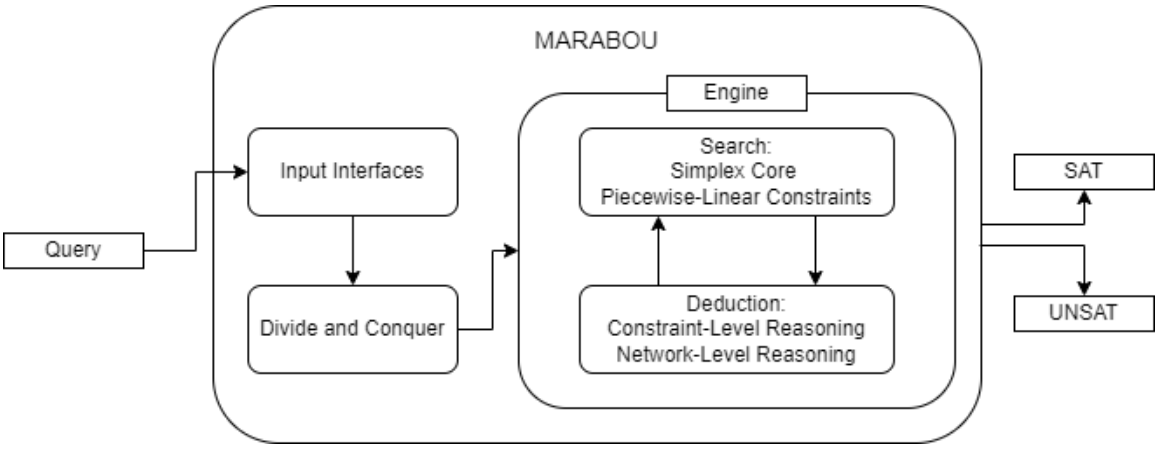

Figure 3.1: Marabou Tool

3.1.1 Simplex Core

The simplex core is the system component in charge of ensuring that variable assignment adheres to linear constraints. It accomplishes this by utilizing a variant of the simplex

algorithm. It changes the assignment of some variable *x*, and thus the assignment of any variable *y* that is connected to *x* by a linear equation, in each iteration. The revised simplex method, in which the various linear constraints are kept in implicit matrix form, and the steepest-edge and Harris' ratio test strategies for variable selection are used to select *x* and determine its new assignment.

Developing an efficient simplex solver is difficult. The linear constraints in Reluplex were delegated to an external solver, GLPK. Their motivation for implementing a new custom solver in Marabou was twofold: first, they discovered in Reluplex that the repeated translation of queries into GLPK and extraction of results from GLPK was a performance limiting factor; second, a black box simplex solver did not provide the flexibility we required in the context of DNN verification. Variable assignments, for example, are typically pressed against their upper or lower bounds in a standard simplex solver, whereas in the context of a DNN, other assignments may be required to satisfy the non-linear constraints. Another example is the deduction capability, which is critical for efficiently verifying a DNN and whose effectiveness may be affected by the simplex solver's internal state.

3.1.2 Piecewise-Linear Constraints

Marabou maintains a set of piecewise-linear constraints that represent the DNN's non-linear functions throughout its execution. Marabou looks for any constraints that are not satisfied by the current assignment in iterations devoted to satisfying these constraints. If such a constraint is discovered, Marabou modifies the assignment to satisfy the constraint. If Marabou detects that a particular constraint is repeatedly not satisfied, it may perform a case-split on that constraint: a process in which the piecewise-linear constraint is replaced by an equivalent disjunction of linear constraints $c_1 \vee ... \vee c_n$. Marabou examines each disjunct individually and checks for satisfiability. If the problem is solved when x is supplanted by some ci, then the initial complaint is also solved; otherwise, the original problem

is unsatisfiable.

Piecewise-linear constraints are represented in their implementation by objects of classes derived from the PiecewiseLinearConstraint abstract class. The two supported instances at the moment are ReLU and Max, but the design is modular in the sense that new constraint types can be easily added. PiecewiseLinearConstraint specifies the interface methods that must be implemented by each supported piecewise-linear constraint.

3.1.3 Constraint- and Network-Level Reasoning

Marabou's performance is dependent on the effective deduction of tighter variable bounds. The deduction is performed at both the constraint and DNN levels, by repeatedly examining linear and piecewise-linear constraints to see if they imply tighter variable bounds.

The getEntailedTightenings() method is used to query the piecewise-linear constraints for tighter bounds when performing constraint-level bound tightening. Linear equations can also be used to calculate tighter bounds. For example, the equation $x = y + z$, as well as the lower bounds $x >= 0$, $y >= 1$, and $z >= 1$, imply the tighter bound $x >= 2$. Marabou encounters many linear equations and uses them for bound tightening as part of the simplex-based search.

Several papers have recently proposed verification schemes based on DNN-level reasoning [14] [21]. Marabou also supports this type of reasoning by storing the initial network topology and performing deduction steps that use this data as part of its iterative search. By (1) instantiating the DNN-level reasoners with the most recent information discovered during the search, such as variable bounds and the state of piecewise-linear constraints, and (2) feeding any new information discovered back into the search procedure, DNN-level reasoning is seamlessly integrated into the search procedure. Marabou currently implements a symbolic bound tightening procedure: upper and lower bounds for each hidden neuron

are expressed as a linear combination of the input neurons based on network topology. Then, if the bounds on the input neurons are sufficiently tight (e.g., as a result of previous deductions), these expressions for upper and lower bounds could insinuate that some of the piecewise-linear activation functions of the hidden neurons are now restricted to one of their linear segments. Additional DNN-level reasoning operations are being implemented.

3.1.4 The Engine

The Engine is Marabou's main class, which contains the main loop. The engine stores and coordinates the solution's various components, such as the simplex core and the piecewise-linear constraints. The main loop is composed of the following steps (using the first rule that applies):

1. Perform a case split on a piecewise-linear constraint if it had to be fixed more than a certain number of times.

2. If the problem has become unsatisfiable, for example, because a lower bound for some variable has been concluded that is greater than its upper bound, negate a previous case split (or if no such case split exists return **UNSAT**).

3. If a linear constraint is violated, take a simplex step.

4. If a piecewise-linear constraint is violated, try to fix it.

5. **SAT** is returned (all constraints are satisfied).

The engine also initiates deduction steps, both at the neuron and network levels, based on various heuristics.

3.1.5 The Divide-and-Conquer Mode and Concurrency

Marabou has a divide-and-conquer (D and C) solving mode, which divides the input region specified in the original query into sub-regions. The desired property is checked independently on each of these sub-regions. By checking each sub-query on a separate

node, the D and C mode naturally lends itself to parallel execution. Furthermore, even when run sequentially, the D and C modes can enhance Marabou's performance overall: the overall duration of solving the sub-queries is often shorter than that of the time of solving the original query because the smaller input regions permit even more efficient deduction steps.

The solver keeps a queue Q of $< query, timeout >$ pairs in response to a query ϕ. Q is initialized with a single element $< \phi, T >$, where T (initial timeout) is a programmable parameter. The solver goes through the following steps to solve ϕ:

1. Take a pair from Q and try to solve it.
2. Return **UNSAT** if the problem is **UNSAT** and Q is empty.
3. Return to step 1 if the problem is **UNSAT** and Q is not empty.
4. Return **SAT** if the problem is **SAT**.
5. If a timeout occurs, partition the input region into sub-queries.

