PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY OF ABOU-BEKR BELKAID – TLEMCEN

Laboratoire de Recherche en Informatique de Tlemcen

FACULTY OF SCIENCES – DEPARTMENT OF COMPUTER SCIENCE

A THESIS

Presented for Obtaining the Degree of:

DOCTORATE

Specialty: Information and knowledge systems engineering and decision aid

By

Wahiba Bachiri

Theme

Optimization and Code Verification for Embedded Systems Based on Intelligent Controllers

Supervised by Seladji Yassamine & Pierre Loïc Garoche

Defended on 16 September 2025 at Tlemcen in Front of the Jury Composed of :

Mr Azeddine Chikh	Full Professor	University of Tlemcen	President
Mrs Yassamine Seladji	Associate Professor	University of Tlemcen	Supervisor
Mr Pierre Loïc Garoche	Full Professor	ENAC, University of Toulouse	Co-Supervisor
Mr Fethalah Hadjila	Associate Professor	University of Tlemcen	Examiner
Mr Matthieu Martel	Full Professor	University of Perpignan Via Domitia	Examiner
Mr Mohammed Yassine Kazi Tani	Associate Professor	ESI, SBA	Examiner
Mr Eric Feron	Full Professor	University of King Abdullah (KAUST)	Guest

Academic Year: 2024/2025

To the relentless seekers of truth—whose wisdom lights the path, and whose love makes the journey worthwhile

Acknowledgements

It is with profound gratitude that I extend my heartfelt thanks to my supervisor, **Mrs Yassamine Seladji** and my co-supervisor, **Mr Pierre Loic Garoche**. Their exceptional mentorship, insightful guidance, and constant encouragement have been the cornerstone of this journey. From the very beginning, they believed in my potential and challenged me to grow, both as a researcher and as a person. Their balanced perspective, approachability, and commitment to academic excellence provided me with clarity and direction at every stage. I have learned immensely from their collaborative spirit and generous support. Their patience, wisdom, and unwavering support during times of doubt and difficulty have left a lasting impact on me, and I feel truly privileged to have worked under their supervision.

I sincerely thank my esteemed examiners **Mr. Fethalah Hadjila, Mr. Mattieu Martel**, and **Mr. Mohammed Yassine Kazi Tani** for their invaluable time, rigorous evaluation, and insightful feedback, which have greatly enhanced the quality of my thesis. Your expertise and constructive criticism have been instrumental in refining my research. To the distinguished guest **Mr. Eric Feron** who honored my defense with their presence, I extend my deepest gratitude for your encouragement and engagement. Your participation made this milestone even more memorable. I am truly humbled by your collective wisdom and support, which have not only strengthened this work but also inspired my academic journey. This achievement is elevated by your contributions, and I will carry your guidance forward with immense appreciation.

I would like to express my deepest gratitude to **Miss Dorra Benkhalifa** for her invaluable support, unwavering encouragement, and insightful guidance throughout my thesis journey. Her expertise and thoughtful advice were instrumental in helping me overcome challenges and refine my work. I am truly honored to have benefited from her mentorship and am profoundly thankful for her role in this academic milestone.

Above all, I owe my deepest thanks to my family, especially my parents, **Bachiri Mohammed** and **Mekkioui Rachida**. Your unwavering love, patience, and belief in me have been the foundation upon which all of this rests. Thank you for the countless sacrifices, your silent strength, and for always standing by my side, even when the road was long and uncertain.

To my siblings, **Badiaa**, **Souad**, **Amel**, **Zoheir**, and **Abdessamad**, and to my friends: **Maria**, and **Wiaam**, thank you for being the pillars of support and joy throughout this journey. Your encouragement, shared laughter, and kind words during challenging moments have helped me stay grounded and motivated. No words can adequately express my gratitude for your endless patience, emotional support, and belief in me during this demanding yet fulfilling chapter of my life.

I gratefully acknowledge my research group colleagues for stimulating discussions, technical collaborations, and moral support during both breakthroughs and challenges. Our shared experiences made this journey more rewarding.

My sincere appreciation goes to the administrative staff of the department of computer science of my university for their tireless assistance in navigating academic requirements, facilitating resources, and creating an environment conducive to research. Their professionalism and kindness often made challenging processes manageable.

This thesis is not just a culmination of my academic efforts, but a reflection of the incredible people who have walked this path with me. Thank you all from the bottom of my heart.

Abstract

The complexity of autonomous systems presents significant challenges for their formal specification and verification, especially when integrating AI controllers based on quantized neural networks (QNNs). These networks utilize fixed-point arithmetic to accommodate the limited computational capabilities of embedded systems. Fixed-point arithmetic allows for the representation of weights, activations, and gradients using low-bit integers, enabling efficient computation with deterministic precision. This is particularly crucial in resource-constrained environments where floating-point units (FPUs) may be unavailable or prohibitively costly in terms of power and latency.

However, this quantization introduces numerical errors due to truncation and rounding, which can propagate through the network and impact decision-making. As a result, verifying QNNs, whether modeled with integers or bit vectors, has been shown to be **PSPACE-complete**.

In this thesis, we present three major contributions to the formal specification and verification of quantized neural networks (QNNs) based intelligent controllers in the domains of autonomous vehicles (AVs) and avionics. First, we propose a formal specification process for AV requirements, which involves a step-by-step transformation of textual requirements into formal properties. Second, we outline a sound and incomplete method for verifying QNNs that does not rely on integer or bit-vector theories. This method combines set theory, rational approximation, and SMT verification to validate the formal properties defined in our first contribution. We evaluate this method in the context of autonomous vehicles using the HIGHWAY-ENV simulator, alongside Z3 and Marabou as SMT solvers. Third, We introduce a quantization method for artificial neural networks (ANNs) that serves as an optimization technique that aims to preserve the properties of ANNs by identifying the largest perturbation that consistently maintains these properties. This perturbation acts as a threshold for applying the precision tuning tool, `Popinns`, which generates the optimized format of the quantized version of ANNs based on the specified threshold using SMT solver. We implement this method in the avionics domain, evaluating it using the ACAS Xu benchmark and Marabou. The results of our contributions demonstrate the efficiency and soundness of our proposed methods compared to traditional SMT solvers.

Keywords: Fixed point arithmetics, Quantized neural network, Formal verification, Specification, Autonomous Vehicles, Avionics, Satisfiability Modulo Theories.

Résumé

La complexité des systèmes autonomes présente des défis importants pour leur spécification et leur vérification formelles, en particulier lorsqu'il s'agit d'intégrer des contrôleurs d'IA basés sur des réseaux neuronaux quantifiés (RNQs). Ces réseaux utilisent l'arithmétique en virgule fixe pour s'adapter aux capacités de calcul limitées des systèmes embarqués. L'arithmétique en virgule fixe permet de représenter les poids, les activations et les gradients à l'aide d'entiers de faible poids, ce qui permet un calcul efficace avec une précision déterministe. Toutefois, cette quantification introduit des erreurs numériques dues à la troncature et à l'arrondi, qui peuvent se propager dans le réseau et influer sur la prise de décision. Par conséquent, la vérification des RQNs, qu'ils soient modélisés avec des entiers ou des vecteurs de bits, s'est avérée **PSPACE-complet**.

Dans cette thèse, nous présentons trois contributions majeures pour la spécification et la vérification formelles des contrôleurs intelligents basés sur des RNQs, appliqués aux domaines des véhicules autonomes (VAs) et de l'avionique. Premièrement, nous proposons un processus de spécification formelle des exigences des VAs, transformant pas à pas des exigences textuelles en propriétés formelles. Deuxièmement, nous décrivons une méthode fiable mais incomplète pour vérifier les RNQs sans recourir aux théories des entiers ou des vecteurs de bits. Cette méthode combine la théorie des ensembles, l'approximation rationnelle et la vérification SMT pour valider les propriétés formelles définies dans notre première contribution. Nous l'évaluons dans le contexte des véhicules autonomes à l'aide du simulateur HIGHWAY-ENV, ainsi que des solveurs SMT Z3 et Marabou. Troisièmement, nous présentons une méthode de quantification pour les RNA qui préserve leurs propriétés en trouvant la plus grande perturbation admissible. Ce seuil permet à l'outil `Popinns` de générer un format optimisé via un solveur SMT. Nous implémentons cette méthode dans le domaine de l'avionique en l'évaluant sur le benchmark ACAS Xu avec Marabou. Les résultats de nos contributions démontrent l'efficacité et la fiabilité de nos méthodes par rapport aux solveurs SMT traditionnels.

Mots clefs : Arithmétique en Virgule Fixe, Réseaux Neuronaux Quantifiés, vérification Formelle, Spécification, Véhicules Autonomes, Avionique, Théories de Modulo de Satisfiabilité.

ملخص

يمثل تعقيد الأنظمة المستقلة تحديات كبيرة في تحديد مواصفاتها الرسمية والتحقق منها، لا سيما عندما يتعلق الأمر بدمج وحدات تحكم الذكاء الاصطناعي القائمة على الشبكات العصبية الكمية. تستخدم هذه الشبكات حساب النقاط الثابتة للتكيف مع قدرة الحوسبة المحدودة للأنظمة المدمجة. يسمح حساب النقاط الثابتة بتمثيل الأوزان والتنشيطات والتدرجات باستخدام أعداد صحيحة منخفضة الترتيب، مما يتيح إجراء عملية حسابية فعالة بدقة حتمية. ومع ذلك، يقدم هذا التحديد الكمي هذا أخطاء عددية بسبب الاقتطاع والتقريب، والتي يمكن أن تنتشر عبر الشبكة وتؤثر على عملية اتخاذ القرار. وبالتالي، فقد ثبت أن التحقق من شبكات ، سواء تم نمذجتها بأعداد صحيحة أو متجهات بت، أمر صعب من حيث الدقة الحتمية و هو مسألة من فئة PSPACE-Complete

في هذه الأطروحة، نقدم ثلاث مساهمات رئيسية في مجال المواصفات والتحقق الرسمي لمتحكمات الذكاء الاصطناعي المعتمدة على شبكات في مجالات المركبات الذاتية والطيران. أولاً، نقترح عملية مواصفات رسمية لمتطلبات المركبات الذاتية، تتضمن تحويل المتطلبات النصية خطوة بخطوة إلى خصائص رسمية. ثانياً، نقدم منهجية سليمة لكن غير كاملة للتحقق من شبكات دون الاعتماد على نظريات الأعداد الصحيحة أو متجهات البت. تجمع هذه الطريقة بين نظرية المجموعات، التقريب النسبي، والتحقق باستخدام نظرية النماذج المُرضية للتحقق من الخصائص الرسمية المحددة في مساهمتنا الأولى. قمنا بتقييم هذه الطريقة في سياق المركبات الذاتية باستخدام محاكي HIGHWAY-ENV إلى جانب حلالي نظرية النماذج المُرضية وهما Z3 و Marabou . ثالثاً، نقدم طريقة تحسين فعالة لشبكات تهدف إلى الحفاظ على خصائص الشبكات العصبية من خلال تحديد أكبر اضطراب يضمن الحفاظ المستمر على هذه الخصائص. يعمل هذا الاضطراب كعتبة لتطبيق أداة ضبط الدقة Popinns ، التي تولد النسخة المكممة الأمثل للشبكة العصبية بناءً على العتبة المحددة باستخدام حلال نظرية النماذج المُرضية. قمنا بتنفيذ هذه الطريقة في مجال الطيران وتقييمها باستخدام معيار Acas Xu وأداة Marabou .

تظهر نتائج مساهماتنا كفاءة وسلامة الطرق المقترحة مقارنة بحلالات نظرية النماذج المُرضية التقليدية.

الكلمات المفتاحية : الحساب ذو النقطة الثابتة، الشبكة العصبية الكمية ،التحقق الصوري ،التخصيص، المركبات الذاتية القيادة ،أنظمة الطيران الإلكترونية، نظرية النماذج المُرضية

List of publications

Journal Papers

1. Wahiba Bachiri, Yassamine Seladji, Pierre-Loïc Garoche,*"Formal Specification and SMT Verification of Quantized Neural Network for Autonomous Vehicles"*,Science of Computer Programming,2025,103316,ISSN 0167-6423,https://doi.org/10.1016/j.scico.2025.103316.

International Conferences and seminars

1. Wahiba Bachiri, Yassamine Seladji, Pierre-Loïc Garoche. *"Formal Verification of Quantized Neural Network,"* 2024 International Conference of the African Federation of Operational Research Societies (AFROS), Tlemcen, Algeria, 2024, pp. 1-5, doi: 10.1109/AFROS62115.2024.11037165.

2. Wahiba Bachiri, *SMT Verification of Quantized Neural Networks.* ENAC (Toulouse), 5 décembre 2024

National Conferences and Communications

1. Wahiba Bachiri, Yassamine Seladji,*Parallel SMT-based Verification of Decision-making Properties using Interval analysis for Artificial Neural Networks*, NCASE24, ENSTA (Algeirs), Online , 17-18 Novembre 2024.

2. Wahiba Bachiri, *SMT Verification of Quantized Neural Network for Autonomous Vehicles*.University of Tlemcen, December 2024.

Posters

- Wahiba Bachiri, Seladji Yassamine, Pierre-Loïc Garoche.*Formal Verification of Quantized Neural Network using Bounded Model Checking*.FEANCISES, ENAC (Toulouse),2022.

- Wahiba Bachiri, Seladji Yassamine, Pierre-Loïc Garoche. *Parallel Verification of Quantized Neural Network for Autonomous Vehicle*. ISCC'24, Oran, 12-13 Novembre 2024.

List of Figures

List of Tables

List of Abbreviations

SMT	**S**atisfiability **M**odulo **T**heories
LP	**L**inear **P**rogramming
ILP	**I**nteger **L**inear **P**rogramming
MILP	**M**ixed **I**nteger **L**inear **P**rogramming
FP	**F**loating-**P**oint
MLP	**M**ultilayer **P**erceptron
ANNs	**A**rtificial **N**eural **N**etwork**s**
QNNs	**Q**uantized **N**eural **N**etwork**s**
AVs	**A**utonomous **V**ehicle**s**
FPGAs	**F**ield **P**rogrammable **G**ate **A**rray
SoC	**S**ystem **o**n **C**hip
DNN	**D**eep **N**eural **N**etwork
UAV	**U**nmanned **A**ircraft **V**ehicle
ACAS Xu	**A**irborne **C**ollision **A**voidance **S**ystem for UAV
IoT	**I**nternet **o**f **T**hings
RL	**R**einforcement **L**earning.
DQN	**D**eep **Q** **N**etwork.
FCNNs	**F**ully **C**onnected **N**eural **N**etworks.
CNN	**C**onvolutional **N**eural **N**etwork.
RNN	**R**eccurent **N**eural **N**etwork.
NN_q	Quantized Neural network
NN_r	Rational Approximation Version of NN_q.
$\widetilde{NN}_r$	Perturbed rational Neural Network.

Contents

Chapter 1

Introduction

1.1 Research Motivation and Context

Autonomous systems are self-operating technologies that intelligently perform tasks without human intervention, relying on sensors, AI, and real-time decision-making [ABG+20, TZW+23, RZB18, SMN19]. These systems are closely related to embedded systems [ELS18, SEI+24], which provide the essential hardware and software foundation for efficiently processing data, controlling actuators, and executing tasks in dynamic environments with low power consumption and reliable, real-time operation [JSNA19, ADAA18, SEI+24].

Intelligent controllers constitute the foundational component of autonomous systems, seamlessly integrating hardware and software to facilitate real-time decision-making[APW89, dJP23]. These systems leverage advanced control algorithms, such as fuzzy logic [DCV21], Bayesian probability [Gad17], neural networks [AAE22], and PID controllers [VSD+21], to enhance precision and adaptability [SMN19, RZB18, TZW+23, XMW+18]. By processing sensor data from LiDAR, cameras, and IoT devices, they facilitate environment perception and autonomous navigation. Machine learning techniques further optimize performance, allowing self-learning and predictive control in dynamic environments [KPK+23, KPK+23].

Great efforts are currently underway to utilize neural networks (NNs) as controllers, driven by their capability to learn complex, nonlinear dynamics and adapt in real time. Recent research highlights advancements in deep reinforcement learning (DRL) , which enable NNs to outperform traditional control methods in robotics and autonomous systems. Recent studies [GDNC15, TKAKP22] demonstrate their efficiency in resource-constrained environments. These innovations pave the way for smarter, self-learning control systems across various industries.[CF17, BL17, YZ18], where reliability and low latency are essential. Key challenges such as energy efficiency and miniaturization [ABG+20, JRBA19, ZDRF21] are being addressed through optimized embedded architectures, including SoCs and FPGAs [REGSV93, DJS16].

The future of these systems lies in creating more intelligent, collaborative, and trustworthy machines that seamlessly integrate into human environments. However, certifications for both security and reliability are essential in these domains. However, numerous studies

have demonstrated that NNs, due to their complex architecture and highly nonlinear nature, often exhibit unpredictable behavior. They can make dangerous control errors under edge-case conditions or when subjected to adversarial perturbations, raising concerns about their deployment in real-world scenarios [SKS19, Spr22].

How to prove the Reliability of Intelligent Controllers Based on Neural Networks?

To enhance the reliability of intelligent controllers based on neural networks, various studies have proposed two fundamental verification paradigms: sound and unsound verification methods [KKR24, UM21]. Unsound verification provides scalable checks using techniques like adversarial testing and heuristic analysis. However, because these methods are unsound, they do not guarantee that the system satisfies all required formal properties, such as correctness or safety. While they can efficiently detect potential issues or improve confidence in the system, their results may include false negatives (missed violations) or false positives (spurious warnings). It treats the neural network as a white box and generates test cases to optimize various coverage criteria, including neuron coverage and condition/decision coverage [SWR+18, PCYJ18]. In contrast, sound verification provides mathematically rigorous proofs of system correctness using formal methods, ensuring no overlooked violations, albeit at the cost of higher computational complexity [KBD+17, GMD+18, ea21].

Formal verification of autonomous systems ensures correctness and safety by mathematically proving that the system meets specified requirements under all possible conditions. Formal methods exhaustively analyze system behavior using techniques such as model checking [Bie21], reachability analysis [EHSH21], theorem proving [LSM+24], and abstract interpretation [CC92]. These methods verify critical properties, including collision avoidance, deadlock freedom, and real-time responsiveness, in autonomous vehicles, drones, and robotics. However, challenges remain in scaling to complex hybrid (discrete-continuous) dynamics and verifying machine learning components, such as neural networks.

Deployment of Neural Networks on Embedded Systems

Another factor affecting the deployment of neural networks in autonomous systems is their high computational demands, memory constraints, and power consumption, all of which are critical in real-time, resource-constrained environments. Neural networks, particularly deep learning models, often require significant processing power and memory, making them challenging to run efficiently on embedded hardware with limited resources [MGL+22]. To tackle these issues, quantization has emerged as a key optimization technique [WMYY24a, NBHP+21], transforming floating-point neural networks performed in floating-point arithmetic into quantized neural networks (QNNs) that are more hardware-friendly.

Quantization reduces the precision of the network's weights and activations (e.g., from 32-bit floating-point to 8-bit integers), significantly decreasing memory footprint and energy consumption while maintaining acceptable accuracy [B+19, X+22, RAK22]. Unlike full-precision networks, QNNs leverage fixed-point arithmetic instead of floating-point

operations, enabling faster computations on embedded processors that lack dedicated floating-point units [LJVD23a, HPP24]. This shift not only reduces memory bandwidth but also accelerates inference by optimizing for fixed-point operations, which are more efficient on edge devices like microcontrollers and Field Programmable Gate Array (FPGAs) [GSZ23]. As a result, it enables efficient execution on autonomous systems while ensuring real-time performance and energy efficiency as illustrated in the Figure 1

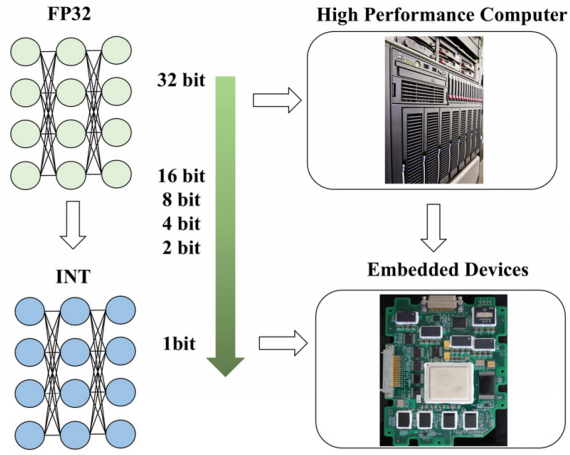


Figure 1: Quantization: Turning the 32-bit floating-point neural network to an 8-bit or even lower bit integer network [WMYY24b].

This thesis focuses on the formal verification of intelligent controllers that use neural networks, specifically quantized neural networks (QNNs). QNNs offer substantial benefits in computational efficiency and memory footprint for embedded deployment. Our analysis will concentrate on two safety-critical autonomous systems that rely on autonomous vehicles and avionics, which will serve as case studies for evaluating our contributions. These domains are particularly suitable for examination due to their robust reliability requirements, real-time operational constraints, and the growing implementation of neural network-based control systems.

1.1.1 Intelligent Controllers for Autonomous Vehicles

Intelligent controllers are advanced systems that enable autonomous vehicles (AVs) to perceive, decide, and act in real-time with minimal human intervention [SEI+24, Ara22]. These systems heavily rely on ANNs across all core AVs components to ensure robust and adaptive performance [INF18, Ara22].

✻ **Neural Networks in AVs Control.** In perception, deep learning models like Convolutional Neural Networks (CNNs) and Vision Transformers (ViTs) process raw sensor data from cameras, LiDAR, and radar for tasks [TV22, BMKL23] such as object detection (e.g., YOLO, Faster R-CNN) [SDB21], and semantic segmentation (e.g., U-Net, DeepLab) [MHU+22]. Additionally, Recurrent Neural Networks (RNNs) and Long Short-Term Mem-

ory (LSTM) networks enhance temporal perception by tracking dynamic objects and predicting trajectories [HXL+19]. For decision-making and planning, Reinforcement Learning (RL)-based policies, including Deep Q-Networks (DQNs) and Proximal Policy Optimization (PPO), facilitate high-level behavior planning in complex traffic scenarios such as lane changes, merges, and intersection navigation [INF18, Ara22].

✻ **Formal Verification of AVs.** In verification, several research studies have formally verified properties of AVs, such as collision avoidance, lane-changing safety, and robustness, using various methods [FBEG16, MI23, Alt10]. For collision avoidance, techniques like reachability analysis (e.g., Hamilton-Jacobi Reachability) [Alt10, BCHT17] and formal model checking have been applied to ensure that AVs maintain safe distances under dynamic conditions [ADK19, GMSL18]. For lane-changing, approaches such as temporal logic (e.g., Linear Temporal Logic (LTL)) have been utilized to validate proper merging behaviors [MRMA20]. Robustness verification against sensor noise and adversarial perturbations has been explored using neural network verification tools (e.g., Marabou, Reluplex) to ensure that AVs operate safely under uncertainty [KBD+17, WIZ+24]. Collectively, these methods enhance the reliability of AVs systems by mathematically proving critical safety properties.

✻ **AVs Simulators.** In the context of autonomous driving, several simulators for autonomous vehicles (AVs) have been developed to model complex driving scenarios in virtual environments [DRC+17, RST+20, SDLK18, Leu18]. These simulators enable rigorous testing of safety and functionality before real-world deployment. They simulate Level 5 autonomy [KEM+23], where the vehicle operates independently under all conditions without human input. Notable tools include CARLA [DRC+17], an open-source simulator that supports perception and control validation, and LGSVL [RST+20], which integrates with ROS/ROS2 for testing full autonomy stacks. WEBOTS [Mic04] offers a high-fidelity environment for prototyping and verification based on formal methods, while AIRSIM [SDLK18] provides photorealistic simulations with APIs for verifying perception and planning. Together, these tools enhance correctness through simulation-based model checking, temporal logic verification, and falsification testing.

✻ **HIGHWAY-ENV.** In this work, we utilized the HIGHWAY-ENV simulator [Leu18], an open-source reinforcement learning (RL) environment designed specifically for autonomous driving research. This simulator is particularly well-suited for the formal verification of QNNs in autonomous driving, especially when compared to more complex simulators like CARLA, AirSim, or LGSVL [DRC+17, RST+20, SDLK18], due to its lightweight, customizable, and deterministic nature. HIGHWAY-ENV [Leu18] offers a simplified yet highly controllable 2D environment optimized for reinforcement learning (RL) and formal verification tasks. This simplicity enables efficient and repeatable testing of QNN-based controllers. Since QNNs introduce quantization errors that may impact decision-making, HIGHWAY-ENV 's deterministic dynamics facilitate precise perturbation analysis and ad-

versarial scenario generation. Additionally, its integration with OpenAI Gym [BCP+16] supports automated testing frameworks, making it easier to apply formal verification tools like Marabou [WIZ+24] for neural network verification.

1.1.2 Intelligent Controllers for Avionics

Modern avionics systems are increasingly adopting intelligent controllers that leverage machine learning algorithms and specially neural networks to enhance aircraft performance, safety, and operational efficiency[WKJC22, BMN+19, CFG+22]. These advanced control systems provide superior handling of complex, nonlinear flight dynamics compared to traditional methods[JEMLS17, DJ10, JPH20].

✻ **Neural Networks for Avionics Control.** In flight control systems, deep neural networks and recurrent architectures like LSTMs enable adaptive flight envelope protection and fault-tolerant control [GWLP22, ZLDC23]. For navigation and guidance, vision-based ANNs process satellite imagery and sensor fusion data to improve GPS-denied navigation [AAM23]. Meanwhile spiking neural networks offer ultra-low-power solutions for onboard processing in unmanned aerial vehicles [SVB+20]. The autopilot systems increasingly employ reinforcement learning to optimize flight trajectories in real-time, considering weather, fuel efficiency, and air traffic constraints [BDKG18]. Collision avoidance systems leverage deep learning for real-time processing of radar data [MWMC10]. For cyber-physical security, ANNs detect anomalies in avionics networks through unsupervised learning, protecting against false data injection and spoofing attacks [Mal19].

✻ **Formal Verification of Flight Control Systems.** The certification of AI-driven systems remains challenging, leading to research on formally verified neural networks and explainable AI techniques that rigorously analyze neural network decision boundaries to ensure they adhere to specified input-output relationships under all possible operating conditions. Robustness verification uses interval bound propagation to certify that navigation and control networks maintain stable performance despite sensor noise, adversarial perturbations, or edge-case inputs [FMP20, MBT+22]. Safety-critical properties like collision avoidance and flight envelope protection are verified through reachability analysis tools that compute provable safe regions in the system's state space [vdBdV18], while temporal logic verification ensures real-time compliance with strict avionics timing constraints [DKH09]. The verification process must also address implementation-specific concerns, including the effects of quantization and fixed-point arithmetic in deployed models.

✻ **Benchmarks and Simulators.** There are several benchmarks and simulation tools specifically designed to support the development and validation of neural networks in avionics applications. Airborne Collision Avoidance System for Unmanned Aircraft (ACAS Xu) [JK19] provide standardized collision avoidance scenarios for formal verification of

ANNs safety properties, Meanwhile FlyNet [DMY+20] offers vision-based flight datasets for perception system evaluation. For simulation, high-fidelity platforms such as AirSim [SDLK18] enable realistic testing of NN-driven flight controllers in virtual environments, complemented by lightweight options like JSBSim [SJA+24] for rapid prototyping. These tools often integrate with reinforcement learning frameworks and hardware-in-the-loop systems to thoroughly validate NN performance under various flight conditions, bridging the gap between theoretical development and real-world aerospace implementation while meeting stringent certification requirements.

✻ **ACAS Xu.** In this thesis we use ACAS Xu system [JK19] is a neural network-based decision-making system designed to help unmanned aerial vehicles (UAVs) avoid mid-air collisions. This system represents a crucial advancement in automated aviation safety [OPM+19, MJ16]. Quantized versions of ACAS Xu neural networks (e.g., 8-bit or 4-bit models) enable faster execution and lower power consumption, which are critical for real-time avionics hardware. However, quantization introduces verification challenges, as reduced precision may affect decision boundaries and robustness. This makes ACAS Xu a key benchmark for studying the trade-offs between efficiency, accuracy, and verifiability in quantized avionics neural networks.

1.2 Research Problem Statement

In the context of formal verification, the formal specification of ANNs plays a crucial role by providing a rigorous mathematical framework for expressing and analyzing system requirements. This process systematically transforms high-level textual specifications into formal properties, including safety properties (e.g., collision avoidance guarantees) [BES16], temporal properties (e.g., Linear Temporal Logic (LTL) or Signal Temporal Logic (STL) [Koy90], and robustness constraints (e.g., adversarial perturbation tolerances) [ZH18, MBT+22]. Formal specifications serve as a bridge between high-level safety objectives and low-level verification, enabling engineers to demonstrate that ANNs comply with critical requirements in all operational scenarios while also identifying potential violations. This is essential for deploying trustworthy AI in safety-critical applications.

Various methods verify floating-point neural networks [KBD+17, WIZ+24, GMD+18], addressing challenges such as the non-linearity of activation functions (e.g., ReLU, tanh, sigmoid), architecture-specific demands (e.g., CNNs, RNNs), and property-specific needs (e.g., robustness, safety). Each of these factors influences the design and precision of the verification method [Sou24].

Generally, the formal verification of neural networks is categorized into two broad types: complete methods and incomplete methods [Sou24, UM21]. Complete methods offer formal guarantees by exhaustively analyzing all possible behaviors of the network within a defined input space. these methods rely on using SMT and Integer Linear Programming

(ILP) which provide rigorous guarantees about network behavior. Notable SMT-based tools include Marabou [WIZ+24], which extends the Reluplex [KBD+17] algorithm to verify properties like robustness and safety in deep neural networks (DNNs) by solving quantized or floating-point constraints, and NeVer [DGPT24], which combines SMT solving with abstraction techniques for scalable verification. For ILP-based approaches, MIPVerify [TXT19] formulates DNN verification as a mixed-integer program to certify adversarial robustness, while Planet [Ehl17] encodes ReLU-based networks into linear constraints for exact verification. These works ensure soundness and completeness and demonstrate how SMT and ILP provide foundational frameworks for DNN verification, balancing precision and computational tractability in Verifying neural network correctness. However it often face scalability limitations due to the high computational complexity of floating-point arithmetic and deep network structures. Incomplete methods [GMD+18, MKE23] trade absolute guarantees for improved scalability by using abstract interpretation, which has emerged as a powerful framework for scalable formal verification of neural networks. These methods use domain-specific abstractions to approximate network behaviors while preserving soundness. Key approaches include interval analysis [LLY+19], which propagates input bounds layer by layer using hyperrectangular over-approximations, and zonotope-based methods [SFG22], which utilize affine arithmetic to capture linear dependencies between neurons for tighter bounds. Advanced domains like polyhedra [MSS+21] further enhance precision by modeling nonlinear activations with polynomial approximations. Tools such as ERAN [SBR+] unify these abstractions within a configurable framework, enabling the verification of safety and temporal properties. Although inherently incomplete due to over-approximation, these methods excel at scaling to large networks, such as ResNet-50 [Koo21], and are widely adopted in autonomous systems certification, as demonstrated by AI2's ReluVal [GMD+18] for collision avoidance verification. Recent extensions combine abstract interpretation with SMT refinements [KBK+23] to mitigate conservatism, thereby striking a practical balance between precision and computational tractability in industrial-scale applications. While these approaches can handle larger networks and more complex properties, they may produce over-approximations or false positives due to relaxed precision constraints.

The complete and incomplete methods, primarily designed for floating-point or ReLU-based networks, often struggle to scale efficiently with low-bit-width quantization (e.g., 4-bit) due to the non-linear and discrete nature of quantization. Furthermore, scalability issues emerge when using integer or bit-vector encodings in SMT verification, which have been shown to be **PSPACE-complete** [HLZ21], resulting in an exponential increase in computational resource requirements as the network size grows. Recent works propose alternative approaches to mitigate these challenges. In [HLZ21], they reduce and simplify the encoding of a Quantized Neural Network (QNN) using fixed-point arithmetic for SMT verification. Similarly, in [ZZC+22], the authors convert non-linear activation functions into integer linear constraints for verification via ILP. Additionally, [HWY+24] employs Gradient-Based Heuristic Search Methods and interval propagation to reduce the search

space before applying integer linear programming. Another strategies focuses on verifying the constraints of the quantization error bound between a Deep Neural Network (DNN) and its quantized counterpart, ensuring that the quantized model adheres to acceptable deviation limits [ZSS23, BMS22, BM24b]. In QEBVerif [ZSS23], the authors utilize a hybrid method based on Differential Reachability Analysis (DRA) and Mixed-Integer Linear Programming (MILP). DRA efficiently approximates the reachable sets of both the DNN and QNN, while MILP refines the error bounds with exact constraints.

The objective of this thesis is to propose a robust formal verification method for intelligent controllers based on neural networks, particularly quantized neural networks, while avoiding the use of integer and bit vector theory. In this thesis, we propose a formal specification method that involves transforming AV requirements into formal properties. Additionally, we proposed a sound, and incomplete verification method to verify QNNs based on SMT and abstract interpretation using interval analysis and set theory to verify safety properties. Furthermore, we design a quantization method that utilizes precision tuning and formal methods to achieve the most optimized precision format while preserving safety properties.

1.3 Research Contributions

1.3.1 Contribution 1: Formal Specification of AVs Textual Requirements

Our aim in the first contribution is to transform the textual and informal requirements of autonomous vehicles (AVs) into formal properties that can be exhaustively verified on neural networks (NNs). We achieve this through five steps designed to extract logical constraints and range values using the concepts of abstract and logical scenarios derived from the AVs' informal requirements. This process helps identify a formal safety property to be verified using SMT verification with HIGHWAY-ENV [BSG25]. We defined all the details and these steps in **Chapter** 5.

1.3.2 Contribution 2: SMT Verification of QNNs for AVs based on Set Theory and Rational Approximation

In the second contribution, we proposed a sound yet incomplete formal verification method for QNNs in autonomous vehicles (AVs). This method focuses on using the rational approximation of QNNs and set-based theory, which involves introducing bounded perturbations to the output of the approximated rational neural network. Additionally, we ensured that the output sets of the perturbed rational neural network include those of both the QNN and its rational approximation. The distance between these output sets is computed using the p-norm and interval propagation. To evaluate our methodology, we utilized the Highway-env autonomous vehicle simulator.

In this contribution, we summarize the experiments with the following points:

- Identification of the most effective norm for validating the proposed verification method in terms of time, memory, and verification results for the requirements of autonomous vehicles (AVs) [BSG25].

- Comparison of the proposed verification method to bit-vector/integer encoding using SMT solvers [BSG25].

- Evaluation of the impact of quantization as an optimization technique for accelerating the verification process of neural networks [BSG24].

- Evaluation of the effects of parallelism by concurrently verifying sub-properties, focusing on execution time and memory usage in our verification method [WB24].

We will details this contribution in **Chapter 6**.

1.3.3 Contribution 3: Design of Quantization Method of ANNs using Precision Tuning and Formal Methods for Avionics.

In the third contribution, we propose a quantiztion method of ANNs based on formal methods and pecision tuning, we focus on preserving safety properties by identifying maximum bounded perturbations and using them in precision tuning to evaluate the error between the (ANN) and its quantized version, enabling us to determine the optimized format of the generated QNN that meets fixed-point constraints. This method is evaluated using Acas Xu [JK19], a neural network-based decision-making system designed to help (UAVs) avoid mid-air collisions, along with its safety properties, and Marabou [WIZ+24] as an SMT solver.

We summarize the ideas behind this method in the following two major phases :

- The first phase begins by verifying the safety properties of the initial neural network using SMT. If the property is violated, we conclude that it is invalid for both the initial neural network and its quantized version. If the property holds, we proceed to ensure its preservation by introducing bounded perturbations to the neural network's output and verifying the same property identifying the maximum robustness threshold that guarantees the property. The output of this phase is the maximum robustness threshold needed to preserve the safety property.

- In the second phase, we consider the output from the previous phase as an error between the NN and its quantized version to use in the precision tuning tool `Popinns` [BBKBM22]. Using this tool, we produce an optimized precision format for the QNN that satisfies both the fixed-point constraints and the safety property based on the specified error. Consequently, if these conditions are met, we can confirm that the QNN also satisfies the safety property; otherwise, it is deemed violated.

We examine the details of this contribution in **Chapter 7** and **8**

1.4 Organization of the Dissertion

The remainder of this dissertation is divided into 3 parts:

Part I : State of the art

This part consists of three chapters that outline the background and related works pertinent to our thesis.

Chapter 2 provides the necessary background knowledge on computer arithmetic to understand and follow the concepts presented in this thesis. We begin with an overview of floating-point arithmetic, followed by essential concepts related to fixed-point arithmetic and its elementary operations. Additionally, we briefly discuss interval analysis and its fundamental operations.

Chapter 3 introduces the concept of neural networks, including their common architectures and activation functions. We also discuss several compression methods applied to artificial neural networks (ANNs) for use in intelligent controllers. Next, we define quantized neural networks (QNNs) and their activation functions.

Chapter 4 introduce common formal methods such as model checking, abstract analysis, static analysis, and SMT. Additionally, we present a summary of existing techniques and tools concerning the formal verification of neural networks and quantized neural networks.

Part II : Specification and SMT Verification of QNN for Autonomous Vehicles

This part is organized into two chapters, addressing the specification and formal verification of QNNs proposed in this thesis.

Chapter 5 Identifies the process of specifying the requirements for autonomous vehicles and transforming them into formal properties for verification.

Chapter 6 outlines a robust formal method for verifying QNNs, based on rational approximation, set-based theory, and SMT verification. We begin with a set-based overview and the general verification process of the proposed method. Next, we present a mathematical proof, followed by a detailed explanation of each step in the process.

Chapter 5 and 6 are revised versions of the following articles:

- Wahiba Bachiri, Yassamine Seladji, Pierre-Loïc Garoche,*Formal Specification and SMT Verification of Quantized Neural Network for Autonomous Vehicles*,Science of Computer Programming,2025,103316,ISSN 0167-6423,
  https://doi.org/10.1016/j.scico.2025.103316.
- Wahiba Bachiri, Yassamine Seladji, Pierre-Loïc Garoche. *"Formal Verification of Quantized Neural Network,"* 2024 International Conference of the African Federa-

tion of Operational Research Societies (AFROS), Tlemcen, Algeria, 2024, pp. 1-5, doi: 10.1109/AFROS62115.2024.11037165.

* Wahiba Bachiri, Seladji Yassamine, Pierre-Loïc Garoche. *Parallel Verification of Quantized Neural Network for Autonomous Vehicle*. ISCC'24, Oran, 12-13 Novembre 2024.

Part III : Design of QNNs by Preserving Their Safety Properties for Avionics

This part is organized into two chapters that present a quantization method of ANNs using formal method and precision tuning.

Chapter 7 explains and defines the quantization method for ANNs, which considers a floating-point neural network as input. We then introduce bounded perturbation, which relies on the maximum output perturbation that satisfies the property. This perturbation serves as input for a precision tuning tool, `Popinns`, designed to identify the optimized QNN format for each neuron that meets the safety property.

Chapter 8 presents the case study used in the experimentation: Acas Xu. We outline the results of our methods through 45 NNs and 10 properties. Then, we discuss the results of δ'_{max} and the results of `Popinns`.

Chapter 9 : Conclusion and Prospectives

In this chapter, we summarize the contributions presented in this dissertation and outline several propositions for future research directions and perspectives.

Part I

State of The Art

Chapter 2

Computer Arithmetic

2.1 Introduction

Computer arithmetic refers to the methods and systems used to perform numerical calculations in digital computing, encompassing both hardware and software implementations of mathematical operations. Unlike idealized mathematical operations that assume infinite precision and continuous ranges, computer arithmetic must address finite storage, limited precision, and discrete representations. This leads to fundamental challenges such as rounding errors, overflow, and underflow [Par02, MM16].

In this chapter, we present some of fundamentals of computer arithmetic to help the reader to gain intuition about situations that may compromise the quality of numerical

computation results and how to avoid these issues. We will focus on floating-point arithmetic, which uses a sign, exponent, and significand to dynamically scale numbers, allowing for a wider range of values. However, this method also introduces complexities such as rounding errors and catastrophic cancellation. We will also explore fixed-point arithmetic, which represents numbers with a fixed number of integer and fractional bits. This approach is often suitable for embedded systems and digital signal processing due to its deterministic performance. Additionally, we will examine interval arithmetic, which bounds errors by working with ranges instead of exact values.

