Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Verification of Neural Networks Containing Non-Linear Activation Functions

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften
der Carl von Ossietzky Universität Oldenburg
zur Erlangung des Grades und Titels eines

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

angenommene Dissertation von

Farzaneh Moradkhani

Gutachter:

Prof. Dr. Martin Fränzle
Prof. Dr. Oliver Kramer

Tag der Disputation:
21.03.2024

Zusammenfassung

Im heutigen technologischen Umfeld repräsentieren *cyber-physische Systeme* (CPS) eine wegweisende Konvergenz physikalischer Prinzipien und logischer Konstrukte. Diese Systeme umfassen verschiedene Komponenten, die von digitalen und analogen Elementen bis hin zur physischen Welt reichen. CPS wird als ein zentrales Forschungsgebiet der Gegenwart anerkannt und ist wegweisend für die künftige Systemgestaltung und -entwicklung. Besonders die funktionale Architektur derzeit entstehender Systeme, wie sie in autonomen Fahrzeugen exemplarisch zu sehen ist, integriert zunehmend Komponenten, die durch künstliche Intelligenz-basierte Modelle wie maschinelles Lernen generiert werden, verbunden durch Echtzeit-Netzwerke. Dieser Wandel weg von konventionellen Ingenieuransätzen führt zu einer heterogenen Kombination von Komponenten und schafft komplexe Wechselwirkungen, die bei der Analyse große Herausforderungen darstellen.

Die Validierung und Verifizierung der Funktionalität von CPS, insbesondere in sicherheitskritischen Bereichen, bringt eine Vielzahl ungelöster Komplexitäten mit sich. Trotz ihres Potenzials fehlt es den gängigen Rechenstrukturen, einschließlich *tiefer neuronaler Netze* (DNNs), an skalierbaren, automatisierten Verifizierungsmechanismen. Die inhärente Größe, Nichtlinearität und Nichtkonvexität von *künstlichen neuronalen Netzen* (ANNs) machen sie für bestehende Verifizierungsmethoden, wie *gemischt-ganzzahlige lineare Programmierung* (MILP) Solver und *Satisfiability Modulo Theories* (SMT) Solver, besonders herausfordernd.

Dieses Dissertationsprojekt widmet sich einer gezielten Erforschung von ANNs mit Aktivierungsfunktionen jenseits der weit verbreiteten *Rectified Linear Unit* (ReLU). Ein wesentlicher Aspekt dieser Forschung liegt in der Anwendung des SMT-Solvers iSAT, der sich durch seine Fähigkeit auszeichnet, boolesche Kombinationen von linearen und nichtlinearen Constraints zu behandeln, selbst wenn sie transzendente Funktionen umfassen. Diese Eigenschaft macht iSAT besonders geeignet, um die Sicherheitseigenschaften von ANNs mit nichtlinearen Transferfunktionen zu verifizieren.

Abstract

In today's technological landscape, *cyber-physical systems* (CPS) represent a pivotal convergence of physical principles and logical constructs. These systems encompass diverse components ranging from digital and analog elements to the physical realm. Recognized as a key domain of contemporary research, CPS is poised to revolutionize future system design and development. Notably, the functional architecture of emerging systems, exemplified by autonomous vehicles, increasingly integrates components generated through artificial intelligence-driven models like machine learning, connected by real-time networks. This shift away from conventional engineering approaches introduces a heterogeneous amalgamation of components, establishing intricate interplays that pose formidable analysis challenges.

Validating and verifying the functionality of CPS, especially in safety-critical domains, presents a host of unresolved complexities. Despite their potential, prevalent computational structures, including *deep neural networks* (DNNs), lack scalable, automated verification mechanisms. The inherent size, non-linearity, and non-convexity of *artificial neural networks* (ANNs) make them particularly challenging for existing verification methods, such as *mixed integer linear programming* (MILP) solvers and *satisfiability modulo theories* (SMT) solvers.