3.1.6 Input Interfaces

Marabou accepts the following interfaces for verification queries:

-Native Marabou format: a user creates a query by using Marabou C++ interface, compiles it, and runs it in the tool. This format can be used to incorporate Marabou into a larger framework.

-Marabou executable: When a user runs a Marabou executable, they pass it to command-line parameters indicating which network and property files should be checked. As of now, network files have been encoded in the NNet format [56], with properties provided in a simple text format.

-Python/TensorFlow interface: the query is passed to Marabou via Python constructs. DNNs stored as TensorFlow protobuf files can also be handled by the Python interface.

3.2 Proof Production for Marabou

As deep neural networks are used more frequently in safety-critical systems, it is essential to ensure that they are correct. As a result, the verification industry has developed a variety of methods and tools for checking DNNs. It is simple to verify when DNN verifiers find an input that causes an error; however, when they indicate that there isn't one, there are no means to confirm that the verification tool isn't in error. The viability of DNN verification is called into doubt because numerous errors have already been found in DNN verification tools. The generation of an easy-to-check proof of unsatisfiability, which demonstrates to the absence of errors, is a new method used for enhancing Simplex-based DNN verifiers with proof production capabilities. The method of producing proofs is based on an effective modification of the well-known Farkas' lemma along with techniques for dealing with piecewise linear functions and errors in numerical accuracy. The technique is applied on top of the Marabou DNN verifier as a proof of concept. The analysis of a safety-critical airborne collision avoidance system demonstrates that proof production is successful almost always and only incurs negligible overhead [57].

3.3 Comparasion to Reluplex

The Marabou tool is vastly superior to its predecessor, Reluplex, as it includes multiple modifications and enhancements. The fully convolutional and connected DNNs with arbitrary piecewise-linear activation functions have native support, this improves on the original Reluplex algorithm, which only supported ReLU activation functions. Support for a divide-and-conquer solving mode is built-in, where this mode automatically adapts itself to parallel processing by conducting sub-queries on distinct nodes; but, even when employed with a single node, it can give significant speedups. It also has a full linear programming core based on simplex, that substitutes the external solver (GLPK), this new core

was designed to integrate seamlessly with the framework and since the GLPK isn't used, a good amount of the overhead in Reluplex is eliminated. The usage of the framework is much simpler to the users as there are various interfaces to feed queries into the solver, it can be in a TensorFlow model contained in a protocol buffer file or a textual format, properties formatted in textual form or in Python can be compiled as well. Finally, it supports network-level deduction and reasoning which enables a more efficient search space optimization [17].

Chapter 4

Proposed Assessment Methodology

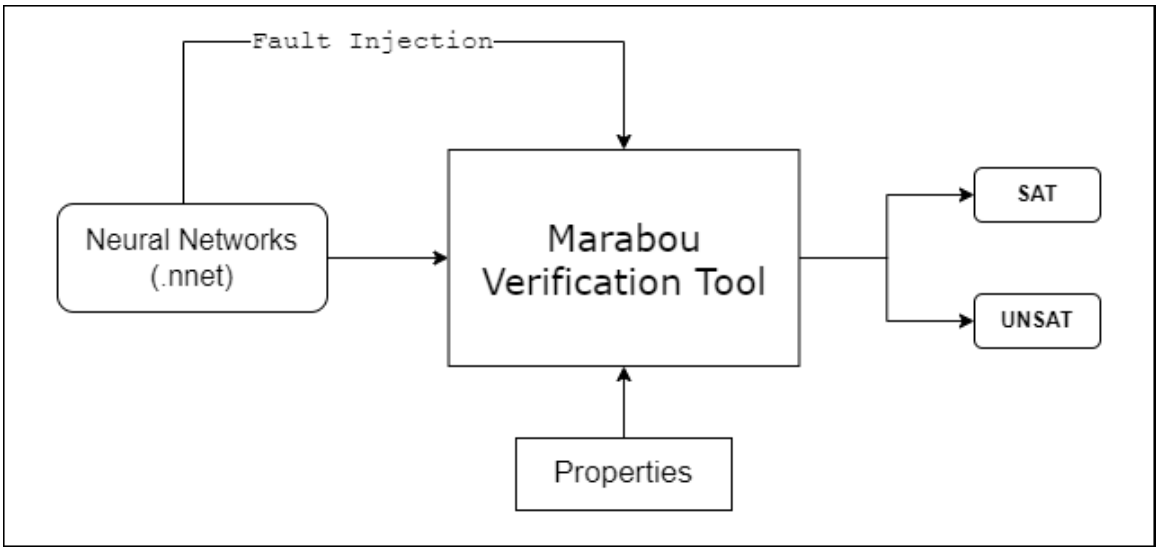

Figure 4.1: Methodology

4.1 Neural Network Comprehension

In a TCAS (Traffic Collision Avoidance System) neural network, the decision-making process is based on input data such as the location, altitude, and velocity of nearby aircraft. The network uses this information to predict potential collisions and generates a set of

possible actions for the pilot to take, such as changing altitude or heading. The decision-making process is typically designed to be as quick and efficient as possible, in order to minimize the risk of a collision. All the variables used to make up the score that will be used later to generate and train the neural networks should be understood because those states and actions will be required to determine the properties needed to verify the neural networks.

4.2 Properties Used

In formal languages like temporal logic, properties for formal verification of neural networks are frequently expressed. The behavior of dynamic systems like neural networks can be specified using temporal logic, a branch of mathematical logic that permits the specification of attributes over time. Using a collection of operators that enable the declaration of logical links between events and states in the system is a typical method for expressing properties in temporal logic. The ”eventually” operator says that a property must hold at some point in the future, whereas the ”always” operator specifies that a property must hold at all times. To build more sophisticated features, these operators can be coupled in a variety of ways.

One often begins by determining the system's needs within which the network would be implemented before writing properties for formal verification of neural networks. The formal qualities that result from this translation can then be verified with a verification tool in this case it's going to be Marabou. Up until all requirements are met, the writing and verifying of properties can be an iterative process, with properties being improved and altered based on the findings of the verification process.

Since the format of neural networks inputted into Marabou is specified to be .nnet, when specifying our properties this format should be taken into consideration. The mean/range values written at the top of the network data are the values used to normalize the network

training data before training the network. Therefore, to change the values of the boundaries of the inputs needed in the properties the equation shown below is used:

$$\frac{input - mean}{range} = normalizedinput \tag{6}$$

However, when it comes to the outputs, the case studies dealt with in this research is based on the advisory actions the aircraft should take. The advisory taken is going to be the advisory output with the least value compared to the others. Hence, if we have a certain number of different outputs each related to a specific action when writing a property to check if the chosen advisory should be taken, the notation written should be the negation of what is needed. For example, if there are five actions (y1, y2, y3, y4, and y5) and we are checking if with certain inputs y1 should be taken, the notation written should be the negation of that statement. The negation statement is written as notions as shown below:

$+y1 - y2 >= 0$

$+y1 - y3 >= 0$

$+y1 - y4 >= 0$

$+y1 - y5 >= 0$

4.3 Network Verification and Validation

Marabou is then used which is a tool for verifying neural networks. It is based on a formal verification technique called ”interval analysis” which allows the tool to reason about the behavior of a neural network over a range of input values, rather than just a single input value. This helps to ensure that the network’s behavior is consistent across a wide range of possible inputs. The Marabou tool takes a neural network in the form of a trained model and a set of input and output constraints as input. It then analyzes the network’s

behavior using interval analysis to determine if there are any input combinations that will cause the network to produce an output that violates the constraints. If such input is found, the tool can provide a counterexample to the user, which shows the specific input values that cause the violation.