The design of computer arithmetic systems involves trade-offs among speed, accuracy, energy efficiency, and hardware complexity [MBdD+18, DBL85]. Applications vary from high-performance scientific computing to low-power edge devices. Ultimately, the choice of arithmetic system depends on the specific problem domain, balancing the need for precision with constraints such as real-time performance and power consumption [MM16].

2.2 Floating-Point Arithmetic

Representing and manipulating real numbers efficiently is essential in many fields of science and engineering. Since the emergence of electronic computing, various methods for approximating real numbers on computers have been developed. However, floating-point arithmetic remains the most widely used method for representing real numbers in modern computers [MM16, MBdD+18].

Simulating an infinite, continuous set (the real numbers) with a finite set (the "machine numbers") is not a straightforward task; it requires finding clever compromises between factors such as speed, accuracy, dynamic range, ease of use, implementation, and memory cost [MBdD+18]. Floating-point arithmetic, with appropriately chosen parameters (radix, precision, extremal exponents, etc.), proves to be a very effective compromise for most numerical applications [DBL85].

2.2.1 Representation of Floating-Point Numbers

Since its inception, IEEE754 [DBL85] has established itself as the scientific standard for specifying floating-point arithmetic. It specifies the bitwise representation for numbers, and specifies parameters for how arithmetic is to be performed.

In this standard, each floating-point number can be represented by five integers [PU20b], (s,β,e,M,p), consisting of :

- Sign $s \in \{0,1\} \cap \mathbb{N}$
- Radix or base $\beta \in \mathbb{N}, \beta \geqslant 2$
- Exponent $e \in \mathbb{Z}$
- Mantissa $M \in \mathbb{N}$

- Precision $p \in \mathbb{N}$, $p \geqslant 2$, p represents the number of "significant digits" , or the length of M .

Floating-point numbers can be formatted with specific precision, such as single or double, as illustrated in Table 2.

Definition 2.1 formally defines a floating-point number. $\mathbb{F}$ represents the set of floating-point numbers, where $\mathbb{F} \subseteq \mathbb{R}$ [MBdD+18].

Definition 2.1. *Let $x \in \mathbb{F}$ a floating point number of base 2 defined as follows :*

$$x = s \,.\, M \,.\, \beta^{e-p+1} \tag{2.1}$$

There is two extremal exponents $e_{min}, e_{max} \in \mathbb{Z}$ such that $e_{min} < e < e_{max}$ and $M = d_0.d_1...d_i...d_{p-1}$, $d_i \in \{0, 1\}$ and $0 \leqslant i < p$. The values of e_{min}, e_{max} and p are given in Table 1

Figure 2 illustrates the representation of a floating-point number in single-precision format ($FP32$), which consists of one bit for the sign, an 8-bits exponent, and a 23-bits mantissa.

32-bit Single Precision Floating Point

s	Exponent	Mantissa
1 bit	8 bits	23 bits

Figure 2: Representation of IEEE 754 single-precision format

In Table 1, we display the various precisions and formats for number representation in the IEEE754 standard.

The first row indicates the total number of bits used to represent a floating-point number, while the second row provides its name. The third and fourth rows specify the number of bits allocated for the mantissa (p) and exponent (e), respectively. The last two rows present the minimum and maximum values that the exponent can assume [PU20b].

Format	binary FP16	binary FP32	binary FP64	binary FP128
Name	Half precision	Single precision	Double precision	Quadruple precision
p	10	23	52	112
e	5	8	11	15
e_{min}	-14	-126	-1122	-16382
e_{max}	$+15$	$+127$	$+1223$	$+16383$

Table 1: Parameters defining the IEEE754 floating-point formats.

Example 2.1. *We assume radix $\beta = 2$, precision $p = 4$, $e_{min} = -7$, and $e_{max} = +8$. The number $416_{10} = 110100000_2$ is a floating-point number. It has one representation only, with integral significand 13_{10} and exponent 8_{10}, since*

$$416 = 13 \,.\, 2^{(8-4+1)} \tag{2.2}$$

IEEE754 [DBL85] defines several types of values: normalized numbers (the default representation for most values, using a hidden leading 1 in the mantissa), denormalized (subnormal) numbers (which preserve gradual underflow by allowing tiny numbers near zero to lose the hidden bit), and special values like signed zero ($\pm$), infinities ($\pm\infty$), and NaN (Not a Number) for invalid operations. Floating-point operations can be performed using different rounding modes to control how results are approximated when they can't be represented exactly. Additionally, the standard specifies five types of exceptions that flag unusual conditions (like division by zero or overflow), which can either trigger hardware interrupts or be silently recorded in status flags [For92, Thé14]. These features work together to provide a robust, predictable system for numerical computation while balancing precision, range, and performance. A detailed and rigorous discussion of floating-point arithmetic can be found in Muller et al.'s Handbook of Floating-Point Arithmetic [MBdD+18].

2.2.2 Elementary Operations in Floating-Point Arithmetics

Processors with hardware-supported floating-point operations require complex circuitry to handle basic IEEE754 standard data operations. Whether performed in software or hardware, these arithmetic operations involve multiple steps [PL03, AASA11].

Floating-Point Addition

The steps for floating-point addition and subtraction are identical, regardless of the format. To add or subtract two floating-point numbers x_1 and x_2, the following operations are performed as follows [HH13, MBdD+18, PL03]:

1. Extract the exponents e_{x1} and e_{x2}.
2. Extract the mantissas M_{x1} and M_{x2}, and convert them into 2's complement numbers, using the signs s_{x1} and s_{x2}.
3. Shift the significand with the smaller exponent right by $|e_{x1} - e_{x2}|$.
4. Perform addition (or subtraction) on the mantissas to get the mantissa of the result, M_r. Remember that the result may require one more significant bit to avoid overflow.
5. If M_r is negative, then take the 2's complement and set S_r to 1. Otherwise set Sr to 0.
6. Shift M_r until the leftmost 1 is in the "hidden" bit position, and add the shift amount to the smaller of the two exponents to form the new exponent e_r.
7. Combine the sign s_r, the exponent e_r, and significand M_r to form the result.

Example 2.2. *The floating-point addition of* $7.875(1.11111 \times 2^2)$ *and* $0.1875(1.1 \times 2^3)$ *is* 8.0625 (1.0000001×2^3). *The fraction and exponent bits are extracted, and the implicit leading* 1 *is prepended. Next, the exponents are compared by subtracting the smaller exponent from the larger one. This difference indicates how many bits the smaller number must be shifted to the right to align*

the implied binary point, ensuring the exponents are equal. The aligned numbers are then added. Since the sum has a mantissa greater than or equal to 2.0*, it is normalized by shifting it right by one bit and incrementing the exponent. In this example, the result is exact, so no rounding is necessary. Finally, the result is stored in floating-point notation by removing the implicit leading one from the mantissa and prepending the sign bit. [HH13]*

Floating-Point Multiplication

To multiply two floating-point numbers x_1 and x_2 , the following steps are performed [AASA11, PL03]:

1. Compute the sign of the result s_r
2. Extract the exponents e_a and e_b
3. Extract the mantissa M_a and M_b
4. Multiply (or divide) the mantissa to form M_r.
5. Add (or subtract) the exponents (in excess-N) to get e_r.
6. Shift Mr until the leftmost 1 is in the "hidden" bit position, and add the shift amount to e_r.
7. Combine the sign s_r, the exponent e_r, and mantissa M_r to form the result

Example 2.3. *let the* IEEE754 *representation of 40 and -7.5 as follow* 0100001000100 *and* 1100000011110 *respectively. After extracting the exponents, we multiply the mantissa* 1.0100 × 1.1110 = 1001011000*. We place the decimal point :* 10.01011000*. Next, we add exponents (*10000100 + 10000001 = 100000101*). The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; So we should subtract the bias from the resultant exponent otherwise the bias will be added twice :* 100000101 − 01111111 = 10000110*. We obtain the sign bit and put the result together:* 11000011010.01011000*. we normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the Exponent by 1 :* 11000011010.01011000 *(before normalizing)* 1100001111.001011000 *(normalized) The result is (without the hidden bit):* 11000011100101100*. The mantissa bits are more than 4 bits; rounding is needed. If we applied the Truncation rounding mode then the stored value is:* 1100001110010 *[AASA11].*

2.2.3 Rounding Errors

Since the set of floating-point numbers is not closed under most operations, such as addition or multiplication. The IEEE 754 standard defines three exceptions [Ple17, PU20b, Thé14] that can occur during these operations:

- An **overflow** occurs when rounding the exact result with no constraint on the exponent range leads to a number larger in magnitude than the largest floating-point number. For instance, adding $(\beta^p - 1)\beta^{e_{max}-p+1}$ to itself would raise an overflow.
- An **underflow** after rounding occurs when rounding the exact result with no constraint on the exponent range leads to a nonzero number, smaller in magnitude than $\beta^{e_{min}}$. For instance, dividing $\beta^{e_{min}-p+1}$ by β would lead to an underflow.
- An **underflow before rounding** occurs when the exact result is nonzero and smaller in magnitude than$\beta^{e_{min}}$.

Floating-point arithmetic allows for a wide dynamic range in numerical computation but inherently introduces rounding errors due to its finite precision representation of real numbers. These errors occur when arithmetic operands have already been rounded, which can lead to a catastrophic loss of significant digits and ultimately result in entirely incorrect outcomes. While each rounding error may be small on its own, their cumulative effect can propagate through computations, rendering the final output meaningless[Ben21, PU20b].

2.3 Fixed-Point Arithmetic

Fixed-point arithmetic is a numerical representation system where numbers are stored with a fixed number of integer and fractional bits, making it ideal for resource-constrained systems like embedded devices and digital signal processors (DSPs) [BM24b, Naj14, Lop14]. Unlike floating-point, fixed-point avoids hardware-intensive operations by maintaining a constant scaling factor, enabling faster and more power-efficient computations. However, its precision and range are inherently limited by the chosen bit allocation, requiring careful design to avoid overflow or excessive quantization errors [ATD05, HPP24].

2.3.1 Representation of Fixed-Point Numbers

Fixed-point arithmetic represents real numbers by separating them into integral and fractional parts with a binary point. Both the integer and fractional parts have a predetermined length, and the position of the binary point can be adjusted as necessary [Par02, HH22].

$\hat{x}_{\langle w,f \rangle}$ represents a fixed-point number. The total width (word width) of $\hat{x}_{\langle w,f \rangle}$ is denoted by $w \in \mathbb{N}$, where $w = k + f$. $k, f \in \mathbb{N}$ represent the number of bits for integer and fractional parts, respectively.

Definition 2.2 (Fixed-point Number). *Let $\mathcal{F}$ represents the set of fixed-point numbers, we define the fixed-point number $\hat{x}_{\langle w,f \rangle} \in \mathcal{F}$ as follow :*

$$Let \quad x \in \mathbb{R}, f \in \mathbb{N}, \quad \hat{x}_{\langle w,f \rangle} = \lfloor x * 2^f \rfloor \tag{2.3}$$

where $\lfloor . \rfloor$ corresponds to the integer part, which is either less than or greater than the nearest whole number.

We focus on the binary format of fixed-point arithmetic because it is widely used in hardware and software implementations. Additionally, developers utilize both signed and unsigned fixed-point formats depending on their intended usage. The signed or unsigned format determines whether the bit pattern representing the fixed-point number should be interpreted as a signed or unsigned integer, respectively [BHL+20a, HH22].

For instance, Fig. 3 shows a 16-bit register storing a fixed-point number $\hat{x}_{\langle 16,10\rangle}$. In this case, 6 bits are allocated for the integer part, including 1 bit for the sign, and 10 bits are allocated for the fractional part.

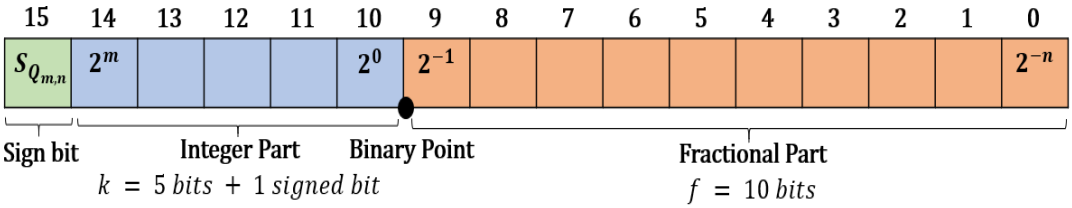


Figure 3: Representation of Fixed-point Number $\hat{x}_{\langle 16,10\rangle}$ on 16-bit.

Example 2.4. *We consider a real number* $x = 1.234$ *, the fixed-point representation of* x *on word width format* w *of 8 bits and fractional part of 5 bits.*

$\hat{x}_{\langle 8,5\rangle} = \lfloor 1.234 * 2^5 \rfloor = 39$

its binary representation defined as follow :

$\hat{x}_{\langle 8,5\rangle} = 001.00111_2$

2.3.2 Elementary Operations in Fixed-Point Arithmetic

Fixed-point operations involve processing numerical data with a predetermined and constant position of the decimal point throughout computations. Unlike floating-point arithmetic, which dynamically adjusts precision, fixed-point arithmetic relies on integer-based manipulation with implicit scaling. Common operations include addition, subtraction, multiplication, and bit shifts, all of which require careful handling of overflow and quantization errors [HH22, Ben22, Naj14, Lop14].

Addition

Let $\hat{x}_{1,\langle w_1,f_1\rangle}$ and $\hat{x}_{2,\langle w_2,f_2\rangle}$ be two fixed-point numbers , where w_1, w_2, f_1, f_2, k_1, and k_2 represent the word width , fractional bits and integer bits of $\hat{x}_{1,\langle w_1,f_1\rangle}$ and $\hat{x}_{2,\langle w_2,f_2\rangle}$ respectively. In fixed-point addition denoted as $\oplus$, it involves aligning the lengths of the fractional parts, denoted as f_1 and f_2, as illustrated in Fig 4. f_r refers to the fractional part of the results. If $f_1 > f_r$, we truncate $f_1 - f_r$ bits. Otherwise, we append $f_r - f_1$ zeros to the right of $\hat{x}_{1,\langle w_1,f_1\rangle}$. The same procedure is applied to $\hat{x}_{2,\langle w_2,f_2\rangle}$. The length of the integer part of the result k_r should be the greater value between k_1 and k_2. If a carry occurs, we increase the number of bits in the integer part by 1. Otherwise, the result will be incorrect [BMS22]. The detailed explanation of fixed-point operations is available in [Naj14].

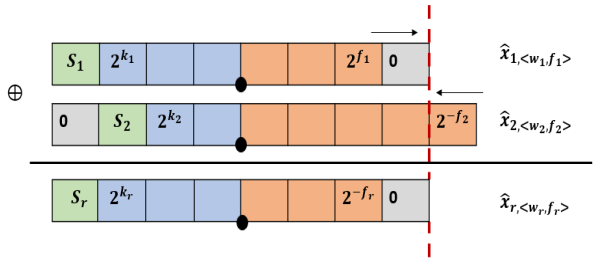


Figure 4: Addition of two fixed-point numbers.

Example 2.5. *In order to add two fixed-point numbers, they must have the same scaling. Adding* 2.25 (0010.0100) *and* 1.50 (0001.1000) *with word width of* 8 *bits,* 4 *integer part bits, and* 4 *fractional part bits, yields* 3.75 (0011.1100) *after binary addition as illustrated in Figure 5. If formats differ, the operand with fewer fractional bits must be shifted left. Overflow can occur if the result exceeds the representable range, requiring careful bit-width selection. Fixed-point addition is efficient but demands alignment and range checks [PU20a].*

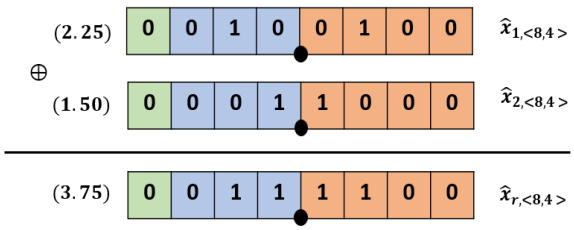


Figure 5: Example of fixed-point addition of two fixed point numbers in format of 8 bits with 4 bits to the fractional parts

Multiplication

Unlike addition and subtraction, multiplication in fixed-point arithmetics imposes no constraints on the formats of the multiplicands.

Let $\hat{x}_{1,\langle w_1,f_1\rangle}$ and $\hat{x}_{2,\langle w_2,f_2\rangle}$ be two fixed-point numbers , where w_1, w_2, f_1, f_2, k_1, and k_2 represent the word width , fractional bits and integer bits of $\hat{x}_{1,\langle w_1,f_1\rangle}$ and $\hat{x}_{2,\langle w_2,f_2\rangle}$ respectively.

We denote $\otimes$ the operator for multiplication in fixed point arithmetic, where $\hat{x}_{r,\langle w_r,f_r\rangle}$ represents the result of this multiplication, such that $\hat{x}_{r,\langle w_r,f_r\rangle} = \hat{x}_{1,\langle w_1,f_1\rangle} \otimes \hat{x}_{2,\langle w_2,f_2\rangle}$.

The number of bits of the fractional f_r and integer k_r parts of the result is defined in Definition 2.3

Definition 2.3. *The number of bits in the integer part of fixed-point multiplication is given by:*

$$k_r = k_1 + k_2 \tag{2.4}$$

And the number of bits in the fractional part of fixed-point multiplication is given by:

$$f_r = f_1 + f_2 \tag{2.5}$$

The figure 6 illustrates fixed-point multiplication in the required format (k_r, f_r). In this figure, 2^{k_1} represents the most significant bit (MSB) and 2^{f_1} the least significant bit (LSB) of $\hat{x}_{1,\langle w_1,f_1\rangle}$ (similarly for $\hat{x}_{2,\langle w_2,f_2\rangle}$).

The result of this multiplication is obtained by computing the binary sum of the intermediate partial products. such that:

- The first intermediate multiplication involves multiplying all bits of $\hat{x}_{1,\langle w_1,f_1\rangle}$ by the rightmost bit of $\hat{x}_{2,\langle w_2,f_2\rangle}$.
- Next, we compute the product of all bits of $\hat{x}_{1,\langle w_1,f_1\rangle}$ with the second bit (from the right) of $\hat{x}_{2,\langle w_2,f_2\rangle}$, then shift this result left by one position.

This process repeats until the last bit of $\hat{x}_{2,\langle w_2,f_2\rangle}$ is processed. the product's magnitude may exceed the representable range of the output format, leading to overflow, which can be mitigated through techniques like saturation (clamping to the maximum/minimum value) or by using extended-precision intermediate storage.

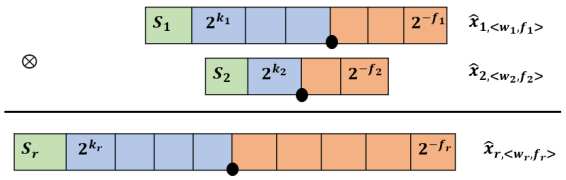


Figure 6: Multiplication of two fixed-point numbers [Naj14].

Example 2.6. *Consider the multiplication example illustrated in Figure 7, where two fixed-point numbers with identical formats are multiplied, having parameters* $f_1 = f_2 = 3$, $k_1 = k_2 = 5$, *and* $w_1 = w_2 = 8$. *The operation produces a fixed-point result* $\hat{x}_{r,\langle 16,6\rangle}$, *where the total number of bits in the result* 16 *equals the sum of the bit widths of both multiplicands, and the number of fractional bits* 6 *equals the sum of the fractional bits from both operands.*

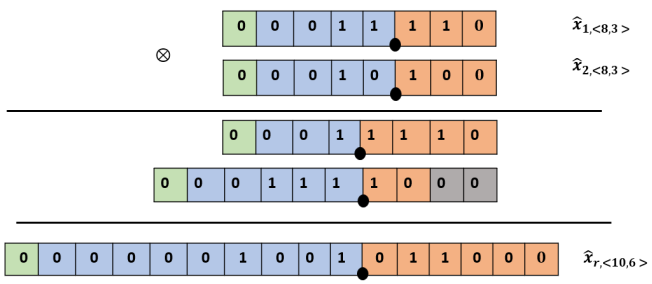


Figure 7: Multiplication of two fixed-point numbers [PU20a] in format of 16 bits with 3 bits of fractional parts

2.3.3 Rounding and Special Values

In fixed-point arithmetic, rounding and the handling of special values are essential to maintaining precision and stability, particularly in systems with limited computational resources such as embedded systems and digital signal processors (DSPs). Rounding becomes necessary when the result of an operation exceeds the number of fractional bits available. Several rounding modes are commonly used: truncation (round toward zero), round toward positive or negative infinity, and round to nearest. Each mode has trade-offs between computational simplicity and error minimization [Naj14, Ben22].

Unlike floating-point arithmetic, fixed-point formats do not support special values such as NaN (Not a Number), positive or negative infinity, or denormalized numbers. Instead, fixed-point systems handle extreme values through overflow and underflow behavior, often using either wrap-around (modulo) arithmetic or saturation. Saturation ensures that values beyond the representable range are clamped to the nearest maximum or minimum value, preventing wrap-around errors that can cause instability in control or signal processing systems [PU20a, PU20b, Lop14].

Additionally, some hardware and libraries implement status flags to indicate overflow or rounding events. Due to the absence of standardized special values, system designers must carefully define application-specific behaviors to handle exceptional cases robustly. This makes rounding and overflow handling not just numerical issues, but also critical design decisions in fixed-point arithmetic [ATD05].

2.4 Interval Arithmetic

Interval arithmetic, introduced by Moore [MY59], provides a framework for bounding numerical errors in computational processes. Unlike floating-point arithmetic, which approximates real numbers with single values, interval arithmetic represents variables as ranges, explicitly accounting for finite precision. In practice, these ranges are confined to intervals, and the computation rules ensure that the true result is always enclosed within the final interval [MKC09].

2.4.1 Representation of Interval Numbers

Definition 2.4 (Interval Number)**.** *Noting that the closed interval denoted by X is the set of real numbers given by*

$$X = [a, b] = \{x \in \mathbb{R}, a \leqslant x \leqslant b\} \tag{2.6}$$

Although various types of intervals, including open and half-open intervals, are prevalent in mathematical discourse, this work will primarily focus on closed intervals. The term "interval" will refer specifically to closed intervals.[MKC09]

Definition 2.5 (Equal intervals)**.** *Two intervals, $X = [a, b]$ and $Y = [c, d]$, s.t. $a, b, c, d \in \mathbb{R}$ are considered equal if they represent the same set. This occurs operationally when their corresponding*

endpoints are equal :

$$X = Y \implies a = c \ \wedge \ b = d \tag{2.7}$$

Definition 2.6 (Degenerate intervals). *A degenerate interval in interval arithmetic is a singular case where the interval collapses to a single real value, indicating no uncertainty or range in the possible value [MKC09]. We considered X as a degenerate interval :*

$$\forall\, a\,,\ b \in \mathbb{R}, \quad X = [a, b] \wedge a = b \tag{2.8}$$

In computations, degenerate intervals simplify operations: for instance, adding $[a, a]$ to $[b, c]$ yields $[a + b, a + c]$, while multiplying two degenerate intervals $[a, a]$ and $[d, d]$ produces another degenerate interval $[a{\times}d, a{\times}d]$. Their role is particularly significant in error tracking, as they allow mixed calculations where some terms are exact (degenerate) and others are uncertain (non-degenerate), enabling algorithms to optimize by reducing redundant range checks for known values.

2.4.2 Elementary Operations in Interval Arithmetic

Given two intervals $[a, b]$ and $[c, d]$, where $a, b, c, d \in \mathbb{R} \ \wedge \ a \leqslant b \wedge c \leqslant d$:

Definition 2.7 (Interval Addition). *We denote $+^{\#}$ the interval addition in interval arithmetics :*

$$[a, b] +^{\#} [c, d] = [a + c, b + d] \tag{2.9}$$

Definition 2.8 (Interval Subtraction). *We denote $-^{\#}$ the interval subtraction in interval arithmetics :*

$$[a, b] -^{\#} [c, d] = [a - d, b - c] \tag{2.10}$$

Definition 2.9 (Interval Multiplication). *We denote $\times^{\#}$ the interval subtraction in interval arithmetics :*

$$[a, b]{\times^{\#}}[c, d] = \big[\min(a{\times}c, a{\times}d, b{\times}c, b{\times}d), \max(a{\times}c, a{\times}d, b{\times}c, b{\times}d)\big] \tag{2.11}$$

min and max computes the minimum and maximum value between real numbers respectively.

Special Cases for Multiplication

- If $[a, b] \subseteq \mathbb{R}^+$ (both positive):

$$[a, b]{\times^{\#}}[c, d] = [a{\times}c, b{\times}d]$$

- If $[a, b] \subseteq \mathbb{R}^-$ (both negative):

$$[a, b]{\times^{\#}}[c, d] = [b{\times}d, a{\times}c]$$

An interval may suffer from inaccuracy due to dependency issues, a common problem in interval computations. This occurs because each occurrence of the same variable in a function is treated independently, leading to overly conservative (and often imprecise) results. For instance, take the function $f(x) = x - x$ with $x \in [-1, 1]$. Naive interval arithmetic produces $[-2, 2]$, while the correct result should be $[0, 0]$ [Mes02, Sou24].

2.4.3 Rational Arithmetic in Interval Arithmetic

Rational arithmetic provides a way to perform exact computations within interval arithmetic by representing numbers as fractions as defined in Definition 2.10.

Definition 2.10 (Rational Number). *let $\overline{x} \in \mathbb{Q}$,represent rational number where $\mathbb{Q}$ refers to the set of rational numbers , it defines as follow :*

$$\forall p, q \in \mathbb{Z}, \text{ and } q \neq 0 \quad \overline{x} = \frac{p}{q} \tag{2.12}$$

His approach avoids the rounding errors inherent in floating-point arithmetic, making it ideal for problems that require absolute precision, such as mathematical proofs or exact geometric algorithms.The operations performed with rational numbers in interval arithmetic are the same as those used for real numbers [VV07, N+61, Joh92].

In Example 2.7, we present a simple interval addition using rational numbers.

Example 2.7 (Addition of two Rational Intervals). *let $\overline{X}_1 = [\frac{1}{3}, \frac{1}{2}]$, and $\overline{X}_2 = [\frac{1}{6}, \frac{1}{4}]$ be two rational interval numbers s.t. :*

$$\overline{X}_1 +^{\#} \overline{X}_2 = [\frac{1}{3}, \frac{1}{2}] +^{\#} [\frac{1}{6}, \frac{1}{4}] = [\frac{1}{2}, \frac{3}{4}] \tag{2.13}$$

However, rational arithmetic has significant limitations, including rapidly growing denominators during computations, which slows down performance, and the inability to represent irrational numbers like π and $\sqrt{2}$ without approximation [AA21]. In contrast, floating-point interval arithmetic approximates real numbers using finite precision, enabling much faster computations while rigorously bounding errors through outward rounding. For instance, the same interval addition performed in floating-point would yield an approximate but efficiently computed result like $[0.5000, 0.7500]$. While floating-point intervals introduce controlled overestimation, they are practical for large-scale numerical simulations, where exact arithmetic would be prohibitively slow [Joh92].

Since we didn't use irrational numbers or computations in our experiments, this dissertation employs rational interval arithmetic instead of floating-point arithmetic to achieve more precise and accurate results.

2.5 Summary

In this chapter, we explore the foundational systems of computer arithmetic that enable numerical computation in digital systems. We begin with floating-point arithmetic, the industry standard for general-purpose computing. This method achieves a wider range through exponent-based scaling but introduces complexities such as rounding errors and special values (NaN, ±inf). Next, we examine fixed-point arithmetic, which represents numbers with a fixed number of integer and fractional bits. This approach offers deterministic performance, making it ideal for embedded and real-time systems, although it has a limited dynamic range. Finally, we analyze interval arithmetic, a rigorous method that represents values as ranges to provide guaranteed error bounds. While this is valuable for critical applications, it can lead to overestimation and increased computational overhead.

This chapter guides readers in selecting suitable arithmetic systems based on application requirements, balancing factors such as speed, accuracy, and reliability in modern computing environments.

Chapter 3

Overview of Neural Networks and Quantization

3.1 Introduction

Artificial neural networks (ANNs) have revolutionized machine learning, leading to significant advancements in image recognition, natural language processing, and autonomous systems. However, their high computational and memory requirements often limit deployment in resource-constrained environments. Quantization addresses this challenge by reducing the precision of weights and activations, thereby enhancing efficiency without a substantial loss in accuracy through fixed-point arithmetic. In this chapter, we will explore the principles of ANNs, their architectures, and activation functions in floating-point arithmetic. We will also examine the optimization methods and

techniques used to deploy ANNs in embedded systems. Finally, we will define quantized neural networks and activation functions in fixed-point arithmetic.

3.2 Artificial Neural Networks

3.2.1 Artificial Neural Network Definition and Composition

Neural networks (NNs) address problems such as classification, handwriting recognition, and object detection.

Artificial Neural Networks (ANNs) primarily consist of an input layer that receives incoming signals, hidden layers that perform intermediate computations, and an output layer that delivers the solution to the problem at hand. An ANN is composed of successive interconnected layers, with each layer being a set of neurons that have no connections between them [GK96].

A neuron takes as input a vector $X = (x_1, \ldots, x_n) \in \mathbb{R}^n$, synaptic weights $W = (w_1, \ldots, w_n) \in \mathbb{R}^n$, and a bias $b \in \mathbb{R}$ to calculate the weighted sum of the synaptic weights, given by $\sum_{i=1}^{n} w_i x_i + b$. This sum is referred to as the affine or pre-activation function. Subsequently, a function f , known as the activation function, is applied to this weighted sum. as illustrated in Figure 8.

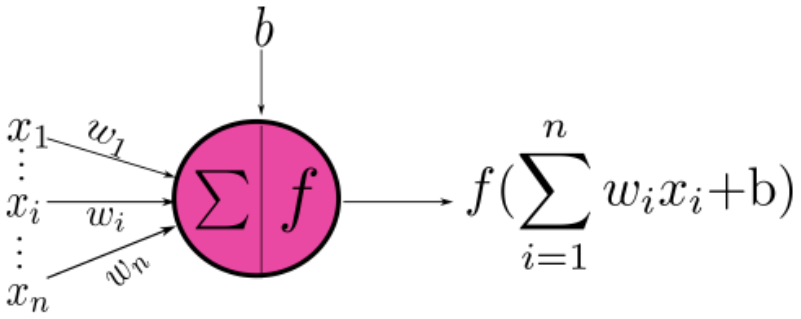


Figure 8: Architecture of a neurone in FCNN [Ben22]

Formally, we define a neural network as follows in Definition 3.1:

Definition 3.1 (Neuron). *In an artificial neural network (ANN) composed of $l \in \mathbb{Z}$ layers and $n \in \mathbb{Z}$ neurons per layer, the function that calculates the output of each neuron is defined by*

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$X \longmapsto u = f(X) = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Where X is the vector of inputs with elements x_i, b is the bias, w_i are the synaptic weights and u is the output of the neuron.

Neural networks are widely used in machine learning for tasks such as classification, regression, and function approximation due to their universal approximation capabilities [HSW89].

3.2.2 Architectures of Neural Networks

Fully Connected Neural Network (FCNN)

Fully connected neural network (FCNN) [SU15] are a type of artificial neural network ANN characterized by having multiple hidden layers. In these networks, connections between nodes do not form cycles, allowing information to flow in a single direction, from the input layer to the output layer. Each layer receives its inputs from the outputs of the previous layer. Figure 9 illustrates an example of FCNN architecture. The definition 3.2 summarizes the concept introduced above.

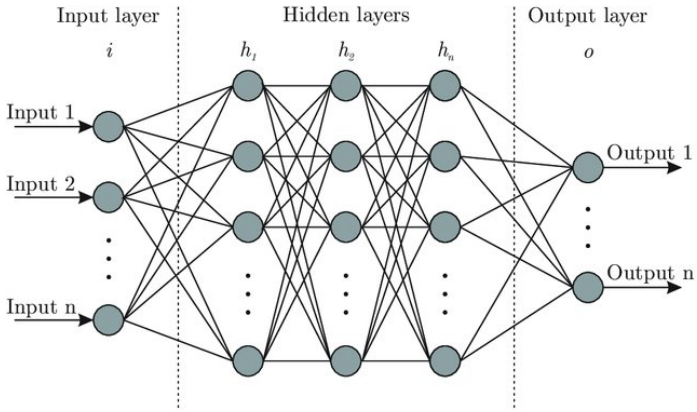


Figure 9: Architecture of FullyConnected Multilayer perceptron[BGF18]

Definition 3.2 (FCNN). *Let $X(x_1, \ldots, x_n) \in \mathbb{R}^n$ be the input vector to a FCNN where n and m represent the number of inputs and output of FCNN respectively. The output vector $Y(y_1, \ldots, y_n) \in \mathbb{R}^m$ is computed by applying successive affine transformations and activation functions across multiple layers. The computation for a single layer is given by:*

$$y_l = \sigma_L(W_L\sigma_{L-1}(W_{L-1}\ldots\sigma_1(W_1X + b_1) + b_{l_1}) + b_L) \tag{3.1}$$

where L is the number of layers, σ is the activation function at layer l, W_l and b_l are the weights and biases of layer l, and X is the input to the network.

While fully connected neural networks exhibit robust performance on tabular data [BLS+24], they face significant challenges when processing high-dimensional data like images [BDPM20] due to their dense connectivity architecture. The necessity for each neuron to connect to all neurons in subsequent layers leads to an excessive number of trainable parameters, resulting in computationally intensive training procedures and substantial memory requirements. These inherent limitations have motivated the development of specialized neural network architectures that are better suited for complex data structures.

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) share fundamental similarities with traditional Artificial Neural Networks (ANNs), consisting of self-optimizing neurons that perform operations like scalar products followed by nonlinear transformations [ON15]. Unlike FCNNs, CNNs process input data through successive layers to produce class scores, with the final layer containing class-specific loss functions. The key distinction lies in CNNs' specialization for image pattern recognition, where their architecture encodes image-specific features while reducing parameter counts compared to traditional ANNs [HUKK19]. This addresses a major limitation of standard ANNs, which struggle with the computational complexity of image data. For instance, even simple 2828 grayscale images like MNIST [Pei21] digits require 784 input weights per neuron in a fully-connected ANN, demonstrating why CNNs' parameter efficiency makes them preferable for image processing tasks [ON15].

CNNs are built upon three fundamental building blocks: convolutional layers for feature extraction, pooling layers for dimensionality reduction, and fully-connected layers for classification. The sequential arrangement of these layers forms a complete CNN architecture, as demonstrated in Figure 10 for the case of MNIST digit classification :

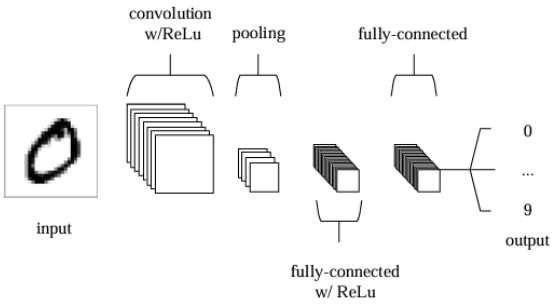


Figure 10: Simple CNN architecture, comprised of five layers [ON15]

- the input layer of a CNN directly receives and processes the raw pixel values of the input image. This layer serves as the entry point for visual data into the network's processing pipeline.

- **The convolutional layer** computes neuron activations by performing scalar products between weight kernels and inputs. These activations are then passed through a Rectified Linear Unit (ReLU), to introduce non-linearity, in contrast to traditional sigmoid functions. This combination of local connectivity and non-linear transformation enables effective feature extraction while maintaining spatial relationships in the input data [KSH12].

- The **pooling layer** perform downsampling on the spatial dimensions of the input, further reducing the number of parameters in that activation [ON15].

- The **Fully connected layers** perform the same functions as those in standard ANNs,

attempting to generate class scores from the activations for classification purposes. It is also suggested that ReLU may be used between these layers to enhance performance.

The success of CNNs in computer vision tasks stems from their ability to preserve spatial relationships while being robust to positional variations, achieving state-of-the-art performance in image classification, object detection, and semantic segmentation. Modern architectures like ResNet [Koo21] and EfficientNet [TL19] have further enhanced these capabilities through innovations such as residual connections and neural architecture search. We refer readers to "A Guide to Convolutional Neural Networks for Computer Vision" by Salman Hameed Khan et al. [KRSB18] for a detailed and formal overview of Convolutional Neural Networks.

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a specialized class of artificial neural networks designed for processing sequential data by maintaining an internal memory of previous inputs through recurrent connections. Unlike feedforward networks, RNNs process inputs sequentially while preserving a hidden state that captures temporal dependencies. This makes them particularly effective for tasks such as time-series analysis, natural language processing, and speech recognition [Sch19].

The recurrent architecture of these networks allows information to persist across time steps, enabling them to learn patterns in variable-length sequences [DTS+23].

As illustrated in Figure 11, a typical Recurrent Neural Network (RNN) consists of three fundamental layers:

- **Input layer** $^{(T)} = (^{(1)}, ^{(2)}, \ldots, ^{(T)})$: A sequence of input vectors at each time step.
- **Hidden layer** $^{(T)} = (^{(1)}, ^{(2)}, \ldots, ^{(T)})$: Stores relevant information from previous time steps up to the current step t.
- **Output layer** $^{(T)} = (^{(1)}, ^{(2)}, \ldots, ^{(T)})$: Computed using the current input $^{(t)}$ and previous hidden state $^{(t-1)}$.

The mathematical formulation of an RNN is given by:

$$^{(t)} = \sigma(\mathbb{W}_x{}^{(t)} + \mathbb{W}_h{}^{(t-1)} + _h) \tag{3.2}$$

$$^{(t)} = \sigma(\mathbb{W}_y{}^{(t)} + _y) \tag{3.3}$$

where $\mathbb{W}_x$ is the input-to-hidden weight matrix, $\mathbb{W}_h$ is the recurrent connection weight matrix, $\mathbb{W}_y$ is the hidden-to-output weight matrix, $_h$, $_y$ are bias vectors for hidden and output layers respectively and $\sigma(\cdot)$ denotes the activation function [DTS+23].

Standard RNNs often struggle with long-term dependencies due to vanishing or exploding gradients during backpropagation through time (BPTT). This limitation has led to

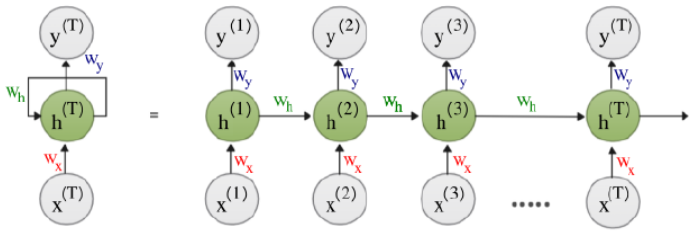


Figure 11: An example of the architecture of RNN [Sou24]

the development of advanced variants like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) [She20]. For more details on recuurent neural networks, or other types of ANNs, the reader can refer to [GBC16].

3.2.3 Activation Functions

Activation functions play a crucial role in neural networks by introducing non-linearity, but their implementation in floating-point arithmetic presents unique computational challenges. This section analyzes the numerical considerations, error propagation, and implementation trade-offs for common activation functions .

Figure 12 illustrates the graphical representation of the most commonly used activation functions in deep neural networks (DNNs).

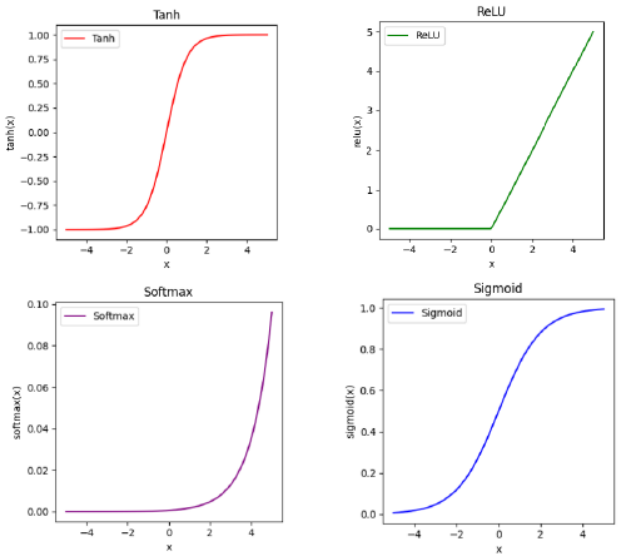


Figure 12: Visualization of the most common activation functions in DNNs

The ReLu Function

The rectified linear unit (ReLU) activation function [GBB11] is the most commonly used activation function in deep neural networks (DNNs). Its unique characteristic is that it activates only positive neurons, as defined in Definition 3.3.