This doctoral project undertakes a focused exploration of ANNs with activation functions beyond the widely-used *rectified linear unit* (ReLU). A significant facet of this research centers on the application of the SMT solver iSAT, uniquely adept at tackling Boolean combinations of linear and non-linear constraint formulas, even extending to transcendental functions. This characteristic renders iSAT suitable for verifying the safety properties of ANNs characterized by non-linear transfer functions.

Contents

1 Introduction **1**
1.1 Related Work 4
1.2 Objectives 5
1.3 Challenges 7
1.4 Structure 8

2 Background **9**
2.1 Neural Networks 10
2.1.1 Overview 10
2.1.2 Feed-Forward Neural Networks 11
2.1.3 Recurrent Neural Networks 18
2.1.4 LSTM Neural Networks 20
2.1.5 Frequently Encountered Issues 22
2.1.6 Frameworks and Libraries 24
2.2 Verification 25
2.2.1 Boolean Satisfiability Problem 25
2.2.2 Satisfiability Modulo Theories 27
2.2.3 iSAT 28
2.2.4 dReal 33

3 Rationale for Verifying Neural Networks **37**
3.1 Why is Verification of Neural Network Important? 37
3.2 Different Types of Formal Methods 38
3.3 Verification Conditions Applicable to Neural Networks 41
3.3.1 Verification Conditions for Individual Systems and Neural Networks 41
3.3.2 Verification Conditions for Pairs of Neural Networks 42
3.3.3 Verification Conditions for Joint Dynamic Behavior of an NN and its Physical Context 43

4 Verification of Neural Networks **45**
4.1 Methodology 46
4.2 Translating to iSAT 46
4.2.1 Encoding a Feed-Forward Neural Network into iSAT 49
4.2.2 Declaration of Input Data into iSAT 50
4.2.3 Encoding LSTMs into iSAT 53
4.3 Property Definition 57
4.4 Optimization 58
4.4.1 Enhancing Boundaries 58
4.4.2 Managing the Numerical Error During Boundary Calculation . . 63
4.4.3 Detecting Constant Sub-Expressions and Optimizing their Encoding 65
4.5 Converter 67

5 Evaluation **69**
5.1 Benchmark 1: MNIST Dataset 70
5.2 Benchmark 2: NGSIM 84
5.3 Benchmark 3: Satellite Collision Detection 89

6 Conclusion and Further Research **95**
6.1 Conclusion 96
6.2 Future Research 99

Bibliography **101**

Appendix A Acronyms **113**

List of Figures

2.1 A single layer feed-forward neural network 12
2.2 A FFNN with n input nodes, 2 output nodes, and 3 hidden layers. 13
2.3 Unit step or binary step activation function 15
2.4 A spectrum of activation functions shaping neural network behavior, from (A) Identity to (D) ReLU, (B) Sigmoid, and (C) Tanh. 17
2.5 Swish activation functions 18
2.6 Recurrent neural network element (left). The behavior at any time point is given by the unfolding of its time-discrete feedback behavior (right). . . 19
2.7 LSTM structure. [YSHZ19] 21
2.8 Interval constraint propagation (ICP) performed on constraint e^x 29
2.9 iSAT code of a bouncing ball [FHT+07, SKB13] 30
2.10 Activation function showcase: Softplus, ELU, SELU, and Swish. Dive into the diverse curves shaping neural network behavior. 33
2.11 A snippet of dReal represented in the SMT-LIB 2.0 format [GAC12]. . . 34

4.1 ONNX Operator graph of a neural network 48
4.2 Declaration of the Input neurons of the translated network, showcasing 7 neurons in iSAT 50
4.3 Declaration of the first layer of the translated network, showcasing 7 neurons in iSAT 51
4.4 Structure of the LSTM with dashed circles representing spaces reserved for auxiliary variables exclusively needed for iSAT encoding [YSHZ19]. . 55
4.5 iSAT code snapshot: X_0_0 to X_0_4 indicate the input values in every time steps which here $\mu = 3$ 56
4.6 iSAT code snapshot: first layer of translated LSTM neurons with auxiliary variables. X_0_0 to X_0_9 indicate the input variables. 57
4.7 An illustrative depiction of the connections between neurons within a layer [Fib22]. 63