The tool is used to verify a set of properties. These properties are formally defined in Chapter 5. The tool looks for an input that would violate each property; thus, an UNSAT result indicates that a property holds, and a SAT result indicates that it does not. The satisfying assignment in the SAT case is an illustration of an input that violates the property. Marabou can be used to verify properties such as network robustness, safety, and fairness. It can also be used to prove the absence of certain types of errors or attacks, such as adversarial examples.

Having the set of neural networks to be verified and the set of properties used to verify these networks, the following algorithm is used to save time.

Listing 4.1: Simplified Version of the Verification script

```
1: for i in range{Neural Networks} {
2:     for j in range {Properties}{
3:         Get a Neural Network
4:         Get a Property
5:         Verify each Neural Network with each Property
6:         Save each verification onto a datalog
7:     }
8: }
```

4.4 Fault Injection

A technique called fault injection is used to assess a system's robustness and dependability by purposefully introducing defects or faults and monitoring how the system responds. Fault injection can be used in the context of neural networks to assess the network's capacity to deal with unpredictable or hostile inputs and to discover any potential flaws or vulnerabilities in the architecture of the neural network. Fault injection can be used with neural networks in a variety of ways, in this study, weight perturbation was used which tests a network's tolerance for malfunctions or errors in the weight parameters by introducing minor random perturbations to the network's weights. We can learn how neural networks behave and perform under different circumstances by employing fault injection techniques, and they can also spot any weaknesses or vulnerabilities that need to be fixed. As a result, neural networks will be more robust and reliable and will be more useful in a variety of applications. Two forms of fault injections will be used on the neural networks; SEUs and MBUs.

4.4.1 Set 1: SEUs

The bit flips in the memory-stored data are modeled as the hardware faults or SEUs. The DNN's parameters, or the weights of each layer, are considered to be data in this context. Flipping the chosen bit on a randomly picked parameter introduces a fault. The bit position and the parameter are both randomly selected in the fault model. The selected fault model replicates SEU or bit flip errors that can influence inputs, outputs, processing elements, or in this case, data stored in memory, and can occur in a variety of embedded platform components, including the processor, dynamic random-access memory (DRAM), buffers, etc.

In Figure 4.2, a hypothetical network parameter in 32-bit floating point format is used as an illustration of how this is accomplished. The 7th most important bit of the parameter is

flipped, changing the value of the stored parameter. Figure 4.2 demonstrates how a random error reduces the number by a cetain factor.

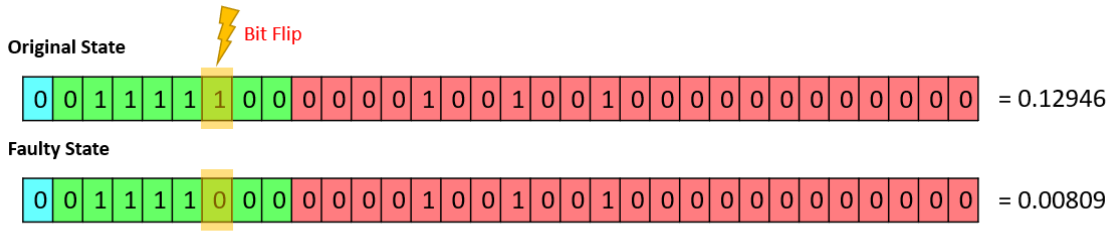

Figure 4.2: Bit flip on a parameter

Before returning the network to its initial state after the defect has been introduced, the correctness of the network is assessed using the testing data set. Marabou is used in this process to verify these defected networks along with the properties written. The network's weights and biases were initially trained and have values that have been determined at the time of inference. This indicates that since nothing alters them, they will not be rewritten during inference. We will only read them. The fault model utilized in this thesis is such that when it affects a data point in the memory, the fault lingers on the data until that specific data point is accessed and overwritten, to put this and the choice of fault model in context.

A neural network's parameters, however, would not be changed at any moment while making an inference. The defect is considered to persist in the affected parameter since network weights and biases are repeatedly read from memory but only once written. In contrast to the weights and biases of the network, the bit flip is only cleared when the parameter is overwritten. It should be acknowledged that using this fault model with deep neural network parameters as an application under test has some drawbacks because the fault is no longer exactly a transient fault if the incorrect data in the memory is never rewritten or corrected. However, there are workarounds like redundancy and voting that can resolve this.

The process of introducing faults and verifying the faulted networks is shown below:

Listing 4.2: Simplified Version of the Verification script for SEU

```
for i in range{Neural Networks} {
    for j in range {Properties}{
        Get a Neural Network
        Select a random weight
        Flip one bit whether it is a sign, exponent, or fraction
        Get a Property
        Verify each Neural Network with each Property
        Save each verification onto a datalog
    }
}
```

4.4.2 Set 2: MBUs

An MBU (Multiple Bit Upset) can happen in a neural network when many bits flip as a result of a single event upset. Any component of the system, including the memory or the processor unit, may experience this. Flipping numerous bits might result in severe computation errors that affect the neural network's output.

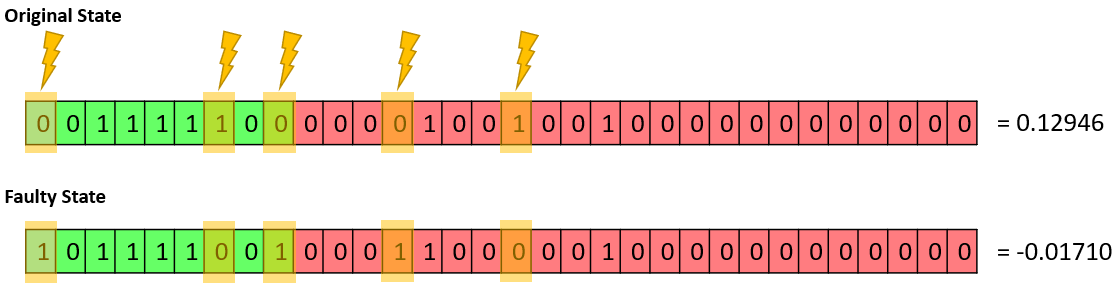

Figure 4.3: Multiple bit flip on a parameter

Multiple faults that can occur simultaneously or sequentially in the neural network parameters can be used to mimic the MBU process in neural networks. This can be done by flipping several bits within a single parameter or several bits within several parameters. By

evaluating the neural network's accuracy on the training dataset, one may assess the impact of MBUs. Figure 4.3 shows how when multiple bits are flipped the value changes too.