Definition 3.3 (ReLU). *Let $a \in \mathbb{R}$, and ReLU activation function such that :*

$$\forall a \in \mathbb{R}, ReLu(a) = max(0, a) = \begin{cases} 0 & if\ a \leqslant 0 \\ a & if\ a > 0 \end{cases} \tag{3.4}$$

The Sigmoid Function

The sigmoid function [DBL12], referred to as "sig," is defined in Definition 3.4 as follows:

Definition 3.4 (Sigmoid).

$$\forall x \in \mathbb{R}\,,\ sig(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

The sigmoid function normalizes input data by producing output values within the range of $[0, 1]$. However, it has some limitations. When inputs are too large or too small, the gradient of the sigmoid function approaches zero, resulting in slow convergence rates during the training of deep neural networks. This issue is known as the vanishing gradient problem.

The Softmax Function

The Softmax function [Led21] is commonly used in classification problems, such as image categorization. It is defined in Definition 3.5 as follows:

Definition 3.5 (Softmax).

$$\forall x \in \mathbb{R} \quad softmax(x) = \frac{e^x}{\sum_{j=1}^{N} e^{x_j}} \tag{3.6}$$

It is primarily a method for normalizing the outputs of a neural network layer so that they sum to one. This normalization allows the output values to be interpreted as class probabilities, facilitating decision-making within a classification context. In other words, the Softmax function transforms the output values into probabilities, enhancing the intuitiveness of the predictions made by networks. This simplification is beneficial in various application domains, including computer vision and natural language processing.

The Tanh Function

The hyperbolic tangent function [GBB11, LBBH98], denoted as tanh, is defined in Definition 3.6 as follows:

Definition 3.6 (Tanh).

$$\forall x \in \mathbb{R}, \quad Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.7}$$

This function normalizes input values to produce outputs in the range of [-1, 1]. Unlike the sigmoid function, which outputs values between 0 and 1, the tanh function is particularly beneficial in binary classification models. It effectively handles significant variations and ensures that data points are well separated, capturing much of their meaning.

3.3 AI-Based Controllers

Autonomous systems, including self-driving vehicles, drones, and industrial robots, depend significantly on advanced control mechanisms to operate efficiently and safely in dynamic environments. Traditional control systems, such as Proportional-Integral-Derivative (PID) controllers, often face challenges in handling complex, nonlinear, and uncertain real-world conditions. Artificial Intelligence (AI) has emerged as a transformative solution, facilitating adaptive, data-driven control strategies that improve performance, robustness, and autonomy.

3.3.1 Neural Network-Based Controllers Types

The integration of Artificial Intelligence (AI) into closed-loop control systems (CLCS) introduces innovative approaches to modeling, tuning, and control. Below are the major types of AI-based controllers, each addressing specific challenges in control system engineering :

AI-Based Process Modeling [SRP22]

AI-based process modeling leverages machine learning algorithms like Artificial Neural Networks (ANNs), including feedforward, recurrent (RNNs/LSTMs)[She20], and convolutional (CNNs) networks [D+14], to create data-driven models that approximate complex system dynamics from sensor data, bypassing traditional physics-based modeling [San12]. Techniques such as Physics-Informed Neural Networks (PINNs) [CMW+21] integrate known physical laws (e.g., differential equations) into training to ensure plausibility, while Fourier Neural Operators (FNOs) [LKA+20] efficiently handle partial differential equations (PDEs) in systems like fluid dynamics. Gaussian Processes (GPs) [See04] offer probabilistic predictions with uncertainty bounds, useful for safety-critical applications, and simpler methods like SVMs or Random Forests provide interpretable alternatives for fault detection. Training relies on high-quality, diverse datasets, often enhanced by transfer learning (pre-training on simulations) or reinforcement learning (RL) for adaptive tuning [KHL+12]. However, challenges include data bias (missing edge cases), computational costs, and the "black-box" nature of deep learning, which complicates safety certification. Hybrid approaches combining AI with physics-based models are increasingly adopted in autonomous vehicles, industrial robotics, and energy systems to balance accuracy, adaptability, and reliability. Future advancements aim for real-time learning and explainable AI to enable broader deployment in critical applications [SRP22].

Full AI-Based Controllers [SRP22]

In this paradigm, ANNs completely replace conventional controllers by directly mapping reference values and sensor inputs to control actions. These controllers excel in handling complex, nonlinear, and multivariable systems—like autonomous vehicles or industrial

robots—by leveraging techniques such as deep reinforcement learning (DRL) for adaptive decision-making and recurrent neural networks (RNNs) for time-series dependencies. These controllers are particularly effective for managing multivariable, nonlinear systems and can identify complex relationships without needing explicit mathematical models. However, training these networks necessitates extensive, unbiased datasets to encompass all operational scenarios. While they offer rapid inference times (less than 10 ms on GPUs and FPGAs), their "black-box" nature raises concerns regarding explainability and safety, as their behavior cannot be fully validated across all edge cases [YG14].

Hybrid Safety-Critical Controllers [SRP22]

To mitigate risks, hybrid designs integrate AI with traditional control methods:

- **Fallback Controllers [SSSH17]**: An AI-based controller operates alongside a conventional (rule-based or model-predictive) controller, with a safety monitor that continuously checks AI outputs against predefined stability thresholds (e.g., acceleration limits in autonomous vehicles). If the AI's decisions deviate from safe bounds—such as suggesting an aggressive maneuver that could lead to a collision—a fail-safe switch automatically reverts control to the fallback system (e.g., a PID or MPC-based controller).

- **Bounded AI Influence [RM19]** : Hybrid control systems combine conventional controllers (e.g., PID, MPC) for stable, safety-compliant baseline performance with AI components (e.g., deep reinforcement learning) that dynamically optimize outputs within predefined limits—enforced via runtime monitoring, constraint layers, or formal verification. This approach is critical in safety-sensitive domains like autonomous vehicles (adaptive path planning within collision-avoidance bounds), industrial robots (precision tuning without exceeding mechanical limits), and medical systems (AI-assisted surgery with fail-safe controls). Emerging techniques, such as run-time assurance (RTA) and neurosymbolic integration, further enhance reliability, ensuring these systems balance AI's adaptability with rigorous safety guarantees. As autonomy advances, hybrid architectures will remain essential for merging innovation with trustworthiness.

  These hybrid approaches are critical for autonomous vehicles, industrial robots, and medical systems, where safety and reliability are paramount.

3.3.2 Compression Techniques of NNs for Deployment on Embedded Systems

Neural network compression refers to a collection of techniques designed to reduce the size, computational cost, and memory footprint of deep learning models while preserving their accuracy as much as possible. These methods are crucial for deploying models on resource-constrained devices such as smartphones, microcontrollers (e.g., Arduino, STM32), and edge AI chips [MPMF23, CWZZ17, DdSJCC24].

Pruning

Pruning removes redundant or less important weights from a neural network, resulting in a sparse model. Unstructured pruning eliminates individual weights (e.g., by setting small weights to zero), while structured pruning removes entire neurons, channels, or layers to achieve hardware-friendly sparsity. Techniques for pruning include magnitude pruning, which eliminates weights closest to zero, and the Lottery Ticket Hypothesis, which identifies sparse subnetworks that can be retrained to full accuracy [HMD15, KNvB+23].

Pruning can reduce model size by up to 90% and improve inference speed, but it requires sparsity-aware hardware (e.g., NVIDIA's Ampere GPUs) to realize these benefits. A key limitation is that unstructured sparsity often necessitates specialized libraries (e.g., TensorFlow's sparse kernels) for deployment. Furthermore, pruning is particularly beneficial for high-performance edge devices where memory and computational resources are constrained, though not as severely as in microcontrollers [Ber21, DdSJCC24, CWZZ17, MPMF23].

Low-Rank Factorization

Low-rank factorization is a highly effective technique for reducing the size and complexity of neural networks NNs without compromising their performance. This approach entails decomposing large, dense weight matrices typically found in deep neural networks (DNN) into two smaller, lower-rank matrices [HMD15, MPMF23]. The fundamental principle of low-rank factorization lies in its ability to combine these two resulting data matrices to approximate the original, thereby facilitating compression. This method not only reduces the model size but also mitigates data processing requirements, enhancing the suitability of convolutional neural networks (CNN) for deployment in resource-constrained environments [CWZZ17]. The objective of this process is to capture the most significant information contained within the network's weights, enabling a more compact representation with minimal performance degradation [Ber21, DdSJCC24].

Matrix decomposition and tensor decomposition are commonly applied techniques in this context. Matrix decomposition in low-rank factorization involves breaking down large weight matrices into simpler matrix forms. A prevalent method is singular value decomposition (SVD), which decomposes a matrix into three smaller matrices, capturing the essential features of the original matrix. This process reduces the number of parameters in DNN models, leading to decreased storage and computational requirements while striving to maintain model performance [DdSJCC24]. Beyond matrix decomposition, tensor decomposition addresses multidimensional arrays (tensors) in DNNs.

Knowledge Distillation

Knowledge distillation is primarily utilized in the field of deep neural networks (DNNs) for model compression and optimization. This process involves transferring knowledge

from a large-scale model, known as the teacher, to a smaller-scale model, referred to as the student. By doing so, it enhances the student's performance without requiring the computational resources of the larger model. At its core, knowledge distillation seeks to extract the informative aspects of the teacher model's behavior and instill this knowledge into the student model. This approach allows the student model to maintain high accuracy while significantly reducing its size and complexity [Ber21, DdSJCC24].

In a teacher-student model framework, a large-scale, well-trained model guides the implementation of a smaller-scale model. The smaller model aims to replicate the outputs of the larger model while utilizing fewer parameters and reducing computational complexity. It is optimized to accurately infer the correct categorization and reproduce the outputs of the large-scale model, including both predictions and intermediate features. The small-scale model can be trained to align with either the softmax output or the feature representations of the large-scale model [GYMT21].

Knowledge distillation is recognized for its ability to encapsulate the functionalities of larger models into formats that are suitable for deployment in resource-constrained environments. However, it presents several complex challenges and drawbacks. A significant concern is the deployment of large pre-trained language models on devices with limited memory, which requires a careful balance to optimize performance while managing system resources effectively . Moreover, the generalization capability of distilled models may be limited when using public datasets that differ from the training datasets, potentially diminishing the model's relevance and accuracy [CH19, MPMF23, DdSJCC24].

Binarization

Binarization of neural networks involves reducing weights and/or activations to binary values (e.g., -1 and +1) to enhance efficiency [HMD15]. This technique, part of neural network quantization, includes weight binarization, activation binarization, or full binarization, with methods like deterministic (sign function), stochastic (probability-based), and XNOR-Net (scaled binary weights). Binarization offers significant memory savings (1/32 of 32-bit weights), faster binary operations (XNOR/popcount), and lower energy consumption, making it ideal for edge devices, mobile applications, and specialized hardware. However, challenges like accuracy loss and training instability are addressed through techniques such as the straight-through estimator (STE) and gradient clipping [Ber21, DdSJCC24].

In this thesis, we applied quantization as an optimization method for neural networks deployed on intelligent controllers.

3.4 Quantized Neural Network

The research of quantized neural networks has attracted a lot of attention from the deep learning community [CBD15, RORF16, Guo18, HPP24, LJVD23a]. Achieving this goal requires quantizing floating-point neural network models into integer neural networks.

With quantized neural networks, we can use bitwise operations instead of floating-point operations for forward and backpropagation thereby saving energy.

The computational operations in these networks primarily involve bitwise operations performed by the arithmetic logic unit (ALU). An ALU consumes significantly less energy than a floating-point unit (FPU). Therefore, for mobile applications where power consumption is crucial, a quantized neural network is preferable to its full-precision counterpart [NFA+]. Neural networks are typically trained using FP32 weights and activations. When performing inference in FP32, the processing elementary operations must support floating-point logic, and 32-bit data must be transferred from memory to the processing units. Both elementary operations and data transfer account for the majority of the energy consumed during neural network inference. Therefore, using a lower-bit fixed-point or quantized representation can yield significant benefits. Low-bit fixed-point representations, such as INT8, not only reduce the amount of data transferred but also decrease the size and energy consumption of elementary operations [Hor14]. This efficiency arises because the cost of digital arithmetic typically scales linearly to quadratically with the number of bits used, and fixed-point addition is more efficient than floating-point addition. [Hor14, AGO+13, FR17].

The key issue in quantization is designing an appropriate quantization mapping function and an effective method for calculating quantization parameters. In uniform quantization, most existing approaches utilize either asymmetric or symmetric quantization mapping functions [GKD+22]. The asymmetric quantization mapping function is defined as follows:

$$x = Q(\hat{x}) = s.\hat{x} + D \tag{3.8}$$

$$\hat{x} = Q^{-1}(x) = round(\frac{x - D}{s}) \tag{3.9}$$

$$s = 2^{f} \tag{3.10}$$

Where Q and Q^{-1} are the quantization mapping function, Q^{-1} is the inverse function of Q, *round* is the rounding operation, x is the floating-point real value, $\hat{x}$ is the integer value after quantization.

s and D are quantization parameters s is the scaling factor, and f represents the number of fractional bits. Power-of-two quantization is a special case of symmetric quantization, where the scaling factor is limited to a power of two. This choice can enhance hardware efficiency, as scaling with s corresponds to simple bit-shifting operations.

D is the zero point, chosen such that the 0 value would exactly map to quantized values. Symmetric quantization is a simplified version of the general asymmetric case [NFA+]. The symmetric quantizer restricts the quantization parameter D to 0 [LDW19].

Definition 3.7 (Quantized neuron in QNNS)**.** *We define in Eq. (3.12) the corresponding formula for the i-th quantized Fully-Connected layer with precision of $\langle w, f \rangle$, where $w, f \in \mathbb{N}$ represents the*

word width bits and fractional bits respectively:

$$\hat{X}_i^{<w,f>} = \hat{b} + \hat{\sigma}_i^{<w,f>}(\hat{W}_i^{<w,f>}\hat{X}_{i-1}^{<w,f>} \tag{3.11}$$

$$\hat{X}_i^{<w,f>} = \hat{b} + \hat{\sigma}_i^{<w,f>}((W_i\, s_w)(X_{i-1}\, s_x)) \tag{3.12}$$

Where $\hat{X}_{i-1}^{<w,f>}$, $\hat{W}_i^{<w,f>}$, $\hat{b}_i^{<w,f>}$, *and* $\hat{\sigma}_i^{<w,f>}$ *represent quantized Input layer, weights, bias, and activation functions respectively in fixed arithmetics through a specified format* $< w, f >$; W_i *and* X_{i-1} *indicates the floating-point weights and inputs of the previous layer respectively.* s_w *and* s_x *represent the scale factors of inputs and weights [ZWW+17, NFA+].*

Example 3.1. *Figure 13 illustrates a representation of a neuron using floating-point arithmetic and its quantized version in fixed-point arithmetic, formatted with 8 bits for the word width and 5 bits for the fractional part.*

Let $X = (x_1, x_2, x_3, x_4) = (0.71, 0.81, 0.67, 0.47)$ *be floating point input vector ,* $W = (0.03, 0.02, 0.04, 0.1)$ *be floating point weight vector , and floating point bias* $b = 0.53$. *The output of the floating point neuron* $y_1 = \sum X \cdot W + b = 0.64$

Quantized input vector $\hat{X} = round(X \cdot 2^f) = (23, 26, 21, 15)$, *Quantized weight vector* $\hat{W} = round(W \cdot 2^f) = (1, 1, 1, 3)$ *and quantized bias* $\hat{b} = round(b \cdot 2^f) = 17$ *The output of the quantized neuron* $\hat{y} = \sum \hat{X} \cdot \hat{W} + \hat{b} = 132$

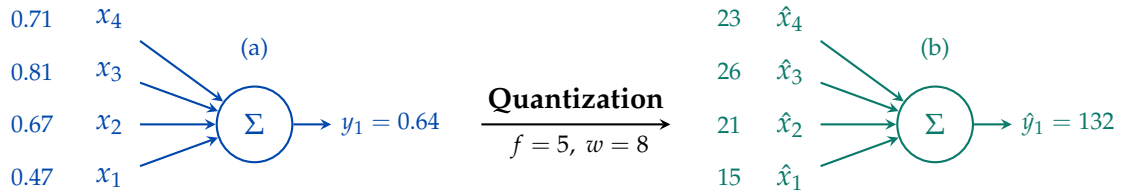


Figure 13: (a): Representation of a neurone in Real arithmetics , (b) : Representation of a neuron of quantized neural network according to a word width of 8 bits and 5 fractional bits.

3.5 Summary

In this chapter, we explored the fundamental concepts of Neural Networks (NNs) and Quantized Neural Networks (QNNs) for embedded systems-based intelligent controllers. In §3.2.2 and §3.2.3, we examined Neural Network architectures and common activation functions. Section §3.3.2 discusses various compression techniques for Artificial Neural Networks (ANNs) in embedded systems. Among the several compression methods, QNNs have emerged as a solution, leveraging reduced precision to enable faster inference and lower energy consumption. We defined QNNs and their activation functions in §4.4.

Ultimately, mastering both NNs and QNNs is essential for building efficient, scalable, and intelligent systems. This chapter lays the foundation for understanding their role in the future of AI-driven technologies.

In the next chapter, we will present common formal methods used for the verification of Neural Networks, along with several works and tools that verify ANNs and QNNs.

Chapter 4

Formal Verification of ANNs and QNNs

4.1 Introduction

Formal verification of quantized neural networks (QNNs) provides mathematical guarantees about their behavior by exhaustively analyzing all possible inputs within specified bounds. Unlike empirical testing, it handles the discrete, non-linear nature of low-bit computations through techniques like SMT solvers and abstract interpretation. Challenges include managing quantization errors, hardware-specific rounding modes, and scaling to large networks. This approach is crucial for safety-critical applications where robustness must be proven, such as autonomous systems. While promising, it faces open

problems in verifying advanced quantization schemes efficiently. In this chapter we will describe a common verification methods and we will describe formal tools and methods for the verification of ANNs as well as QNNs

4.2 Common Formal Verification Methods

Formal methods [MBT+22, BES16, KDM+17, FCAF17, WDF+14, WLBF09] are mathematically rigorous techniques used to specify, design, and verify software and hardware systems, ensuring their correctness and reliability. Common approaches include model checking (automated verification against temporal logic properties), theorem proving (interactive proof development), abstract interpretation (static analysis for approximating program behavior), and SAT/SMT solvers like Z3, which facilitate automated reasoning. These techniques are essential in safety-critical domains such as aerospace, cybersecurity, and hardware design, where errors can lead to severe consequences. Although some methods require expert input, advances in automation are enhancing their accessibility and adoption.
Before delving into common formal verification methods, let us clarify what verification means by focusing on two key principal properties:

1. *Soundness* [Bun19]: A verification algorithm is considered sound if it only accepts properties that are actually true. An unsound verification algorithm may incorrectly classify properties for which counterexamples exist as true.

2. *Completeness*[Bun19]: A verification algorithm is considered complete if it can prove any true property. This is a stringent requirement, as some properties may require nearly exhaustive enumeration for proof. Most complete methods tend to be computationally intensive. In contrast, incomplete but sound methods can prove some true properties, but not all. As a result, false properties, along with some true properties that the algorithm cannot prove due to its limitations, will remain in an unknown status.

4.2.1 Model Checking

Model checking [BK08] is a verification technique that systematically explores all possible system states in a brute-force manner. A model checker [CES09], which is the software tool that performs this verification, examines each potential system scenario to confirm that a given model meets specific properties. One of the significant challenges is to analyze the largest possible state spaces manageable with current hardware, such as processors and memory. State-of-the-art model checkers [BBF+13, GL94, CCD+14] can handle state spaces ranging from approximately 10^8 to 10^9 states through explicit state-space enumeration. By employing advanced algorithms and specialized data structures, larger state spaces—from 10^{20} to even 10^{476} states—can be addressed for particular problems. Furthermore, subtle

errors that may go undetected through emulation, testing, and simulation can potentially be revealed using model checking [Bie21, SdBG+19].

The system model is typically generated automatically from a model description specified in a suitable programming language, such as C or Java, or in hardware description languages . It's important to note that the property specification defines what the system should and should not do, while the model description outlines how the system behaves. The model checker evaluates all relevant system states to determine whether they satisfy the desired properties [BK08]. If it encounters a state that violates a property, the model checker provides a counterexample that illustrates how the model reached this undesired state. This counterexample outlines an execution path from the initial system state to the state that violates the property being verified. Using a simulator, the user can replay the violating scenario, thereby obtaining valuable debugging information and adjusting the model or the property as needed, as illustrated in Figure 14.

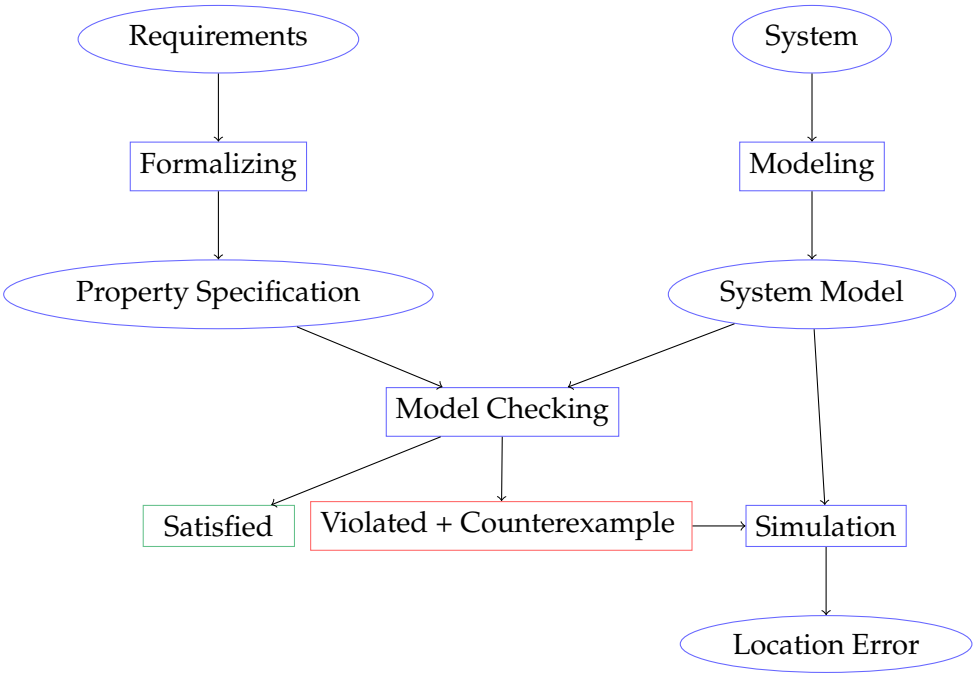


Figure 14: Schematic view of the model-checking approach [BK08]

Example 4.1 (Concurrency and Atomicity [BK08])**.** *We aim to analyze the following concurrent program, in which three processes:* **INC**, **DEC**, *and* **RESET**, *cooperate. They operate on a shared integer variable, x, which has an arbitrary initial value and can be accessed (read) and modified (written) by each of the individual processes. The processes are :*

$$\text{PROC } \textbf{INC} = \quad \textbf{\textit{while}}\ true\ \textbf{\textit{do if}}\ x < 200\ \textbf{\textit{then}}\ x := x{+}1\ \textbf{\textit{fi od}}$$

$$\text{PROC } \textbf{DEC} = \quad \textbf{\textit{while}}\ true\ \textbf{\textit{do if}}\ x < 0\ \textbf{\textit{then}}\ x := x{-}1\ \textbf{\textit{fi od}}$$

$$\text{PROC } \textbf{RESET} = \quad \textbf{\textit{while}}\ true\ \textbf{\textit{do if}}\ x = 200\ \textbf{\textit{then}}\ x := 0\ \textbf{\textit{fi od}}$$

Process **INC** *increments x if its value is less than 200,* **DEC** *decrements x if its value is at least 1, and* **RESET** *resets x when it reaches 200. These actions occur repetitively.*

Example 4.1 examines whether the variable x consistently remains within the range [0, 200]. At first glance, this appears to be true; however, a closer analysis reveals a flaw: when x = 200, Process Dec checks if x > 0 (which passes), then Process Reset sets x to 0, and finally, Process Dec decrements x to -1. This scenario occurs because the operations are not atomic—they execute in separate steps, contrary to the intuitive assumption of atomicity. Consequently, x can fall outside the expected bounds.

Model Checking Process

In the application of model checking to a design, several distinct phases can be identified:

- **Modeling phase** The prerequisite inputs for model checking are a model of the system being analyzed and a formal specification of the property to be verified.

  System models provide precise behavioral descriptions, typically represented as finite-state automata [CL89] with states (capturing variable values, program counters, etc.) and transitions (defining state evolution). For practical systems, these are implemented using modeling languages (e.g., C, Java, or VHDL extensions), with concurrent systems requiring particularly careful abstraction. Simulation [Bal95] helps detect basic modeling errors early, reducing later verification costs. Rigorous verification relies on unambiguous property specification, often using temporal logic [Koy90, Roz11], an extension of propositional logic with time-based operators, to formalize critical properties like functional correctness, safety ("no bad events"), liveness ("good events eventually occur"), and real-time constraints. Model checking validates whether the system satisfies these temporal logic formulas.

  Verification involves checking that the design meets the identified requirements; in other words, *verification* is about ensuring that "we are building the thing right". *Validation*, on the other hand, assesses whether the formal model aligns with the informal conception of the design; thus, *validation* ensures that "we are verifying the right thing" [BK08].

- **Running phase** The model checker must first be initialized by appropriately setting various options and directives for exhaustive verification. Following this, the actual model checking occurs. This process is fundamentally algorithmic, involving the verification of the property's validity across all states of the system model [CLM89].

- **Analysis phase** There are essentially three possible outcomes: the specified property is either valid in the given model, it is not valid, or the model is too large to fit within the computer's physical memory limits [ABM98].

1. In case the property is valid, the following property can be checked, or, in case all properties have been checked, the model is concluded to possess all desired properties.
2. Whenever a property is falsified, the negative result may have different causes. There may be a modeling error, i.e., upon studying the error it is discovered that the model does not reflect the design of the system. This implies a correction of the model, and verification has to be restarted with the improved model.
3. When dealing with models too large for available memory, several strategies can be employed: symbolic techniques like binary decision diagrams [WBCG00] and partial order reduction [Min99] exploit structural regularities; rigorous abstractions preserve key properties while reducing model size, sometimes requiring different abstractions for different properties; and probabilistic verification trades full precision for partial state-space exploration, often with negligible accuracy loss - with the most effective reduction methods [AP18].

4.2.2 Static Analysis by Abstract Interpretation

Static program analysis [Hen14, Wög05] aims to automatically determine whether a program satisfies specific properties, such as "the program never dereferences a null pointer," "the program never divides by zero," and "user-specified assertions are never violated." Abstract interpretation [Hen14, CC77, Cou21] provides the mathematical foundation for designing such analyses. Developed by Patrick Cousot and Radhia Cousot in the late 1970s, it offers a unifying framework for static program analysis by approximating the behavior of programs.

Abstract interpretation involves replacing a precise element of a program with a less detailed abstraction in order to infer the program's properties [ea17, CC92]. This process leads to a loss of certain safe information, which can limit the ability to draw conclusions about all programs. The abstract interpretation of a program can help identify runtime errors, such as division by zero and overflow, as well as issues related to access to shared variables and dead code [Bou17].

Figure 15 illustrates the main principle of the approach: the set R represents the non-computable set of reachable states. Abstract interpretation aims to over-approximate this set with a simpler set A that includes R. Set A should be sufficiently precise to demonstrate the absence of bugs; specifically, the intersection of A with the set of error states $E2$ is empty, indicating that every state in $E2$ is proven to be unreachable. In contrast, A intersects with the set $E1$, while R does not, suggesting that the over-approximation A is too coarse to establish the unreachability of the errors in $E1$ [Hen14].

First, we need to introduce standard definitions needed for describing abstract domain.

Definition 4.1 (Partially ordered set)**.** *A Partially ordered set or poset $\mathcal{P}$ equiped with partial order relation $(\mathcal{P}, \sqsubseteq)$, where $\sqsubseteq$ is:*

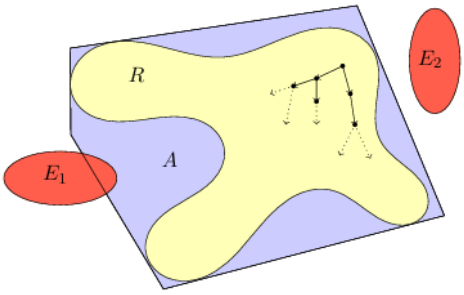


Figure 15: Abstract Interpretation computes over-approximation of the set R. The errors in $E2$ are proved unreachable, while the non-empty intersection of A with $E1$ raises a false alarm [Hen14].

- *Reflexive if* $\forall X \in \mathcal{P}, X \sqsubseteq X$
- *Transitive if* $\forall X, Y, Z \in \mathcal{P}, X \sqsubseteq Z \wedge Z \sqsubseteq Y \implies X \sqsubseteq Y$
- *Antisymmetric if* $\forall X, Y \in \mathcal{P}, X \sqsubseteq Y \wedge Y \sqsubseteq X \implies X = Y$

Definition 4.2 (Lower and Upper Bounds). *Let $\mathcal{P}'$ be a subset of poset $(\mathcal{P}', \sqsubseteq)$. An element $u \in \mathcal{P}$ is an upper bound of $\mathcal{P}' \iff \forall a \in \mathcal{P}'.\ a \sqsubseteq u$.*

Similarly, An element $l \in \mathcal{P}$ is an lower bound of $\mathcal{P}' \iff \forall a \in \mathcal{P}', l \sqsubseteq a$.

A least upper bound $\sqcup$ of a subset $\mathcal{P}'$ is an upper bound of $\mathcal{P}'$ that is less than or equal to any other upper bound of $\mathcal{P}'$. This is often represented as $\sqcup\mathcal{P}'$. Conversely, a greatest lower bound of a subset $\mathcal{P}'$ is a lower bound $\sqcap$ of $\mathcal{P}'$ that is greater than or equal to any other lower bound of $\mathcal{P}'$. This is often represented as $\sqcap\mathcal{P}'$

Definition 4.3 (Latice). *A complete lattice $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap)$ is a poset in which every pair of element $x, y \in \mathcal{P}$ has a least upper bound $x \sqcup y$ and great upper bound $x \sqcap y$*

Definition 4.4 (Monotonic Function). *Consider two posets $(\mathcal{P}_1, \sqsubseteq_1)$ and $(\mathcal{P}_2, \sqsubseteq_2)$. A function $f : \mathcal{P}_1 \to \mathcal{P}_2$ is called monotonic if it satisfies the following condition:*

$$\forall x, y \in \mathcal{P}_1, x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y) \tag{4.1}$$

Definition 4.5 (Galois Connexion). *Let $(\mathcal{P}, \sqsubseteq)$ and $(\mathcal{P}^\#, \sqsubseteq^\#)$ two complete lattices, connected by two functions: α and γ, such that :*

- $\alpha : X \to X^\#$ *is monotonic,*
- $\gamma : X^\# \to X$ *is monotonic,*
- $\forall x \in X, \forall x^\# \in X^\#, \alpha(x) \sqsubseteq^\# x^\# \iff x \sqsubseteq \gamma(x^\#)$

Function α is called the abstraction function, and γ is the concretization function. A concrete value $x \in X$ can be abstracted by $\alpha(x)$(which is the best abstraction for x), whose concretization includes $x : x \sqsubseteq \gamma \circ \alpha(x)$. Alternatively, an abstract value $x^\#$ may represent a concrete value $\gamma(x^\#$, whose abstraction is included in $x^\# : \alpha \circ \gamma(x^\#) \sqsubseteq^\# x^\#$.

Definition 4.6 (Concretization-based Abstract Domain)**.** *Let $\mathcal{P}$ be an ordered set of elements to be abstracted. X is called concrete domain. We assume that $(X, \sqsubseteq, \sqcup, \sqcap)$ is a lattice, where $\sqsubseteq$ is the partial order, and $\sqcup$ and $\sqcap$ are respectively the join and meet operators. An abstract domain over the concrete domain X is a pair $(X^\#, \gamma)$, where γ is a function from $(X^\#$ to X called concretization function, and $(X^\#, \sqsubseteq^\#, \sqcup^\#, \sqcap^\#)$ is a lattice satisfying the following properties:*

- *$\sqsubseteq^\#$ is a sound approximation of $\sqsubseteq$:*

$$\forall x^\#, y^\# \in X^\#,\ x^\# \sqsubseteq^\# y^\# \rightarrow \gamma(x^\#) \sqsubseteq \gamma(y^\#) \tag{4.2}$$

  In other words, we can also say that the concretization function γ is monotonic.

- *The abstract join operator $\sqcup^\# : X^\# \times X^\# \rightarrow X^\#$ is a sound abstraction of $\sqcup : X \times X \rightarrow X$*

$$\forall x^\#, y^\# \in X^\#,\ \gamma(x^\#) \sqcup \gamma(y^\#) \sqsubseteq \gamma(x^\# \sqcup^\# y^\#) \tag{4.3}$$

- *Similarly, the abstract meet operator $\sqcap^\# : X^\# \times X^\# \rightarrow X^\#$ abstracts $\sqcap : X \times X \rightarrow X$ so that:*

$$\forall x^\#, y^\# \in X^\#,\ \gamma(x^\#) \sqcup \gamma(y^\#) \sqsubseteq \gamma(x^\# \sqcup^\# y^\#) \tag{4.4}$$

  An element $x^\# \in X^\#$ represents an element $\gamma(x^\#) \in X$, and is a sound abstraction of any x such that $x \sqsubseteq \gamma(x^\#)$. Intuitively, we can say that $x^\#$ is more precise than $y^\#$ if $x^\# \sqsubseteq^\# y^\#$, since it represents a set which is smaller.

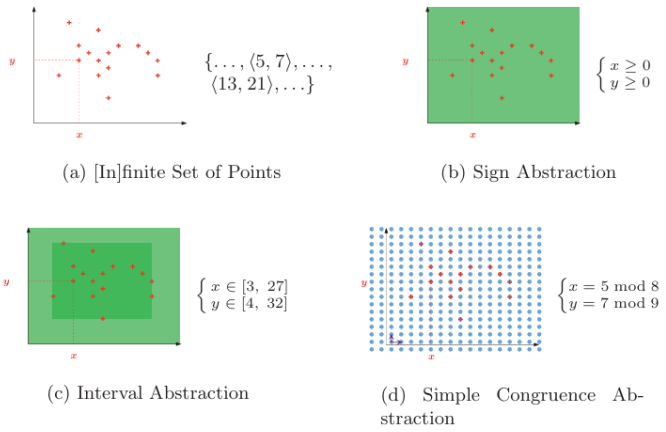
Numerical Abstract Domains

Assume a program has two integer variables, X and Y. We can further abstract by only considering the values x and y of these variables in a given state. This abstraction reduces the trace semantics to a set of points (pairs of values), as illustrated in the plane in Figure 16(a). We will now informally illustrate several effective abstractions of an infinite set of points.

- **Non-relational Abstractions:** The non-relational, attribute-independent, or Cartesian abstractions [CC79] involve ignoring the potential relationships between the values of the X and Y variables. Consequently, a set of pairs is approximated through projection as a pair of sets. Each of these sets may still be infinite and, in general, not exactly computable, necessitating further abstractions.

  The sign abstraction [CC79] , illustrated in Figure 16(b), represents set of integers with their signs, thereby disregarding their absolute values. The interval abstraction [CC76] , shown in Figure 16(c), offers a more precise approximation, representing a set of integers by its minimal and maximal values, including $-\infty$ and $+\infty$, as well as the empty set if necessary.

  The congruence abstraction [Gra89], which generalizes the parity abstraction [CC79], is not directly comparable, as demonstrated in Figure 16(d).

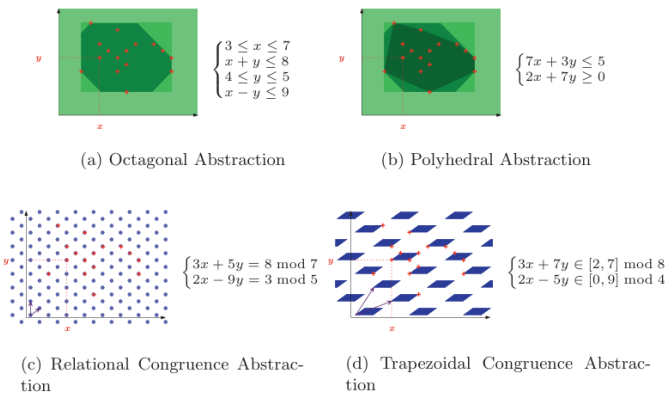


(a) [In]finite Set of Points (b) Sign Abstraction

(c) Interval Abstraction (d) Simple Congruence Abstraction

Figure 16: Non-relational Abstractions [Cou01]

- **Relational Abstractions:** Relational abstractions are more precise than non-relational ones in that some of the relationships between values of the program states are preserved by the abstraction. For example, the polyhedral abstraction[SPV17] illustrated in Figure 17(b) approximates a set of integers by its convex hull. Only non-linear relationships between the values of the program variables are abstracted. The use of an octagonal abstraction[Min06], illustrated in Figure 17(a), is less precise, as it retains only some shapes of polyhedra, or equivalently, considers only linear relations between any two variables with coefficients of $+1$ or -1 (of the form $\pm x \pm y \leqslant c$, where c is an integer constant). A non-comparable relational abstraction is the linear congruence abstraction [BTV03], as illustrated in Figure 17(c). The trapezoidal linear congruence abstraction [Mas92], depicted in Figure 17(d), combines non-relational dense approximations (such as intervals) with relational sparse approximations (such as congruences).



(a) Octagonal Abstraction (b) Polyhedral Abstraction

(c) Relational Congruence Abstraction (d) Trapezoidal Congruence Abstraction

Figure 17: Relational Abstractions [Cou01]

4.2.3 Satisfiability Modulo Theories

The Boolean satisfiability decision problem, commonly referred to as SAT problem which is *NP-complete* [Coo23]. It involves finding a satisfying assignment for a given formula in propositional logic, which utilizes only Boolean variables and the logical connectives: and ($\wedge$), or ($\vee$), and negation ($\neg$) [MZ09]. An assignment assigns either true or false to each variable in the formula, and a satisfying assignment is one for which the formula evaluates to true.

Example 4.2. *The formula $A \wedge B$ has the satisfying assignment of both A and B being true*

Despite its exponential worst-case complexity, very efficient algorithms have been developed, allowing current Boolean satisfiability solvers to address many practical SAT problems of industrial size. The majority of state-of-the-art solvers are based on the DPLL architecture. [BHvM09, Tin02] provide a detailed description of this architecture.

Propositional logic can express many properties, but some are more easily expressed in a richer language[DMB11], while others cannot be expressed at all within propositional logic. Satisfiability Modulo Theories (SMT) extends SAT by incorporating a background theory or a combination of background theories. SAT was enhanced to support atoms from a theory $\mathcal{T}$ in addition to propositional literals, resulting in the decision problem known as satisfiability modulo theory (SMT). A background theory enriches the language with a new set of formulas that replaces atoms used in SAT.

For instance, the theories of Linear Integer Arithmetic (LIA) and Linear Rational Arithmetic (LRA) include atoms of the form $(a_1x_1 + \cdots + a_nx_n \bowtie C)$, where $(a_1, \ldots, a_n, C)$ are integer constants, $(x_1, \ldots, x_n)$ are variables (interpreted over $\mathbb{Z}$ for LIA and $\mathbb{R}$ or $\mathbb{Q}$ for LRA) , and $\bowtie$ is a comparison operator such as $=, <, \leqslant, >, \geqslant$. LIA and LRA are widely used in the field of software verification, along with other theories like bit vectors, arrays, and uninterpreted functions.

Example 4.3. *ϕ is a formula in the theory of integers.*

$$\phi = (x < 1) \wedge (x > 1)$$

Satisfiability modulo theories such as LIA or LRA is NP-complete. However, tools based on the *DPLL($\mathcal{T}$)* architecture [GHN+04] effectively solve many practical SMT problems. These tools provide a satisfying assignment when the problem is satisfiable.

DPLL

We briefly summarize the principle of the *DPLL* algorithm for solving the SAT problem. The procedure alternates between two phases:

- **Decision:** This phase heuristically selects a Boolean variable and assigns it a value of either true or false.

- **Boolean Constraint Propagation:** This phase propagates the consequences of the previous decision to other variables. It is common to encounter a conflict due to an incorrect choice in the decision phase. In such cases, the algorithm identifies the reason for the conflict (an erroneous decision) and *backtracks*, which involves unsetting the variables assigned between the incorrect decision and the current assignment, before restarting with a new decision.