5.1 Instances of handwritten numbers of MNIST dataset [LCB10]. 71

5.2 Comparison of the solver time of iSAT2, iSAT3, and dReal4 for the verification process of $property1$ of networks using the sigmoid function. Red lines indicate a 12-hour time limit, and the blue dotted line shows where solver times converge. A logarithmic scale on both axes enhances the visibility of swift resolutions. 77
5.3 Comparison of the solver time of iSAT2, iSAT3, and dReal4 for the verification process of $property1$ of networks using the Tanh function. Red lines indicate a 12-hour time limit, and the blue dotted line shows where solver times converge. A logarithmic scale on both axes enhances the visibility of swift resolutions. 83
5.4 Visualization of the distance between cars 84
5.5 Visualization of the density of objects orbiting the low Earth orbit [CFS, Eur] 90

6.1 Architecture and execution of our study's methodology 97

List of Tables

2.1 Arithmetic operators available in iSAT. Operators marked with an * are only available in iSAT3 [isa10] 31
2.2 Recently released activation functions 32
2.3 commands available in the SMT-LIB 2.0 format [BST+10] 34

4.1 File format in different frameworks 46
4.2 Properties of the graph object in ONNX [onn] 47

5.1 Solver time of iSAT2 applied to $property1$ of the network on the MNIST dataset using the sigmoid function (in seconds) 74
5.2 Solver time of iSAT3 applied to $property1$ of the network on the MNIST dataset using the sigmoid function (in seconds) 75
5.3 Solver time of dReal applied to $property1$ of the network on the MNIST dataset using the sigmoid function (in seconds) 76
5.4 Percentage of aborted verification runs for networks using the sigmoid function grouped by the solver and optimization level for $property1$. . . . 78
5.5 The highest number of layers for which each solver successfully obtained results while assessing $property1$ employing the sigmoid function. 78
5.6 Solver time of iSAT2 applied to $property1$ of the network on the MNIST dataset using the Tanh activation function (in seconds) 79
5.7 Solver time of iSAT3 applied to $property1$ of the network on the MNIST dataset using the Tanh activation function (in seconds) 80
5.8 Solver time of dReal for $Property$ 1 of networks (in second) 81
5.9 Percentage of aborted verification runs of the $property1$ for networks using the hyperbolic tangent function grouped by solver and optimization level 82
5.10 The highest count of network layers that each solver successfully addressed for $property1$ using the hyperbolic tangent function. T signifies the termination of the solver process due to a timeout. 82
5.11 NGSIM database [Adm07] 84
5.12 The result of verifying the NGSIM case study 89

5.13 Memory usage and time for solving of iSAT2 and iSAT3 on satellite collision detection data-set for *property*1 92

Chapter 1

Introduction

Cyber-physical systems [Jaz14] merge physical and computational elements into an intricately integrated system, empowering physical objects with intelligence and adaptability beyond their inherent properties. These systems are anticipated to play a pivotal role in shaping future smart technologies, encompassing areas such as smart cities, advanced supply chains, autonomous transportation, and automated healthcare.

The functional architectures of CPSes, like those seen in autonomous vehicles [KKLR13], are increasingly being shaped by machine learning methods rather than traditional engineering approaches. Although such systems offer significant advantages in analyzing and predicting vast and intricate datasets, they lack strong mathematical guarantees concerning essential functionalities, unlike most historically employed architectures. Consequently, integrating machine-learning-enabled CPSes into CPS architectures, which involve a diverse array of functional building blocks interacting in heterogeneous combinations, poses a formidable challenge in terms of analyzing and detecting potential errors. One of the primary methods for testing cyber-physical systems and their associated functional design models involves utilizing simulation tools [DLV11]. Through simulations, developers aim to provide evidence that these systems will behave as intended. For instance, companies working on self-driving vehicles rely on simulation results (pertaining to software, hardware, etc.) to establish safety cases and meet functional safety standards. However, a crucial question arises: can simulation-based testing alone suffice for critical systems? Will it effectively identify potentially unsafe, inefficient, or incomplete features?