To mitigate the impact of MBUs, for every neural network we randomly selected a single weight in the output layer where we randomly flipped multiple bits at once and verified the faulted networks with the properties. The process used can be shown below:

Listing 4.3: Simplified Version of the Verification script for MEU

```
1: for i in range{Neural Networks} {
2:     for j in range {Properties}{
3:         Get a Neural Network
4:         Select a random weight
5:         Flip multiple bits no matter whether it is a sign, exponent, or fraction
6:         Get a Property
7:         Verify each Neural Network with each Property
8:         Save each verification onto a datalog
9:     }
10: }
```

A Multiple Multiple Bit Upset (MMBU) event occurs when several Multiple Bit Upsets (MBUs) take place simultaneously in a system. An MMBU event is a situation in which a single energetic particle or radiation event causes multiple bits in various sections of the system to be affected or flipped. MMBUs can cause several failures to occur simultaneously across many components or subsystems, making them more severe than individual MBUs. The system's operation could be significantly disrupted by these faults, requiring additional steps for error detection, correction, and recovery. Similar to how the MBUs were introduced, MMBUs were caused by randomly selecting several weights and then flipping multiple random bits. The procedure is demonstrated below:

Listing 4.4: Simplified Version of the Verification script for MMEU

```
1: for i in range{Neural Networks} {
```

```
    for j in range {Properties}{
        Get a Neural Network
        Select multiple random weights
        Flip multiple bits no matter whether it is a sign, exponent, or fraction
        Get a Property
        Verify each Neural Network with each Property
        Save each verification onto a datalog
    }
}
```

Chapter 5

Experimental Setup and Results

5.1 Vertical CAS case study

A system known as a "Vertical Collision Avoidance System" (VCAS) is made to stop aircrafts from crashing in the vertical direction, particularly with reference to height. When two aircrafts are on a collision trajectory and their heights are not coordinated, it is frequently employed. VCAS uses a barometric altimeter, Global Positioning System (GPS), and Automatic Dependent Surveillance-Broadcast (ADS-B) data to detect the position, velocity, and altitude of aircraft nearby. The resolution advisory (RA), which informs the pilots of both aircraft of the suggested course of action to take in order to prevent a collision, is then generated using this information to predict probable collisions. This could be a "climb/descend" command or a climb or descend instruction. Then, it is the pilots' duty to adhere to the RA and maintain a safe distance from other aircraft.

Five dimensions make up the state space, as shown in Table 5.1. The vertical encounter geometry is described by the variables h, $\dot{h}_{own}$, and $\dot{h}_{int}$. In order to prevent an NMAC, the aircraft must be separated vertically once the horizontal separation is lost, which is indicated by the variable τ, which also serves as a countdown to that moment. The previous

advisory provided is represented by the variable *s adv*, which enables the system to consistently select fresh alerts. Figure 5.1 depicts the geometry of the encounter. The nine pilot advisories that make up the action space, which is detailed in Table 5.2, are transmitted once every second and control the ownship's acceleration. The other advisories presumptively assume that the pilot accelerates at an authorized acceleration until they reach the goal vertical rate, at which time the pilot stops accelerating, but COC permits the ownship to select any acceleration. The aircraft is presumed to maintain the existing vertical rate if it currently complies with the advisory.

In our case study, we work on 9 neural networks where each neural network is dependable on the previous action it had, hence, Neural networks 1, 2, 3, 4, 5, 6, 7, 8, and 9 are Neural networks with COC, DNC, DND, DES1500, CL1500, SDES1500, SCL 1500, SDES2500, and SCL2500 as their previous advisory respectively. The neural networks are made up of 4 inputs, 9 outputs, and 7 layers, wherein each hidden layer there are 45 nodes each fully connected to the next layer.

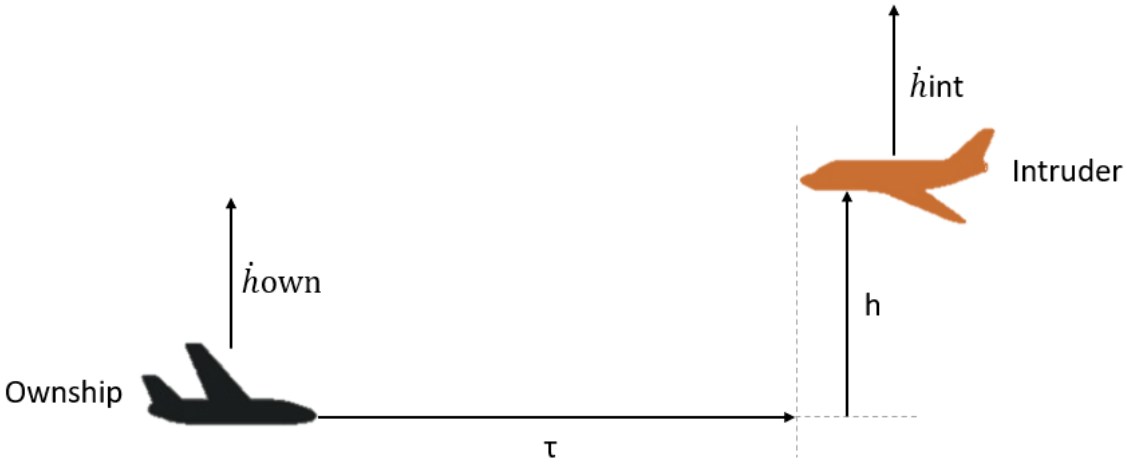

Figure 5.1: VerticalCAS Aircraft encounter geometry

Table 5.1: VerticalCAS state variables

Variable	Neural Network Representation	Description
h	x0	Altitude between the two ships
$\dot{h}_{own}$	x1	Ownship vertical rate
$\dot{h}_{int}$	x2	Intruder vertical rate
τ	x3	Time to loss of horizontal separation
s adv	-	Previous advisory

Table 5.2: VerticalCAS advisories

Advisory Description	Neural Network Representation	Description
COC	y0	Clear of Conflict
DNC	y1	Do Not Climb
DND	y2	Do Not Descend
DES1500	y3	Descend >= 1500 ft/min
CL1500	y4	Climb >= 1500 ft/min
SDES1500	y5	Strengthen Descent to >= 1500 ft/min
SCL1500	y6	Strengthen Climb to >= 1500 ft/min
SDES2500	y7	Strengthen Descent to >= 2500 ft/min
SCL2500	y8	Strengthen Climb to >= 2500 ft/min

5.1.1 Experimental Analysis

The five properties used to verify the neural networks of this system are:

Property 1:

- Mathematical notation expressions:

  $0.48125 \leq x0 \leq 0.5, -0.5 \leq x1 \leq 0.5, -0.5 \leq x2 \leq 0.5, 0.375 \leq x3 \leq 0.5$

  $+y0-y2 \geq 0, +y0-y3 \geq 0, +y0-y4 \geq 0, +y0-y5 \geq 0, +y0-y6 \geq 0, +y0-y7 \geq 0, +y0 - y8 \geq 0$

  $+y1-y2 \geq 0, +y1-y3 \geq 0, +y1-y4 \geq 0, +y1-y5 \geq 0, +y1-y6 \geq 0, +y1-y7 \geq 0, +y1 - y8 \geq 0$

- Description: If the intruder is really far and is above the ownship, the score of a COC or DNC advisory will be minimum.

- Input constraints: $7700 \leq h \leq 8000, -100 \leq \dot{h}_{own} \leq 100, -100 \leq \dot{h}_{int} \leq 100, 35 \leq \tau \leq 40$

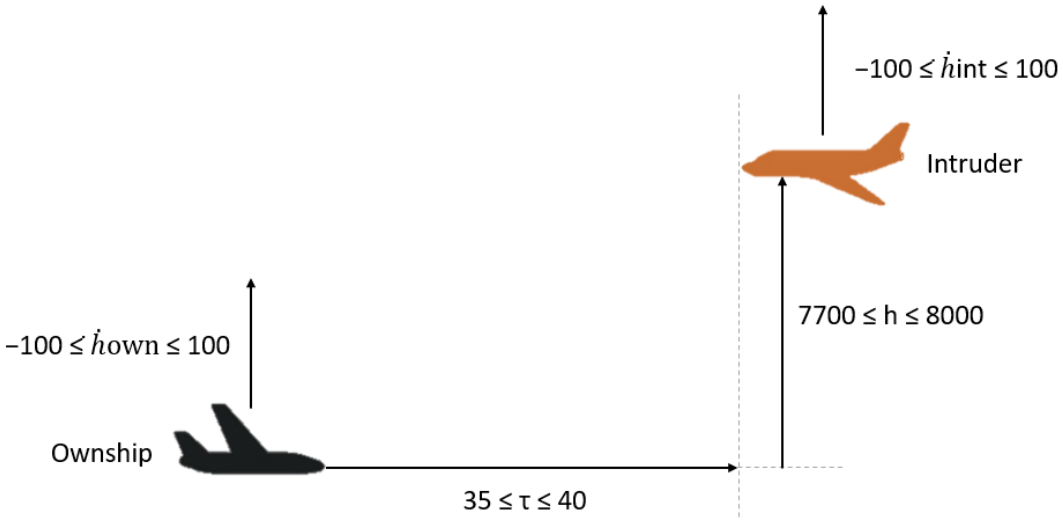

Figure 5.2: Property 1 encounter geometry

- Desired output property: the score for COC or DNC advisory is the minimal score.
- Tested on: all 9 networks.