The algorithm terminates when either there are no more decision steps to apply (i.e., all variables are fully assigned) with no conflicts, indicating the answer is *SAT*, or when the conflict arises from reasons beyond a simple bad decision indicating the answer is *UNSAT* [Hen14].

DPLL($\mathcal{T}$)

Suppose we have an SMT formula that includes atoms from a specific theory ($\mathcal{T}$). The core principle of *DPLL($\mathcal{T}$)* is to integrate a SAT solver with a T-solver tailored to the selected theory.

The *DPLL SAT* solver and the $\mathcal{T}$-solver interact by exchanging information as illustrated in Figure 18. the *SAT* solver abstracts the SMT formula into a SAT formula with the same Boolean structure and provides satisfying assignments for this Boolean abstraction to the $\mathcal{T}$-solver. The $\mathcal{T}$-solver then checks whether this assignment is satisfiable within the theory. Similar to Boolean Constraint Propagation, Theory Propagation propagates the effects of decisions based on the theory in use. If the formula is found to be unsatisfiable in the theory, it returns an *unsat* clause to prune the search space of the *SAT* solver. The *unsat* clause is a crucial aspect of the algorithm, as it is returned to the *SAT* solver as a conjunction of Boolean assignments that are collectively unsatisfiable, providing a second reason for backtracking (the first reason arises from conflicts identified through Boolean Constraint Propagation and Theory Propagation). Therefore, it is essential to obtain "good" (or "short") *unsat* clauses to ensure that backtracking is efficient [Hen14, GHN+04].

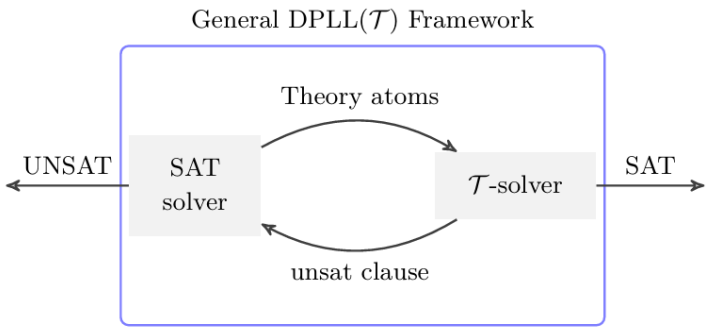


Figure 18: *DPLL($\mathcal{T}$)* framework [Hav18]

SMTLIB

The SMT-LIB standard outlines how to specify theories and communicate with SMT solvers [BFT16]. Two relevant background theories are the theory of fixed-length bit-vectors and the theory of floating-point arithmetic. The theory of fixed-length bit-vectors allows for reasoning about bit-vectors of a specified length using common operations such as bitwise AND, OR, and NOT, as well as basic arithmetic operations that interpret the bit-vectors as binary numbers. The theory of floating-point arithmetic enables reasoning about floating-point numbers according to the IEEE754 floating-point standard [DBL85].

4.2.4 Linear Programming

Linear programming problems involve a linear cost function, which consists of multiple variables that need to be minimized or maximized, subject to a set of constraints. These constraints are represented as linear inequalities involving the variables of the cost function. The cost function is often referred to as the objective function. Linear programming is closely related to linear algebra; however, the most significant difference is that linear programming typically employs inequalities in the problem statement instead of equalities [Sch00].

Definition 4.7 (Linear programming Problem)**.** *A linear programming problem is the problem of minimizing (or maximizing) a linear function subject to a finite number of linear constraints. Given variables $\lambda = [\lambda_1, \ldots, \lambda_n]^T$, an LP problem in standard form can be given as shown*

$$\begin{aligned} \textbf{\textit{Maximize (or Minimize)}} \quad Z(\lambda) = & \sum_{j=1}^{n} c_j \lambda_j \\ \textbf{\textit{subject To}} \quad & \sum_{j=1}^{n} a_{i,j} \lambda_j \leqslant b_i, \quad \forall i \in \{1 \ldots m\} \\ & \lambda \geqslant 0 \end{aligned} \tag{4.5}$$

Where $Z(\lambda)$ is the linear objective function, $\sum_{j=1}^{n} a_{i,j}\lambda_j \leqslant b_i, (\forall i \in \{1 \ldots m\})$ is the set of linear constraints, and $a_{i,j}$ and c_j are real coefficients. The linear constraints combined with a linear objective are called Linear Programs (LPs), and systems that solve them are called LP solvers. Note that if a constraint is not in the form of equality then we can add a non-negative variable, also called slack variable.

Any vector λ satisfying the constraints of the linear programming problem is called a feasible solution of the problem. Every linear programming problem falls into one of three categories :

- *Infeasible*. A linear programming problem is infeasible if a feasible solution to the problem does not exist; that is, there is no vector λ for which all the constraints of the problem are satisfied

- *Unbounded*. A linear programming problem is unbounded if the constraints do not sufficiently restrain the cost function so that for any given feasible solution, another feasible solution can be found that makes a further improvement to the cost function.

- *Has an optimal solution*. Linear programming problems that are not infeasible or unbounded have an optimal solution; that is, the cost function has a unique minimum (or maximum) cost function value. However, This does not mean that the values of the variables that yield that optimal solution are unique.

Integer Linear Programming (ILP) is a specialized form of Linear Programming (LP) where variables are constrained to integer values. Like LP, ILP involves optimizing an objective function—either maximizing or minimizing it—while satisfying linear equality and inequality constraints [Ben21]. However, ILP imposes additional integrality conditions on some or all variables, making the problem more complex. Although finding an optimal solution for ILP is NP-hard, many large-scale ILP problems can still be solved efficiently in practice [CWM02].

Mixed Integer Linear Programming (MILP) is an extension of Linear Programming where some variables are integers while others can be continuous, making it ideal for problems combining discrete and continuous decisions. In contrast, Integer Linear Programming (ILP) requires all variables to be integers, restricting it to pure discrete optimization. While MILP can model complex real-world scenarios like production planning with both whole-unit constraints and variable resource allocation, ILP is better suited for problems requiring strict whole-number solutions like employee scheduling or network design. Both are NP-hard, but MILP problems often have computational advantages since their continuous variables can sometimes relax the solution space compared to purely discrete ILP formulations. Modern solvers like Gurobi and CPLEX handle both problem types using similar techniques (branch-and-bound, cutting planes), though ILP problems typically require more intensive computation due to their fully constrained nature.

LP Solving

Linear Programming (LP) problems are typically solved using either the Simplex method [NM65] or Interior-Point methods [LMS94], each offering distinct advantages. The Simplex method [NM65], developed by George Dantzig in 1947, it performs iteratively row operations on the simplex table. At each iteration, the method moves from a current basic feasible solution to another basic feasible solution which improves the objective function value. The method terminates when it cannot decrease the objective function value any more. While it is highly efficient for most practical problems and provides valuable sensitivity analysis, its worst-case complexity is exponential, which can lead to slower performance in certain degenerate cases [Bor12].

In contrast, Interior-Point methods [LMS94], pioneered by Narendra Karmarkar in 1984, approach the optimal solution by traversing the interior of the feasible region instead of its

vertices. These methods utilize barrier functions to navigate toward the optimum while remaining within the feasible space, offering polynomial-time complexity and superior performance for large-scale, dense problems. However, they may lack the intuitive geometric interpretation and post-optimality analysis that the Simplex method provides [RTV05].

Popular commercial solvers such as Gurobi [GO21], and CPLEX [Ilo10] implement optimized versions of both methods, automatically selecting the best approach based on the problem structure. Open-source alternatives like GLPK [Lua06], and COIN-OR [Sal02] also support these techniques, making LP accessible for academic and smaller-scale applications.

4.3 Formal Verification Methods for ANNs

The main challenge we encounter when verifying AI controllers based on artificial neural networks (ANNs) and their associated interpretation lies in effectively managing the composition and relationship between its inputs and outputs.
Fortunately, there is a considerable amount of ongoing research dedicated to the formal verification of neural networks, which specifically tackles this challenge, see e.g. [Ehl17, WIZ+24, GMD+18, LLX+24, GWZ+21, HKWW17a, KBD+17].

Several factors influence the design of neural network verification methods, including :

(1) The architecture of the networks (e.g., depth, layer types, and connectivity patterns).

(2) The activation functions used (such as ReLU, sigmoid, or tanh, which affect linearity and differentiability).

(3) The norms employed for measuring robustness and distances (like L_1, L_2, or L_∞).

(4) The datasets utilized, which determine input distributions and generalization requirements [Sou24].

(5) The computational complexity, scalability to large networks, and the trade-off between precision and efficiency are crucial.

Considerations of These factors collectively shape the development of tailored verification techniques to ensure reliability and robustness in neural network deployments. From these methods, we identify Two major formal verification categories for verifying neural networks, complete methods [KBD+17, MSB22, WIZ+24, FJ18, DJST18] and incomplete methods [BP23, HRS+23, WXT+13, SBR+, PRA19].

4.3.1 Complete Methods

A sound and complete verification framework of ANNs guarantees that a method either proves that the property holds or finds a counterexample to this property. They are based on SMT [BSST09] or Mixed Integer Linear Programming MILP [MSB22] search engines and other complete methods based on global optimization or combinations of exact and

approximate analyses [RHK18, BLT+20]. The main issue with this approach is scalability. For instance, neural networks used for computer vision tasks contain millions of parameters. While large neural networks theoretically enable the generation of formal specifications, in practice, these formalizations are difficult for modern solvers to reason about.

- ***SMT-based approaches and Tools*** Common methods have been widely employed SMT-based approaches [HKWW17b, KBD+17, PT12, SdBG+19] to verify the safety of ANNs. The main advantage of these techniques is their soundness and completeness. However, a significant drawback is their limited scalability due to the NP-HARD nature of the problem [KBD+17]. They are also constrained when dealing with large artificial neural networks (ANNs) and their complexities, and they are inefficient for quantized networks.

  The first formal verification method for neural networks was introduced by Pulina and Tacchella [PT10], focusing on verifying safety bounds for outputs of fully-connected networks with sigmoid activations. Their approach discretized the activation functions into piecewise linear approximations using interval arithmetic, encoding the network and safety constraints into an SMT problem. Parallel work by Scheibler et al. [SWWB15] applied similar SMT-based bounded model checking to the cart-pole system, but could only verify basic properties at minimal unrolling depths, further demonstrating the computational challenges of early SMT approaches to neural network verification. Other proposed SMT-based tools are implemented for the formal verification of ANNs; we mention:

  - RELUPLEX [KBD+17] Is a specialized SMT-based solver for verifying deep neural networks with ReLU activation functions. It extends the classical Simplex algorithm to handle the piecewise-linear constraints introduced by ReLUs through a novel *case-splitting* approach. The tool transforms neural network verification problems into a series of linear programming subproblems, where infeasible ReLU states are incrementally excluded via conflict analysis. RELUPLEX was the first verifier to successfully analyze the safety properties of fully-connected feed-forward networks used in airborne collision avoidance systems [JK19]. While limited to ReLU activations, its sound and complete procedure established foundational techniques for later neural verification tools.
  - MARABOU [WIZ+24] is the successor of RELUPLEX , an open-source, SMT-based neural network verification framework developed by the Weizmann Institute. It analyzes properties of deep neural networks (DNNs) by encoding them as SMT queries over real arithmetic and piecewise-linear constraints. MARABOU supports both *complete* verification (providing formal guarantees for properties like robustness or safety) and *incomplete* counterexample search, handling networks with ReLU, sigmoid, and max-pooling activations. Key features include

parallelization, support for ONNX and TensorFlow models, and integration with abstraction refinement techniques.

In this thesis, we used MARABOU as an SMT verification tool to evaluate our contributions.

– DLV [HKWW17b] developed by Huang et al. for verifying local robustness against adversarial attacks in neural networks. Their layer-by-layer approach works for general feedforward and convolutional architectures with arbitrary activation functions by:

  * Reducing the infinite input neighborhood to finite representative points
  * Propagating constraints between layers
  * Verifying consistent outputs across all points

  Implemented in their open-source DLV tool using Z3 SMT solver [DMB08a], the method successfully found adversarial examples in seconds for some MNIST[Pei21] and CIFAR-10 [KH+10] cases but faced scalability challenges with larger inputs like ImageNet, highlighting the computational complexity of complete verification.

All these mentioned tools used floating point arithmetic using IEEE754 with precision of 64 bits, where rational numbers was available only in Marabou with GNU Multiple Precision Arithmetic Library (GMP) library [Lib23]

- ***MILP-based approaches and Tools*** MILP-based formal methods enable exact verification of neural networks by encoding them as optimization problems. These problems utilize continuous variables for activations and integer variables for discrete behaviors, with linear constraints modeling the network's semantics. These methods provide precise guarantees for properties such as adversarial robustness and output safety, leveraging powerful solvers like GUROBI [Gur24] and CPLEX [VS10]. Although they are **NP-HARD**, these methods effectively manage diverse architectures and activation functions, making them valuable in safety-critical domains.

  Fischetti and Jo [FJ18] proposed a MILP approach to find adversarial examples and maximize neuron activations in ReLU networks, using layer-wise optimization with asymmetric bounds. Their method worked well on small MNIST networks ($\leqslant$ 70 neurons) but scaled poorly to larger ones. Bunel et al. [FJ18] introduced BAB, a branch-and-bound framework unifying prior methods (PLANET, RELUPLEX), with BABSB [BTT+18] outperforming other variants by smarter input splitting, especially on deeper networks like ACAS Xu. Dutta et al. [DJST18] developed SHERLOCK, combining MILP with local search to bound outputs of ReLU networks efficiently—scaling to thousands of neurons but sometimes requiring days. While monolithic MILP solves small cases faster, SHERLOCK and BABSB show significant speedups over RELUPLEX, highlighting MILP's versatility for exact verification despite scalability challenges.

Finally, Tjeng et al. [TXT19] proposed MIPVERIFY, a MILP-based tool for finding minimal adversarial examples (using $L1, L2, and L_\infty$ distances) in ReLU networks. Their key innovation was progressive bound tightening: first using fast interval arithmetic, then resorting to exact LP bounds only for uncertain ReLUs. This approach, tested on MNIST/CIFAR-10 networks with thousands of neurons, proved $100 - 1000$x faster than RELUPLEX, with runtime depending mainly on ambiguous ReLUs rather than total network size.

There are other complete verification methods for neural networks (NNs) that do not rely on SMT or MILP. We refer to DEEPGO [KKH18], a global optimization approach for verifying Lipschitz-continuous networks (including ReLU, sigmoid, and tanh activations), which computes output bounds 36 to 100 times faster than Sherlock and RELUPLEX and can scale to million-neuron MNIST networks. Concurrently, NEURIFY [WPW+18] has been introduced, enhancing precision by maintaining input neuron dependencies during ReLU analysis.

4.3.2 Incomplete Methods

Incomplete verification methods can analyze large neural networks efficiently, typically within minutes for networks with thousands of neurons, while maintaining soundness using floating-point arithmetic. However, they may produce false positives. These methods support diverse architectures, including recurrent networks, and fall into two main categories: abstract interpretation-based approaches and other techniques using simulation, and linear approximations.

- ***abstract Interpretation based methods and approaches***: The field began with Gehr et al.'s AI2 [GMDC+18], which introduced abstract interpretation for verifying ReLU networks using intervals, zonotopes, and polyhedra, demonstrating practical verification (<10 seconds) for MNIST networks up to 18K neurons. Subsequent advances include:

  - DEEPZ [BP23] (Singh et al., 2018): A specialized zonotope domain handling floating-point soundness, verifying 88.5K-neuron CIFAR-10 networks in minutes.
  - DEEPPOLY [HRS+23] (Singh et al., 2019): Combined concrete/symbolic bounds with adaptive ReLU approximations, outperforming DeepZ in speed and precision.
  - KPOLY [WXT+13] : Jointly approximates k ReLUs, enabling analysis of $107K$-neuron networks in < 8 minutes.

  These methods, implemented in ERAN [SBR+] tool, leverage GPU acceleration (170× speedup) and symbolic propagation (Li et al.) to scale to near-million-neuron networks. Key trade-offs persist between precision (reduced false positives) and scalability for complex perturbations (L1, rotations).

These tools default to floating-point IEEE754 (usually 64-bit double-precision) for efficiency. However, floating-point arithmetic can introduce rounding errors, which may compromise soundness if not properly controlled. To address this, they employ directed rounding, interval arithmetic (explicitly tracking error bounds), and affine arithmetic or zonotopes (propagating linear error terms).

Other approaches simplify verification by constructing smaller over-approximating networks:

- Prabhakar & Afzal [PRA19]: Merge ReLU neurons layer-wise using interval arithmetic, reducing ACAS Xu verification time via MILP encoding (tested with $\leqslant 32$ abstract neurons), though precision depends on partitioning strategy.
- Sotoudeh & Thakur [ST20]: Generalized to convex abstract domains (beyond intervals) and activations (Leaky ReLU, sigmoid/tanh), with conditions to ensure soundness or network modifications.

- ***Other incomplete methods*** : Recent research has developed efficient but incomplete methods to estimate neural network robustness bounds, trading exactness for scalability. Weng et al.'s Fast-Lin/Fast-Lip algorithms use symbolic linear approximations and Lipschitz constants to verify ReLU networks 10,000× faster than exact methods while maintaining reasonable precision (2-3× bound looseness). Zhang et al. extended this approach via CROWN [WZX+21a] to support diverse activations (sigmoid, tanh) with 19-20% tighter bounds.

4.4 Formal Verification Methods for QNNs

Existing verification methods of ANNs struggle with QNNs due to their discrete, non-differentiable operations (e.g., rounding), which violate the smoothness assumptions required by most techniques. Complete methods (MILP/SMT) face exponential complexity from bit-precise encoding, while incomplete approaches (abstract interpretation) lose precision when handling discontinuities. Additionally, the lack of standardized tools, hardware-specific semantics (e.g., overflow behavior), and insufficient theoretical frameworks for fixed-point arithmetic further limit applicability. Although emerging solutions like bit-precise SMT solvers and specialized abstract domains show promise, current methods remain inadequate for scalable, sound QNN verification, a critical gap as quantized models dominate edge AI deployments.

4.4.1 Formal Verification methods of 1-bit QNNs

There is restricted research on the verification of quantized neural networks (QNNs) compared to floating-point neural networks, particularly 1-bit quantized neural networks, which refer to binarized neural networks (BNNs) [HCS+16]. BNNs employ binary weights

and activations to drastically reduce computational latency and memory usage, making them attractive for edge devices and real-time applications. However, their verification remains underexplored compared to their full-precision counterparts.

Some researchers have approached BNN verification by reducing the problem to Boolean satisfiability (SAT) instances [RORF16, CNHR18], leveraging the discrete nature of binary operations to apply formal logic-based methods. While this approach is theoretically sound, its practical applicability is limited due to the NP-hardness of SAT solving, which scales poorly with network depth and width.

While symbolic interval propagation provides efficient over-approximations, BNN verification remains challenged by combinatorial explosion and limited applicability to real-world mixed-precision models [HCS+16]. Recent advances [NSS+16] focus on hybrid methods combining SAT with abstraction and GPU acceleration to improve scalability, with future directions extending these techniques to practical low-bit quantized networks.

4.4.2 Formal Verification methods of multiple-bit QNNs

Recently, there has been significant effort in verifying multiple-bit quantized neural networks (QNNs) [GHL20, BMS22, HLZ21, MSB22, ZSS23, HWY+24].

As a first step, researchers in [GHL20, BMS22, BM24a] have focused on determining the appropriate bit-width allocation for the fractional and integer parts of each neuron or layer in Quantized Neural Networks (QNNs). This precision-tuning process aims to ensure both optimal efficiency (minimizing hardware resource usage) and soundness (preserving network behavior relative to the original floating-point model). Their approaches leverage formal methods such as interval arithmetic to bound numerical errors and SMT solvers to verify the adequacy of selected bit-widths under worst-case scenarios. For instance, Giacobbe et al. in [GHL20] proposed a layer-wise mixed-precision optimization framework that combines gradient-based bit-width search with formal guarantees, while Benmaghnia et al. in [BMS22] introduced fixed-point encoding strategies with provable error bounds using SMT-based validation. These methods address the tension between quantization-induced precision loss and computational tractability, particularly for safety-critical applications where over-approximation of errors is unacceptable. However, challenges persist in scaling these techniques to deep networks and reconciling them with hardware-specific constraints (e.g., FPGA DSP block configurations).

Additionally, zhang et al. in [HWY+24, ZSS23] focus on synthesizing tight error bounds between floating-point DNNs and their quantized counterparts, employing techniques such as interval propagation, differential reachability analysis and SMT-based error minimization to rigorously quantify precision loss. These methods prioritize optimizing bit-width allocation per neuron or layer, often using gradient-based search or mixed-integer programming to balance computational efficiency with formal guarantees on approximation error. However, this line of work largely overlooks the verification of semantic properties (e.g., robustness, fairness, or safety) that depend on the quantized network's behavior in real-world

contexts. For instance, while [ZSS23] derives worst-case error bounds for fixed-point arithmetic, it does not verify whether these errors violate critical specifications under adversarial inputs or distribution shifts. Recent efforts like [DKSL20] bridge this gap by combining error-bound synthesis with adversarial robustness verification, but challenges persist in scaling such approaches to large-scale QNNs with heterogeneous quantization policies (e.g., mixed 4/8-bit models). A unified framework for joint precision optimization and property verification remains an open research direction, particularly for safety-critical applications where both numerical soundness and functional correctness are non-negotiable.

In [HLZ21], the authors proposed heuristic optimizations including symbolic simplification, abstraction refinement, and parallel SMT solving to reduce the verification complexity of SMT encodings with fixed-point arithmetic, adversarial perturbations, and bit-vector constraints which has shown to be **PSPACE-complete**. While effective for moderate-sized networks, these methods struggle with large QNNs due to exponentially growing constraint complexity. Subsequent work has addressed this through hardware-conscious approaches that co-optimize bit-widths and verification such as FPGA-aware precision tuning [HYL+24].

Instead of only reducing the coding of a QNN by fixed-point arithmetic, as proposed in [HLZ21], or verifying the constraints of the quantization error bound between a deep neural network (DNN) and its quantized version, as outlined in QEBVerif [ZSS23]. In this thesis, we focus on verifying safety properties of QNN behaviours using SMT-based verification method that avoids the need for integer or bit-vector encoding, even with reduced heuristics.

4.5 Summary

In this chapter, we presented formal verification techniques for neural networks (NNs) and quantized neural networks (QNNs), focusing on their mathematical foundations, practical challenges, and applications. We began by examining common formal methods such as abstract analysis, abstract interpretation, model checking, satisfiability modulo theories, and mixed linear integer programming. Then, we highlighted several approaches dedicated to the verification of NNs: complete, incomplete, and hybrid methods. Finally, we explored recent works that address the formal verification of QNNs. In the next chapter, part II, we will examine the first contribution of this thesis which consist of specifying QNNs requirements for Autonomous vehicles.

Part II

Specification and SMT Verification of QNN for Autonomous Vehicles

Chapter 5

Specification of Autonomous Vehicles requirements for verifying NNs

5.1 Introduction

There is a growing research area focused on the formal specification of autonomous systems [LFD+19, WMA12, FDG+19, RPL+20]. This involves transforming the textual requirements of autonomous systems, as mentioned in a user requirements document written in natural languages, into a formal specification [FLSM22b, FLSM22a]. This specification can then be converted into various properties and formulas, such as LTL

properties [FDK12], first-order logic formulas [FRNS17], and Signal Temporal Logic properties [Aré19]. Recently, there has been a particular emphasis on the formal specification of requirements and rules for autonomous vehicles that includes the use of real-world and simulator metrics [GGKT23, STRP23].

In this chapter, we detail the first contribution of this dissertation. We will begin with a literature review on the formal specification of autonomous vehicle (AV) requirements. Next, we will present the AV simulator HIGHWAY-ENV as a case study. We will then explain the process of transforming AVs' textual requirements into formal properties to be verified exhaustively on neural networks. Finally, we will demonstrate our proposed process using AVs' textual requirements inspired by HIGHWAY-ENV.

5.2 Formal Specification of Autonomous Vehicles

Formal specification of autonomous vehicles (AVs) provides a mathematically rigorous framework to define and verify safety-critical behaviors, using temporal logic for time-dependent rules ("always yield to pedestrians") [PW24], contract-based design [TFA24] for component interfaces, and hybrid systems modeling [ASL91] for vehicle dynamics [KSA+18]. These specifications enable exhaustive verification through model checking, theorem proving, and runtime monitoring [RFNV20], addressing challenges like state-space explosion and neural network verification while complying with standards ISO 21448, UL 4600 [GWWR23]. By bridging theoretical precision with practical deployment needs, formal methods ensure AVs meet provable safety guarantees a mid real-world uncertainty, resolving ambiguity in requirements and enabling compositional analysis of complex perception-planning-control stacks.

Formal methods encompass the process of translating abstract specifications into a system control algorithm or program to ensure that the behaviour of the controlled system meets these specifications. In response to this necessity, numerous methods have been proposed in the literature to formalize textual requirements of autonomous vehicles and formulate formal safety properties [ZHZ+21, GGKT23, FDG+19, RPL+20, WNSO23]. These methods involve generating abstract scenarios and subsequently transforming them into logical or concrete scenarios suitable for linear temporal logics (LTL) and signal temporal logic (STL) properties [Aré19, FDG+19]. These properties, which rely on time, are used to present the specifications of autonomous vehicles for testing in real traffic simulation tools [RPL+20, DRC+17]. A formal verification framework utilizing SMT-based model checking was introduced in [ZHZ+21] which abstracts autonomous vehicle's behaviour to SMT predicates, enabling the identification of counterexamples that violate the initial requirements based on AV standard rules.

In contrast to the approaches outlined previously, which used formal verification to identify safety violations in autonomous vehicles (AVs) by comparing them to AV rules or using real-world traffic simulation or testing, our method involves generating logical

scenarios from AV textual requirements that are converted into SMT predicates for exhaustive verification of AI controller-based neural networks, in particular quantized neural networks (QNN), using SMT-based model checking.

5.3 Case Study: HIGHWAY-ENV

HIGHWAY-ENV [Leu18] is a Python library that provides a collection of environments for reinforcement learning (RL) tasks focused on autonomous driving and tactical decision-making scenarios such as highway driving, parking, and intersections 23. It's built on top of OpenAI Gym (now Gymnasium), making it compatible with most RL algorithms.

It uses a Deep Q-Network (DQN) reinforcement learning algorithm [HVP+18] that employs a Multi-Layer Perceptron (MLP) architecture for ANNs. This function exclusively controls the autonomous or ego vehicle (EV), while the other vehicles, referred to as normal (non-ego) vehicles (NVs), are initialised to operate randomly.

Each vehicle in the environment is characterized by five input kinematic variables:

- p_i: indicates the presence of the i^{th} vehicle on the road.
- x_i: represents its position on the x axes in meters (m).
- y_i: represents its position on the y axes in meters (m)
- v_{xi}: denotes the velocity of the vehicle on the x axes in meters per second (m/s)
- v_{yi}: denotes the velocity of the vehicle on the y axes in meters per second (m/s)

The neural network takes as input 25 features corresponding to five vehicles, including the EV), each vehicle represented by five kinematic variables. The kinematic values of the EV are absolute, while the kinematic values of the four NVs are relative to the EV.

The corresponding neural network produces five Q-values representing the probability of each possible action. According to the DQN algorithm, the agent selects the action associated with the highest Q-value, which can be one of the following:

- `lane left`: Represents the first output of the neural network; it indicates moving one lane to the left. The vehicle will smoothly change lanes if the left lane is available and safe.
- `idle`: Represents the second output of the neural network; it indicates to maintain the current lane and speed. The vehicle continues at its current velocity without acceleration or deceleration.
- `lane right`: Represents the third output of the neural network, indicating a lane change to the right. The vehicle will smoothly change lanes if the right lane is available and safe.

- `faster`: Represents the fourth output of the neural network: accelerate in the current lane. This increases the vehicle's speed, up to a maximum speed limit.
- `slower`: Represents the fifth output of the neural network: "Decelerate in the current lane." This output decreases the vehicle's speed, but it will not allow the vehicle to go negative or reverse.

The kinematic variables are normalized between -1 and 1 before being propagated in the *NN*, and the feature values of the non-ego vehicles (NVs) are relative to the EV except for those of the EV, which are absolute as described in the documentation [1].

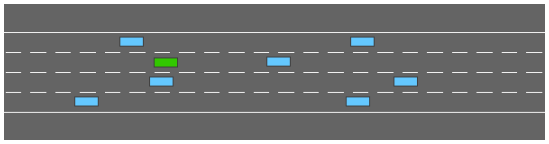


Figure 19: Highway Environment

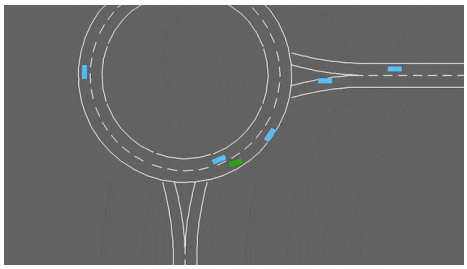


Figure 20: Roundabout Environment

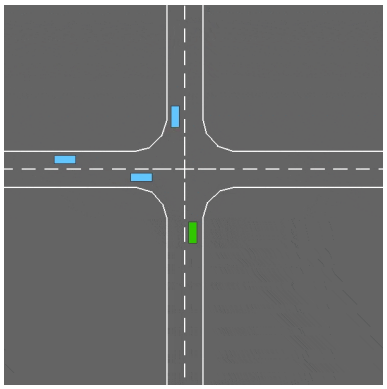


Figure 21: Intersection Environment

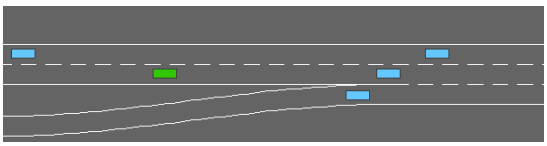


Figure 22: Merge Roads Environment

Figure 23: HIGHWAY-ENV Environments [Leu18]

5.4 Process of Transforming Textual Requirements into Formal Properties for AVs

In user requirement documents, engineers often rely on scenarios to describe the intended behaviour of their system. This is particularly relevant for autonomous systems, where explaining the full requirements using only natural languages can be tedious. Additionally, translating user requirements into logical statements for autonomous vehicles is inherently challenging due to the complexity of balancing technical, legal, and ethical

[1] https://highway-env.farama.org/

considerations. User requirements, such as safety and adherence to traffic laws, often lack precision and are context-dependent [HZ19].

Therefore, we propose a method for specifying Autonomous Vehicle (AV) requirements, which transforms textual and informal requirements into formal properties through four key steps. First, we reformulate the textual requirement if it is unclear or unavailable. Next, we analyze the textual requirement and identify abstract scenarios or AV observations that satisfy the constraints defined in the requirements by examining the neural network's output against the specified values. Subsequently, we generate logical scenarios by converting relevant observations and value ranges into predicates. Finally, we formalize these predicates into verifiable properties encoded as SMT predicates. If the SMT solver verifies the property, we conclude that the formal property correctly corresponds to the original textual requirement. Otherwise, we determine that the textual requirement is inaccurate and must be reformulated, taking into account the defined values, as illustrated in Figure 24.

This approach is inspired by the abstract scenarios proposed in the literature [ZHZ+21, GGKT23]. However, our goal is not to identify violations or counterexamples of requirements with respect to given rules for self-driving cars. Instead, we aim to exhaustively verify these scenarios on an autonomous vehicle controller based on neural networks to demonstrate their reliability. The steps of our method are defined in Figure 24 as follows:

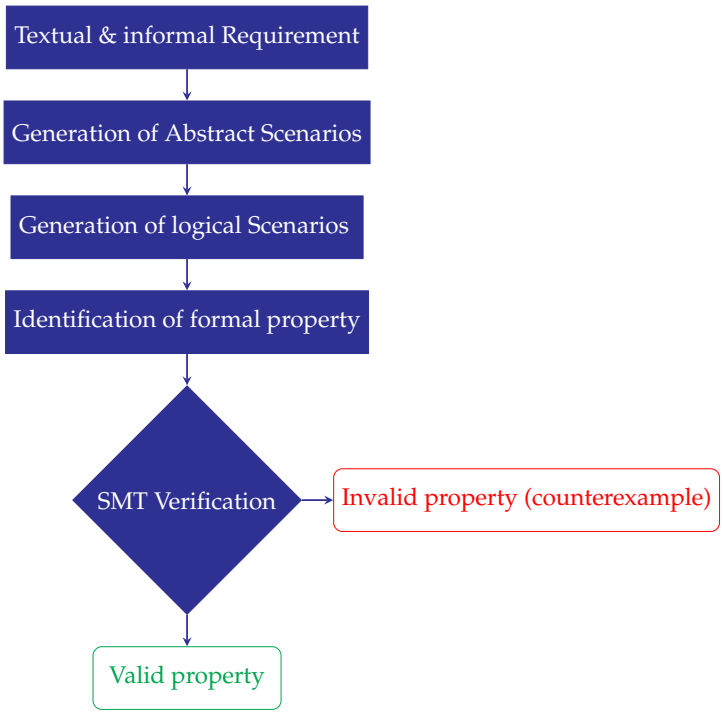


Figure 24: Process of identifying formal properties from textual requirement

5.4.1 Textual and Unformal Requirement

It also refers to functional scenarios or mission-level requirements that are written in natural language and focus on the behavior of the EV in relation to the surrounding vehicles. In this step, we manually reformulate the AV's textual requirements if they are unavailable; otherwise, we skip this step and move on to the second step. Due to the absence of textual requirements for the AVs that correspond to the behavior of the EV in the HIGHWAY-ENV [Leu18] simulator, we first observed the behavior of the EV in a specific and well-defined situation by running the trained model of the EV using the highway-env simulator [Leu18]. Based on these observations, we manually described and formulated the textual requirements, with a focus on clarity to ensure that each requirement included measurable values for verification. These observations included the actions and values of each variable of the ego vehicle (EV) and three surrounding non-ego vehicles (NVs). at each timestep t_i, as mentioned in [Leu18].

Example 5.1 (Running Example). *Figure 25, taken from the video recording of the* HIGHWAY-ENV *simulator [Leu18], depicts a highway with three lanes showing a* `slower` *predicted action involving the EV and three surrounding NVs.*

Textual requirement of `slower` *action* ***TRQ1: The EV shall decelerate smoothly when it is in the middle lane and surrounded by three vehicles: one at least 20 meters in front, one at least 15 meters to the left, and one at least 15 meters to the right, provided that the relative speed of all surrounding vehicles is within the range of -2 m/s to 2 m/s for a continuous period of at least 5 timesteps.***

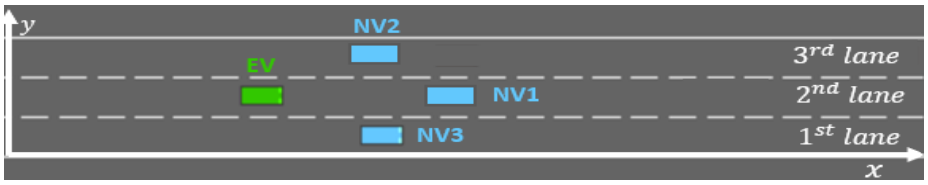


Figure 25: A scene of a video recorder for `slower` action of HIGHWAY-ENV environment

5.4.2 Generation of Abstract Scenarios

Abstract scenarios represent a set of observations that capture the state (values of each variable) of the EV and the surrounding vehicles on the road at each timestep t_i. To transform textual requirements into abstract scenarios, we follow two steps:

1. *Generation of constraints from textual requirements:* From the textual requirements, we extract the constraints that enable us to select only the set of variable values for the EV and NVs that meet our textual requirements. To accomplish this, we identified the action performed by the trained EV model, which represent its outputs. Additionally, we specify the number of NVs present on the road ($L \in \mathbb{Z}$), the position of each vehicle in the lanes ($y_i \in \mathbb{Q}$), the minimum relative distance of the NVs compared to

the EV ($x_i \in \mathbb{Q}$), and the speed range of all vehicles ($vx_i \in \mathbb{Q}$) for a defined number of timesteps ($k \in \mathbb{Z}$).

In the running example, we define the constraints from *TRQ1* as follows:

- "The EV shall decelerate smoothly" $\Longrightarrow$ The trained model of the EV in a highway environment should select the action `slower`, indicating that the last output has the highest value compared to the other outputs (actions).
- "when it is in the middle lane and surrounded by three vehicles" $\Longrightarrow$ $(y_0 = 4m \ \wedge \ L = 3)$
- "one at least 20 meters in front" $\Longrightarrow$ NV1 on the same road as the EV, $(y_1 = 0\,m \wedge x_1 \geqslant 20\,m)$.
- "one at least 10 meters to the left" $\Longrightarrow$ NV2 in the left lane $(y2 = -4\,m \wedge x_2 \geqslant 10\,m)$.
- "one at least 20 meters to the right" $\Longrightarrow$ NV3 in the right lane $(y3 = 4\,m \wedge x_3 \geqslant 20\,m)$.
- "provided that the relative speed of all surrounding vehicles is within the range of -2 m/s to 2 m/s" $\Longrightarrow$ $(\forall i \in \{1, \ldots, L\}, vx_i \in [-2, 2](m/s))$.

2. *Selection of abstract scenarios*: Based on the constraints defined in the previous step, we select a set of k successive variable values for the EV and the NVs, which are obtained by running the trained neural network of the EV at each timestep t_i. In our running example, *TRQ1*, we select arbitrary a set of five successive variable values that meet these constraints.

   For instance, for x_1 of NV1, we select the following values (respectively for the other variables for each vehicle):

$$(NV1) : x_1(t_1) = 33(m),\ x_1(t_2) = 34(m),\ x_1(t_3) = 36(m),\\ x_1(t_4) = 36,5(m),\ x_1(t_5) = 37(m) \tag{5.1}$$

5.4.3 Generation of Logical Scenarios

Logical scenarios provide a detailed representation of abstract scenarios by specifying the value ranges of locations and variables for each vehicle using formal notation. These state space variables describe the entities and their relationships. In our case, we extract the minimum and maximum values for certain variables from the abstract scenarios outlined in the running example. We then define a logical scenario that considers the value ranges for specific variables of each vehicle while fixing the others. In HIGHWAY-ENV [Leu18], We set the variable p, which indicates the presence of the vehicle, to 1 if the vehicle is on the road and to 0 otherwise. Additionally, the vertical speed vy of all vehicles is set to 0

In the running example, we define S_L1 as the logical scenario corresponding to the textual requirement *TRQ1* in Eq. 5.2 as follows:

$$S_L1 = \begin{cases} EV: & p_0 = 1, x_0 = 24.5, y_0 = 4, vx_0 \in [23, 25] \\ NV1: & p_1 = 1, x_1 \in [33, 39], y_1 = 0, vx_1 \in [0.4, 0.6] \\ NV2: & p_2 = 1, x_2 \in [10, 20], y_2 = -4, vx_2 \in [-2, -1.5] \\ NV3: & p_3 = 1, x_3 =\in [29, 38], y_3 = 4, vx_3 \in [1.5, 1.7] \\ NV4: & p_4 = x_4 = y_4 = vx_4 = 0 \end{cases} \tag{5.2}$$

According to HIGHWAY-ENV [Leu18] observations, the values of the EV are absolute, while the NV observations are relative to the EV.

For example, $x_1 = x_0 - x_{a1}$, where x_{a1} and x_0 refer to the absolute positions of NV1 and the EV, respectively. In this context, x_1 represents the relative position of NV1. We fix the absolute position x of the EV at 24.5 m as an example, so any value within the range of [-100, 100] (as defined in the HIGHWAY-ENV documentation) would also be acceptable and would function the same as the chosen value.

5.4.4 Identification of Formal Property

We can identify the formal property associated with the logical scenarios by transforming the variables and their value ranges of each vehicle into predicates. In our example, we define the formal decision property, denoted as φ_1, of the logical scenario S_L1 in Eq. 5.3 as follows:

Let us define the input vector $X = (p_0, x_0, y_0, vx_0, vy_0, \ldots, vy_4) \in \mathbb{R}^{25}$.

$$\begin{aligned} \boldsymbol{\varphi_1}: & \forall X \in \mathbb{R}^{25}, \text{ s.t.} \\ & (p_0 = 1 \wedge x_0 = 24.5, y_0 = 4 \wedge vx_0 \in [23, 25] \wedge vy_0 = 0) \\ & \wedge (p_1 = 1 \wedge x_1 \in [33, 39] \wedge y_1 = 0 \wedge vx_1 \in [0.4, 0.6] \wedge vy_1 = 0) \\ & \wedge (p_2 = 1 \wedge x_2 \in [10, 20] \wedge y_2 = -4 \wedge vx_2 \in [-2, -1.5] \wedge vy_2 = 0) \\ & \wedge (p_3 = 1 \wedge x_3 \in [29, 38] \wedge y_3 = 4 \wedge vx_3 \in [1.5, 1.7] \wedge vy_3 = 0) \\ & \wedge (p_4 = x_4 = y_4 = vx_4 = vy_4 = 0) \\ & \implies \bigwedge_{j \in [0,4], j \neq 4} (out_4 > out_j) \end{aligned} \tag{5.3}$$

where $out \in \mathbb{R}^m$ is defined as $out = NN(X)$ and out_4 refers to the probabilistic value of the `slower` action.