A notable incident occurred in March 2018 when a Tesla Model X, operating with

autopilot driving assistance, was involved in a fatal accident, colliding with a concrete median on Highway 101 in Mountain View, California, after millions of test miles [Dic18, Lam18]. Ensuring the societal acceptance of intelligent cyber-physical systems depends on mitigating such risks adequately.

Both the research community and the industry are thus intensely pursuing the idea of complementing extensive simulation and testing with formal verification of essential safety properties in machine-learning-enabled CPSes, particularly in the context of *automated vehicles* (AV). This approach aims to address concerns and enhance confidence in the safety and reliability of smart systems. However, this presents a significant challenge: artificial neural networks are trained using samples and heuristic optimization techniques, leading to an approximation of a mathematically complex statistical inference. The statistical nature of this inference and the inherent imprecision in ANNs introduce uncertainties into their reasoning process, affecting the output of each network. Due to these uncertainties and the absence of an accessible closed-form specification of their function, post-training verification methods become crucial for assessing the functional well-behavedness and reliability of machine-learning-based systems.

Currently, the only hope for providing reasonable guarantees of these systems' functionality is through a functional verification method, inspecting the behavior of the trained neural network. This becomes particularly vital when dealing with large and complex neural networks. Ensuring high safety standards in smart safety-critical devices like automated vehicles necessitates the use of formal methods in functional verification [ZJ15, CW96]. However, the hybrid discrete-continuous system structure and the heterogeneity of components in safety-critical cyber-physical systems pose significant challenges when applying functional verification techniques. This issue is already relevant to traditional engineered designs, but it becomes even more pronounced with the integration of machine learning components, particularly artificial neural networks, in cyber-physical systems for environment analysis. The incorrect outputs from neural networks can potentially lead to dangerous situations and result in significant consequences, particularly in critical systems.

As a result, there is an urgent need to verify ANNs, which poses a significant, yet mostly unsolved, scientific challenge. Ensuring the reliability and safety of these machine learning components is of paramount importance in mitigating potential hazards and avoiding costly outcomes.

An ANN is a network structure comprising interconnected elements known as neurons [AJO+18, AKB00]. Each neuron has inputs, and outputs, and performs simple localized data-flow operations. While the network's structure, including the number, types, layout, and interconnections of neurons, is typically predefined manually by a designer, neural networks learn significant aspects of their function during a training process. This training

involves adjusting parameters within the neuron functions to achieve specific goals.

Once a neural network is trained, it is possible to translate it into a closed-form functional equation that describes its actual function. This translation can be achieved by taking the characteristic equations of each neuron function, similar to the approach used in digital circuit verification for many years, along with the interconnect structure. By logically combining these characteristic equations, we obtain an exact representation of the input-output activation function of the analog combinatorial circuit that the *feed-forward neural network* (FFNNs)) [CW96, BG94] represents, which might be potentially deep.

Formal verification [Bje05] of a neural network then involves conducting a mathematical analysis of the properties of this resulting mathematical function. By performing this verification, we can gain insights into the network's behavior and ensure its reliability and correctness for various applications. The latter analysis can, in principle, be mechanized using interactive or automatic formal verification techniques. However, computational challenges arise when dealing with neural networks with a substantial number of neurons, which is often the case in practice. This results in unwieldy mathematical descriptions of the network function, containing as many mathematical operators as the number of neurons in the network, and often even more due to the need to describe each neuron as a composition of simpler mathematical functions and the sharing of subnetwork outputs facilitated by the network interconnect.