Property 2:

- Mathematical notation expressions:

  $-0.5 \leq x0 \leq -0.48125, -0.5 \leq x1 \leq 0.5, -0.5 \leq x2 \leq 0.5, 0.375 \leq x3 \leq 0.5$

  $+y0-y1 \geq 0, +y0-y3 \geq 0, +y0-y4 \geq 0, +y0-y5 \geq 0, +y0-y6 \geq 0, +y0-y7 \geq 0, +y0-y8 \geq 0$

  $+y2-y1 \geq 0, +y2-y3 \geq 0, +y2-y4 \geq 0, +y2-y5 \geq 0, +y2-y6 \geq 0, +y2-y7 \geq 0, +y2-y8 \geq 0$

- Description: If the intruder is really far and is under the ownship, the score of a COC or DND advisory will be minimum.
- Input constraints: $-8000 \leq h \leq -7700, -100 \leq \dot{h}_{own} \leq 100, -100 \leq \dot{h}_{int} \leq 100, 35 \leq \tau \leq 40$.

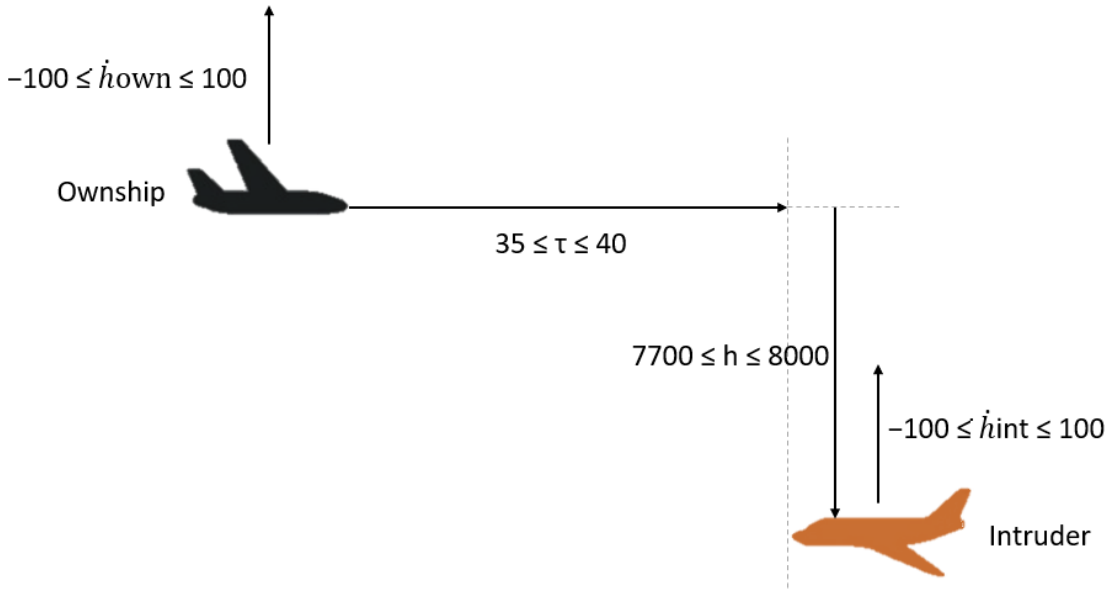

Figure 5.3: Property 2 encounter geometry

- Desired output property: the score for COC or DND advisory is the minimal score.
- Tested on: all 9 networks.

Property 3:

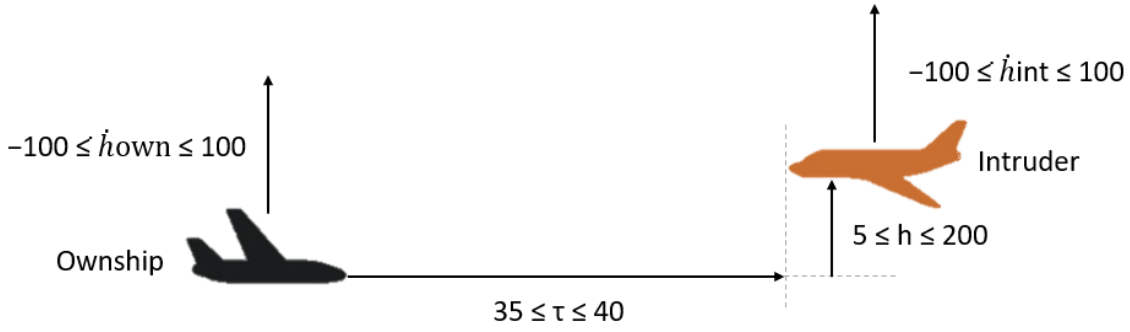

Figure 5.4: Property 3 encounter geometry

- Mathematical notation expressions:

  $0.0003125 \leq x0 \leq 0.0125, -0.5 \leq x1 \leq 0.5, -0.5 \leq x2 \leq 0.5, 0.375 \leq x3 \leq 0.5$

$+y7-y0 \geq 0, +y7-y1 \geq 0, +y7-y2 \geq 0, +y7-y3 \geq 0, +y7-y4 \geq 0, +y7-y6 \geq 0, +y7 - y8 \geq 0$

$+y5-y0 \geq 0, +y5-y1 \geq 0, +y5-y2 \geq 0, +y5-y3 \geq 0, +y5-y5 \geq 0, +y5-y6 \geq 0, +y5 - y8 \geq 0$

- Description: If the intruder is close and is on top of the ownship, the score of SDES 1500 or SDES 2500 advisory will be minimum.
- Input constraints: $5 \leq h \leq 200, -100 \leq \dot{h}_{own} \leq 100, -100 \leq \dot{h}_{int} \leq 100, 35 \leq \tau \leq 40$.
- Desired output property: the score for SDES 1500 or SDES 2500 advisory is the minimal score.
- Tested on: all 9 networks.