5.4.5 SMT Verification

In this step, we encode the neural network (NN) and the formal property defined in the previous step using SMT-LIB. We then verify them using an SMT solver.

The results of the SMT verification can be categorized in two ways:

- **Satisfied Property**: If the result is UNSAT, the property is satisfied, indicating that the textual requirements defined in the first step describe the correct behavior of the specified AV's neural network-based controller.

- **Violated Property**: If the result is SAT with a counterexample, the property is violated, suggesting that the textual requirements established in the first step do not accurately describe the correct behavior of the specified AV's neural network-based controller.

In the running example, TRQ1 ensures the correct behavior of deceleration applied in the HIGHWAY-ENV.

5.5 Experimentation and Evaluation

To validate our proposed method for specifying Autonomous Vehicle (AV) requirements, we conducted a series of experiments applying a four-step formalization process to real-world AV requirements. We evaluated the correctness and robustness of the generated formal properties using SMT-based verification. This section presents the experimental setup and results, demonstrating how our method ensures consistency between textual requirements and their formal representations.

Setup

We implemented our process using Python 3, modifying the HIGHWAY-ENV [Leu18] simulator to execute our step-by-step process with Stable Baselines3 [RHG+21] and OpenAI Gym [BCP+16]. We used z3 as SMT solver. All experiments were conducted on an Apple M1 Ultra with 20 CPU cores and 128 GB of RAM.

Results & Evaluation

Returning to the *running example* defined in § 5.4.2, we extract the SMT predicates P_i from the property φ_1 of *TRQ1*, as defined in Eq. (5.3) in § 5.4.4, where we normalize the feature values to [-1,1] as a preprocessing step before propagating them in the neural

network as defined in Eq. (5.4) as follow :

$$
\begin{aligned}
P_0 = \;& (p_0 = 1.0) \wedge (x_0 = 1.0) \wedge (y_0 = 0.33) \wedge (vx_0 \geqslant 0.2875 \wedge vx_0 \leqslant 0.3125) \wedge (vy_0 = 0) \\
P_1 = \;& (p_1 = 1.0) \wedge (x_1 \geqslant 0.165 \wedge x_1 \leqslant 0.195) \wedge (y_1 = 0.0) \\
& \wedge (vx_1 \geqslant 0.005 \wedge vx_1 \leqslant 0.0075) \wedge (vy_1 = 0) \\
P_2 = \;& (p_2 = 1.0) \wedge (x_2 \geqslant 0.05 \wedge x_2 \leqslant 0.1) \wedge (y_2 = -0.33) \\
& \wedge (vx_2 \geqslant -0.025 \wedge vx_2 \leqslant -0.01875) \wedge (vy_2 = 0) \\
P_3 = \;& (p_3 = 1.0) \wedge (x_3 \geqslant 0.145 \wedge x_3 \leqslant 0.19) \wedge (y_3 = 0.33) \\
& \wedge (vx_3 \geqslant 0.01875 \wedge vx_3 \leqslant 0.02125) \wedge (vy_3 = 0) \\
P = \;& ((out4 > out0) \wedge (out4 > out1) \wedge (out4 > out2) \wedge (out4 > out3)) \\
\varphi_1 = \;& \bigwedge_{0 \leqslant i \leqslant 3} P_i \implies P
\end{aligned}
\tag{5.4}
$$

where out_i represents the outputs of the NN as defined in Eq. (5.3) in §5.4.2 We presents the negation of φ_1 using predicates P_i in E.q. (5.5) as follow :

$$\neg\varphi_1 : P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge \neg P \tag{5.5}$$

We encode the neural network (NN) and ($\neg\varphi_1$) using SMTLib, as demonstrated in Listing 5.1. The encoding process begins with declaring the variables (Lines 0-6). Next, we encode the rational NN by defining each neuron by layer, where "layer10" refers to the first neuron in the first layer (Lines 7-18). After that, we assign the neurons of the last layer to the output variables. Finally, we encode the negation of the property (Lines 20-28).

```
0  declaration of variables
1  (declare-fun p0 () Real)
2  (declare-fun x0 () Real)
3  (declare-fun y0 () Real)
4  (declare-fun vx0 () Real)
5  (declare-fun vy0 () Real)
6              .....
7  encoding of rational neural network by defining each neurone in
       each layer in the network
8  (assert (= layer10  ...)
9  (assert (= layer11  ...)
10 (assert (= layer12  ...)
11             ...
12 (assert (= layer30 ...)
13             ...
14 (assert (= out0  layer30))
15 (assert (= out1  layer31))
16 (assert (= out2  layer32))
17 (assert (= out3  layer33))
```

```
(assert (= out4  layer34))
encoding of the negation of the property
(assert (= p0 (/  1.00   1.00)))
(assert (= x0 (/  1.00   1.00)))
(assert (= y0 (/  33.00   100.00)))
(assert (>= vx0 (/  2875.00   10000.00)))
(assert (<= vx0 (/  3125.00   10000.00)))
(assert (= vy0 (/  00.00   1.00)))
            .....
(assert (or (>= out0 out4 ) (>= out1 out4) (>= out2 out4) (>=
   out3 out4)
```

Listing 5.1: SMTLIB encoding of the neural network and the formal property φ_1.

We then applied our method to another requirement that refers to the `slower` action denoted *TRQ2* and presented in Figure 26.

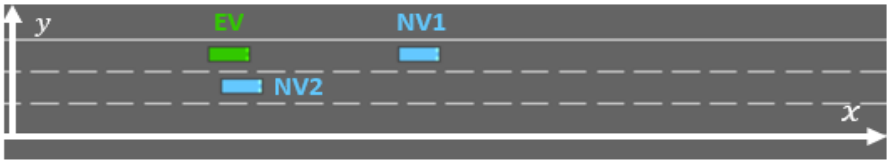


Figure 26: Textual requirement of `slower` action (*TRQ2*): *The EV shall decelerate smoothly when it is in the the 3rd lane (left lane) and surrounded by at least two cars: the first at least 20 meters in front, the secound at least 7 meters to the right, provided that the relative speed of all surrounding vehicles is within the range of -2.5 m/s to 1.5 m/s for a continuous period of at least 5 timesteps.*

We extract abstract scenarios from *TRQ2* such that :

- The EV is positioned in the 3rd ($y = 8$ m), with an initial speed between 83 and 90 km/h (23 and 25 m/s, respectively).
- It is surrounded by NV1 and NV2 : NV1 is in front of the EV (y=0 m), with a relative position at least $(x >= 20m)$, defining the safety distance. NV2 is in the second lane (y=4 m), with a relative position at least $(x >= 7m)$ and a minor relative acceleration ranging from -9 to 3.6 km/h ($-2.5m/s$ to $1.5m/s$).
- NV3 and NV4 are not on the road, so their kinematic feature values are all set to zero.

Next, we generate logical scenarios from a set of successive variable values that meet these constraints. After that, we define the formal property of *TRQ2*, denoted as φ_2, as follows:

$$P'_0 = (p_0 = 1) \land (y_0 = 8.0) \land (vx_0 \geqslant 23 \land vx_0 \leqslant 25) \land (vy_0 = 0.0)$$
$$P'_1 = (p_1 = 1) \land (x_1 \geqslant 32 \land x_1 \leqslant 39) \land (y_1 = 0.0) \land (vx_1 \geqslant -2.50 \land vx_1 \leqslant -2) \land (vy_1 = 0.0)$$
$$P'_2 = (p_2 = 1) \land (x_2 \geqslant 7 \land x_2 \leqslant 10) \land (y_2 = 4.0) \land (vx_2 \geqslant 0.8 \land vx_2 \leqslant 1.1) \land (vy_2 = 0.0)$$
$$P = ((out4 > out1) \land (out4 > out2) \land (out4 > out3) \land (out4 > out5))$$
$$\varphi_2 = \bigwedge_{0 \leqslant i \leqslant 2} P'_i \implies P$$

We encoded both the neural network (NN) and φ_2 using SMT-Lib and verified them using the Z3 SMT solver. Since φ_2 is satisfied, we conclude that *TRQ2* describes the correct behavior of deceleration performed by the highway environment agent in the highway environment.

5.6 Summary

This chapter focused on specifying the requirements for autonomous vehicles (AVs) by converting textual requirements into formal properties that can be rigorously verified for neural networks. Given the safety-critical nature of AV systems, it is essential to ensure that their underlying neural networks adhere to strict behavioral constraints. We outlined a step-by-step methodology to systematically generate formal properties from high-level textual specifications, ensuring they were both interpretable and verifiable using Satisfiability Modulo Theories (SMT) solvers. By leveraging SMT-based techniques, we enabled precise and automated verification of neural network compliance with these properties, bridging the gap between natural language requirements and formal, machine-checkable constraints.

In the next chapter, we will build upon these foundations by presenting a novel verification method tailored for quantized neural networks (QNNs). Since QNNs are widely used in embedded and resource-constrained AV systems due to their efficiency, verifying their correctness is crucial. Our approach will utilize the formal properties identified in this chapter to develop a scalable and sound verification framework, ensuring that QNNs meet the required safety and performance standards despite the challenges introduced by quantization.

Chapter 6

SMT Verification of QNNs using Set Theory and Rational Approximation

6.1 Introduction

Neural networks have been widely used in autonomous vehicles and a significant amount of research has been conducted to formally verify them [KBD+17, GPP23, GMD+18, MKE23, HKWW17a, WIZ+24, LLX+24, GWZ+21]. These networks are often limited to resource-constrained devices due to their demand for computational power and storage space, as well as their lack of floating-point units in certain configurations [HLZ21, BHL+20b]. In such cases, the weights and biases of real neural networks are

compressed and converted to integers through a process called quantization. This process utilizes fixed-point arithmetic for elementary operations and activation functions [BMS22]. Additionally, quantization allows for neural network computations to be performed using smaller word sizes, such as 8 and 16 bits. This reduction in word size helps to decrease training time, inference latency, and memory storage requirements [NHP+21, JKC+18]. However, the verification of QNNs faces scalability issues when using integers or bit-vector encodings, as it is proven to be **PSPACE-complete** [HLZ21].

In this chapter, we propose a sound and incomplete formal verification method that combines rational approximations using set-based theory with SMT-based verification of rational arithmetic programs. This approach is specifically designed to verify the properties that characterize the behavior of QNNs, rather than those that define the error between DNNs and their quantized versions [ZSS23].

Firstly, we will outline our approach and describe each step in detail. Next, we will provide an overview of the method and introduce the concept of δ-robustness. Finally, we will evaluate our method using the HIGHWAY-ENV simulator and the formal properties discussed in the previous chapter.

6.2 Process of SMT Verification of QNNs

In this chapter, we introduced a sound yet incomplete formal verification method for QNNs in Autonomous Vehicles (AVs). Our approach utilizes rational approximations of QNNs along with set-based analysis, applying bounded perturbations to the output of the approximated rational neural network. Additionally, we ensure that the output sets of the perturbed rational network encompass those of both the original QNN and its rational approximation. To quantify the divergence between these output sets, we employ p-norm distance metrics and interval propagation techniques.

The objectives of our method are summarized in the following steps, which define the process of our approach as illustrated in Figure 27.

- ***Inputs:*** We take as inputs a quantized neural network, denoted as NN_q, with a precision format of bits (specifying integer and fractional bits), along with the AVs formal safety property, denoted as P, defined in the previous chapter.

- ***Step1:*** we approximate NN_q using rational arithmetic to avoid fixed-point arithmetic complexities, thereby enabling efficient verification (From a **PSPACE-complete** [HLZ21]. to **NP-hard**problem [KBD+17]). The output of this step is a rational NN denoted NN_r. We will explain this step in §6.3.2

- ***Step2*** we encode both NN_r and the negation of the formal decision property P to verify that the property holds for NN_r using SMT verification, which indicates that the SMT solver returns `unsat`.

This is a necessary condition but not a sufficient one, such that if a counterexample is found (`sat`), the property is violated for NN_r, and the verification process fails resulting in incomplete result.

- ***Step3*** In case of success (`unsat`), we proceed with the analysis of a perturbed rational neural network, denoted as $\widetilde{NN}_r$. This involves characterizing the largest feasible perturbation, δ_{max}, such that $\delta_{max} \geqslant \varepsilon_p$, where ε_p represents the largest approximated distance between NN_r and NN_q. This distance is computed using p-norm and interval arithmetic, as clarified in §6.3.3.

  In this step, our goal is to verify that $\widetilde{NN}_r$ ensures its output sets encompass those of both NN_q and NN_r.

  The verification process involves encoding both $\widetilde{NN}_r$ and the negation of the safety property ($\neg P$). We will elaborate these steps in §6.3.3.

- ***Output:*** If the result of the SMT verification returns `sat`, we conclude that the property is violated on $\widetilde{NN}_r$ and unproved on NN_q, but it could still be satisfiable because we used interval arithmetic in computing ε_p, where this distance is overapproximated. However, if it returns `unsat`, We can conclude that the property is satisfied in both $\widetilde{NN}_r$ and NN_q.

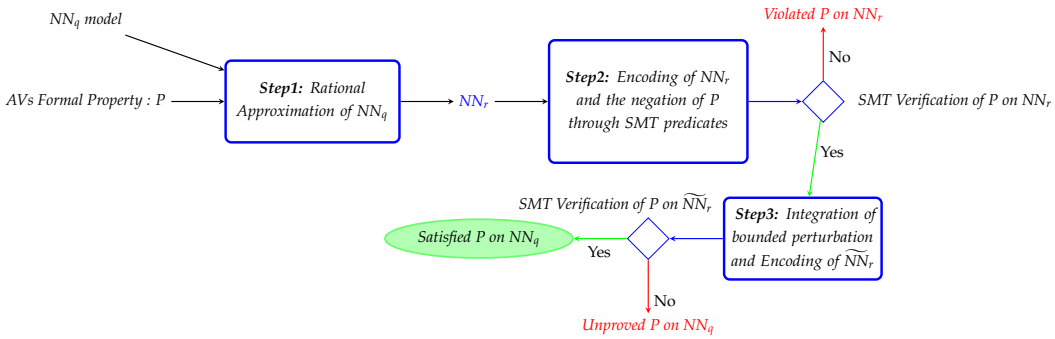


Figure 27: Overall Framework for SMT Verification of QNNs

6.3 SMT Verification of QNNs using Rational Approximation and Set Theory

The formal verification of neural networks using SMT-based model checking[GMD+18, MKE23, TLM+19] faces challenges in verifying QNNs due to computational complexity and the reliance on integer or bit-vector encodings, which have been proven to be **PSPACE-complete** [HLZ21].

Instead of only reducing the coding of a QNN by fixed-point arithmetic, as proposed in [HLZ21], or verifying the constraints of the quantization error bound between a deep neural network (DNN) and its quantized version, as outlined in QEBVerif [ZSS23], which

employs a hybrid approach based on Differential Reachability Analysis (DRA) and Mixed-Integer Linear Programming (MILP), we propose an efficient and sound verification method which combines rational approximations using set-based theory with SMT-based verification of rational arithmetic programs. We are inspired by the combination of methods proposed in the context of verifying dynamical systems [KGF23].

6.3.1 Overview of the Methodology

The set-based overview of our verification method is illustrated in Figure 28. Instead of analyzing the validity of a property P directly on a quantized neural network (NN_q), we verify it on its rational approximation, denoted NN_r, due to the complexity of performing precise fixed-point arithmetic operations. This approximation allows us to leverage the well-established theory and efficient computational methods available for rational neural networks. The distance ε_p between the original NN_q and its rational counterpart NN_r is evaluated using standard p-norm distance metrics, including the L_1-norm ($||.||_1$), squared L_2-norm ($||.||_2^2$), and L_∞-norm ($||.||\infty$), combined with interval arithmetic to rigorously bound approximation errors.

Once the validity of P is established over NN_r, we further refine our analysis by introducing a perturbed version of the rational neural network, denoted $\widetilde{NN}_r$. This perturbed model is constructed by adding bounded, small perturbations to the output of NN_r, simulating potential deviations that may arise from quantization errors or other uncertainties. Our objective is to determine the largest admissible perturbation $\delta_{\max,p}$ that can be applied to NN_r while still preserving the validity of P over $\widetilde{NN}_r$. Crucially, this perturbation bound must be sufficiently large to encompass the original NN_q, ensuring that the property holds for the quantized model as well.

To establish formal guarantees, we compare the computed approximation error ε_p (the distance between NN_q and NN_r) with the maximum allowable perturbation $\delta_{\max,p}$. If $\varepsilon_p \leqslant \delta_{\max,p}$, we can conclusively assert that the output set of NN_q satisfies P, as the quantization-induced deviations remain within the tolerance that preserves the property. This approach provides a robust and scalable framework for verifying quantized neural networks while circumventing the computational intractability of direct analysis on low-precision arithmetic.

We explain the soundness of our method by introducing the concept of δ-robustness and demonstrating its role in verifying QNN using set theory.

δ-Robustness over Set-based Theory

Let P be a formal property over a set $S \subseteq \mathbb{F}^m$, ie. it can defined either as a subset of S or as a predicate over the output space: $P : \mathbb{F}^m \to \mathbb{B}$. We denote by $S \models P$ the validity

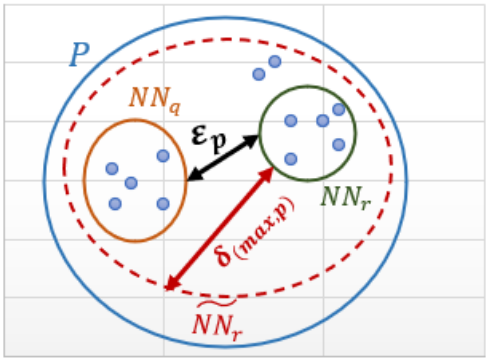


Figure 28: General View of Verifying QNN

of the property P for the set S: $\forall s \in S, P(s)$ holds.

$$\forall x_q \in \mathbb{F}^n, NN_q(x_q) \models P \tag{6.1}$$

$$\forall x_r \in \mathbb{Q}^n, NN_r(x_r) \models P \tag{6.2}$$

We aim to prove that P holds for the quantized neural network NN_q (cf. Eq. (6.1)).

However, instead of analyzing (6.1) and NN_q, we preferred to work on an approximated rational version of NN_q, denoted NN_r.

There is a notion of distance between two neural networks that are intended to *compute the same function*. In our case, we utilize the p-norm distance to define a maximum bounded pointwise distance, ε_p, between their outputs, as given in Definition 6.1.

Definition 6.1 (*NN* distance). *For all* $NN_1, NN_2 : \mathbb{R}^n \to \mathbb{R}^m$, *for all input set* $I \subseteq \mathbb{R}^n$, *let* $\varepsilon_p \in \mathbb{R}^+$ *s.t*

$\varepsilon_p = \max_{x \in I} \left(||NN_2(x) - NN_1(x)||_p \right)$.

The notion of δ-robustness consists of adding bounded perturbations to the output set of a neural network, by creating a new perturbed output set that always satisfies the property P. It is important to ensure that the difference between the initial and perturbed output sets is always limited by a threshold δ (as defined in Definition 6.2). This threshold gradually increases until it reaches the maximum threshold $\delta_{\max}$ as specified in Definition 6.3.

Definition 6.2 (δ-robust property). *A property* P *is* δ-robust *for a neural network* $NN : \mathbb{R}^n \to \mathbb{R}^m$ *and an input set* $I \subseteq \mathbb{R}^n$, *iff* $\forall x \in I, \forall o \in \mathbb{R}^m, ||NN(x) - o||_p < \delta \implies o \models P$.

Remark 6.1. *The special case* $o = NN(x)$ *denotes the validity of* P *for* NN. *This is a necessary condition for* δ-*robustness:* $\forall x \in \mathbb{R}^n, NN(x) \models P$. *In our case, we first verify Eq.* (6.2).

Remark 6.2 (Monotonicity). *δ-robustness is a monotonic property: $\forall \delta_1, \delta_2 \in \mathbb{R}^+$, with $\delta_1 < \delta_2$, a δ_2-robust property is also δ_1-robust.*

Let $R(P, NN) \in \mathbb{R}^+$ be the robustness of property P for neural network NN: the maximum δ such that P is δ-robust for NN is given in Def (6.3).

The max δ represents the maximum bounded perturbations that we can add to the initial neural network while still ensuring that the property P holds, as mentioned in Definition 6.3.

Definition 6.3 ($\max \delta$). *Let P be a property of $\mathbb{R}^m$, $NN : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network, $I \subseteq \mathbb{R}^n$ an input set for NN.*

We define $R(P, NN)$ as

$$\max \left\{ \delta \in \mathbb{R}^+ \;\middle|\; \forall x \in I, \forall o \in \mathbb{R}^m, ||NN(x) - o||_p < \delta \implies o \models P \right\}.$$

Proposition 6.1. *All elements of $\mathbb{R}^m$ that are less than $R(P, NN)$ distant from the image set of NN satisfy P:*

$$\left\{ \tilde{o} \in \mathbb{R}^m \,\middle|\, \exists x \in \mathbb{R}^n, ||NN(x) - \tilde{o}||_p \leqslant R(P, NN) \right\} \subseteq P \tag{6.3}$$

By applying the Def. 6.2 and Def. 6.3 on the main problem, defined in Eq. (6.1), we can state the main result :

Theorem 6.1 (Soundness). *For all $NN_1, NN_2 : \mathbb{R}^n \to \mathbb{R}^m$, for all input set $I \subseteq \mathbb{R}^n$, let ε_p and $R(P, NN_2) \subseteq \mathbb{R}^+$ be two scalars satisfying respectively Defs. 6.1 and 6.3.*

The condition :

$$\varepsilon_p \leqslant R(P, NN_2)$$

is a sufficient condition for $\forall x \in I, NN_1(x) \models P$.

Proof 6.1. *Since $\varepsilon_p \leqslant R(P, NN_2)$, we have*

$$\max_{x \in I} \left(||NN_2(x) - NN_1(x)||_p \right) \leqslant \max\{\delta \in \mathbb{R}^+ \,|\forall x \in I, \forall o \in \mathbb{R}^m, ||NN_2(x) - o||_p < \delta \implies o \models P\}$$

This implies the following property on image sets $NN_1(I)$ and $NN_2(I)$ with respect to P:

$$\{NN_1(x) \mid x \in I\} \subseteq \{y \in \mathbb{R}^m \mid x \in I, \varepsilon_p \leqslant ||NN_2(x) - y||_p \leqslant R(P, NN_2)\} \subseteq P.$$

6.3.2 Rational Approximation of QNN

In this contribution, we have chosen to verify P using an approximate rational neural network instead of directly verifying it on the QNN. This decision is based on the complexity of fixed-point arithmetic when performing elementary operations such as addition or multiplication. In fixed-point arithmetic, we need to consider shifts and avoid overflows for each operation result. However, by using rational arithmetic, we can avoid these calculations and simplify the process.

Quantized neural networks rely on fixed-point arithmetic with integer operations and shifts. Each variable, neural net weight, or bias is associated with a format $\langle w, f \rangle$, where w denotes the word width in bits and $f < w$ denotes the fractional part. Each NN_q can be expressed as a NN_r by representing all integers x in the format $\langle w, f \rangle$ with the rational scalar $x \times 2^{-f}$.

Similarly, all fixed-point arithmetic operations, including shifts, are replaced with their rational equivalents when encoding the NN_r using SMT, as defined in [PWB+20]. In this approach, the neural network is encoded by introducing intermediate variables to hold the results of the compositions of the inputs, hidden layers, and output in relation to the property to be verified.

6.3.3 SMT Verification of Formal Properties of QNNs through Set Theory

To ensure that the output sets of NN_q and NN_r include $\widetilde{NN}_r$, we first compute ε_p, which is the maximum distance between NN_q and NN_r using interval analysis and the p-norm. We then compare this value with the largest perturbation $\delta_{max,p}$ achieved by $\widetilde{NN}_r$ that can preserve the property P, representing the maximum p-norm distance between $\widetilde{NN}_r$ and NN_r.

In the following, we explain $repsilon_p$, $\delta_{max,p}$, and the relationship between them to verify QNN using set theory.

Characterization of the maximum p-norm Distance between the NN_q and NN_r

The approximation of NN_q by NN_r leads to numerical errors between the two functions. In order to over-approximate the distance ε_p between the two sets of images, as illustrated in Figure 28, we first need to select an appropriate norm and then use a method to compute this distance.
While static analysis techniques uses interval or affine arithmetic [GP15] to provide sound over-approximations of such distances,

We rely on interval propagation in our work as a sound yet incomplete method for computing ε_p, which represents the maximum p-norm distance between rational and fixed-point neural networks.

We firstly define the interval rational neural network as follows:

Definition 6.4 (Interval Rational Neural Network)**.** *Let $I_r \subseteq \mathbb{Q}$ be the set of rational interval s.t. $\forall x_r^{\#} \in I_r : x_r^{\#} = [lb_r, ub_r]$, with $lb_r, ub_r \in \mathbb{Q} \wedge lb_r \leqslant ub_r$*

Note that the interval rational neural network denoted $NN_r^{\#}$ is defined as follows:

$$\begin{aligned} NN_r^{\#} \colon I_r^n &\longrightarrow I_r^m \\ X_r^{\#} &\longmapsto Y_r^{\#} = NN_r^{\#}(X_r^{\#}) \end{aligned}$$

We define quantized interval neural Network as follows :

Definition 6.5 (Interval Quantized Neural Network)**.** *Let $I_q \subseteq \mathbf{F}$ be the set of fixed-point interval s.t. $\forall x_q^{\#} \in I_q : x_q^{\#} = [lb_q, ub_q]$, with $lb_q, ub_q \in \mathbf{F} \wedge lb_q \leqslant ub_q$*

We define interval quantized neural network denoted $NN_q^{\#}$ as follows:

$$\begin{aligned} NN_q^{\#} \colon I_q^n &\longrightarrow I_q^m \\ X_q^{\#} &\longmapsto Y_q^{\#} = NN_q^{\#}(X_q^{\#}) \end{aligned}$$

To apply interval propagation of rational and fixed-point neural networks using interval arithmetics, we firstly transform rational and fixed point scalar weights and bias to interval ones using *degenerate interval* where $ub = lb$.

According to the Def. 6.4 and Def. 6.5. we define ε_p which represent the maximum p-norm distance between NN_q and NN_r as follows:

Definition 6.6 (ε_p approximation)**.** *Let $X_q^\# \in I_q^n$ and $X_r^\# \in I_r^n$ be an interval vector of inputs of $NN_q^\#$ and $NN_r^\#$ respectively, $\varepsilon_p \in \mathbf{Q}$ be the maximum p-norm distance between the two interval neural networks.*

$$\varepsilon_p \triangleq (||NN_r^\#(x) - NN_q^\#(x)||_p^\#). \tag{6.4}$$

where $||.||_p^\#$ represents the interval p-norm distance computed using interval analysis, where ε_p represents the upper bound of the computed interval to represent the maximum p-norm distance.

In our experiments, we used the 1-norm, ∞-norm and squared 2-norm (also known as squared Euclidean distance) instead of the 2-norm. This decision was made because we encountered performance issues encoding the square root using SMTLIB.

Identification of the Robustness of Formal Properties through $\widetilde{NN}_r$

First, we check that the property holds for NN_r (cf. Eq. (6.2)) for all input domain I.

$$\forall x_r \in I, P(NN_r(x_r)) \tag{6.5}$$

δ_{max} is defined introducing $\widetilde{NN}_r$, adding (bounded) perturbations λ_i to NN_r.

$$\begin{aligned} \widetilde{NN}_r &\triangleq \{\forall x_r \in \mathbb{Q}^n, \exists \tilde{y}_r \in \mathbb{Q}^m, \lambda_i \in \mathbb{R}^m, s.t. : \tilde{y}_r = \widetilde{NN}_r(x_r) + \lambda_i\} \\ \delta_{\max,p} &\triangleq \max_{\delta} \forall x_r \in I, ||NN_r(x_r) - \widetilde{NN}_r(x_r)|| \leqslant \delta \implies P(\widetilde{NN}_r(x_r)) \end{aligned}$$

Since we want to apply Theorem. 6.1 and check that $\delta_{\max,p} > \varepsilon_p$, a sufficient condition is to replace δ with ε_p. Eventually, we verify the following property:

$$\forall x_r \in I, ||NN_r(x_r) - \widetilde{NN}_r(x_r)||_p \leqslant \varepsilon_p \implies P(\widetilde{NN}_r(x_r)) \tag{6.6}$$

The semantics of the neural network are formalized using SMT predicates with SMTLIB, The proof of the property (6.6) is performed with the SMT solver by searching for a model of its negation. An `unsat` result ensures its validity: P holds for NN_q.

6.4 Experimentations and Evaluation

6.4.1 Setup and Configuration

We implemented our tool using Python 3, utilizing the HIGHWAY-ENV [Leu18] and Stablebaseline3 [RHG+21] libraries for the highway environment and trained DQN model, respectively. For model checking verification, we used SMTLib [BFT16] as an encoding SMT language and z3 [dMB08b] as the SMT solver to evaluate the experiments.
We performed all experiments on an Apple M1 Ultra with 20 CPU cores (16 performance and 4 efficiency) and 128 GB of RAM.
We quantized the MLP policy function into 13 versions with different word-width formats, ranging from 8 to 21 bits using the Post-training quantization (PTQ) algorithm. This

algorithm converts a pre-trained floating-point neural network model into a fixed-point network without re-training the original NN model. We have set the integer part to be 4 bits, as defined in [NHP+21, Eq.(1), page 8].This guarantees that the highest value in the operations conducted on real neural networks can be expressed using 3 bits, along with an additional 1 bit for the sign.

HIGHWAY-ENV configuration. To evaluate our approach, we utilized HIGHWAY-ENV [Leu18] with DQN reinforcement learning algorithms [HVP+18] serving as the controller for the autonomous/ego vehicle (EV).
The DQN agent predicts the appropriate action for each observation using a policy function that utilizes a multilayer Perceptron (MLP) architecture with 2 hidden layers of 80 neurones and ReLu activation function as defined in Table 2. All experiments are publicly available [1].

MLP Policy Function	Input layer : [5,5] hidden layer : [80,80] activation function: ReLU output number : 5

Table 2: Configuration of MLP policy function

6.4.2 Results and Evaluation

As defined in § 6.3.1, our objective for the formal verification of QNN is to demonstrate that the output set of $\widetilde{NN}_r$ includes the output sets of NN_r and NN_q in order to verify the property φ_1. we define in Listing 1.6, the encoding of $\widetilde{NN}_r$ using the notion of δ-robustness with 1-norm, and the SMT predicates P_i using SMTLIB [BFT16] as follow :

```
declaration of variables
            .....
encoding of rational neural network
            .....
(declare-fun lb1 () Real) ...
(declare-fun epsilon1 () Real)
(declare-fun delta_p1  () Real)
            ...
(assert (= ypr1 (+ yr1  lb1)))
(assert (= ypr2 (+ yr2  lb2)))
(assert (= ypr3 (+ yr3  lb3)))
(assert (= epsilon1  0.00169097))
(assert (= delta_p1 (+ (abs lb1) (abs lb2) (abs lb3))))
(assert (<= delta_p1 epsilon1))
(assert (= p0 1.0))
(assert (= x0 1.0))
```

[1] https://gitfront.io/r/Wahiba/XT2hSmq47QPp/AV-QNN-Verif/

```
16 (assert (= y0 0.33))
17 (assert (and (>= vx0 0.2875) (<= vx0 0.3125)))
18 ...
19 (assert (or (>= ypr0 ypr4 ) (>= ypr1 ypr4) ...
```

Listing 6.1: SMTLIB encoding of $\widetilde{NN}_r$ and the specification using 1-norm.

Firstly, we declare the inputs, outputs, and neurons by layer. Then, we encode the rational neural network NN_r (lines 0-3). Next, we declare the perturbation variables added to NN_r as epsilon1 and delta_p1, which refer to ε_1 and $\delta_{\max,1}$ respectively, as defined in §. 6.3.3 (lines 4-7).

Next, we represent the encoding of $\widetilde{NN}_r$ with bounded perturbations $lb1, lb2,$ and $lb3$ (lines 8-10).

Then we bound ε_1, the distance between NN_q and NN_r (line 11). As described in §. 6.3.3. Then the 1-norm of the perturbation is determined (line 12). Lastly, we encode the condition on $\delta_{\max,1}$ (line 13) and the negation of φ_1, denoted $\neg\varphi_1$, using the perturbed rational neural network $\widetilde{NN}_r$ (lines 14-19).

Evaluation

In a first phase, we evaluated our verification method on the *running example* defined in Chapter 5, § 5.4.1 using SMT solver z3. We chose z3 over other SMT verification tools for neural networks, such as Marabou [WIZ+24], because z3 is not limited to linear constraints like Marabou. This allows us to explore both linear and nonlinear activation functions, squared Euclidean norms, and even complex properties. This flexibility enhances our framework's ability to integrate different methods, improving the efficiency of our approach. We firstly performed this verification on the approximation rational versions (NN_r) to verify the Eq. (6.5), using various bit formats from 8 to 21 bits. The property φ_1 is satisfied by the approximate rational version NN_r. This validation allows us to proceed with the approach and verify the perturbed rational version $\widetilde{NN}_r$ as illustrated in Table 3. In these experiments, We compared the ε_p computed using the interval 1-norm, squared Euclidean distance (2-norm), and the interval ∞-norm, denoted as $||\cdot||_1$, $||\cdot||_2^2$, and $||\cdot||_\infty$ respectively for 14 versions of NN_q. In the table, each norm is associated to a status – whether the property was proved or not according to Eq. (6.6) – and the computed δ-robustness criteria: the maximum allowable distance between NN_q and NN_r. Table 3 illustrates the SMT verification results of Eq. (6.6) when using both the ∞-norm and the 1-norm, as well as the maximum allowable distance ϵ_p between the NN_q and NN_r for the property φ_1.

φ_1 is satisfied for $\widetilde{NN}_r$'s 20 th through 21st word-width versions when using the ∞-norm. However, using the 1-norm, φ_1 is satisfied for the 19th through 21st word-width versions of $\widetilde{NN}_r$ with an ε_1 value of 0.0430. Unfortunately, the 2-norm could not satisfy φ_1 for all $\widetilde{NN}_r$ versions due to its complexity. The squared Euclidean distance presents challenges in SMT verification because it involves non-linear quadratic constraints, which are computationally

Table 3: Results of SMT verification for φ_1 across different word-widths of $\widetilde{NN}_r$ through $||\cdot||_1$, $||\cdot||_2^2$ amd $||\cdot||_\infty$ norms. St. stands for status.

width bit	Fraction bit	$\|\|\cdot\|\|_1$		$\|\|\cdot\|\|_2^2$		$\|\|\cdot\|\|_\infty$	
		st.	ε_1	st.	ε_2	st.	ε_∞
8	4	✗	32.9120	✗	169.6407	✗	28.6142
9	5	✗	25.9961	✗	113.4889	✗	23.5460
10	6	✗	15.7948	✗	37.7527	✗	13.4192
11	7	✗	8.7454	✗	10.5711	✗	7.0012
12	8	✗	4.7008	✗	3.1157	✗	3.8150
13	9	✗	2.5821	✗	0.8923	✗	2.0224
14	10	✗	1.2985	✗	0.2201	✗	0.9973
15	11	✗	0.7046	✗	0.0630	✗	0.5302
16	12	✗	0.3389	⊥	0.0148	✗	0.2579
17	13	✗	0.1645	⊥	0.0035	✗	0.1263
18	14	✗	0.0849	⊥	0.0009	✗	0.0646
19	15	✓	0.0430	⊥	0.0002	✗	0.0320
20	16	✓	0.0210	⊥	0.0001	✓	0.0165
21	17	✓	0.0108	⊥	0.000006	✓	0.0081

more difficult to manage than linear ones. SMT solvers struggle with quadratic terms, leading to issues such as undecidability and a significant increase in search space, especially when applied to large neural networks. [NPSS10]. The cases in which the property is violated are related to the large value of the maximum allowable distance, ε_p computed using interval propagation with interval arithmetis. This large value does not satisfy Eq. (6.6) and results in counterexamples for φ_1, given a specified word bit width.

We then applied our approach to another requirement that refers to the `slower` action denoted *TRQ2* and denoted φ_2 which is defined in Chapter 5, §5.5.

In Table 4, we observe that φ_2 is satisfied according to Eq. (6.6) for the word widths of NN_q ranging from 18 to 21 bits when evaluating the 1-norm, squared 2-norm, and ∞-norm using interval propagation to compute ε_p. However, φ_2 is violated for word widths ranging from 8 to 17 bits due to the overapproximation in interval propagation, which results in large values of ε_p.

The results illustrated in Table 4 and Table 3 demonstrate the applicability of the approach across several word-width versions of NN_q. Combining set-distance computation through p-norm ε_p using interval propagation and SMT solving, we were able to validate exhaustively the validity of these two abstract scenarios defined formally in φ_1 and φ_2. Our approach performs better on medium to large word-width quantized neural networks, ranging from 19 to 21 word widths for φ_1 and from 18 to 21 word widths for φ_2, where the property is successfully satisfied. However, it has limitations when dealing with small word-width quantized versions due to the overapproximation of the maximum p-norm

Table 4: Results of SMT verification for φ_2 across different word-widths of $\widetilde{NN}_r$ through $||\cdot||_1$, $||\cdot||_2^2$ amd $||\cdot||_\infty$ norms. St. stands for status.

width bit	Fraction bit	$\|\|\cdot\|\|_1$		$\|\|\cdot\|\|_2^2$		$\|\|\cdot\|\|_\infty$	
		st.	ε_1	st.	ε_2	st.	ε_∞
8	4	✗	36.5442	✗	134.6583	✗	19.5401
9	5	✗	29.1435	✗	86.0401	✗	15.4381
10	6	✗	21.0094	✗	45.0675	✗	10.6069
11	7	✗	11.5460	✗	13.6545	✗	5.9140
12	8	✗	5.8403	✗	3.5051	✗	3.0766
13	9	✗	3.0156	✗	0.9321	✗	1.5405
14	10	✗	1.5256	✗	0.2377	✗	0.7696
15	11	✗	0.7499	✗	0.0576	✗	0.3949
16	12	✗	0.3867	✗	0.0153	✗	0.1947
17	13	✗	0.1843	✗	0.0035	✗	0.0946
18	14	✓	0.0948	✓	0.0009	✓	0.0490
19	15	✓	0.0461	✓	0.0002	✓	0.0240
20	16	✓	0.0246	✓	0.0001	✓	0.0126
21	17	✓	0.0119	✓	0.0000	✓	0.0060

distance ε_p, where the property is violated. The value of ε_p decreases as the word-width format increases. This is expected since the rational approximation representation of NN_r closely resembles its corresponding version in fixed-point arithmetic when ε_p becomes smaller, but significantly deviates when ε_p becomes larger.

6.4.3 Discussion

According to the tables above defined in §6.4.2, we can confirm the validity of our approach as an alternative way to verify quantized neural network for decision-making property.

We evaluate our approach to answer the following research questions:

- **Q1:** What is the best norm for validating our approach regarding time, memory, and verification results for the requirements of autonomous vehicles (AVs)?
- **Q2:** How does our approach compare to the bit-vector/integer encoding using SMT solvers?
- **Q3:** Does quantization improve verification execution time and reduce memory usage?
- **Q4:** How can integrating parallelism in our approach improve it by dividing the main property into subproperties?

Q1 : The Best Norm for Validating our Approach

To define the best norm used, Table 5 and Table 6 show the verification time execution (in seconds) and memory usage (in MB) of each $\widetilde{NN}_r$ under the 1-norm, squared 2-norm, and ∞-norm for φ_1 and φ_2. "$\perp$ " and "-" indicate timeout and unavailable, respectively. We also considered the execution time for computing the maximum distance between NN_q and NN_r using interval propagation, which takes 1.25 (s) for both properties.