The complexity of calculating properties for such large networks primarily arises from the presence of a significant fraction of non-linear functions, attributable to the *activation functions* (AFs) [SSA17] of neurons. Non-linear activation functions are essential for introducing non-linearity into the overall activation function, allowing the definition of non-monotonic functions or non-convex regions that underlie decisions. Without these non-linear AFs, neural networks would struggle to learn even the simplest non-monotonic Boolean operators, like *exclusive or* (XOR).

Furthermore, they would be ill-suited for modeling classification tasks involving complex input data types, such as images, videos, audio, speech, etc. However, the non-linear nature of activation functions gives rise to a massive non-convex constraint problem in the verification of neural networks [KBD+17b], making it difficult to find scalable automatic solving methods for this challenge. As a result, addressing verification problems in large neural networks with non-linear activation functions remains a formidable task in the field of neural network research.

1.1 Related Work

Despite its youth, the verification of neural networks has garnered increasing attention due to the widespread deployment of these models in critical applications such as autonomous vehicles and healthcare. Researchers are exploring various techniques to ensure the reliability and safety of neural networks, including formal methods, robustness analysis, and adversarial testing. As the complexity and scale of neural networks continue to grow, addressing the verification challenges becomes paramount for maintaining trust in AI systems. Two primary methodologies are commonly employed to tackle property verification in artificial neural networks: mixed integer linear programming and SAT modulo theory utilizing dedicated SMT solvers. These approaches play crucial roles in ensuring the reliability and safety of neural network models by verifying properties such as robustness, safety constraints, and general performance. By leveraging mixed integer linear programming and SAT modulo theory, researchers aim to enhance the trustworthiness and deployment readiness of ANNs across diverse applications. Several studies have utilized SMT solvers directly for property verification in neural networks, including [BR23, BR23, KBD+17b, BIL+16, HKWW17, LM17, Ehl17]. For instance, the Reluplex architecture introduced in [KBD+17a] deals with ReLU-based networks, where the primary non-linearity is the piecewise linear ReLU node (essentially a clipped identity function). To handle the complexity arising from ReLU activation functions, Reluplex employs a lazy theorem proving and conflict-driven clause learning approach. The architecture takes a neural network description as input along with the desired property of the network's activation function, expressed as an SMT formula. This property is later appended to the SMT formulation of the network's activation function. By integrating *linear programming* (LP) [DT03] to address linear weighting nodes and employing a dedicated constraint propagator for ReLU nodes within an SMT core for linear arithmetic, the compound formula's satisfiability or unsatisfiability is determined. Another study [GZZ+23] presents an SMT-based method to formally verify deep neural networks' robustness against occlusions, a significant semantic perturbation. By formulating the occlusion robustness verification problem and introducing efficient encoding techniques within neural networks, the approach enables scalable verification using off-the-shelf tools. However, these methods are limited to piecewise linear ReLU-type neural networks.

While it might seem straightforward to extend the underlying verification scheme to more general activation functions by employing more versatile SMT solvers capable of addressing the corresponding non-linear arithmetic fragments, such approaches have proven to be non-scalable for networks of non-trivial size, especially those encountered in real-world problems. [PT10, PT11] use piecewise linear functions to approximate the sigmoid

activation function. For verification, the SMT-Solver HYSAT [FH07] — a predecessor of iSAT — was used. It was shown that a safety-critical property that is verifiable for a consistent abstraction also holds for the original neural network. The downside of this approach is its lack of scalability, as only small networks with just 20 hidden nodes could be verified [PT10, PT12].

Regarding the verification of more intricate neural network architectures, such as *recurrent neural networks* (RNNs) [MC01], notable research efforts have already been undertaken. Akintunde et al. [AKLP19] have achieved formal verification of RNNs by transforming them into feed-forward Neural Networks and converting the resulting verification problem into a mixed-integer linear program. Jacoby et al. [JBK20b] proposed a method for the formal verification of systems consisting of a stateful agent implemented as an RNN interacting with an environment. Their approach involves utilizing inductive invariants to reduce RNN verification to FFNN verification. However, both of these techniques are currently applicable only to piecewise linear, ReLU-type neural networks.