Property 4:

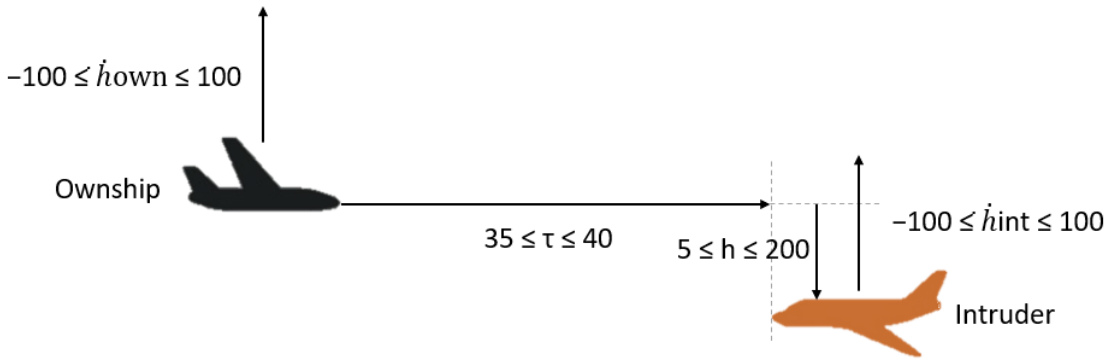

Figure 5.5: Property 4 encounter geometry

- Mathematical notation expressions:

$-0.0125 \leq x0 \leq -0.0003125, -0.5 \leq x1 \leq 0.5, -0.5 \leq x2 \leq 0.5, 0.375 \leq x3 \leq$

$0.5 + y8 - y0 \geq 0, + y8 - y1 \geq 0, + y8 - y2 \geq 0, + y8 - y3 \geq 0, + y8 - y4 \geq 0, + y8 - y5 \geq 0, + y8 - y7 \geq 0$

$+y6 - y0 \geq 0, + y6 - y1 \geq 0, + y6 - y2 \geq 0, + y6 - y3 \geq 0, + y6 - y4 \geq 0, + y6 - y5 \geq 0, + y6 - y7 \geq 0$

- Description: If the intruder is really far and is under the ownship, the score of SCL 1500 or SCL 2500 advisory will be minimum.
- Input constraints: $-200 \leq h \leq 5, -100 \leq \dot{h}_{own} \leq 100, -100 \leq \dot{h}_{int} \leq 100, 35 \leq \tau \leq 40$.
- Desired output property: the score for SCL 1500 or SCL 2500 advisory is the minimal score.
- Tested on: all 9 networks.

Property 5:

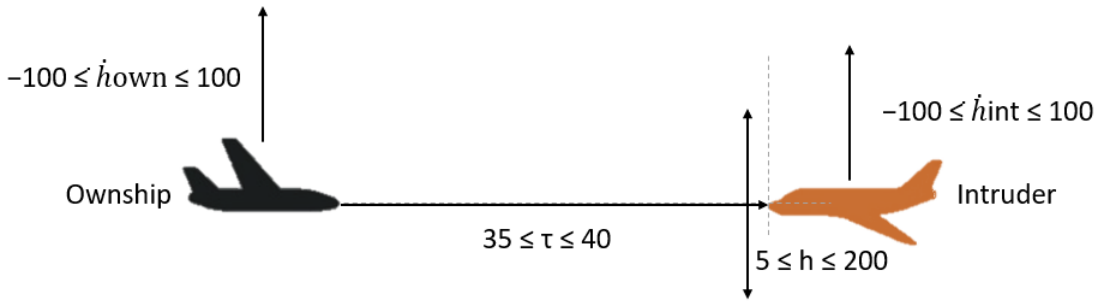

Figure 5.6: Property 5 encounter geometry

- Mathematical notation expressions:

$-0.0125 \leq x0 \leq 0.0125, -0.5 \leq x1 \leq 0.5, -0.5 \leq x2 \leq 0.5, 0.375 \leq x3 \leq 0.5$

$+y0 - y1 \geq 0, + y0 - y2 \geq 0, + y0 - y3 \geq 0, + y0 - y4 \geq 0, + y0 - y5 \geq 0, + y0 - y6 \geq 0, + y0 - y7 \geq 0, + y0 - y8 \geq 0$

- Description: If the intruder is close no matter the direction relative to the ownship, the score of a COC advisory will not be minimum.
- Input constraints: $-200 \leq h \leq 200, -100 \leq \dot{h}_{own} \leq 100, -100 \leq \dot{h}_{int} \leq 100, 35 \leq \tau \leq 40$.
- Desired output property: the score for COC advisory is the minimal score.
- Tested on: all 9 networks.

5.1.2 Discussion

Table 5.3: VCAS networks verification

SAT/UNSAT	**Properties**				
Neural Network	**Property 1**	**Property 2**	**Property 3**	**Property 4**	**Property 5**
Neural Network 1	sat	sat	unsat	unsat	unsat
Neural Network 2	sat	sat	unsat	unsat	unsat
Neural Network 3	sat	sat	unsat	unsat	unsat
Neural Network 4	sat	sat	unsat	unsat	unsat
Neural Network 5	sat	sat	unsat	unsat	unsat
Neural Network 6	sat	sat	sat	sat	sat
Neural Network 7	sat	sat	sat	sat	sat
Neural Network 8	sat	sat	sat	sat	sat
Neural Network 9	sat	sat	sat	sat	sat

As shown in Table 5.3, the neural networks are numbered with the previous advisory such that Neural Network 1 and Neural Network 4 had COC and DES1500 as their previous advisories, respectively. Deducing from this table we can see how all networks didn't withstand properties 1 and 2 meaning that the verification tool, Marabou, was able to find certain input variables that violate the property's constraints. However, when it comes to the other properties, properties 3, 4, and 5, five of the nine previous advisory networks were able to hold. This can be seen in two different ways: first, the previous advisories that didn't hold the property shared something in common which was a strengthened action whether it

Table 5.4: VCAS networks verification time (ms)

	Properties				
Neural Network	**Property 1**	**Property 2**	**Property 3**	**Property 4**	**Property 5**
Neural Network 1	5699	15002	22166437	15178568	107298086
Neural Network 2	375	4705	2291973	2730288	37830697
Neural Network 3	4984	13875	1568735	5057202	50684448
Neural Network 4	9603	12280	33190052	12784288	186951416
Neural Network 5	19124	12102	734657	5132020	18544046
Neural Network 6	11013	11094	37803	24645	233303402
Neural Network 7	11856	7516	19282674	16362	52015568
Neural Network 8	11635	12882	10320873	9426	74683999
Neural Network 9	4313	10618	3285058	105597	1514197

was a descent or climb. The other point of view could be the properties themselves, where properties 1 and 2 focused on when the intruder is far away from the ownship, and the other three properties focused on when the intruder is close to the ownship.

Here are some case scenarios where the result is SAT which means the scenario that occurred isn't what is meant to be:

Scenario 1 (Neural Network 1 with Property 2):

- Input assignment:

  x0 = -0.497418

  x1 = -0.117776

  x2 = -0.271089

  x3 = 0.454533

- Output:

  y0 = 0.139961

  y1 = -0.230524

  y2 = 0.043327

  y3 = 0.043327

  y4 = 0.040944

y5 = -0.284707

y6 = -0.266676

y7 = -0.276966

y8 = -0.260034

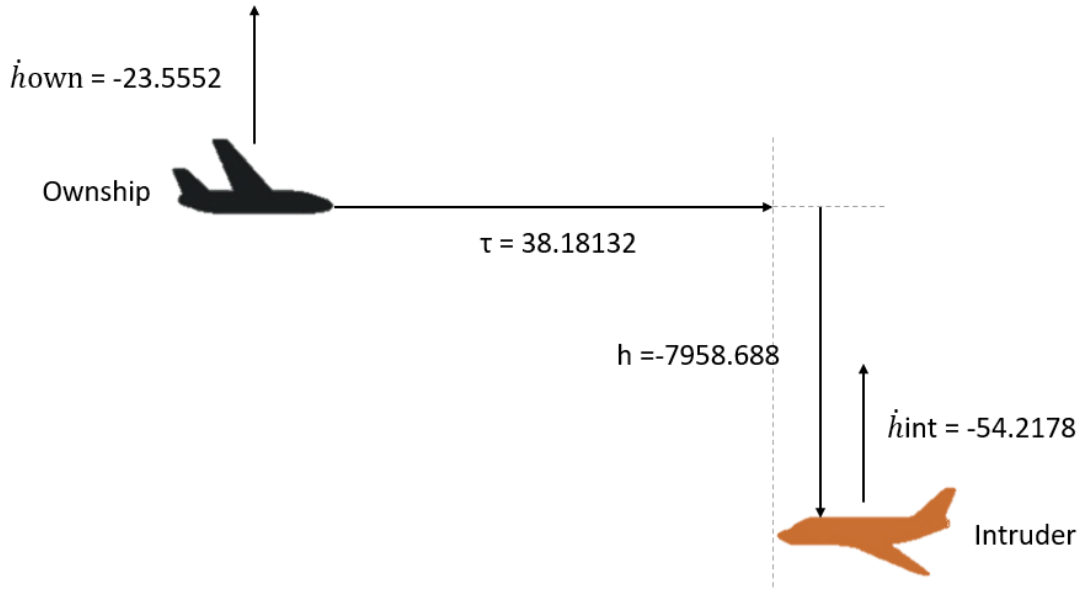

Figure 5.7: Scenario 1 encounter geometry

As seen in the figure the COC advisory had the greatest value which means its the least advisory to be taken which is incorrect in this case.

Scenario 2 (Neural Network 7 with Property 3):

- Input assignment:

  x0 = 0.004138

  x1 = -0.037832

  x2 = 0.180846

  x3 = 0.375000

- Output:

y0 = 0.122725

y1 = 0.110774

y2 = 0.112923

y3 = 0.114561

y4 = 0.113009

y5 = 0.122774

y6 = 0.116796

y7 = 0.133331

y8 = 0.104014

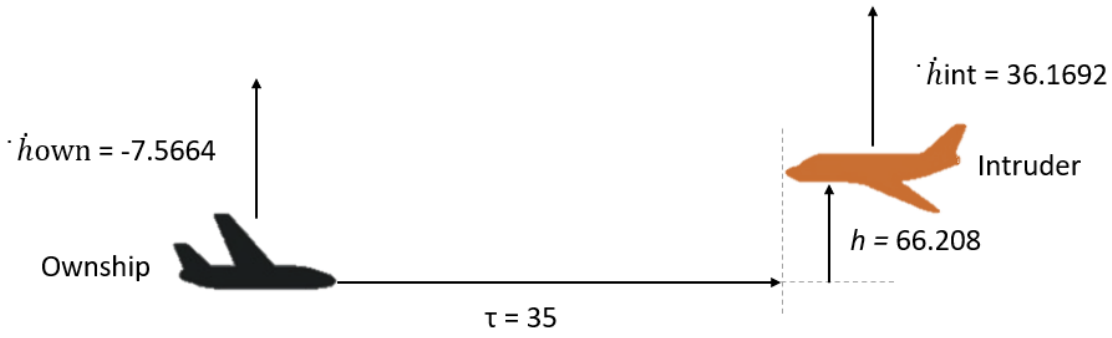

Figure 5.8: Scenario 2 encounter geometry

As seen in the figure the SDES2500 advisory had the greatest value which means its the least advisory to be taken which is incorrect in this case.

After verifying the VCAS neural network we introduced two different fault injection techniques: SEUs and MBUs to check if the neural network's behavior is affected by these changes.

For the SEU test, we used three different neural networks, neural network 1 (previous advisory is COC), neural network 4 (previous advisory is DES1500), and neural network 9 (previous advisory is SCL2500). For each network, 36 random weights neurons were

Table 5.5: Verified Networks with SEUs

Properties	Number of weights changed	Bit Change	Result
Neural Network 1			
Property 1	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 2	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 3	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT
Property 4	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT
Property 5	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT
Neural Network 4			
Property 1	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 2	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 3	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT
Property 4	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT
Property 5	36	Sign	UNSAT
		Exponent	UNSAT
		Fraction	UNSAT

Table 5.6: Verified Networks with SEUs (continuation)

Properties	Number of weights changed	Bit Change	Result
Neural Network 9			
Property 1	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 2	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 3	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 4	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT
Property 5	36	Sign	SAT
		Exponent	SAT
		Fraction	SAT

Table 5.7: Verified Networks with MBUs (Network 1)

Properties	Number of weights changed	Result
Property 1	45	SAT
Property 2	45	SAT
Property 3	45	UNSAT
Property 4	45	UNSAT
Property 5	38	UNSAT
	7	SAT

chosen at a time and flipped one bit of its value. We approached the neural networks in three different approaches where we flipped 1. the sign bit, 2. a random exponent bit, and 3. a random fraction bit.

Tables 5.5 and 5.6 represent the verification outcome with our five properties when a sign bit, exponent bit, or fraction bit is flipped at a random weight in neural networks 1, 4, and 9 respectively. We can observe that the neural network exhibits the same behavior as if the fault does not exist, which means that an SEU does not affect these neural networks.

For the MBU test, we focused on one neural network, neural network 1 (the previous advisory is COC). For this specific network, 45 random weights neurons were chosen at a time and flipped multiple bits of their value. The bits chosen were randomly selected no matter if they were a sign bit, an exponent bit, or a fraction bit.

Table 5.7 represents the effects of fault injections on this system's neural network. Fault injection was done on the chosen network mentioned earlier changing one single neuron's weight and re-verifying the system at a time. It can be seen that the system behaved the same when fault injections occurred. However, that was not the case always, when tested again with other random neurons' weights being changed (around 225 tests) the networks acted the same for all properties but one, which was property 5 where there were some instances that a property changed from UNSAT to SAT. We also ran some MBU tests on

Neural Network 9 (previous advisory is SCL2500) to check if the SAT outcomes changed to UNSAT meaning that the neural network is better, but as expected there weren't any changes.

The impacts of multiple bit upsets (MBUs) were more obvious and significant observations were made in the extensive MBU test carried out on various neural networks, namely neural networks 1, 4, and 9. Multiple neurons were injected with MBU faults during the test, changing the weights of those neurons where the previous test was only injecting one random neuron. These MBUs had a significant effect on how the neural networks were verified. The verification process frequently timed out, indicating the serious consequences of the introduced flaws. This occurrence emphasizes how susceptible neural networks are to MBUs and the potential havoc they might cause with the verification process.

Moreover, the test outcomes revealed intriguing cases where the satisfiability results of the neural networks were altered. Some properties that were initially determined as UNSAT showed a transition to SAT outcomes. While this could indicate unexpected behavior caused by the injected faults, it also underscores the importance of thorough verification to identify potential vulnerabilities and ensure the proper functioning of the neural networks. The instances where the SAT results changed to UNSAT were of great relevance. These occurrences demonstrate that the neural networks displayed an unexpected behavior which made the neural network work as wanted which wasn't the case with the original neural network.

Finally, these findings underline the significance of carefully assessing and testing neural networks in a variety of fault scenarios, including MBUs, to guarantee their efficiency in safety-critical applications. Knowing how the networks react to such failures helps us deploy them confidently since we can rely on them to handle unforeseen circumstances while maintaining functionality and dependability.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Currently, one of the most critical challenges is to incorporate neural networks into embedded systems, particularly in safety-critical applications. These types of systems have strict requirements for safety, security, availability, and timing. While neural networks are powerful for processing data, they also introduce unpredictability because they calculate output based on probability distributions. This unpredictability makes it difficult to ensure that the system behaves accurately and predictably, which is crucial for identifying worst-case outcomes. Additionally, incorporating fault tolerance into embedded system design becomes even more complicated when using neural networks. Nevertheless, researchers can investigate and reinforce neural networks against potential faults to minimize their impact to an acceptable threshold.