Execution time and memory usage for φ_1 in *TRQ1* are defined only for 1-norm and ∞-norm, as they validate the approach, unlike the squared euclidean norm as illustrated in Table 5. We observe that the ∞-norm outperforms the 1-norm in terms of execution time for all valid perturbed rational versions. However, when it comes to memory usage, we notice a negligible difference between the two norms. They have almost identical values when dealing with this problem size.

Table 5: Time (T) and memory (M) evaluation of our approach through $||\cdot||_1$ and $||\cdot||_\infty$ norms for φ_1

width bit	Frac bit	$\|\|\cdot\|\|_1$		$\|\|\cdot\|\|_\infty$	
		T(s)	M(mb)	T(s)	M(mb)
19	15	713.35	80.09	-	-
20	16	1025.39	81.13	649.68	80.65
21	17	1087.26	89.26	684.64	89.23

Similar results are observed for φ_2 in TRQ2. In Table 6, we observe that the squared Euclidean distance takes about five times the verification time of the 1-norm for all valid $\widetilde{NN}_r$ versions. Here, the ∞-norm is also significantly the most efficient.

Table 6: Time (T) and memory (M) evaluation of our approach through $||\cdot||_1$, $||\cdot||_2^2$ and $||\cdot||_\infty$ norms for φ_2

width bit	Frac. bit	$\|\|\cdot\|\|_1$		$\|\|\cdot\|\|_2^2$		$\|\|\cdot\|\|_\infty$	
		T(s)	M(mb)	T(s)	M(mb)	T(s)	M(mb)
18	14	229.74	77.09	9956.69	285.05	70.88	77.46
19	15	266.46	78.01	10314.47	288.95	81.07	78.21
20	16	279.05	79.06	1807.54	285.88	155.33	79.38
21	17	628.95	85.51	118176.07	365.44	482.88	84.85

We applied our approach to verify the Eq. (6.6) for φ_1 using Marabou, focusing solely on the 1-norm and ∞-norm. This is possible because we can encode these norms as linear equations, as defined in Table 7. We selected φ_1 for this verification because it requires significantly more time compared to φ_2 for both the 1-norm and ∞-norm, as shown in Tables 4 and 5.

Table 7: Time (T) and memory (M) evaluation of our approach with $||\cdot||_1$ and $||\cdot||_\infty$ norms for φ_1 using Marabou. St. stands for status.

width bit	Frac. bit	$\|\|\cdot\|\|_1$			$\|\|\cdot\|\|_\infty$		
		St.	T(s)	M(mb)	St.	T(s)	M(mb)
15	11	✗	0.068	0.323	✗	0.072	0.297
16	12	✗	0.068	0.323	✗	0.068	0.297
17	13	✗	0.069	0.297	✗	0.065	0.297
18	14	✗	0.070	0.297	✗	0.066	0.297
19	15	✓	0.067	0.297	✗	0.065	0.297
20	16	✓	0.069	0.297	✓	0.067	0.323
21	16	✓	0.069	0.297	✓	0.068	0.323

In Table 7, we observed the same verification results (St.) as defined in Table 3 for z3, but with respect to time verification and memory usage. Marabou demonstrates its efficiency in this context by incorporating several optimizations. These include the Reluplex [KBD+17] algorithm for efficiently handling ReLU activations, symbolic bound propagation to reduce the feasible search space and techniques that accelerate verification time while minimizing memory usage. When comparing the norms, we also note that the ∞-norm is less resource-intensive than the 1-norm for Marabou.

RQ1: We notice that the ∞-norm is the best norm for time and result verification for φ_1 and φ_2 using z3 and Marabou. However, for memory usage, there is no significant difference between the 1-norm and the ∞-norm. We have also observed that the exhaustive-scenario verification of $TRQ1$ takes more time and memory usage for verification compared to that of $TRQ2$. This is mainly due to the number of bounded range features. In $TRQ1$, we had 4 observed vehicles on the road with 7 bounded range features, whereas in $TRQ2$, we only had 3 vehicles with 5 bounded range features.

Q2: Our Approach VS Bitvector/Integer Encoding using SMT Solvers.

We compare our verification method using the ∞-norm with the integer encoding of the HIGHWAY-ENV MLP for the φ_1 property, using the z3 and *Yices2* [Dut14] SMT solvers.

We choose φ_1 because it requires significantly more verification time compared to φ_2, as shown in Table 5 and Table 6. We used z3 and *Yices2* because they refer to different SMT approaches. z3 represents the lazy approach that works more incrementally and reasons about the Boolean part of the formula first, only delving into the background theories when necessary [dMB08c, HBJ+14]. *Yices2* represents a hybrid approach that combines elements of both lazy and eager approaches. It differs from the lazy approach in that it considers the entire formula, including the background theories, during the verification process [HBJ+14].

According to Table 8, we observed that our method using z3 is more efficient in terms of verification time where it takes 1087.26 (s) as a maximum value to solve φ_1 However, with integer/bit-vector encoding using z3, and *Yices2* SMT solvers, it takes more than 24 hours.

Table 8: Verification time evaluation of our approach using z3 for φ_1 compared to SMT Solvers (z3, Yices2) using integer/bit-vector encoding, where $\perp$ denotes a verification time of over 24 hours.

width bits	Fractional bits	Our method using z3 (s)	SMT Solvers using Integer/bit-vector encoding (ms)
19	15	713.35	$\perp$
20	16	1025.39	$\perp$
21	17	1087.26	$\perp$

RQ2: Our approach performs better than the direct encoding of NN_q through integer/bit-vector theory using state of the art solvers such as z3, and *Yices2* for φ_1 .

Q3: Does quantization improve verification execution time and reduce memory usage?

To determine the effect of quantization on accelerating the verification process, we conduct a comparison between the original neural networks of the HIGHWAY-ENV benchmarks prior to quantization, using our proposed verification method. We apply the ∞-norm and choose the word width format that results in the least execution time and memory usage as defined in Table 9.

Table 9: Identification of the impact of quantization on the verification process

Before Quantization				
Properties		V. Results	Time (s)	Memory (mb)
φ_1		✓	1656.25	133.75
φ_2		✓	912.11	125.30
After Quantization				
Properties	p-norm	V. Results	Time (s)	Memory (mb)
φ_1	1-norm	✓	565.76	120.08
	∞-norm	✓	269.98	121.48
φ_2	1-norm	✓	430.21	79.77
	∞-norm	✓	160.07	79.62

We noticed that verifying QNNs using our method is sound, efficient, and less time-consuming and memory-usage compared to the SMT verification of the initial neural

network before quantization. This holds true for both norms used and the both properties. After quantization, the verification time is more than 5 times faster than the verification of the initial neural network. However, there is a consistent difference of approximately 50 (MB) in terms of memory usage.

> **RQ3**: Finally, we can assume that quantization accelerates the verification process and protects the verification results for medium to large word widths, even for large neural networks.

Q4: How Can Integrating Parallelism in our Approach, Improving it by Dividing the Main Property into Subproperties?

Since the decision-making safety properties are a combination of SMT predicates connected by the $\vee$ operator, we integrate parallel SMT verification into our approach to accelerate the verification process in terms of time and memory usage. We create subproperties from the main safety property and consider each subproperty as a standalone entity, running them separately in parallel to reduce the search space for the SMT engine.

Example 6.1. *φ_1 is defined as follows:*

$$\varphi_1 : \forall x \in D_1, D \subseteq \mathbb{Q}^n, \exists y \in \mathbb{Q}^m, y = NN(x), s.t. \forall i \in \{0, \ldots, m\}, y_3 > y_i$$

We create 4 sub-properties from φ_1 as follow to be verified each one separatly in parallel:

$$\varphi_{1.1} : (a_i \leqslant x_i \leqslant b_i \wedge \cdots \wedge a_i \leqslant x_i \leqslant b_i), a_i, b_i \in D_1 \subseteq \mathbb{Q}^m \implies (y_3 > y_0) \tag{6.7}$$

$$\varphi_{1.2} : (a_i \leqslant x_i \leqslant b_i \wedge \cdots \wedge a_i \leqslant x_i \leqslant b_i), a_i, b_i \in D_1 \subseteq \mathbb{Q}^m \implies (y_3 > y_1). \tag{6.8}$$

$$\varphi_{1.3} : (a_i \leqslant x_i \leqslant b_i \wedge \cdots \wedge a_i \leqslant x_i \leqslant b_i), a_i, b_i \in D_1 \subseteq \mathbb{Q}^m \implies (y_3 > y_2). \tag{6.9}$$

$$\varphi_{1.4} : (a_i \leqslant x_i \leqslant b_i \wedge \cdots \wedge a_i \leqslant x_i \leqslant b_i), a_i, b_i \in D_1 \subseteq \mathbb{Q}^m \implies (y_3 > y_4). \tag{6.10}$$

We apply parallelism to all QNN formats; however, in Table 10, we outline the maximum values for time verification and memory usage for both the ∞-norm and 1-norm concerning the properties φ_1 and φ_2.

Table 10: Identification of the impact of parallelism on the SMT verification of QNNs using our method for decision-making properties φ_1 and φ_2

SMT Verification			Without Parallel		With Parallel	
Properties	p-norms	V.Results	T(s)	M(mb)	T(s)	M(mb)
φ_1	$\|\|.\|\|_1$	✓	595.45	59.95	153.46	42.2
	$\|\|.\|\|_\infty$	✓	193.95	59.23	84.24	43.05
φ_2	$\|\|.\|\|_1$	✓	565.76	79.07	0.13	39.38
	$\|\|.\|\|_\infty$	✓	314.97	78.16	0.12	39.29

We observed in Table 10 that parallel SMT verification accelerates execution time and reduces memory usage compared to ordinary SMT verification. Our experiments demonstrate a speedup of over 3.5 times in verification time, along with a reduction in memory usage for both the 1-norm and ∞-norm properties when using parallelism compared to ordinary verification.

> **RQ4:** We have noticed that verifying our approach using parallelism, by dividing the main property into sub-properties and running them separately in parallel, enhances efficiency, accelerates verification time, and reduces memory usage.

6.5 Summary

In this chapter, we presented an SMT verification method for Quantized Neural Networks (QNNs) that is based on set theory and rational approximations. Our approach abstracts quantization zones as geometric regions and approximates activation functions using computationally efficient rational functions, in contrast to traditional bit-vector or integer-based verification methods.

Experimental validation demonstrated a 3.2x speedup in verification time for typical QNN architectures, achieving near-complete coverage of safety properties.

In the next chapter, we will explain and detail a quantization method of ANNs based on formal methods and precision tuning, which involves analyzing a neural network that relies on the maximum output perturbation that satisfies the property. This perturbation serves as input for the precision tuning tool `Popinns` designed to identify the optimized QNN format for each neuron that garantees the safety property.

Part III

Design of QNNs by Preserving Their Safety Properties for Avionics

Chapter 7

Design of Quantization Method for ANNs using Precision Tuning and Formal Methods

7.1 Introduction

Neural networks are typically trained using high-precision floating-point arithmetic on server-class machines equipped with graphics processing units (GPUs). However, this high precision can be prohibitively expensive for resource-constrained embedded systems [LJVD23b]. To enhance efficiency in terms of area, latency, or memory usage, trained neural networks are often converted to lower-precision numerical formats,

such as fixed-point or reduced floating-point precision, through a process known as precision tuning.

In this chapter, we propose a quantization method as an optimization technique for ANNs, focusing on precision tuning and formal methods that guarantee safety properties. Considering a real neural network as the input, we introduce a novel formal verification method that relies on the maximum output perturbation that satisfies the specified property. This perturbation serves as input for a precision tuning tool, `Popinns`, designed to identify the optimized quantized neural network (QNN) format for each neuron while ensuring compliance with the safety property.

We explain the proposed method in detail, outlining several steps, the underlying theory, and finally discussing precision tuning in artificial neural networks (ANNs) and its role in verifying QNNs.

7.2 Process of designing QNN using formal methods and precision tuning tool

In this chapter, we propose a quantization method for ANNs based on formal methods and precision tuning. This approach aims to generate an optimized format for QNNs that guarantees safety properties.

Here, we take a trained Neural Network model (*NN*) as input and introduce a bounded perturbation to the output of the *NN* that satisfies the safety property.

The maximum perturbation we can achieve is denoted as $\delta'_{\max}$. Once $\delta'_{\max}$ is determined, it can be processed as an error in another precision tuning tool called `Popinns`, which identifies the optimized QNN format according to a threshold denoted as ϵ. Generally, we select an arbitrary error that represents the difference between the NN and its quantized version, with the aim of solving the generated fixed-point constraints to obtain the ideal optimized format. In our case, we define $\delta'_{\max}$ as the error ϵ. If we can identify an optimized format according to the generated constraints, we consider the QNN verified; otherwise, we fail to achieve quantization and cannot assume that the QNN is verified.

We outline our approach based on the steps illustrated in Figure 31 as follows:

- ***Inputs*** : We take as inputs a trained model of neural network denoted *NN* and a formal safety property *P*. In this dissertion, we evaluate our approach using Acas Xu Neural Networks and their properties [JK19].
- ***Step1:*** We verify the property *P* on the neural network using SMT.. If the property is violated, we consider it invalid for the neural network, and we cannot proceed to the next steps. Otherwise, we move on to ***Step 2***. In this step, we use Marabou as an SMT NN verification tool.
- ***Step2:*** After ensuring the satisfiability of *P* on *NN*, we aim to preserve it by introducing a bounded perturbation to the output of *NN*. We will reverify *P* with respect

to a maximum distance, $\delta'_{\max}$, between NN and $\widetilde{NN}$, using p-norms. This approach ensures that we always maintain the satisfiability of the property, which we refer to as δ-robustness. The output of ***Step 2*** is the $\delta'_{\max}$ that we can achieve to preserve the satisfiability of the safety property as illustrated in Figure 29

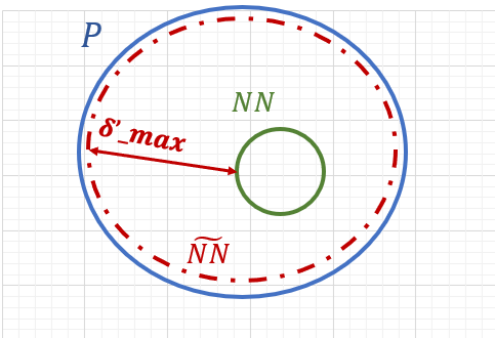


Figure 29: Set-based overview of preserving the safety property using the notion of δ-robustness.

- ***Step3:*** We define the error $\delta'_{\max}$ as the difference ϵ between the NN_r and its generated quantized version, NN_q. This error serves as the input for the precision tuning tool, `Popinns` [BM24a]. Our goal in this step is to generate an optimized format precision for the NN_q that satisfies fixed-point constraints based on the specified error $\delta'_{\max}$, as illustrated in Figure 30.

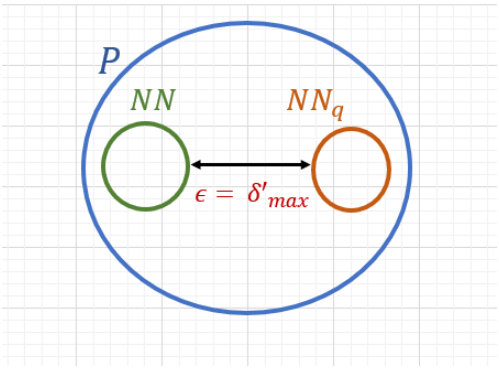


Figure 30: Set-based overview for Verifying QNN

The `Popinns` [BM24a] tool introduces a novel method for generating fixed-point code from a DNN written in TensorFlow 2.0, with formal guarantees on the output error bounds. This technique has been implemented in a prototype and is distinguished by its use of a formal semantics to model the propagation of round-off errors throughout the network. By leveraging this formal framework, `Popinns` can minimize the size of the fixed-point formats while ensuring that a user-defined error threshold is respected. This is achieved by solving a system of constraints derived from the network structure and precision analysis. We will explain the core of `Popinns` in §7.4.2 .

- ***Outputs:*** If we could obtain an optimized format of NN_q in accordance with $\delta'_{\max}$,

we can ensure that this NN_q also satisfies the safety property P. Conversely, if we are unable to quantize the neural network according to the specified threshold or error, $\delta'_{\max}$, we would be unable to verify the NN_q with our approach.

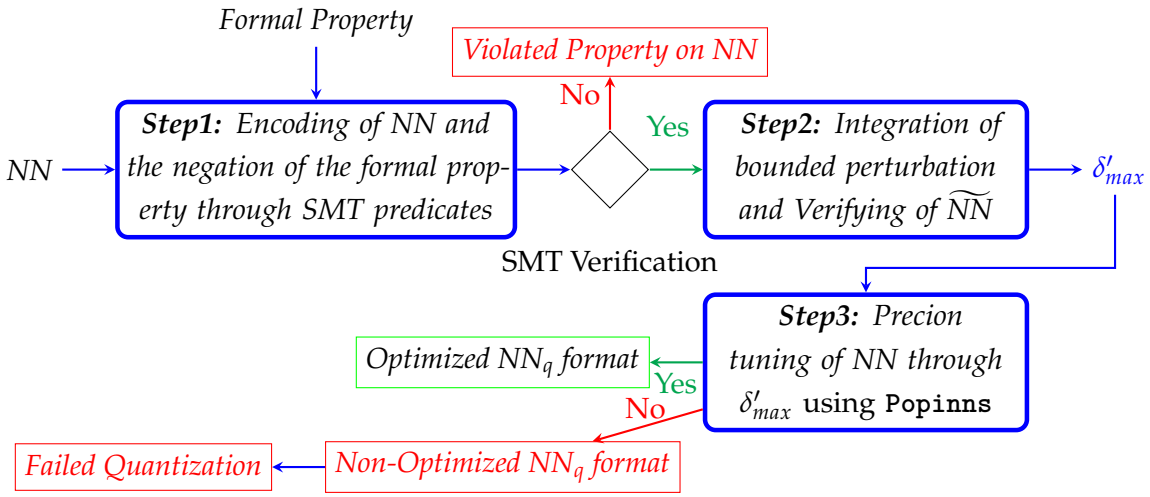


Figure 31: Overall Framework for Designing Quantization Method for Floating Point Neural Networks

7.3 Identification of the Maximum Perturbation Added to ANN for Preserving Safety Properties

We denote n and m the number of inputs and outputs of a neural network respectively. Let P be a formal property over a set $S \subseteq \mathbb{R}^m$, ie. it can defined either as a subset of S or as a predicate over the output space: $P : \mathbb{R}^m \to \mathbb{B}$. We denote by $S \models P$ the validity of the property P for the set S: $\forall s \in S, P(s)$ holds.

$$\forall x \in \mathbb{R}^n, NN(x) \models P \tag{7.1}$$

Firstly, We have to prove that P holds for the neural network NN (cf. Eq. (7.1)). The notion of δ-robustness consists of adding bounded perturbations to the output set of a neural network, by creating a new perturbed output set that always satisfies the property P. It is important to ensure that the difference between the initial and perturbed output sets is always limited by a threshold δ' (as defined in Definition 6.2 in chapter 6 §6.3.1). This threshold gradually increases until it reaches the maximum threshold $\delta'_{\max}$ as specified in Definition 6.3 in chapter 6 §6.3.1.

First, we check that the property holds for NN (cf. Eq. (7.1)) for all input domain I.

$$\forall x \in I, P(NN(x)) \tag{7.2}$$

We add bounded perturbations γ_i to the output of NN, to create the perturbed rational neural network denoted $\widetilde{NN}$ as defined in Eq. (7.3) δ'_{max} represents the maximum p-norm distance that the $\widetilde{NN}$ can reach to satisfy the property as defined in eq (7.4)

$$\widetilde{NN} \triangleq \{\forall x \in \mathbb{R}^n, \exists \tilde{y} \in \mathbb{R}^m, \gamma_i \in \mathbb{R}^m, s.t. : \tilde{y} = NN(x) + \gamma_i\} \tag{7.3}$$

$$\delta'_{\max,p} \triangleq \max_{\delta}{}' \forall x \in I, ||NN(x) - \widetilde{NN}(x)|| \leqslant \delta' \implies P(\widetilde{NN}(x)) \tag{7.4}$$

Since we want to apply Theorem. 6.1 (in chapter 6 §) and find the δ'_{max} that satisfy the property as defined in E.q. 7.5 :

$$\forall x \in I, ||NN(x) - \widetilde{NN}(x)||_p \leqslant \delta'_{max} \implies P(\widetilde{NN}(x)) \tag{7.5}$$

We used incremental verification to find δ'_{max} The semantics of the neural network are formalized using SMT predicates. The proof of the property (7.5) is performed with the SMT solver by searching for a model of its negation. An `unsat` result ensures the validity of the property.

7.4 Precision Tuning of ANNs

A further difficulty arises from the fact that DNNs are typically trained on desktop computers with high computational power before being deployed on target architectures with significantly lower processing capabilities. Consequently, it is essential to perform the necessary arithmetic conversion without degrading the network's performance. Precision tuning is one proposed solution to reduce the complexity of complex DNN architectures while minimally affecting the accuracy of the initial DNN.

7.4.1 Precision Tuning Vs. Quantization

Precision tuning [IM19] is a broad process that involves adjusting the numerical precision of neural network parameters and computations to improve efficiency while maintaining accuracy. This can include reducing the bit-width of floating-point representations [Ple17], such as moving from 32-bit to 16-bit floating-point formats. Quantization [GKD+22, NFA+, HMD15] is a specific form of precision tuning that typically involves mapping continuous floating-point values to a discrete set of lower-precision values, often integers, such as 8-bit fixed-point representations. While quantization generally implies discretization and is widely used to reduce memory and computational requirements, precision tuning encompasses both quantization and other methods of precision reduction that do not necessarily convert values to integer formats [J+18, NBHP+21, WMYY24a].

Quantization is a technique that significantly reduces the memory footprint. It involves reducing the number of bits used to code each model weight, so that the total memory footprint is reduced by the same factor. Several advantages are possible if operations are carried out using integer rather than floating-point formats. An important advantage is that integer operations require far fewer computations on most processor cores, including micro-controllers especially in cases where there is no FPUs available, so floating-point instructions have to be emulated in software, resulting in a significant overhead.

In this thesis, we use the precision tuning tool `Popinns` to quantize neural network controllers, such as the ACAS Xu network. Our contribution to the development of `Popinns` addresses a key limitation in its current design: at this stage, `Popinns` performs quantization based on arbitrary error thresholds provided by the user, without any formal guarantees regarding the preservation of the neural network's functional behavior. For instance, in the case of classification tasks, `Popinns` does not ensure that the classifications produced by the original floating-point network are preserved in the fixed-point implementation.

To overcome this, we propose a novel approach that identifies the maximum allowable error threshold denoted as δ'_{max}, that preserves the critical properties of the ACAS Xu network. By computing this threshold in advance and providing it as input to `Popinns`, we enable the tool to perform quantization in a way that ensures the functional correctness of the resulting fixed-point network. This targeted strategy improves the robustness of the quantized model and avoids the risks associated with selecting error thresholds in an arbitrary or uninformed manner.

In the following section, we will detail the architecture of `Popinns` and its main features.

7.4.2 Case Study: The `Popinns` Tool

In this section, we give an overview of the tool `Popinns` as shown in Figure 32. It is important to note that `Popinns` is not publicly available as an open-source tool, as it is still under restricted access and active development. However, we have collaborated closely with the authors of the tool to conduct our work and extend its capabilities for our purposes.

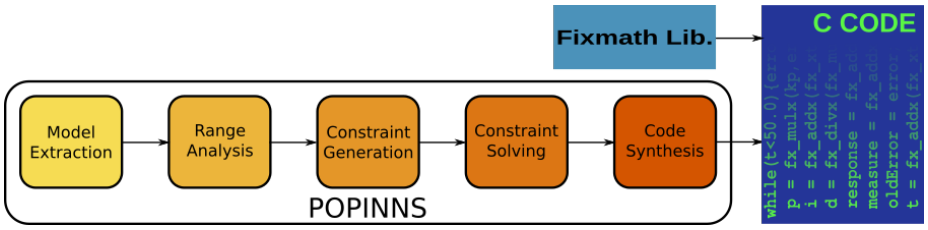


Figure 32: Workflow used by `Popinns` to synthesize fixed-point code for DNNs [BM24b].

The main steps of `Popinns` are the following: [BM24a, BM24b, BM24c]

1. *Model Extraction:* `Popinns` takes as input a Tensorflow model[1] and translates it into its internal representation. Currently, the layers accepted are `dense`, `conv2d`, `max_pooling2d`, `up_sampling2d` and `flatten`. The `ReLU` activation function is also handled by the tool.

   For example, the code below defines a model made of a convolutional layer followed by a dense layer with ReLU (the flatten layer translates the matrix resulting from the convolution into a vector.) This model takes as inputs images of size `height` × `width`= 16×16 and classify them in different classes, `numclass`= 6.

[1] https://www.tensorflow.org/

```
height = 16 ; width = 16 ;
 num_classes = 6
input_shape = (height, width, 1)
model = keras.Sequential(
    [
    keras.Input(shape=input_shape),
    layers.Conv2D(1, kernel_size=(3, 3)),
    layers.Flatten(),
    layers.Dense(num_classes,
                 activation="relu")
    ]
)
```

Once the model is trained, the following command is all that is needed to generate the code in fixed-point arithmetic.

```
threshold = 8
popinns(model,1,height,width,imgs,threshold)
```

In the sequence above `imgs` is an array containing several images of size `heightwidth`. It corresponds typically to a subset of the training set and is used to perform the dynamic range analysis. This dynamic analysis consists of running the DNN with a set of input data and taking, for each output, the join of the values obtained at each run. This gives the most significant bits of the values arising at each point of the model (this step is crucial to generate the system of constraints described in the next paragraph) Finally, `threshold` denotes the accuracy required for the fixed-point model which is set by the user. In our example, an accuracy of 2^{-8} is required, which means that the errors between the outputs of the original model and those of the fixed-point code synthesized by **`Popinns`** must be less than 2^{-8}.

2. *Range analysis:* Once the model is extracted, a range analysis is performed. In the current version of **`Popinns`**, this analysis is dynamic but we plan to make it static using affine forms [dFS04]. The dynamic analysis consists of running the DNN with a set of input data and taking, for each output, the join of the values obtained at each run. This gives an under-approximation of the possible values which is acceptable in practice.

3. *Constraint Generation:* The third step of **`Popinns`** is the generation of the constraints. The constraints are inequalities between linear expressions among integer variables and constants. They are not linear because they also contain implications to encode the min and max operations. The variables are the precision of the inputs of each layer as well as the precision in which each operation is carried out inside each layer.

   For constraint generation, **`Popinns`** consider a DNN made of ℓ layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{\ell-1}$, such that the inputs and outputs of each layer have a depth of one. Let $\mathcal{I}_k$ and $\mathcal{O}_k$ respectively be the number of inputs and outputs of $\mathcal{L}_k$, $0 \leqslant k < \ell$. Let x_j^k, $0 \leqslant j < \mathcal{I}_k$,

(resp. y_i^k, $0 \leqslant i < \mathcal{O}_k$) be the j^{th} input (resp. i^{th} output) of Layer $\mathcal{L}_k$, $0 \leqslant k < \ell$ and let $\varepsilon(x_j^k)$ and $\varepsilon(y_i^k)$ be the errors associated to these inputs and outputs (note that $\mathcal{O}_k = \mathcal{I}_{k+1}$ for $0 \leqslant k < \ell - 1$).

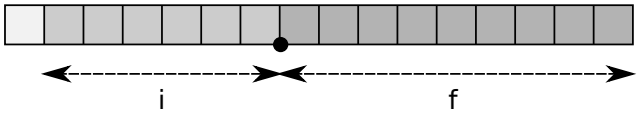


Figure 33: A fixed-point number in format $Q_{\mathsf{i},\mathsf{f}}$, with $\mathsf{i} = 6$ and $\mathsf{f} = 9$. The leftmost bit is used for the sign.

The unknowns of the system of constraints are the precisions (fractional sizes f of Figure 33) ${}^k_{x_j} \in \mathbb{Z}$ and ${}^k_{y_i} \in \mathbb{Z}$ on the inputs and outputs of each layer, i.e. $\varepsilon(x_j^k) \leqslant 2^{-{}^k_{x_j}}$, resp. $\varepsilon(y_i^k) \leqslant 2^{-{}^k_{y_i}}$, as well as the working precision ${}^k_i \in \mathbb{Z}$ of the neurons, $0 \leqslant i < \mathcal{O}_k$. In this way, `Popinns` only have integer constraints. Recall that this greatly simplifies the resolution by `Z3` compared to real or floating-point constraints.

Let us consider the case of fully connected (dense) layers in `Popinns`. For this type of layer, dedicated constraints are formulated to ensure that quantization and rounding errors introduced during the weighted summation neither cause overflow nor lead to unacceptable loss of numerical accuracy. In a typical fully connected layer k, each output neuron receives a set of input activations x_i^k and produces an output y_i^k by computing a sum of products with learned weights, optionally followed by a non-linear activation function such as ReLU.

The principle is that the internal sum for each neuron must be represented with a fixed-point format sufficient to accommodate the full dynamic range of the accumulated products, while each input activation must preserve sufficient precision to limit the propagation of roundoff errors. This balance ensures that all intermediate computations remain within safe numeric bounds and that the final output precision is achieved without exceeding the prescribed global error threshold given by the user.

Consequently, `Popinns` defines the internal format ${}^k_i$ for each output neuron such that it can store the desired output precision ${}^k_{y_i}$, together with the integer range required by the maximum product magnitude, plus additional bits to account for the accumulation of n summed products and an extra guard margin so that

$$ {}^k_i \geqslant {}^k_{y_i} + \mathsf{i}_\Psi + \lfloor \log_2(n) \rfloor + 1 $$

where i_Ψ represents the integer range of the largest input activation, and the term $\lfloor \log_2(n) \rfloor$ accounts for the bit-growth due to the addition of n terms.

Similarly, each input activation must maintain a format wide enough to ensure that the product with its corresponding weight does not introduce excessive roundoff error. This is formalized by the following constraint.

$$ {}^k_{x_i} \geqslant {}^k_{y_i} + \mathsf{i}_{\Omega_i} + \lfloor \log_2(n) \rfloor + 1, \tag{7.6} $$

where i_{Ω_i} denotes the integer range of the largest absolute weight associated with neuron i. These static parameters are defined as shown

$$\Omega_i = \max_j |w_{ij}|, \quad \Psi = \max_j |x_j|, \quad \varepsilon_{w_i} = \max_j \varepsilon(w_{ij}), \quad \varepsilon_x = \max_j \varepsilon(x_j),$$

and the rounding errors fix the necessary fractional precision:

$$\overset{k}{i} = \lceil \log_2(\varepsilon_{w_i}) \rceil, \quad \overset{k}{x_i} = \lceil \log_2(\varepsilon_x) \rceil \ . \tag{7.7}$$

Combining equations (7.6) and (7.7), the final constraints for each output neuron y_i are given by

$$\left\{ \overset{k}{y_i} \leqslant \max\left(\mathrm{i}_\Psi - \overset{k}{i}, \mathrm{i}_{\Omega_i} - \overset{k}{x_i}, 1 - \overset{k}{i} - \overset{k}{x_i}, \overset{k}{i}\right) + \log_2(n) + 3 \right\} \ . \tag{7.8}$$

4. *Constraint solving:* The solution of the generated constraints is computed by the `z3` optimizing SMT solver [DMB08a]. To optimize the solution the solver needs a cost function and several relevant functions may be defined for this purpose as presented in the work of [BM22]. In **`Popinns`**, they minimize the total number of bits needed to represent the fractional parts of the fixed-point numbers. Then their cost function for a DNN N made of ℓ layers $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{\ell-1}$ is

$$\mathsf{cost}(N) = \sum_{0 \leqslant k < \ell} \left(\sum_{0 \leqslant j < \mathcal{I}_k} \overset{k}{x_j} + \sum_{0 \leqslant i < \mathcal{O}_k} \overset{k}{x_j} + \sum_{0 \leqslant i < \mathcal{O}_k} \overset{k}{i} \right) \ . \tag{7.9}$$

5. *Code synthesis:* The last step consists of synthesizing the fixed-point code implementing the DNN. **`Popinns`** uses the fixed-point formats found by the solver to synthesize a fixed-point `C` code relying on the `Fixmath` library[2] for the fixed-point operations. A fixed-point number is represented by a k-bit signed integer X, combined with a scale factor $f \in \mathcal{Z}$. Then X represents the real value x defined by

$$x = X \cdot 2^{-f} \ . \tag{7.10}$$

We denote $Q_{i,f}$ the format of a given fixed-point number represented using a k-bit integer associated to a scaling factor f , where $k = i + f$. In **`Popinns`**, the formats $Q_{i,f}$ of the fixed-point numbers and variables are determined thanks to the range analysis which yields the sizes i of the integer part and thanks to the solution to the system of constraints which gives the sizes f of the fractional parts.

7.5 Summary

In this chapter, we highlighted the process of the quantization method based on formal methods and precision tuning . We then proved the theory behind it, which is based on δ-robustness, set-based theory, SMT verification, and precision tuning.

[2] https://savannah.nongnu.org/projects/fixmath/

Finally, we explained the role of precision tuning and how we integrate it into our proposed method.

We also discussed `Popinns` as the precision tuning tool we used to Verify QNNs through the specified threshold ϵ.

In the next chapter, we will evaluate our approach and demonstrate the results using the Acas Xu NNs [JK19]. We will explore its properties and delve into each step explained in the general process.

Chapter 8

Evaluation of Quantization Method for Acas Xu

8.1 Introduction

In this chapter, we evaluate the experiments conducted using our quantization proposed method defined in the previous chapter. Additionally, we will discuss the results of $\delta'_{\max}$ and the precision tuning of *NN* in Acas Xu Neural Networks, examining its properties as a case study.

8.2 Case Study : Acas Xu

The ACAS Xu system (Airborne Collision Avoidance System for Unmanned Aircraft) is a neural network-based decision-making system designed to help unmanned aerial vehicles (UAVs) avoid mid-air collisions. This system represents a crucial advancement in automated aviation safety.

8.2.1 Acas Xu Neural Networks

Creating the initial ACAS Xu state-action lookup tables required an immense storage capacity, amounting to hundreds of gigabytes of floating-point data [KBD+22, SB23]. Such storage demands could limit practical deployment. To address this, downsampling techniques were applied, reducing the table size to 2 GB. However, this reduced size might still be too large, particularly for certified avionics systems used in UASs [OLS+19]. As a solution, neural networks were introduced to further compress the data.

This approach involved 45 individual neural networks, each designed with six layers, ReLU activation functions, and 50 neurons per layer [KBD+22]. Each neural network is associated with a specific pair (λ, β), where λ indicates the prior advisory ($a_{prev} \in \{COC, WL, WR, SL, SR\}$, as shown in Table 11, and β corresponds to the time to loss of vertical separation ($\tau \in \{0, 1, 5, 10, 20, 40, 60, 80, 100\}$) in seconds. For instance, $N_{2,3}$ represents the network associated with the previous advisory WL and a time to loss of vertical separation of 5 seconds.

The inputs to these networks consist of state variables $(\varphi, \theta, \psi, v_{own}, v_{int})$, as detailed in Table 11, and the outputs are turning advisories listed in Table 11. The selection of a specific neural network can change during runtime, depending on the time to loss of vertical separation and the last command issued. This approach minimizes decision oscillations near boundaries by encouraging consistency with the previous advisory. When both aircraft remain at the same altitude, τ is always zero, limiting the selection to five networks based solely on the prior command. Otherwise, all 45 networks are available for switching.

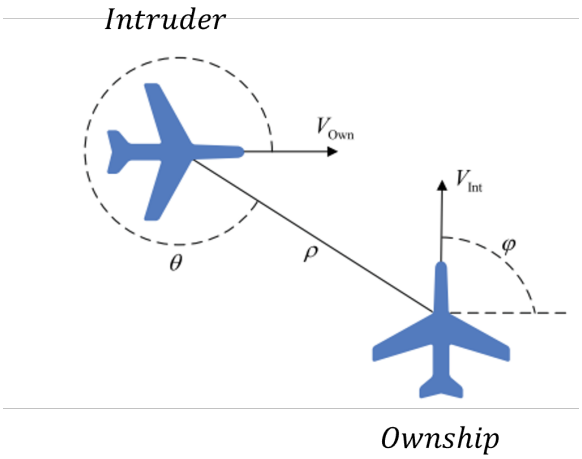


Figure 34: Geometry for ACAS Xu horizontal logic table [KBD+17]

Table 11: Description of the inputs and actions performed by Acas Xu NNs

Inputs	Units	Description	Action	Description
φ	ft	distance between ownship and intruder	SL	strong left turn at 3.0 deg/s
ϕ	rad	angle to intruder w.r.t ownship heading	WL	weak left turn at 1.5 deg/s
ψ	rad	heading of intruder w.r.t ownship v_{own}	COC	clear of conflict (do nothing)
v_{int}	ft/s	ownship velocity	WR	weakright turn at 1.5 deg/s
	ft/s	intruder velocity	SR	strong right turn at 3.0 deg/s

8.2.2 Acas Xu properties

The ACAS Xu system has been designed with specific formal properties to ensure its reliability, robustness, and safety in collision avoidance scenarios. These properties, often discussed in verification studies, outline criteria for evaluating the behavior of the neural networks under various conditions. Collectively, the 10 properties ensure that ACAS Xu operates reliably, robustly, and safely, leveraging its 45 networks to handle complex, real-world scenarios in a modular and efficient manner, as defined in Table 12.

Table 12: Summary of Property Descriptions

Property	Description	Tested On	Desired Output
φ_1	If the intruder is distant and slower than ownship, the score of a COC advisory will always be below a certain fixed threshold.	All 45 networks	COC score $\leqslant 1500$
φ_2	If the intruder is distant and slower than ownship, the score of a COC advisory will never be maximal.	Nx,y for all $x \geqslant 2$ and y	COC score is not maximal
φ_3	If the intruder is directly ahead and moving towards the ownship, the score for COC will not be minimal.	All networks except N1,7, N1,8, N1,9	COC score is not minimal
φ_4	If the intruder is directly ahead and moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.	All networks except N1,7, N1,8, N1,9	COC score is not minimal
φ_5	If the intruder is near and approaching from the left, the network advises “strong right”.	N1,1	"Strong right" score is minimal
φ_6	If the intruder is sufficiently far away, the network advises COC.	N1,1	COC score is minimal
φ_7	If vertical separation is large, the network will never advise a strong turn.	N1,9	“Strong right” and “strong left” are never minimal
φ_8	For a large vertical separation and a previous “weak left” advisory, the network will either output COC or continue advising “weak left”.	N2,9	“Weak left” score or COC score is minimal
φ_9	Even if the previous advisory was “weak right”, the presence of a nearby intruder will cause the network to output a “strong left” advisory instead.	N3,3	“Strong left” score is minimal
φ_{10}	For a far away intruder, the network advises COC.	N4,5	COC score is minimal

8.3 Preserving Acas Xu Properties

Setup

We conducted our experiments using 45 neural networks from Acas Xu in ONNX format. For formal verification, we employed Marabou as an SMT solver for neural networks, utilizing its Python version to integrate the notion of δ'_{max}, ensuring compatibility with Python 3 and Marabou.

All experiments were performed on an Apple M1 Ultra equipped with 20 CPU cores and 128 GB of RAM.

To process with the precision tuning tool `Popinns`, which works with TensorFlow models, we convert the ONNX models into TensorFlow models.

8.3.1 Results of SMT Verification of Acas Xu ANNs using Marabou

The table 13 illustrates the δ'_{max} of each neural network according to its verified property. **SAT** refers to a violated property, **TO** indicates a timeout, and $\perp$ means that the property does not apply to the corresponding neural network.

All the properties that have a $\delta'_{max} > 0$ are already verified with $\delta'_{max} = 0$ and are satisfiable.

According to Table 13, we observe that φ_1 has the highest δ'_{max} , while the minimum δ'_{max} value belongs to φ_6 for the first ACAS Xu NNs.

There are some properties that are not verified in their corresponding NNs, which return SAT with a counterexample (e.g., φ_3 and φ_4 for "ACASXU_1_7").

From 0 to 10^{-6}, we observed that the *NN* is robust up to this distance, allowing for the verification of several properties simultaneously. For example, for the first Acas Xu NN, we can assume that 10^{-6} verifies the properties φ_1, φ_2, φ_3, φ_4, φ_5, and φ_6, as it represents the minimum distance δ'max covered.

Table 13: Results of $\delta'_{\max}$ According to Acas Xu NNs and its properties

ACAS Xu NNs	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	φ_9	φ_{10}
ACASXU_1_1	4.008799	SAT	0.0010	0.0012	0.0010	1e-06				
ACASXU_1_2	4.00789	SAT	0.0015	0.0011						
ACASXU_1_3	4.00699	SAT	0.0012	0.0030						
ACASXU_1_4	4.0068	SAT	0.0058	0.0058						
ACASXU_1_5	4.0062	SAT	0.0060	0.0057						
ACASXU_1_6	4.0074	SAT	0.0028	0.0027						
ACASXU_1_7	4.0097	5e-06	SAT	SAT						
ACASXU_1_8	4.0083	1e-05	SAT	SAT						
ACASXU_1_9	4.0097	1e-05	SAT	SAT						
ACASXU_2_1	3.9263	SAT	0.01	0.0255						
ACASXU_2_2	3.7594	SAT	2.5e-05	0.020						
ACASXU_2_3	3.9318	SAT	0.0123	0.0203						
ACASXU_2_4	3.963	SAT	0.014	0.0236						
ACASXU_2_5	3.7594	SAT	0.014	0.0225						
ACASXU_2_6	3.8847	SAT	0.0217	0.0225						
ACASXU_2_7	3.6812	SAT	0.0216	0.0227						
ACASXU_2_8	3.5343	SAT	0.0215	0.0207						
ACASXU_2_9	3.7412	SAT	0.0202	0.0196				SAT		
ACASXU_3_1	3.75	SAT	0.0269	0.045						
ACASXU_3_2	3.9453	SAT	0.0001	0.0408						
ACASXU_3_3	3.9453	SAT	0.0297	0.0451					0.0025	
ACASXU_3_4	3.916	SAT	0.0258	0.0299						
ACASXU_3_5	3.9062	SAT	0.0225	0.0223						

ACAS Xu NNs	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	φ_9	φ_{10}
ACASXU_3_6	$\geqslant$ 3.0	SAT	0.0212	0.0232						
ACASXU_3_7	3.75	SAT	0.02324	0.0217						
ACASXU_3_8	3.75	SAT	0.01719	0.0217						
ACASXU_3_9	$\geqslant$ 2.1709	SAT	0.02375	0.0241						
ACASXU_4_1	3.75	SAT	0.005	0.0211						
ACASXU_4_2	3.9553	SAT	1e-05	0.0234						
ACASXU_4_3	3.4375	SAT	0.01875	0.0246						
ACASXU_4_4	3.4375	SAT	0.0231	0.0122						
ACASXU_4_5	3.4375	SAT	0.025	0.0262						0.00375
ACASXU_4_6	1.375	SAT	0.0208	0.0206						
ACASXU_4_7	3.4375	SAT	0.021094	0.021						
ACASXU_4_8	3.04	SAT	0.0205	0.0221						
ACASXU_4_9	1.1387	SAT	0.0202	0.02						
ACASXU_5_1	3.9101	SAT	0.006	0.0524						
ACASXU_5_2	3.7812	SAT	0.0175	0.0459						
ACASXU_5_3	3.9675	SAT	0.0268	0.044						
ACASXU_5_4	3.9439	SAT	0.0233	0.0244						
ACASXU_5_5	3.8672	SAT	0.0279	0.0266						
ACASXU_5_6	3.7812	SAT	0.0232	0.0223						
ACASXU_5_7	2.6641	SAT	0.0221	0.0211						
ACASXU_5_8	2.75	SAT	0.0197	0.0207						
ACASXU_5_9	3.4375	SAT	0.020713	0.0195						

8.3.2 Identification of the Maximum Perturbation for Preserving Acas Xu Properties

From Table 13, we extract only the minimum δ'_{max} of all the properties applied to each Acas Xu NNs. We define the adequate bits for each one as the threshold we aim to provide to the `Popinns` tool for generating the optimized format. As outlined in Table 14 The number of bits is calculating using the Eq. 8.1 :

$$bits = ceil(abs(log(\delta'_{max})/log(2))) \tag{8.1}$$

Table 14: Minimum δ'_{max} of all the properties applied to Acas Xu NNs.

Acas Xu NNs (onnx)	properties	δ'_{max}	bits
ACASXU_v2a_1_1	φ_1 φ_2 φ_3 φ_4 φ_5 φ_6	1e-06	20
ACASXU_v2a_1_2	φ_1 φ_2 φ_3 φ_4	0.0011	10
ACASXU_v2a_1_3	φ_1 φ_2 φ_3 φ_4	0.0012	10
ACASXU_v2a_1_4	φ_1 φ_2 φ_3 φ_4	0.0058	8
ACASXU_v2a_1_5	φ_1 φ_2 φ_3 φ_4	0.0057	8
ACASXU_v2a_1_6	φ_1 φ_2 φ_3 φ_4	0.0027	9
ACASXU_v2a_1_7	φ_1 φ_2 φ_3 φ_4	5e-06	18
ACASXU_v2a_1_8	φ_1 φ_2 φ_3 φ_4	1e-05	17
ACASXU_v2a_1_9	φ_1 φ_2 φ_3 φ_4 φ_7	1e-05	17
ACASXU_v2a_2_1	φ_1 φ_2 φ_3 φ_4	0.01	7
ACASXU_v2a_2_2	φ_1 φ_2 φ_3 φ_4	2.5e-05	16
ACASXU_v2a_2_3	φ_1 φ_2 φ_3 φ_4	0.0123	7
ACASXU_v2a_2_4	φ_1 φ_2 φ_3 φ_4	0.014	7
ACASXU_v2a_2_5	φ_1 φ_2 φ_3 φ_4	0.014	7
ACASXU_v2a_2_6	φ_1 φ_2 φ_3 φ_4	0.0217	6
ACASXU_v2a_2_7	φ_1 φ_2 φ_3 φ_4	0.0216	6
ACASXU_v2a_2_8	φ_1 φ_2 φ_3 φ_4	0.0207	6
ACASXU_v2a_2_9	φ_1 φ_2 φ_3 φ_4 φ_8	0.0196	6
ACASXU_v2a_3_1	φ_1 φ_2 φ_3 φ_4	0.0269	6
ACASXU_v2a_3_2	φ_1 φ_2 φ_3 φ_4	0.0001	14
ACASXU_v2a_3_3	φ_1 φ_2 φ_3 φ_4 φ_9	0.0025	9
ACASXU_v2a_3_4	φ_1 φ_2 φ_3 φ_4	0.0258	6
ACASXU_v2a_3_5	φ_1 φ_2 φ_3 φ_4	0.0223	6
ACASXU_v2a_3_6	φ_1 φ_2 φ_3 φ_4	0.0212	6
ACASXU_v2a_3_7	φ_1 φ_2 φ_3 φ_4	0.0217	6
ACASXU_v2a_3_8	φ_1 φ_2 φ_3 φ_4	0.01719	6
ACASXU_v2a_3_9	φ_1 φ_2 φ_3 φ_4	0.02375	6
ACASXU_v2a_4_1	φ_1 φ_2 φ_3 φ_4	0.005	8
ACASXU_v2a_4_2	φ_1 φ_2 φ_3 φ_4	1e-05	17

Acas Xu NNs (onnx)	properties	δ'_{max}	bits
ACASXU_v2a_4_3	φ_1 φ_2 φ_3 φ_4	0.01875	6
ACASXU_v2a_4_4	φ_1 φ_2 φ_3 φ_4	0.0122	7
ACASXU_v2a_4_5	φ_1 φ_2 φ_3 φ_4 φ_{10}	0.00375	9
ACASXU_v2a_4_6	φ_1 φ_2 φ_3 φ_4	0.0206	6
ACASXU_v2a_4_7	φ_1 φ_2 φ_3 φ_4	0.021	6
ACASXU_v2a_4_8	φ_1 φ_2 φ_3 φ_4	0.0205	6
ACASXU_v2a_4_9	φ_1 φ_2 φ_3 φ_4	0.02	6
ACASXU_v2a_5_1	φ_1 φ_2 φ_3 φ_4	0.006	8
ACASXU_v2a_5_2	φ_1 φ_2 φ_3 φ_4	0.0175	6
ACASXU_v2a_5_3	φ_1 φ_2 φ_3 φ_4	0.0268	6
ACASXU_v2a_5_4	φ_1 φ_2 φ_3 φ_4	0.0233	6
ACASXU_v2a_5_5	φ_1 φ_2 φ_3 φ_4	0.0266	6
ACASXU_v2a_5_6	φ_1 φ_2 φ_3 φ_4	0.0232	6
ACASXU_v2a_5_7	φ_1 φ_2 φ_3 φ_4	0.0211	6
ACASXU_v2a_5_8	φ_1 φ_2 φ_3 φ_4	0.0197	6
ACASXU_v2a_5_9	φ_1 φ_2 φ_3 φ_4	0.0195	6

In Table 14, we observed that 20 is the highest number of bits for the threshold we aim to reach to optimize the QNN format. However, 6 is the minimum number of bits reserved for the threshold.

Achieving an optimized format of QNN through a 20-bit threshold, which represents the error between the NN and its quantized version NN_q, poses a challenge for the precision tuning tool `Popinns`.

8.4 Precision Tuning of Acas Xu Networks using `Popinns`

We evaluated our fixed-point code generation tool, `Popinns`, on the Acas Xu neural networks. The Acas Xu system comprises a total of 45 neural networks, each trained to support collision avoidance in aircraft [JK19]. For the purposes of this study, we focus on a single representative network to illustrate the methodology and results.

Our objective in this step is to generate the fixed-point version of the neural network according to the precision previously computed using the δ'_{max} parameter. The novelty of this approach lies in ensuring that the number of fractional bits is sufficient to preserve all verified network properties when converting from floating-point to fixed-point arithmetic. In contrast, previous work with `Popinns` performed precision optimization and code generation using randomly chosen precisions, without guarantees regarding the functional correctness of the resulting numerical programs or neural networks [BM24c].

The network used in our experiments has the architecture shown in Table 15. It is a fully connected feedforward network with seven layers, where all parameters are trainable.

The total model size is approximately 52 KB.

Layer	Output Shape	Number of Parameters
dense	(None, 50)	300
dense_1	(None, 50)	2,550
dense_2	(None, 50)	2,550
dense_3	(None, 50)	2,550
dense_4	(None, 50)	2,550
dense_5	(None, 50)	2,550
dense_6	(None, 5)	255
Total	-	13,305

Table 15: ACAS-Xu neural network architecture used in the experiments.

The Acas Xu networks were originally trained using 32-bit floating-point (FP32) precision. Using `Popinns`, we synthesized fixed-point implementations of these networks and determined the minimum δ'_{max} required to guarantee that all relevant safety properties are maintained. For the network evaluated, this analysis indicated that a minimum of 20 fractional bits is sufficient, defining the precision of the fractional component in the fixed-point representation as defined in Eq. 8.1.

Table 16 summarizes the fractional bits needed for each property, together with the corresponding maximum deviation $\delta_{\max}$. Entries marked with "-" denote properties that were either not evaluated for this network or are considered irrelevant, and "/" represents that the property has been violated, and it returned "SAT".

Network (onnx)	Property	$\delta'_{\max}$	Minimum Fractional Bits
ACASXU_v2a_1_1	φ_1	4×10^{1}	2
ACASXU_v2a_1_1	φ_2	/	/
ACASXU_v2a_1_1	φ_3	1.0×10^{-3}	10
ACASXU_v2a_1_1	φ_4	1.2×10^{-3}	10
ACASXU_v2a_1_1	φ_5	1.0×10^{-3}	10
ACASXU_v2a_1_1	φ_6	1.0×10^{-6}	20
ACASXU_v2a_1_1	φ_7	-	-
ACASXU_v2a_1_1	φ_8	-	-
ACASXU_v2a_1_1	φ_9	-	-
ACASXU_v2a_1_1	φ_{10}	-	-

Table 16: Minimum number of fractional bits required in the fixed-point representation to satisfy Acas Xu properties. "-" indicates properties not evaluated or irrelevant for this model. "/" indicates that the property has already been violated "SAT"

8.4.1 Execution Results

The execution results of **Popinns** at each stage of the analysis for the evaluated network are as follows:

1. `Model Loading:` The TensorFlow model was loaded in approximately 0.0023 seconds.
2. `Range Analysis:` The output ranges of all neurons were computed in 0.019 seconds.
3. `Constraint Generation:` The constraints were generated to preserve the network properties. Output shapes per layer were 50 neurons for hidden layers and 5 neurons for the output layer. This step took 0.98 seconds.
4. `Constraint Solving:` The Z3 solver was used to determine the minimum bit-widths required to satisfy all properties, which took 0.866 seconds.
5. `Code Synthesis:` Floating-point code was generated in 0.007 seconds, while fixed-point code generation took 1.54 seconds.

The results demonstrate that **Popinns** generates fixed-point implementations of neural networks while ensuring that all relevant properties are preserved. Despite the conversion from FP32 to a fixed-point representation with 20 fractional bits, all verified properties of the ACAS-Xu network are maintained. The majority of the computation time is devoted to constraint solving and fixed-point synthesis, which is expected given the combinatorial nature of the optimization problem.

8.4.2 Generated Fixed-Point Code

The fixed-point C code was generated by **Popinns** using the Fixmath library[1] to perform fixed-point arithmetic. This library provides efficient operations on fixed-point numbers, enabling the neural network to execute with reduced precision while maintaining all verified network properties.

An excerpt of the generated fixed-point code is presented in Listing 8.1, illustrating both the allocation of the layer variables and the computations performed in the sixth layer.

Listing 8.1: Fixed-point code generated by **Popinns** showing memory allocation and sixth layer computation of the ACAS-XU network.

```
#include<stdio.h>
#include<stdlib.h>
...
#define MAX(x,y) ((x)>(y)?(x):(0))
#define RELU(x) ((x)>(0)?(x):(0))

```

[1] https://www.nongnu.org/fixmath/doc/index.html

```
int main(int argc, char *argv[]) {
    fixed_t tmp;
    int count=0;
    fixed_t *t;
    fixed_t ****x = (fixed_t****)malloc(8*sizeof(fixed_t***));

    x[0] = (fixed_t***)malloc(1*sizeof(fixed_t**));
    for(int h=0;h<1;h++) {
        x[0][h] = (fixed_t**)malloc(5*sizeof(fixed_t*));
        for(int i=0;i<5;i++) {
            x[0][h][i] = (fixed_t*)malloc(1*sizeof(fixed_t));
        }
    };

    % ... continues for layers 1 to 7 ...
...
    tmp = 0;
for (int j=0; j<50; j++) {
    tmp = fx_addx(tmp, fx_mulx(W_6[0*50+j], x[6][0][j][0], 8));
}; // fx_k_d_i=8 f_k_i=19 fy_k_d_i=6
x[7][0][0][0] = RELU(fx_xtox(tmp, 19, 6));

tmp = 0;
for (int j=0; j<50; j++) {
    tmp = fx_addx(tmp, fx_mulx(W_6[1*50+j], x[6][0][j][0], 8));
}; // fx_k_d_i=8 f_k_i=19 fy_k_d_i=6
x[7][0][1][0] = RELU(fx_xtox(tmp, 19, 6));

tmp = 0;
for (int j=0; j<50; j++) {
    tmp = fx_addx(tmp, fx_mulx(W_6[2*50+j], x[6][0][j][0], 8));
}; // fx_k_d_i=8 f_k_i=19 fy_k_d_i=6
x[7][0][2][0] = RELU(fx_xtox(tmp, 19, 6));

tmp = 0;
for (int j=0; j<50; j++) {
    tmp = fx_addx(tmp, fx_mulx(W_6[3*50+j], x[6][0][j][0], 8));
}; // fx_k_d_i=8 f_k_i=19 fy_k_d_i=6
x[7][0][3][0] = RELU(fx_xtox(tmp, 19, 6));

tmp = 0;
for (int j=0; j<50; j++) {
    tmp = fx_addx(tmp, fx_mulx(W_6[4*50+j], x[6][0][j][0], 8));
}; // fx_k_d_i=8 f_k_i=19 fy_k_d_i=6
x[7][0][4][0] = RELU(fx_xtox(tmp, 19, 6));
    % ... weights for next layer initialization ...
```

In the generated code, arithmetic operations are performed using the fixed-point functions provided by the library. Specifically, `fx_addx(a,b)` performs the addition of two fixed-point numbers `a` and `b`, while `fx_mulx(a,b,shift)` multiplies two fixed-point numbers with a specified shift to adjust the fractional precision.

The function `fx_xtox(value,input_frac_bits,output_frac_bits)` converts a fixed-point value `value` from one fractional bit representation (`input_frac_bits`) to another (`output_frac_bits`), allowing precise control over precision throughout the network. The ReLU activation function is implemented using the macro `RELU(a)`, which returns the maximum of `a` and zero. Together, these operations enable the neural network to execute efficiently with reduced precision, specifically using 20 bits for the fractional part, while preserving the ACAS-Xu safety properties. For instance, `fx_xtox(tmp,22,8)` converts the variable `tmp` from 22 fractional bits to 8 fractional bits by shifting the value right by 14 bits. This conversion preserves the real value while reducing the fractional resolution.

The comments in the code, e.g., `fx_k_d_i=8 f_k_i=19 fy_k_d_i=6`, indicate the precision of the fixed-point variables. Briefly, `fx_k_d_i` represents the number of fractional bits of the weight, `f_k_i` the number of fractional bits used internally for accumulation, and `fy_k_d_i` the fractional bits of the output neuron. These parameters correspond to the constraints defined in equations (7.6)–(7.8), ensuring that products, sums, and outputs maintain sufficient precision to satisfy the network's properties while minimizing bit-width.

The results obtained demonstrate that **`Popinns`** can successfully generate fixed-point implementations of ACAS-Xu neural networks while preserving all verified safety properties. The preliminary experiments, performed on a single representative network, are promising and indicate the feasibility of deploying these networks on embedded platforms.

The immediate goal is to extend this approach to all 45 ACAS-Xu networks. By optimizing each network, we aim to enable their execution on constrained embedded systems, such as STM32 microcontrollers.

A longer-term objective is the deployment of ACAS-Xu networks on FPGA platforms. To achieve this, we plan to leverage high-level synthesis (HLS) tools, such as Xilinx Vitis HLS, and use the `ap_fixed` library to implement the fixed-point arithmetic efficiently. These tools will allow us to generate hardware designs with controlled precision, meeting both performance and safety requirements.

Additionally, we intend to refine the fixed-point constraints used during the fixed-point code synthesis process. By improving the precision allocation strategy, we aim to further minimize the bit-width requirements without violating the network properties. This refinement will optimize memory usage, computation speed, and overall energy efficiency, which are critical factors for both embedded and FPGA deployments.

8.5 Summary

In this chapter, we present the main results of optimizing the quantization format of artificial neural networks (ANNs) using formal methods and the precision tuning tool `Popinns`, based on the computed δ'_{max}. Our goal is to generate C fixed-point code with an optimized format that preserves the safety properties of Acas Xu.

Chapter 9

Conclusion and Perspectives

The formal specification and verification of AI controllers based on neural networks—particularly quantized neural networks (QNNs)—remains a critical challenge in ensuring the safety and reliability of autonomous systems. While neural networks offer powerful function approximation capabilities, their black-box nature and non-linear, discontinuous behavior (exacerbated by quantization) complicate formal analysis.

In this chapter, we will summarize the contributions of this dissertation, outline the difficulties encountered in realizing this work, and conclude with future directions and our proposed perspectives.

9.1 Summary of Contributions

In this thesis, we present three major contributions that focus on the formal specification and verification of AI-based controllers, particularly QNNs in the domains of autonomous vehicles and avionics.

In Chapter 2, we began the dissertation by introducing a brief theoretical foundation of various concepts related to computer arithmetic. This includes floating-point arithmetic, fixed-point arithmetic, and interval arithmetic. We elaborate on the elementary operations used in each of these arithmetic types and clarify each concept with a corresponding example.

In Chapter 3, we introduced the architecture of neural networks and their activation functions. We highlighted the different compression techniques applied to neural networks for deployment on controllers, and we defined quantized neural networks and their activation functions in fixed-point arithmetic.

In Chapter 4, we began by explaining common formal verification methods, such as Abstract Interpretation, SMT, and linear programming, highlighting the potential of each one and their advantages. and where we use them. We then presented a summarized survey of the verification of neural networks using the aforementioned formal methods, providing a clear overview. Notably, there is limited research on the verification of quantized neural networks (QNNs). We outlined the research works on the formal verification of QNNs and focused on the key differences between our contributions and the ideas behind these related works.

In chapter 5, we explained our first contribution, which consists of introducing a formal specification method for transforming AVs' textual requirements into formal properties. This method involves several steps: first, we generate abstract scenarios from the textual requirements according to specified constraints; next, we create logical scenarios that define range values for variables; and finally, we define SMT predicates to reformulate a formal property. We verify this property using SMT on the corresponding neural network that addresses the action to be performed as described in the textual requirement. If the property is verified, we assume that the formal property generated from the initial textual requirement is correct; otherwise, if the property is violated, it indicates that the textual requirement is poorly formed. We evaluated our method using HIGHWAY-ENV as an AV simulator to illustrate and extract the scenarios defined in the textual requirements.

In chapter 6, we described the second contribution of this thesis, which presents a formal verification method for QNNs simulating the controller of AVs. This method combines set-based theory with satisfiability in what we call (δ)-robustness to verify decision-making properties. We outlined the process and steps of our approach, and we evaluated our work using HIGHWAY-ENV, the Z3 [dMB08b] SMT solver, and Marabou [WIZ+24]. Our approach has demonstrated soundness and validity across multiple versions of quantized neural networks, different p-norms, and various scenarios, showcasing its efficiency compared to SMT verification of QNNs through integer encoding.

In Chapter 7, we explained an optimized method which consist of designing a quantization method for ANNs in the avionics domain using formal methods and precision tunning. We proposed a formal verification of QNNs using the QNN as input (Figure 27). This contribution considers a real neural network as the input. We introduce a dif-

ferent formal verification method that relies on the maximum output perturbation δ'_{max} that satisfies the property. This perturbation serves as input for a precision tuning tool, `Popinns` [BM24a], designed to identify the optimized QNN format for each neuron that meets the safety property. We explained the process and steps of this contribution and outlined the core of `Popinns` and its main role in our work.

In Chapter 8, we experimented with the proposed optimized method using Acas Xu [JK19] and its properties. We began by identifying the $\delta' max$ for each property corresponding to each neural network. We then defined the appropriate $\delta' max$ for each neural network by considering all the properties applied to it, ultimately determining the minimum δ'_{max} among them.

To proceed with the next step of using the precision tuning tool `Popinns`, we first converted Acas Xu ONNX files to TensorFlow models for processing in `Popinns` and considered the δ'_{max} of each Acas Xu neural network as a defined threshold.

9.2 Future Work and Perspectives

The work conducted in this dissertion paves the way for numerous new perspectives, some of which are outlined below :

9.2.1 Using Appropriate SMT Neural Network Solver Tools for Rational NNs

We aim to improve our method for verifying rational neural networks by utilizing state-of-the-art neural network verification tools such as nneum [Bak21], CROWN-Beta [WZX+21b], and ERAN [SBR+]. Our goal is to select the most appropriate tool based on key metrics, including runtime efficiency, memory usage, and architectural complexity support, to determine the optimal choice for different scenarios. Our analysis will compare their performance on modern architectures and safety-critical properties while assessing the trade-offs between precision and scalability. Ultimately, we aim to develop a practical framework for selecting verification tools based on model type, resource constraints, and certification requirements, thereby enhancing reliable deployment in real-world systems.

9.2.2 Tight Error Bound Approximation between NN_q and NN_r

The limit of our work initially relies on using interval propagation with interval arithmetic to compute the maximum distance between NN_q and NN_r, denoted as ε_p. This results in overapproximated values that violate the properties for small to medium word width formats. As prospective, we plan to explore a variety of advanced methods to compute a tighter quantization error bound for neural network controllers, addressing a critical challenge in their reliable deployment. One key direction involves investigating interval arithmetic and affine arithmetic techniques to better capture the propagation of numerical errors through quantized networks. Additionally, we will examine zonotope-

based reachability analysis, which provides a rigorous framework for bounding the output perturbations caused by quantization.

9.2.3 Exploring Other Types of Properties

Our current verification methods focus on decision-making properties in autonomous systems. However, to meet the increasing demand for rigorous safety guarantees in dynamic environments, we aim to expand our framework to support Signal Temporal Logic (STL) [DDG+17] and Linear Temporal Logic (LTL) [Roz11]. These formalisms facilitate the verification of time-sensitive behaviors that are critical for autonomy, such as real-time collision avoidance (STL) and mission sequencing in robotics (LTL).

Major challenges include integrating temporal logic with neural network verification, such as through bounded model checking or differentiable STL loss functions, and ensuring compatibility with industry standards. By incorporating STL/LTL, our approach will enhance the certification of AI-driven systems—for example, ensuring that an autonomous vehicle yields within 2 seconds of detecting a pedestrian—while bridging the gap between learning-based control and symbolic planning. Future work will explore GPU-accelerated temporal monitors, hybrid neuro-symbolic reasoning, and standardized benchmarks for real-world deployment. This extension positions our methods at the forefront of verifiable autonomy, where safety, real-time performance, and scalability converge.

9.2.4 Scalability: Extending to More Complex Networks and Activation Functions

Our framework primarily employs fully connected neural networks. To enhance scalability, we suggest two main research directions:

- **Diversifying Activation Functions and Layer Types**: We aim to extend support for activation functions such as sigmoid, tanh, and softmax, including their abstract transformations in fixed-point arithmetic, to generalize our methodology. Additionally, we plan to incorporate advanced layers (e.g., normalization and residual connections) to enable the analysis of complex architectures similar to GPT-4 [AAA+23] and PaLM [ADF+23].

- **Broadening Network Architectures**: Beyond DQN, we aim to explore complex architectures such as CNN, RNN (Reccurent Neural Network) and RL algorithms, as well as the types of data in the domain of autonomous vehicles. We also intend to investigate real data, including images and sensor values.

  These advancements will significantly expand the applicability of our methods across diverse neural network paradigms.

9.2.5 Parallelization and GPU Utilization

Combining symbolic execution [GWZ+18] with GPU-accelerated solvers [LS13] aims to enhance the scalability of formal verification by parallelizing computationally intensive tasks. This approach leverages symbolic execution to systematically explore program paths, generating complex path conditions that represent feasible execution traces. Meanwhile, GPU acceleration exploits massive parallelism to speed up two key bottlenecks: (1) constraint solving (e.g., through batched SMT queries processed concurrently on GPU cores) and (2) path exploration (e.g., via parallelized ReLU splits in neural networks [WOZ+20, KPKH17]. The main components of this hybrid architecture include GPU-optimized SMT solvers (e.g., cuSAT [CSKHK13] for bit-vector theories, benefiting from warp-level parallelism in CUDA) and hybrid CPU-GPU task distribution frameworks [PSK11]. In this setup, CPUs manage control flow, symbolic state tracking, and heuristic-driven path prioritization, while GPUs handle embarrassingly parallel workloads such as SAT/SMT solving, linear algebra operations, and neural activation pattern enumeration.

Further innovations, such as just-in-time kernel compilation for SMT queries and adaptive batching strategies, could optimize GPU occupancy. This approach is particularly promising for verifying quantized neural networks (QNNs), as the bit-precise nature of computations aligns well with GPU-accelerated bit-vector reasoning. By leveraging parallelism across layers, neurons, and input regions, GPU-enhanced symbolic execution could achieve orders-of-magnitude speedups compared to CPU-only methods, making exhaustive verification of real-world deep learning models more feasible. Future work may also explore integration with probabilistic symbolic execution [GDV12] or differentiable SMT solvers [BBS+19] to further bridge formal methods and machine learning.

9.2.6 Integration of Verification with QNN Training

A promising direction in Quantized Neural Network (QNN) verification is closing the gap between training and verification. Traditionally, verification is applied after training, treating the QNN as a static object. However, integrating verification during training can produce models that are inherently more verifiable and robust.

Major approaches include verification-aware training (e.g., adding formal constraints to loss functions [FHL+19]), quantization-aware verification [ZCS+24], and self-verifying architectures (e.g., layers with built-in safety guarantees [WM22]). Challenges like computational cost and accuracy-verifiability trade-offs persist, but tools like differentiable bound propagation (auto LiRPA [XSZ+20]) and neural-symbolic methods offer promising solutions. This co-design paradigm aims to produce QNNs that are inherently robust, scalable, and deployable in safety-critical systems.

9.2.7 Integrate our methods in a Verification Tool

We aim to structure the implementation of our methods to be more generalized, organized, and efficient, serving as a reference tool for verifying QNNs.

We plan to extend our method to create an efficient tool that supports multiple benchmarks and neural network formats, such as H5py, ONNX, and NNET, while addressing various properties in a modular and dynamic manner. In this implementation, we prioritize well-formed and comprehensive documentation, along with concise steps to achieve results.

By the end of this dissertion, we are confident that our contributions represent a promising step toward addressing the challenges of specifying and verifying quantized neural networks with respect to both scalability and precision. This work opens up numerous research avenues for further improvement and application.

Bibliography

[AA21] Ravi P Agarwal and Hans Agarwal. Origin of irrational numbers and their approximations. *Computation*, 9(3):29, 2021.

[AAA+23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[AAE22] Ahmed Jaber Abougarair, Mohamed KI Aburakhis, and Mohamed M Edardar. Adaptive neural networks based robust output feedback controllers for nonlinear systems. *International Journal of Robotics and Control Systems*, 2(1):37–56, 2022.

[AAM23] Muhammad Yeasir Arafat, Muhammad Morshed Alam, and Sangman Moh. Vision-based navigation techniques for unmanned aerial vehicles: Review and challenges. *Drones*, 7(2):89, 2023.

[AASA11] Mohamed Al-Ashrafy, Ashraf Salem, and Wagdy Anis. An efficient implementation of floating point multiplier. In *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, pages 1–5, 2011.

[ABG+20] Jyotika Athavale, Andrea Baldovin, Ralf Graefe, Michael Paulitsch, and Rafael Rosales. Ai and reliability trends in safety-critical autonomous systems on ground and air. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 74–77, 2020.

[ABM98] Paul E Ammann, Paul E Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*, pages 46–54. IEEE, 1998.

[ADAA18] Basman M Hasan Alhafidh, Amar I Daood, Mohammed M Alawad, and William Allen. Fpga hardware implementation of smart home autonomous

system based on deep learning. In *Internet of Things–ICIOT 2018: Third International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 3*, pages 121–133. Springer, 2018.

[ADF+23] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[ADK19] Johan Arcile, Raymond Devillers, and Hanna Klaudel. Verifcar: a framework for modeling and model checking communicating autonomous vehicles. *Autonomous agents and multi-agent systems*, 33:353–381, 2019.

[AGO+13] Davide Anguita, Alessandro Ghio, Luca Oneto, Francesc Xavier Llanas Parra, and Jorge Luis Reyes Ortiz. Energy efficient smartphone-based activity recognition using fixed-point arithmetic. *Journal of universal computer science*, 19(9):1295–1314, 2013.

[Alt10] Matthias Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

[AP18] Gul Agha and Karl Palmskog. A survey of statistical model checking. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(1):1–39, 2018.

[APW89] Panos J Antsaklis, Kevin M Passino, and SJ Wang. Towards intelligent autonomous control systems: Architecture and fundamental issues. *Journal of Intelligent and Robotic Systems*, 1:315–342, 1989.

[Ara22] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):740–759, 2022.

[Aré19] Nikos Aréchiga. Specifying safety of autonomous vehicles in signal temporal logic. In *2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, June 9-12, 2019*, pages 58–63. IEEE, 2019.

[ASL91] Panos J Antsaklis, James A Stiver, and Michael Lemmon. Hybrid system modeling and autonomous control systems. In *International Hybrid Systems Workshop*, pages 366–392. Springer, 1991.

[ATD05] Behzad Akbarpour, Sofiène Tahar, and Abdelkader Dekdouk. Formalization of fixed-point arithmetic in HOL. *Formal Methods Syst. Des.*, 27(1-2):173–200, 2005.

[B+19] Stanley Bak et al. Improved quantization for neural network verification. *Logical Methods in Computer Science*, 15(3), 2019.

[Bak21] Stanley Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*, volume 12673 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2021.

[Bal95] Osman Balci. Principles and techniques of simulation validation, verification, and testing. In *Proceedings of the 27th conference on Winter simulation*, pages 147–154, 1995.

[BBF+13] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[BBKBM22] Sofiane Bessaï, Dorra Ben Khalifa, Hanane Benmaghnia, and Matthieu Martel. Fixed-Point Code Synthesis Based on Constraint Generation. In *Workshop on Design and Architectures for Signal and Image Processing*, Budapest, Hungary, June 2022.

[BBS+19] Mislav Balunovic, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. Learning to solve smt formulas. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10317–10328, 2019.

[BCHT17] Somil Bansal, Mo Chen, Sylvia Herbert, and Claire J Tomlin. Hamilton-jacobi reachability: A brief overview and recent advances. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2242–2253. IEEE, 2017.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[BDKG18] Harald Bayerlein, Paul De Kerret, and David Gesbert. Trajectory optimization for autonomous flying base station via reinforcement learning. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2018.

[BDPM20] S. H. Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. Impact of fully connected layers on performance of convolutional neural networks for image classification. *Neurocomputing*, 378:112–119, 2020.

[Ben21] Dorra Ben Khalifa. *Fast and efficient bit-level precision tuning. (Analyse statique pour le réglage de la précision numérique)*. PhD thesis, University of Perpignan, France, 2021.

[Ben22] Hanane Benmaghnia. *Synthèse de code virgule fixe pour les réseaux de neurones. (Fixed-point code synthesis for neural networks)*. PhD thesis, University of Perpignan, France, 2022.

[Ber21] Anthony Berthelier. *Etudes techniques de compression de réseaux de neurones pour sa mise en place dans une architecture embarquée de type Smartphone*. Theses, Université Clermont Auvergne, December 2021.

[BES16] Alessandro Bernardini, Wolfgang Ecker, and Ulf Schlichtmann. Where formal verification can help in functional safety analysis. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2016.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BGF18] Facundo Bre, Juan M. Gimenez, and Víctor D. Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158:1429–1441, 2018.

[BHL+20a] Marek S. Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamaric. An SMT theory of fixed-point arithmetic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 13–31. Springer, 2020.

[BHL+20b] Marek S. Baranowski, Shaobo He, Mathias Lechner, Thanh Son Nguyen, and Zvonimir Rakamaric. An SMT theory of fixed-point arithmetic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 13–31. Springer, 2020.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[Bie21] Armin Biere. Bounded model checking. In *Handbook of satisfiability*, pages 739–764. IOS press, 2021.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BL17] Jaâfar Berrada and Fabien Leurent. Modeling transportation systems involving autonomous vehicles: A state of the art. *Transportation Research Procedia*,

27:215–221, 2017. 20th EURO Working Group on Transportation Meeting, EWGT 2017, 4-6 September 2017, Budapest, Hungary.

[BLS+24] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. Deep neural networks and tabular data: A survey. *IEEE Trans. Neural Networks Learn. Syst.*, 35(6):7499–7519, 2024.

[BLT+20] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(42):1–39, 2020.

[BM22] Dorra Ben Khalifa and Matthieu Martel. Constrained precision tuning. In *8th International Conference on Control, Decision and Information Technologies, CoDIT*, pages 230–236, 2022.

[BM24a] Dorra Ben Khalifa and Matthieu Martel. Efficient implementation of neural networks usual layers on fixed-point architectures. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES*. ACM, 2024.

[BM24b] Dorra Ben Khalifa and Matthieu Martel. Floating to fixed-point conversion of deep neural networks with guaranteed error bounds. In Teresa Alsinet, Xavier Vilasís-Cardona, Daniel García, and Elena Álvarez, editors, *Artificial Intelligence Research and Development - Proceedings of the 26th International Conference of the Catalan Association for Artificial Intelligence, CCIA 2024, Barcelona, Spain, 2-4 October 2024*, volume 390 of *Frontiers in Artificial Intelligence and Applications*, pages 136–139. IOS Press, 2024.

[BM24c] Dorra Ben Khalifa and Matthieu Martel. Rigorous floating-point to fixed-point quantization of deep neural networks on STM32 micro-controllers. In *10th International Conference on Control, Decision and Information Technologies, CoDIT*, pages 1201–1206. IEEE, 2024.

[BMKL23] Durga Prasad Bavirisetti, Herman Ryen Martinsen, Gabriel Hanssen Kiss, and Frank Lindseth. A multi-task vision transformer for segmentation and monocular depth estimation for autonomous vehicles. *IEEE Open Journal of Intelligent Transportation Systems*, 4:909–928, 2023.

[BMN+19] Petros S. Bithas, Emmanouel T. Michailidis, Nikolaos Nomikos, Demosthenes Vouyioukas, and Athanasios G. Kanatas. A survey on machine-learning techniques for uav-based communications. *Sensors*, 19(23):5170, 2019.

[BMS22] Hanane Benmaghnia, Matthieu Martel, and Yassamine Seladji. Fixed-point code synthesis for neural networks. *CoRR*, abs/2202.02095, 2022.

[Bor12] Karl Heinz Borgwardt. *The simplex method: a probabilistic analysis*, volume 1. Springer Science & Business Media, 2012.

[Bou17] Jean-Louis Boulanger. 11 - verification and validation. In Jean-Louis Boulanger, editor, *Certifiable Software Applications 2*, pages 115–161. Elsevier, 2017.

[BP23] Nazar Beknazarov and Maria Poptsova. Deepz: a deep learning approach for z-dna prediction. In *Z-DNA: Methods and Protocols*, pages 217–226. Springer, 2023.

[BSG24] Wahiba Bachiri, Yassamine Seladji, and Pierre-Loïc Garoche. Formal verification of quantized neural network. In *2024 International Conference of the African Federation of Operational Research Societies (AFROS)*, pages 1–5, 2024.

[BSG25] Wahiba Bachiri, Yassamine Seladji, and Pierre-Loïc Garoche. Formal specification and smt verification of quantized neural network for autonomous vehicles. *Science of Computer Programming*, 245:103316, 2025.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[BTT+18] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in neural information processing systems*, 31, 2018.

[BTV03] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31:129–168, 2003.

[Bun19] Rudy Bunel. *Formal verification of neural networks*. PhD thesis, University of Oxford, UK, 2019.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. 01 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs. *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of*

Programming Languages, San Antonio, Texas, USA, January 1979, pages 269–282. ACM Press, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 334–342. Springer, 2014.

[CES09] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[CF17] G.E. Cantarella and A. Di Febbraro. Transportation systems with autonomous vehicles: models and algorithms for equilibrium assignment. *Transportation Research Procedia*, 27:349–356, 2017. 20th EURO Working Group on Transportation Meeting, EWGT 2017, 4-6 September 2017, Budapest, Hungary.

[CFG+22] Soledad Le Clainche, Esteban Ferrer, Sam Gibson, Elisabeth Cross, Alessandro Parente, and Ricardo Vinuesa. Improving aircraft performance using machine learning: a review. *CoRR*, abs/2210.11481, 2022.

[CH19] Jang Hyun Cho and Bharath Hariharan. On the efficacy of knowledge distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4794–4802, 2019.

[CL89] John Carroll and Darrell Long. Theory of finite automata: with an introduction to formal languages. 1989.

[CLM89] Edmund M Clarke, David E Long, and Kenneth L McMillan. Compositional model checking. 1989.

[CMW+21] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review. *Acta Mechanica Sinica*, 37(12):1727–1738, 2021.

[CNHR18] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Harald Ruess. Verification of binarized neural networks via inter-neuron factoring - (short paper). In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*, volume 11294 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 2018.

[Coo23] Stephen A Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 138–156. Springer, 2001.

[Cou21] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

[CSKHK13] Surendran Cherukodan, G Santhosh Kumar, and S Humayoon Kabir. Using open source software for digital libraries: A case study of cusat. *The Electronic Library*, 31(2):217–225, 2013.

[CWM02] Samit Chaudhuri, Robert A Walker, and John E Mitchell. Analyzing and exploiting the structure of the constraints in the ilp approach to the scheduling problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):456–471, 2002.

[CWZZ17] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[D+14] Emily L Denton et al. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 2014.

[DBL85] *IEEE standard for binary floating-point arithmetic - IEEE standard 754-1985*. Beuth, 1985.

[DBL12] Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Process. Mag.*, 29(6):82–97, 2012.

[DCV21] Catalin Dumitrescu, Petrica Ciotirnae, and Constantin Vizitiu. Fuzzy logic for intelligent control system using soft computing applications. *Sensors*, 21(8):2617, 2021.

[DDG+17] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51:5–30, 2017.

[DdSJCC24] Pierre V. Dantas, Waldir Sabino da Silva Jr., Lucas Carvalho Cordeiro, and Celso Barbosa Carvalho. A comprehensive review of model compression techniques in machine learning. *Appl. Intell.*, 54(22):11804–11844, 2024.