In our work, we proposed a technique that can be employed to evaluate the reliability of neural networks in safety-critical situations, which entails leveraging particular traits and fault injection methods. The SMT-based framework tool that was used to make this work possible is Marabou which is able to convert network attributes into constraint satisfaction problems and respond to queries about the behavior of the network. We were able to come

up with five different properties that were used to verify the VCAS case study which was inspired by the well-known system ACAS Xu and deduced which properties withstood and which didn't.

We also were able to understand the use of fault injection testing to see if the system was able to recover from errors that were introduced purposely. There were two different fault injection methods used in this research, SEU and MBU. Both methods showed two different outcomes, where when SEU was used and the system was verified again with the five properties written; the system behaved the same. However, when MBU was applied, there were some cases where an UNSAT outcome is now SAT, meaning that the neural network isn't operating as expected.

6.2 Future Work

The comprehension and robustness of neural networks in safety-critical systems could be improved by exploring various aspects of future research topics in the areas of neural network verification and fault injection. Potential research areas include:

1. Advanced verification methods to assure the accuracy, security, and dependability of neural networks, advanced methods, and algorithms are being developed. This could entail looking into new formal verification frameworks, scaling enhancements, and automated formal property generation for verification.

2. Investigating adversarial attacks on neural networks and creating efficient defense mechanisms will increase their resilience to harmful input perturbations. Designing more durable neural network topologies, training strategies, and adversarial detection approaches may be the main areas of this study.

3. Verification of deep reinforcement learning by extending verification methods to these algorithms, which integrate reinforcement learning algorithms and neural networks. The goal of this research is to guarantee the dependability and safety of these systems in

challenging situations.

The broad domain of neural network verification and fault injection contains many possible areas for research, and these are only a few of them. In order to assure the reliable and trustworthy operation of neural networks as they develop and find applications in safety-critical domains, more research and innovation in these fields are necessary.

Bibliography

[1] J. Kuchar and A. C. Drumm, "The traffic alert and collision avoidance system," *Lincoln laboratory journal*, vol. 16, no. 2, p. 277, 2007.

[2] U. Durak, D. Müller, F. Möcke, and C. B. Koch, "Modeling and simulation based development of an enhanced ground proximity warning system for multicore targets," in *Proceedings of the Model-driven Approaches for Simulation Engineering Symposium*, pp. 1–12, 2018.

[3] Z. Yan, Y. Shi, W. Liao, M. Hashimoto, X. Zhou, and C. Zhuo, "When single event upset meets deep neural networks: Observations, explorations, and remedies," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 163–168, IEEE, 2020.

[4] A. Chabot, I. Alouani, R. Nouacer, and S. Niar, "A memory reliability enhancement technique for multi bit upsets," *Journal of Signal Processing Systems*, vol. 93, pp. 439–459, 2021.

[5] X. Sun, H. Khedr, and Y. Shoukry, "Formal verification of neural network controlled autonomous systems," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pp. 147–156, 2019.

[6] D. Corsi, E. Marchesini, and A. Farinelli, "Formal verification of neural networks for safety-critical tasks in deep reinforcement learning," in *Uncertainty in Artificial Intelligence*, pp. 333–343, PMLR, 2021.

[7] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning," *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 2021.

[8] L. Deng, D. Yu, *et al.*, "Deep learning: methods and applications," *Foundations and trends® in signal processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[9] A. G. H.-O. M. Learning, "with scikit-learn and tensorflow: Concepts, tools, and techniques to build intelligent systems," 2017.

[10] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618," *Genetic Programming and Evolvable Machines*, vol. 19, no. 1-2, pp. 305–307, 2018.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[13] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[14] 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)*, pp. 3–18, IEEE, 2018.

[15] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pp. 3–29, Springer, 2017.

[16] 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*, pp. 97–117, Springer, 2017.

[17] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, *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*, pp. 443–452, Springer, 2019.

[18] B. J. Taylor, M. A. Darrah, and C. D. Moats, "Verification and validation of neural networks: a sampling of research in progress," in *Intelligent Computing: Theory and Applications*, vol. 5103, pp. 8–16, SPIE, 2003.

[19] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pp. 243–257, Springer, 2010.

[20] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," *arXiv preprint arXiv:1805.02242*, 2018.

[21] 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)*, pp. 1599–1614, 2018.

[22] 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.

[23] M. J. Kochenderfer, *Decision making under uncertainty: theory and application*. MIT press, 2015.

[24] M. L. Puterman, "Markov decision processes," *Handbooks in operations research and management science*, vol. 2, pp. 331–434, 1990.

[25] K. D. Julian and M. J. Kochenderfer, "Guaranteeing safety for neural network-based aircraft collision avoidance systems," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pp. 1–10, IEEE, 2019.

[26] R. Bellman, "On the theory of dynamic programming," *Proceedings of the national Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.

[27] K. D. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.

[28] 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.

[29] 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)*, pp. 1–10, IEEE, 2016.

[30] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8609–8613, IEEE, 2013.

[31] C. Barrett and C. Tinelli, *Satisfiability modulo theories*. Springer, 2018.

[32] P. Marwedel, *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature, 2021.

[33] E. Dubrova, *Fault-tolerant design*. Springer, 2013.

[34] B. Johnson, "Fault-tolerant microprocessor-based systems," *IEEE Micro*, vol. 4, no. 06, pp. 6–21, 1984.

[35] J. F. Ziegler, "Terrestrial cosmic rays," *IBM journal of research and development*, vol. 40, no. 1, pp. 19–39, 1996.

[36] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[37] C. Torres-Huitzil and B. Girau, "Fault and error tolerance in neural networks: A review," *IEEE Access*, vol. 5, pp. 17322–17341, 2017.

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[39] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[40] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[41] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[42] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[43] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 367–379, 2016.

[44] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[45]

[46] "Edge tpu - run inference at the edge nbsp;—nbsp; google cloud."

[47] "Tensorflow lite: Ml for mobile and edge devices."

[48] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[49] A. Ltd., "Defining the future of computing."

[50] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std*, vol. 754, no. 2008, pp. 1–70, 2008.

[51] N. Carlini, G. Katz, C. Barrett, and D. L. Dill, "Provably minimally-distorted adversarial examples," *arXiv preprint arXiv:1709.10207*, 2017.

[52] D. Gopinath, G. Katz, C. S. Pasareanu, and C. Barrett, "Deepsafe: A data-driven approach for checking adversarial robustness in neural networks," *arXiv preprint arXiv:1710.00486*, 2017.

[53] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *arXiv preprint arXiv:1709.02802*, 2017.

[54] L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer, "Toward scalable verification for safety-critical deep networks," *arXiv preprint arXiv:1801.05950*, 2018.

[55] Y. Kazak, C. Barrett, G. Katz, and M. Schapira, "Verifying deep-rl-driven systems," in *Proceedings of the 2019 workshop on network meets AI & ML*, pp. 83–89, 2019.

[56] Sisl, "Sisl/nnet: Documentation and scripts related to the .nnet file format. this file format specifies a simple text file to define feed-forward, fully-connected, relu activated neural networks. example networks in this format can be found in the reluplex repository.."

[57] O. Isac, C. Barrett, M. Zhang, and G. Katz, "Neural network verification with proof production," in *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 38–48, 2022.