[dFS04] Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004.

[DGPT24] Stefano Demarchi, Dario Guidotti, Luca Pulina, and Armando Tacchella. Never2: learning and verification of neural networks. *Soft Comput.*, 28(19):11647–11665, 2024.

[DJ10] Travis Dierks and Sarangapani Jagannathan. Output feedback control of a quadrotor uav using neural networks. *IEEE Transactions on Neural Networks*, 21(1):50–66, 2010.

[dJP23] Armando de Jesús Plasencia-Salgueiro. Deep reinforcement learning for autonomous mobile robot navigation. In Ahmad Taher Azar and Anis Koubaa, editors, *Artificial Intelligence for Robotics and Autonomous Systems Applications*, volume 1093 of *Studies in Computational Intelligence*, pages 195–237. Springer, 2023.

[DJS16] Nikil D. Dutt, Axel Jantsch, and Santanu Sarma. Toward smart embedded systems: A self-aware system-on-chip (soc) perspective. *ACM Trans. Embed. Comput. Syst.*, 15(2):22:1–22:27, 2016.

[DJST18] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[DKH09] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19:332–377, 2009.

[DKSL20] Kirsty Duncan, Ekaterina Komendantskaya, Robert Stewart, and Michael Lones. Relative robustness of quantized neural networks against adversarial attacks. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[DMB08a] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[dMB08c] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[DMY+20] Zhao Dong, Jing Men, Zhiwen Yang, Jason Jerwick, Airong Li, Rudolph E Tanzi, and Chao Zhou. Flynet 2.0: drosophila heart 3d (2d+ time) segmentation in optical coherence microscopy images using a convolutional long short-term memory neural network. *Biomedical Optics Express*, 11(3):1568–1579, 2020.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[DTS+23] Susmita Das, Amara Tariq, Thiago Santos, Sai Sandeep Kantareddy, and Imon Banerjee. Recurrent neural networks (rnns): architectures, training tricks, and introduction to influential research. *Machine learning for Brain disorders*, pages 117–138, 2023.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[ea17] Aws Albarghouthi et al. Probabilistic abstract interpretation. *ESOP*, 2017.

[ea21] Gagandeep Singh et al. Abstract interpretation for neural networks. *PLDI*, 2021.

[Ehl17] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017.

[EHSH21] Michael Everett, Golnaz Habibi, Chuangchuang Sun, and Jonathan P How. Reachability analysis of neural feedback loops. *IEEE Access*, 9:163938–163953, 2021.

[ELS18] Ferdinand Englberger, Thomas Latzel, and Prodromos Sotiriadis. An autonomous robot for embedded systems and robotics. In *2018 12th European Workshop on Microelectronics Education (EWME)*, pages 35–39. IEEE, 2018.

[FBEG16] Joseph Funke, Matthew Brown, Stephen M Erlien, and J Christian Gerdes. Collision avoidance and stabilization for autonomous vehicles in emergency scenarios. *IEEE Transactions on Control Systems Technology*, 25(4):1204–1216, 2016.

[FCAF17] Lucas E. R. Fernandes, Vinicius Custodio, Gleifer V. Alves, and Michael Fisher. A rational agent controlling an autonomous vehicle: Implementation and formal verification. In Lukas Bulwahn, Maryam Kamali, and Sven Linker, editors, *Proceedings First Workshop on Formal Verification of Autonomous Vehicles, FVAV@iFM 2017, Turin, Italy, 19th September 2017*, volume 257 of *EPTCS*, pages 35–42, 2017.

[FDG+19] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: a language for scenario specification and scene generation. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 63–78. ACM, 2019.

[FDK12] Ioannis Filippidis, Dimos V. Dimarogonas, and Kostas J. Kyriakopoulos. Decentralized multi-agent control from local LTL specifications. In *Proceedings of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA*, pages 6235–6240. IEEE, 2012.

[FHL+19] Jiameng Fan, Chao Huang, Wenchao Li, Xin Chen, and Qi Zhu. Towards verification-aware knowledge distillation for neural-network controlled systems. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[FJ18] Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

[FLSM22a] Marie Farrell, Matt Luckcuck, Oisín Sheridan, and Rosemary Monahan. Fretting about requirements: Formalised requirements for an aircraft engine controller. In Vincenzo Gervasi and Andreas Vogelsang, editors, *Requirements Engineering: Foundation for Software Quality - 28th International Working Conference, REFSQ 2022, Birmingham, UK, March 21-24, 2022, Proceedings*, volume 13216 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2022.

[FLSM22b] Marie Farrell, Matt Luckcuck, Oisín Sheridan, and Rosemary Monahan. Towards refactoring fretish requirements. In Jyotirmoy V. Deshmukh, Klaus

Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 272–279. Springer, 2022.

[FMP20] Mahyar Fazlyab, Manfred Morari, and George J Pappas. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *IEEE Transactions on Automatic Control*, 67(1):1–15, 2020.

[For92] Surveyors' Forum. comments on "what every computer scientist should know about floating point arithmetic". *ACM Comput. Surv.*, 24(3):319, 1992.

[FR17] Ambrose Finnerty and Hervé Ratigner. Reduce power and cost by converting from floating point to fixed point. *WP491 (v1. 0)*, pages 9–10, 2017.

[FRNS17] Predrag Filipovikj, Guillermo Rodríguez-Navas, Mattias Nyberg, and Cristina Seceleanu. Smt-based consistency analysis of industrial systems requirements. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1272–1279. ACM, 2017.

[Gad17] S Andrew Gadsden. An adaptive pid controller based on bayesian theory. In *Dynamic Systems and Control Conference*, volume 58288, page V002T12A005. American Society of Mechanical Engineers, 2017.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[GDNC15] Shigen Gao, Hairong Dong, Bin Ning, and Lei Chen. Neural adaptive control for uncertain nonlinear system with input saturation: State transformation based output feedback. *Neurocomputing*, 159:117–125, 2015.

[GDV12] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176, 2012.

[GGKT23] Srajan Goyal, Alberto Griggio, Jacob Kimblad, and Stefano Tonetta. Automatic generation of scenarios for system-level simulation-based verification

of autonomous driving systems. In Marie Farrell, Matt Luckcuck, Mario Gleirscher, and Maike Schwammberger, editors, *Proceedings Fifth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2023, Leiden, The Netherlands, 15th and 16th of November 2023*, volume 395 of *EPTCS*, pages 113–129, 2023.

[GHL20] Mirco Giacobbe, Thomas A. Henzinger, and Mathias Lechner. How many bits does it take to quantize your neural network? In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 79–97. Springer, 2020.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, pages 175–188. Springer, 2004.

[GK96] Christoph Goller and Andreas Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96), Washington, DC, USA, June 3-6, 1996*, pages 347–352. IEEE, 1996.

[GKD+22] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-power computer vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[GL94] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[GMD+18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 3–18. IEEE Computer Society, 2018.

[GMDC+18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE S&P*, 2018.

[GMSL18] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Formal verification of an autonomous wheel loader by model checking. In

Proceedings of the 6th Conference on Formal Methods in Software Engineering, pages 74–83, 2018.

[GO21] LLC Gurobi Optimization. Gurobi optimizer reference manual. 2021.

[GP15] Eric Goubault and Sylvie Putot. A zonotopic framework for functional abstractions. *Formal Methods Syst. Des.*, 47(3):302–360, 2015.

[GPP23] Dario Guidotti, Laura Pandolfo, and Luca Pulina. Verifying neural networks with SMT: an experimental evaluation. In *19th IEEE International Conference on e-Science, e-Science 2023, Limassol, Cyprus, October 9-13, 2023*, pages 1–2. IEEE, 2023.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[GSZ23] Refka Ghodhbani, Taoufik Saidani, and Hafedh Zayeni. Deploying deep learning networks based advanced techniques for image processing on fpga platform. *Neural Computing and Applications*, 35(26):18949–18969, 2023.

[Guo18] Yunhui Guo. A survey on methods and theories of quantized neural networks. *CoRR*, abs/1808.04752, 2018.

[Gur24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[GWLP22] Kai Guo, Na Wang, Datong Liu, and Xiyuan Peng. Uncertainty-aware lstm based dynamic flight fault detection for uav actuator. *IEEE Transactions on Instrumentation and Measurement*, 72:1–13, 2022.

[GWWR23] Martin Götze, Manuela Witt, Nina Willer, and Florian Raisch. Safety in use and automated driving in consideration of the new iso 21448. *ATZ worldwide*, 125(4):38–43, 2023.

[GWZ+18] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S Pasareanu, and Sarfraz Khurshid. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*, 2018.

[GWZ+21] Xingwu Guo, Wenjie Wan, Zhaodi Zhang, Min Zhang, Fu Song, and Xuejun Wen. Eager falsification for accelerating robustness verification of deep neural networks. In Zhi Jin, Xuandong Li, Jianwen Xiang, Leonardo Mariani, Ting Liu, Xiao Yu, and Nahgmeh Ivaki, editors, *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 345–356. IEEE, 2021.

[GYMT21] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

[Hav18] Joel Havermark. Bit-vector approximations of floating-point arithmetic, 2018.

[HBJ+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanovic, Clark W. Barrett, and Cesare Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 680–695. Springer, 2014.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4107–4115, 2016.

[Hen14] Julien Henry. *Static Analysis by Abstract Interpretation and Decision Procedures. (Analyse statique de programme par interprétation abstraite et procédures de décision)*. PhD thesis, University of Grenoble, France, 2014.

[HH13] David Money Harris and Sarah L. Harris. 5 - digital building blocks. In David Money Harris and Sarah L. Harris, editors, *Digital Design and Computer Architecture (Second Edition)*, pages 238–293. Morgan Kaufmann, Boston, second edition edition, 2013.

[HH22] Sarah L. Harris and David Harris. 5 - digital building blocks. In Sarah L. Harris and David Harris, editors, *Digital Design and Computer Architecture*, pages 236–297. Morgan Kaufmann, 2022.

[HKWW17a] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017.

[HKWW17b] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017.

[HLZ21] Thomas A. Henzinger, Mathias Lechner, and Dorde Zikelic. Scalable verification of quantized neural networks. In *The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3787–3795. AAAI Press, 2021.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.

[HPP24] Geonho Hwang, Yeachan Park, and Sejun Park. On expressive power of quantized neural networks under fixed-point arithmetic. *arXiv preprint arXiv:2409.00297*, 2024.

[HRS+23] Md Shakhawat Hossain, Md Mahmudur Rahman, M Mahbubul Syeed, Mohammad Faisal Uddin, Mahady Hasan, Md Aulad Hossain, Amel Ksibi, Mona M Jamjoom, Zahid Ullah, and Md Abdus Samad. Deeppoly: deep learning-based polyps segmentation and classification for autonomous colonoscopy examination. *IEEE Access*, 11:95889–95902, 2023.

[HSW89] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[HUKK19] M. Hasan, S. Ullah, M. J. Khan, and K. Khurshid. Comparative analysis of svm, ann and cnn for classifying vegetation species using hyperspectral thermal infrared data. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W13:1861–1868, 2019.

[HVP+18] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3223–3230. AAAI Press, 2018.

[HWY+24] Pei Huang, Haoze Wu, Yuting Yang, Ieva Daukantas, Min Wu, Yedi Zhang, and Clark W. Barrett. Towards efficient verification of quantized neural networks. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 21152–21160. AAAI Press, 2024.

[HXL+19] Lian Hou, Long Xin, Shengbo Eben Li, Bo Cheng, and Wenjun Wang. Interactive trajectory prediction of surrounding road users for autonomous driving using structural-lstm network. *IEEE Transactions on Intelligent Transportation Systems*, 21(11):4615–4625, 2019.

[HYL+24] Hantao Huang, Ziang Yang, Jia Yao Christopher Lim, Jung Hau Foo, and Chia-Lin Yu. Hardware-aware mixed-precision quantization, January 4 2024. US Patent App. 17/852,484.

[HZ19] Rasheed Hussain and Sherali Zeadally. Autonomous cars: Research results, issues, and future challenges. *IEEE Commun. Surv. Tutorials*, 21(2):1275–1313, 2019.

[Ilo10] IBM Ilog. User's manual for cplex. *http://www. ilog. com/*, 2010.

[IM19] Arnault Ioualalen and Matthieu Martel. Neural network precision tuning. In *Quantitative Evaluation of Systems: 16th International Conference, QEST 2019, Glasgow, UK, September 10–12, 2019, Proceedings 16*, pages 129–143. Springer, 2019.

[INF18] David Isele, Alireza Nakhaei, and Kikuo Fujimura. Safe reinforcement learning on autonomous vehicles. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–6, 2018.

[J+18] Benoit Jacob et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[JEMLS17] Sergey Jatsun, Oksana Emelyanova, Andres Santiago Martinez Leon, and Svetlana Stykanyova. Control fligth of a uav type tricopter with fuzzy logic controller. In *2017 Dynamics of Systems, Mechanisms and Machines (Dynamics)*, pages 1–5, 2017.

[JK19] Kyle Julian and Mykel Kochenderfer. Reluplex-based verification of neural network controllers for acas xu. *DASC*, 2019.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2704–2713. Computer Vision Foundation / IEEE Computer Society, 2018.

[Joh92] Jeremy R Johnson. Real algebraic number computation using interval arithmetic. In *Papers from the international symposium on Symbolic and algebraic computation*, pages 195–205, 1992.

[JPH20] Fan Jiang, Farhad Pourpanah, and Qi Hao. Design, implementation, and evaluation of a neural-network-based quadcopter uav system. *IEEE Transactions on Industrial Electronics*, 67(3):2076–2085, 2020.

[JRBA19] Maksim Jenihhin, Matteo Sonza Reorda, Aneesh Balakrishnan, and Dan Alexandrescu. Challenges of reliability assessment and enhancement in autonomous systems. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2019.

[JSNA19] Farha Jahan, Weiqing Sun, Quamar Niyaz, and Mansoor Alam. Security modeling of autonomous systems: A survey. *ACM Comput. Surv.*, 52(5), September 2019.

[KBD+17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[KBD+22] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods Syst. Des.*, 60(1):87–116, 2022.

[KBK+23] Suhas Kotha, Christopher Brix, J. Zico Kolter, Krishnamurthy Dvijotham, and Huan Zhang. Provably bounding neural network preimages. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[KDM+17] Maryam Kamali, Louise A. Dennis, Owen McAree, Michael Fisher, and Sandor M. Veres. Formal verification of autonomous vehicle platooning. *Sci. Comput. Program.*, 148:88–106, 2017.

[KEM+23] Manzoor Ahmed Khan, Hesham El-Sayed, Sumbal Malik, Muhammad Talha Zia, Muhammad Jalal Khan, Najla Alkaabi, and Henry Alexander Ignatious. Level-5 autonomous driving - are we there yet? A review of research literature. *ACM Comput. Surv.*, 55(2):27:1–27:38, 2023.

[KGF23] Elias Khalife, Pierre-Loïc Garoche, and Mazen Farhood. Code-level formal verification of ellipsoidal invariant sets for linear parameter-varying systems. In Kristin Yvonne Rozier and Swarat Chaudhuri, editors, *NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May*

16-18, 2023, Proceedings, volume 13903 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2023.

[KH+10] Alex Krizhevsky, Geoff Hinton, et al. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

[KHL+12] Said G Khan, Guido Herrmann, Frank L Lewis, Tony Pipe, and Chris Melhuish. Reinforcement learning and optimal adaptive control: An overview and implementation examples. *Annual reviews in control*, 36(1):42–59, 2012.

[KKH18] Maxat Kulmanov, Mohammed Asif Khan, and Robert Hoehndorf. Deepgo: predicting protein functions from sequence and interactions using a deep ontology-aware classifier. *Bioinformatics*, 34(4):660–668, 2018.

[KKR24] Sudeep Kanav, Jan Křetínskỳ, and Sabine Rieder. A literature review on verification and abstraction of neural networks within the formal methods community. *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, pages 39–65, 2024.

[KNvB+23] Andrey Kuzmin, Markus Nagel, Mart van Baalen, Arash Behboodi, and Tijmen Blankevoort. Pruning vs quantization: Which is better? In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems LA, USA*, 2023.

[Koo21] Brett Koonce. Resnet 50. In *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*, pages 63–72. Springer, 2021.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[KPK+23] Kailash Kumar, Suyog Vinayak Pande, T Ch Anil Kumar, Parvesh Saini, Abhay Chaturvedi, Pundru Chandra Shaker Reddy, and Krishna Bikram Shah. Intelligent controller design and fault prediction using machine learning model. *International Transactions on Electrical Energy Systems*, 2023(1):1056387, 2023.

[KPKH17] Juyong Kim, Yookoon Park, Gunhee Kim, and Sung Ju Hwang. Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *International conference on machine learning*, pages 1866–1874. PMLR, 2017.

[KRSB18] Salman Hameed Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. *A Guide to Convolutional Neural Networks for Computer*

Vision. Synthesis Lectures on Computer Vision. Morgan & Claypool Publishers, 2018.

[KSA+18] Bernhard Kaiser, Daniel Schneider, Rasmus Adler, Dominik Domis, Felix Möhrle, Axel Berres, Marc Zeller, Kai Höfig, and Martin Rothfelder. Advances in component fault trees. In *Safety and Reliability–Safe Societies in a Changing World*, pages 815–823. CRC Press, 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[LDW19] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks. *arXiv preprint arXiv:1909.13144*, 2019.

[Led21] Johannes Lederer. Activation functions in artificial neural networks: A systematic overview. *CoRR*, abs/2101.09957, 2021.

[Leu18] Edouard Leurent. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

[LFD+19] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.*, 52(5):100:1–100:41, 2019.

[Lib23] The GNU Multiple Precision Arithmetic Library. https://gmplib.org/, 2023.

[LJVD23a] Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. Sound mixed fixed-point quantization of neural networks. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–26, 2023.

[LJVD23b] Debasmita Lohar, Clothilde Jeangoudoux, Anastasia Volkova, and Eva Darulova. Sound mixed fixed-point quantization of neural networks. *ACM Trans. Embed. Comput. Syst.*, 22(5s):136:1–136:26, 2023.

[LKA+20] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[LLX+24] Haiyi Liu, Shaoying Liu, Guangquan Xu, Ai Liu, and Dingbang Fang. NNTBFV: simplifying and verifying neural networks using testing-based formal verification. *Int. J. Softw. Eng. Knowl. Eng.*, 34(2):273–300, 2024.

[LLY+19] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26*, pages 296–319. Springer, 2019.

[LMS94] Irvin J Lustig, Roy E Marsten, and David F Shanno. Interior point methods for linear programming: Computational state of the art. *ORSA Journal on Computing*, 6(1):1–14, 1994.

[Lop14] Benoit Lopez. *Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (Optimal implementation of linear filters in fixed-point arithmetic)*. PhD thesis, Pierre and Marie Curie University, Paris, France, 2014.

[LS13] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63:443–466, 2013.

[LSM+24] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*, 2024.

[Lua06] Louis Luangkesorn. Introduction to r-glpk. 2006.

[Mal19] Yassine Maleh. Machine learning techniques for iot intrusions detection in aerospace cyber-physical systems. In *Machine Learning and Data Mining in Aerospace Technology*, pages 205–232. Springer, 2019.

[Mas92] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *Proceedings of the 6th international conference on Supercomputing*, pages 226–235, 1992.

[MBdD+18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic (2nd Ed.)*. Springer, 2018.

[MBT+22] Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[Mes02] Frederic Messine. Extensions of affine arithmetic: Application to unconstrained global optimization. *J. UCS*, 8:992–1015, 01 2002.

[MGL+22] Mario Merone, Alessandro Graziosi, Valerio Lapadula, Lorenzo Petrosino, Onorato d'Angelis, and Luca Vollero. A practical approach to the analysis and optimization of neural networks on embedded systems. *Sensors*, 22(20):7807, 2022.

[MHU+22] Khan Muhammad, Tanveer Hussain, Hayat Ullah, Javier Del Ser, Mahdi Rezaei, Neeraj Kumar, Mohammad Hijji, Paolo Bellavista, and Victor Hugo C de Albuquerque. Vision-based semantic segmentation in scene understanding for autonomous driving: Recent achievements, challenges, and outlooks. *IEEE Transactions on Intelligent Transportation Systems*, 23(12):22694–22715, 2022.

[MI23] Fernando V Monteiro and Petros Ioannou. Safe autonomous lane changes and impact on traffic flow in a connected vehicle environment. *Transportation research part C: emerging technologies*, 151:104138, 2023.

[Mic04] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[Min99] Marius Minea. Partial order reduction for model checking of timed automata. In *International Conference on Concurrency Theory*, pages 431–446. Springer, 1999.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19:31–100, 2006.

[MJ16] Guido Manfredi and Yannick Jestin. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9, 2016.

[MKC09] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[MKE23] Omar El Mellouki, Mohamed Ibn Khedher, and Mounim A. El-Yacoubi. Abstract layer for leakyrelu for neural network verification based on abstract interpretation. *IEEE Access*, 11:33401–33413, 2023.

[MM16] Paolo Montuschi and Jean-Michel Muller. Modern computer arithmetic. *Computer*, 49(9):12, 2016.

[MPMF23] Giosuè Cataldo Marinò, Alessandro Petrini, Dario Malchiodi, and Marco Frasca. Deep neural networks compression: A comparative survey and choice recommendations. *Neurocomputing*, 520:152–170, 2023.

[MRMA20] Sebastian Maierhofer, Anna-Katharina Rettinger, Eva Charlotte Mayer, and Matthias Althoff. Formalization of interstate traffic rules in temporal logic. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 752–759. IEEE, 2020.

[MSB22] Samvid Mistry, Indranil Saha, and Swarnendu Biswas. An MILP encoding for efficient verification of quantized deep neural networks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(11):4445–4456, 2022.

[MSS+21] Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. Scaling polyhedral neural network verification on gpus. *Proceedings of Machine Learning and Systems*, 3:733–746, 2021.

[MWMC10] A Miele, T Wang, JA Mathwig, and M Ciarcia. Collision avoidance for an aircraft in abort landing: Trajectory optimization and guidance. *Journal of optimization theory and applications*, 146:233–254, 2010.

[MY59] Ramon E Moore and CT Yang. Interval analysis i. *Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, USA*, 1959.

[MZ09] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[N+61] Ivan Niven et al. *Numbers: rational and irrational*, volume 1. Random House New York, 1961.

[Naj14] Mohamed Amine Najahi. *Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks. (Synthèse de programmes certifiés en arithmètique à virgule fixe, et son application à des briques de base d'algèbre linéaire)*. PhD thesis, University of Perpignan, France, 2014.

[NBHP+21] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoit Miramond, and Vincent Gripon. Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9):2984, 2021.

[NFA+] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. arxiv 2021. *arXiv preprint arXiv:2106.08295*, 4.

[NHP+21] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoît Miramond, and Vincent Gripon. Quantization and deployment of deep neural networks on microcontrollers. *Sensors*, 21(9):2984, 2021.

[NM65] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[NPSS10] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto L. Sangiovanni-Vincentelli. Calcs: SMT solving for non-linear convex constraints. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 71–79. IEEE, 2010.

[NSS+16] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[OLS+19] Matthew Osborne, Jennifer Lantair, Zain Shafiq, Xingyu Zhao, Valentin Robu, David Flynn, and John Perry. Uas operators safety and reliability survey: Emerging technologies towards the certification of autonomous uas. In *2019 4th International Conference on System Reliability and Safety (ICSRS)*, pages 203–212, 2019.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[OPM+19] Michael P. Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. Acas xu: Integrated collision avoidance and detect and avoid capability for uas. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.

[Par02] Behrooz Parhami. Number representation and computer arithmetic. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 317–333. Academic Press, 2002.

[PCYJ18] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. *GetMobile Mob. Comput. Commun.*, 22(3):36–38, 2018.

[Pei21] Jiaming Pei. Mnist. https://doi.org/10.21227/pch7-tk58, November 2021. Accessed on YYYY-MM-DD.

[PL03] S. Paschalakis and P. Lee. Double precision floating-point arithmetic on fpgas. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, pages 352–358, 2003.

[Ple17] Antoine Plet. Contribution to error analysis of algorithms in floating-point arithmetic. (contribution à l'analyse d'algorithmes en arithmétique à virgule flottante), 2017.

[PRA19] Pavithra Prabhakar and Zahra Rahimi Afzal. Abstraction based output range analysis for neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[PSK11] Manolis Papadrakakis, George Stavroulakis, and Alexander Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid cpu–gpu architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13-16):1490–1508, 2011.

[PT10] Luca Pulina and Armando 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*, pages 243–257. Springer, 2010.

[PT12] Luca Pulina and Armando Tacchella. Challenging SMT solvers to verify neural networks. *AI Commun.*, 25(2):117–135, 2012.

[PU20a] Larry D. Pyeatt and William Ughetta. Chapter 8 - non-integral mathematics. In Larry D. Pyeatt and William Ughetta, editors, *ARM 64-Bit Assembly Language*, pages 239–292. Newnes, 2020.

[PU20b] Larry D. Pyeatt and William Ughetta. Chapter 9 - floating point. In Larry D. Pyeatt and William Ughetta, editors, *ARM 64-Bit Assembly Language*, pages 293–321. Newnes, 2020.

[PW24] Aditya Parameshwaran and Yue Wang. Temporal logic guided safe navigation for autonomous vehicles. *IFAC-PapersOnLine*, 58(28):1067–1072, 2024. The 4th Modeling, Estimation, and Control Conference – 2024.

[PWB+20] Ivan Papusha, Rosa Wu, Joshua Brulé, Yanni Kouskoulas, Daniel Genin, and Aurora C. Schmidt. Incorrect by construction: Fine tuning neural networks for guaranteed performance on finite sets of examples. *CoRR*, abs/2008.01204, 2020.

[RAK22] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. A comprehensive survey on model quantization for deep neural networks. *CoRR*, abs/2205.07877, 2022.

[REGSV93] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[RFNV20] Nijat Rajabli, Francesco Flammini, Roberto Nardone, and Valeria Vittorini. Software verification and validation of safe autonomous cars: A systematic literature review. *IEEE Access*, 9:4797–4819, 2020.

[RHG+21] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[RHK18] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. *arXiv preprint arXiv:1805.02242*, 2018.

[RM19] Francesca Rossi and Nicholas Mattei. Building ethically bounded ai. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9785–9789, 2019.

[RORF16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2016.

[Roz11] Kristin Y Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.

[RPL+20] Stefan Riedmaier, Thomas Ponn, Dieter Ludwig, Bernhard Schick, and Frank Diermeyer. Survey on scenario-based safety assessment of automated vehicles. *IEEE Access*, 8:87456–87477, 2020.

[RST+20] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*, pages 1–6. IEEE, 2020.

[RTV05] Cornelis Roos, Tamás Terlaky, and J-Ph Vial. Interior point methods for linear optimization. 2005.

[RZB18] Ugo Rosolia, Xiaojing Zhang, and Francesco Borrelli. Data-driven predictive control for autonomous systems. *Annu. Rev. Control. Robotics Auton. Syst.*, 1:259–286, 2018.

[Sal02] Matthew J Saltzman. Coin-or: an open-source library for optimization. *Programming languages and systems in computational economics and finance*, pages 3–32, 2002.

[San12] T Sands. Physics-based control methods. *Advances in Spacecraft Systems and Orbit Determination; InTech Publishers: London, UK*, pages 29–54, 2012.

[SB23] Sanaz Sheikhi and Stanley Bak. Closed-loop ACAS xu neural network verification. In *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23), San Antonio, Texas, USA, May 9, 2023*, volume 96 of *EPiC Series in Computing*, pages 1–8. EasyChair, 2023.

[SBR+] Gagandeep Singh, Mislav Balunovic, Anian Ruoss, Christoph Müller, Jonathan Maurer, Adrian Hoffmann, Maximilian Baader, Matthew Mirman, Timon Gehr, Petar Tsankov, et al. Eran user manual.

[Sch00] Mark Schulze. Linear programming for optimization. 09 2000.

[Sch19] Robin M Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv preprint arXiv:1912.05911*, 2019.

[SDB21] Abhishek Sarda, Shubhra Dixit, and Anupama Bhan. Object detection for autonomous driving using yolo [you only look once] algorithm. In *2021 Third international conference on intelligent communication technologies and virtual mobile networks (ICICV)*, pages 1370–1374. IEEE, 2021.

[SdBG+19] Luiz H. Sena, Iury Valente de Bessa, Mikhail Y. R. Gadelha, Lucas C. Cordeiro, and Edjard Mota. Incremental bounded model checking of artificial neural networks in CUDA. In *IX Brazilian Symposium on Computing Systems Engineering, SBESC 2019, Natal, Brazil, November 19-22, 2019*, pages 1–8. IEEE, 2019.

[SDLK18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics: Results of the 11th International Conference*, pages 621–635. Springer, 2018.

[See04] Matthias Seeger. Gaussian processes for machine learning. *International journal of neural systems*, 14(02):69–106, 2004.

[SEI+24] Sedat Sonko, Emmanuel Augustine Etukudoh, Kenneth Ifeanyi Ibekwe, Valentine Ikenna Ilojianya, and Cosmas Dominic Daudu. A comprehensive review of embedded systems in autonomous vehicles: Trends, challenges, and future directions. *World Journal of Advanced Research and Reviews*, 21(1):2009–2020, 2024.

[SFG22] Christian Schilling, Marcelo Forets, and Sebastián Guadalupe. Verification of neural-network control systems by integrating taylor models and zonotopes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8169–8177, 2022.

[She20] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[SJA+24] Abderahim Salhi, Joseph E Jabour, Thomas L Arnolds, James E Ross, and Haley R Dozier. Leveraging jsbsim and gymnasium: A reinforcement learning approach for air combat simulation. In *World Congress in Computer Science, Computer Engineering & Applied Computing*, pages 271–283. Springer, 2024.

[SKS19] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In Necmiye Ozay and Pavithra Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 147–156. ACM, 2019.

[SMN19] Shervin Shahrdar, Luiza Menezes, and Mehrdad Nojoumian. A survey on trust in autonomous systems. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Computing*, pages 368–386, Cham, 2019. Springer International Publishing.

[Sou24] Asma Soualah. *Scaling up the static analysis of neural networks using affine forms*. Theses, Université de Perpignan, October 2024.

[Spr22] Christopher Sprague. *Efficient and Trustworthy Artificial Intelligence for Critical Robotic Systems*. PhD thesis, Kungliga Tekniska högskolan, 2022.

[SPV17] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 46–59, 2017.

[SRP22] Julius Schöning, Adrian Riechmann, and Hans-Jürgen Pfisterer. AI for closed-loop control systems: New opportunities for modeling, designing, and tuning control systems. In *ICMLC 2022: 14th International Conference on Machine Learning and Computing, Guangzhou, China, February 18 - 21, 2022*, pages 318–323. ACM, 2022.

[SSSH17] Tsubasa Sasaki, Kenji Sawada, Seiichi Shin, and Shu Hosokawa. Model based fallback control for networked control system via switched lyapunov function. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(10):2086–2094, 2017.

[ST20] Matthew Sotoudeh and Aditya V Thakur. Abstract neural networks. In *Static Analysis: 27th International Symposium, SAS 2020, Virtual Event, November 18–20, 2020, Proceedings 27*, pages 65–88. Springer, 2020.

[STRP23] Qunying Song, Kaige Tan, Per Runeson, and Stefan Persson. Critical scenario identification for realistic testing of autonomous driving systems. *Softw. Qual. J.*, 31(2):441–469, 2023.

[SU15] Alexander G Schwing and Raquel Urtasun. Fully connected deep structured networks. *arXiv preprint arXiv:1503.02351*, 2015.

[SVB+20] Rasmus Stagsted, Antonio Vitale, Jonas Binz, Leon Bonde Larsen, Yulia Sandamirskaya, et al. Towards neuromorphic control: A spiking neural network based pid controller for uav. RSS, 2020.

[SWR+18] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE, Montpellier, France*, pages 109–119. ACM, 2018.

[SWWB15] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *MBMV*, pages 30–40, 2015.

[TFA24] Nadra Tabassam, Martin Fränzle, and Muhammad Waleed Ansari. A contract-based design methodology for safety in autonomous vehicles. In *Proceedings of KES-STS International Symposium*, pages 91–105. Springer, 2024.

[Thé14] Laurent Thévenoux. *Synthèse de code avec compromis entre performance et précision en arithmétique flottante IEEE 754. (Code Synthesis to Optimize Accuracy and Speed in Floating-Point Arithmetic)*. PhD thesis, University of Perpignan, France, 2014.

[Tin02] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *European Workshop on Logics in Artificial Intelligence*, pages 308–319. Springer, 2002.

[TKAKP22] Ehsan Tanghatari, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. Distributing dnn training over iot edge devices based on transfer learning. *Neurocomputing*, 467:56–65, 2022.

[TL19] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[TLM+19] Hoang-Dung Tran, Diego Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-based reachability analysis of deep neural networks. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years -*

Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings, volume 11800 of *Lecture Notes in Computer Science*, pages 670–686. Springer, 2019.

[TV22] Tolga Turay and Tanya Vladimirova. Toward performing image classification and object detection with convolutional neural networks in autonomous driving systems: A survey. *IEEE Access*, 10:14076–14119, 2022.

[TXT19] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[TZW+23] Yang Tang, Chaoqiang Zhao, Jianrui Wang, Chongzhen Zhang, Qiyu Sun, Wei Xing Zheng, Wenli Du, Feng Qian, and Jürgen Kurths. Perception and navigation in autonomous systems in the era of learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 34(12):9604–9624, 2023.

[UM21] Caterina Urban and Antoine Miné. A review of formal methods applied to machine learning. *CoRR*, abs/2104.02466, 2021.

[vdBdV18] Remon van den Brandt and CC de Visser. Safe flight envelope uncertainty quantification using probabilistic reachability analysis. *IFAC-PapersOnLine*, 51(24):628–635, 2018.

[VS10] Jose M. G. Vilar and Leonor Saiz. Cplexa: a *Mathematica* package to study macromolecular-assembly control of gene expression. *Bioinform.*, 26(16):2060–2061, 2010.

[VSD+21] Vagisha Vartika, Swati Singh, Subhranil Das, Sudhansu Kumar Mishra, and Sitanshu Sekhar Sahu. A review on intelligent pid controllers in autonomous vehicle. *Advances in Smart Grid Automation and Industry 4.0: Select Proceedings of ICETSGAI4. 0*, pages 391–399, 2021.

[VV07] Xenia Vamvakoussi and Stella Vosniadou. How many numbers are there in a rational numbers interval? constraints, synthetic models and the effect of the number line. *Reframing the conceptual change approach in learning and instruction*, pages 265–282, 2007.

[WB24] Pierre-Loïc Garoche Wahiba Bachiri, Yassamine Seladji. Parallel verification of quantized neural network for autonomous vehicle. 2024. ISCC'24, 12-13 Novembre 2024, Tlemcen, Algeria.

[WBCG00] Poul F Williams, Armin Biere, Edmund M Clarke, and Anubhav Gupta. Combining decision diagrams and sat procedures for efficient symbolic

model checking. In *International Conference on Computer Aided Verification*, pages 124–138. Springer, 2000.

[WDF+14] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, and Kerstin Dautenhahn. Formal verification of an autonomous personal robotic assistant. In *2014 AAAI Spring Symposia, Stanford University, Palo Alto, California, USA, March 24-26, 2014*. AAAI Press, 2014.

[WIZ+24] Haoze Wu, Omri Isac, Aleksandar Zeljic, Teruhiro Tagomori, Matthew L. Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, Pei Huang, Ori Lahav, Min Wu, Min Zhang, Ekaterina Komendantskaya, Guy Katz, and Clark W. Barrett. Marabou 2.0: A versatile formal analyzer of neural networks. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2024.

[WKJC22] A. N. Wilson, Abhinav Kumar, Ajit Jha, and Linga Reddy Cenkeramaddi. Embedded sensors, communication technologies, computing platforms and machine learning for uavs: A review. *IEEE Sensors Journal*, 22(3):1807–1826, 2022.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[WM22] Ruigang Wang and Ian R Manchester. Youla-ren: Learning nonlinear feedback policies with robust stability guarantees. In *2022 American Control Conference (ACC)*, pages 2116–2123. IEEE, 2022.

[WMA12] Danny Weyns, Sam Malek, and Jesper Andersson. FORMS: unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.*, 7(1):8:1–8:61, 2012.

[WMYY24a] Lu Wei, Zhong Ma, Chaojie Yang, and Qin Yao. Advances in the neural network quantization: A comprehensive review. *Applied Sciences*, 14(17):7445, 2024.

[WMYY24b] Lu Wei, Zhong Ma, Chaojie Yang, and Qin Yao. Advances in the neural network quantization: A comprehensive review. *Applied Sciences*, 14(17), 2024.

[WNSO23] Yifan Wang, Masaki Nakamura, Kazutoshi Sakakibara, and Yuki Okura. Formal specification and verification of an autonomous vehicle control system

by the ots/cafeobj method (S). In Shi-Kuo Chang, editor, *The 35th International Conference on Software Engineering and Knowledge Engineering, SEKE 2023, KSIR Virtual Conference Center, USA, July 1-10, 2023*, pages 363–366. KSI Research Inc., 2023.

[Wög05] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, Tech. rep., Technische Universität Wien, 2005.

[WOZ+20] Haoze Wu, Alex Ozdemir, Aleksandar Zeljic, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. Parallelization techniques for verifying neural networks. In # *PLACEHOLDER_PARENT_METADATA_VALUE*#, volume 1, pages 128–137. TU Wien Academic Press, 2020.

[WPW+18] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. pages 6369–6379, 2018.

[WXT+13] Fei Wen, Zhuo Xu, Shaobo Tan, Weimin Xia, Xiaoyong Wei, and Zhicheng Zhang. Chemical bonding-induced low dielectric loss and low conductivity in high-k poly (vinylidenefluoride-trifluorethylene)/graphene nanosheets nanocomposites. *Acs Applied Materials & Interfaces*, 5(19):9411–9420, 2013.

[WZX+21a] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 29909–29921, 2021.

[WZX+21b] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network verification. In *NeurIPS*, 2021.

[X+22] Kaidi Xu et al. Verification-aware quantization for neural networks. *IEEE Transactions on Computers*, 71(8):1871–1885, 2022.

[XMW+18] Weiming Xiang, Patrick Musau, Ayana A. Wild, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Joel A. Rosenfeld, and Taylor T. Johnson. Verification for machine learning, autonomy, and neural networks survey. *CoRR*, abs/1810.01989, 2018.

[XSZ+20] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33:1129–1141, 2020.

[YG14] Anil Kumar Yadav and Prerna Gaur. Ai-based adaptive control and design of autopilot system for nonlinear uav. *Sadhana*, 39:765–783, 2014.

[YZ18] Joe Yin and Zhiqiang Zhu. Flight autonomy impact to the future avionics architecture. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–7, 2018.

[ZCS+24] Yedi Zhang, Guangke Chen, Fu Song, Jun Sun, and Jin Song Dong. Certified quantization strategy synthesis for neural networks. In *International Symposium on Formal Methods*, pages 343–362. Springer, 2024.

[ZDRF21] Osama Zaki, Matthew W. Dunnigan, Valentin Robu, and David Flynn. Reliability and safety of autonomous systems based on semantic modelling for self-certification. *Robotics*, 10(1):10, 2021.

[ZH18] Zhihao Zheng and Pengyu Hong. Robust detection of adversarial attacks by modeling the intrinsic properties of deep neural networks. *Advances in neural information processing systems*, 31, 2018.

[ZHZ+21] Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott A. Mahlke, and Z. Morley Mao. A systematic framework to identify violations of scenario-dependent driving rules in autonomous vehicle software. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(2):15:1–15:25, 2021.

[ZLDC23] Jiaxin Zhao, Pingli Lu, Changkun Du, and Fangfei Cao. Active fault-tolerant strategy for flight vehicles: Transfer learning-based fault diagnosis and fixed-time fault-tolerant control. *IEEE Transactions on Aerospace and Electronic Systems*, 60(1):1047–1059, 2023.

[ZSS23] Yedi Zhang, Fu Song, and Jun Sun. Qebverif: Quantization error bound verification of neural networks. 13965:413–437, 2023.

[ZWW+17] Shuchang Zhou, Yuzhi Wang, He Wen, Qinyao He, and Yuheng Zou. Balanced quantization: An effective and efficient approach to quantized neural networks. *J. Comput. Sci. Technol.*, 32(4):667–682, 2017.

[ZZC+22] Yedi Zhang, Zhe Zhao, Guangke Chen, Fu Song, Min Zhang, Taolue Chen, and Jun Sun. Qvip: an ilp-based formal verification approach for quantized neural networks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.