In the context of more general and flexible classes of RNNs, which include *long short-term memory* (LSTM) networks [She20] and *generative adversarial networks* (GANs) [PYY+19], the presence of layers with nonlinear transfer functions, such as sigmoid and *hyperbolic tangent function* (tanh), poses challenges. Mohammadinejad et al. [MPDW21] proposed a differential verification method for verifying RNNs with nonlinear activation functions. Their verification goal is to certify the approximate equivalence of two structurally similar neural network functions, rather than verifying behavioral invariants. This method offers a promising approach to handling RNNs with various nonlinear activation functions, facilitating their formal verification and enhancing the applicability of verification techniques to a wider range of neural network architectures. [TCY+23] introduces an extended star reachability method to verify the robustness of recurrent neural networks for safety-critical applications with the NNV tool [LCTJ23].

Recent research has expanded the scope of neural network verification to include other architectures, notably convolutional neural networks [TBXJ20], reinforcement learning [CMF21], and semantic segmentation networks [TPM+21]. This exploration employs diverse set-based reachability tools, including JuliaReach [BFF+19], among others, showcasing the versatility of verification methodologies across various neural network structures.

1.2 Objectives

The primary objective of this thesis is to provide automated formal verification of neural networks, with an emphasis on both well-known feed-forward neural networks and

more intricate structures like recurrent neural networks, particularly LSTM networks. The verification process involves ensuring compliance with formal safety specifications. To achieve this, we propose an automatic verification strategy that involves extending the core capabilities of the SMT solver iSAT [FHT+07] to handle networks with nonlinear activation functions, such as sigmoid and tanh. iSAT is well-suited for this task due to two key reasons: (1) its specific design to handle complex Boolean combinations of non-linear arithmetic facts, including transcendental functions, and (2) its integrated mechanisms for *bounded model checking* (BMC) [BCCZ99]. By leveraging these features, our approach aims to facilitate efficient and rigorous verification of neural networks, broadening the scope of formal verification techniques to encompass a wider range of activation functions and network structures.

The principal aim of this project is to establish a rigorous and formal procedure for assessing the outputs generated by both feed-forward and recurrent neural networks. To accomplish this, the project was structured around the pursuit of the following key milestones:

- **Encoding neural networks into iSAT:** The initial step in this project involves encoding the neural networks into expressions compatible with the iSAT solver. This involves formulating the mathematical representation of the network's architecture, including its layers, connections, and activation functions. By mapping the neural network components into iSAT-compatible expressions, we can effectively utilize the solver's capabilities for formal verification and analysis. This step is crucial in enabling iSAT to process and assess the network's behavior and performance based on the specified safety specifications.

- **Conduct a reachability analysis of the recurrent neural network:** Following the encoding of the neural network into iSAT expressions, the project proceeds to conduct a reachability analysis when the network is an RNN. This analysis involves examining the network's behavior and its capability to reach specific states or outcomes from given initial conditions. By performing reachability analysis, we can systematically explore the network's response to different inputs and evaluate its ability to achieve desired or critical states. The reachability analysis plays a crucial role in verifying whether the network meets the specified formal safety specifications. By thoroughly analyzing the reachable states and trajectories, we can gain insights into the network's behavior and identify potential issues or vulnerabilities. This step allows us to assess the network's reliability and performance under various scenarios, helping ensure that it behaves as intended and adheres to the desired safety constraints.

- **Evaluate the scalability of the developed techniques:** The evaluation of the scalability of the developed techniques is an important aspect of this project. It involves assessing the performance and efficiency of the verification procedures as the complexity and size of the neural networks increase. To evaluate scalability, we conduct a series of experiments and analyses with varying network sizes and architectures. We measure the computational resources (such as time and memory) required to perform the verification process for each network configuration. Additionally, we examine the solver's ability to handle larger and more complex networks, including those with a higher number of layers, neurons, and nonlinear activation functions.

1.3 Challenges

The project encountered substantial challenges attributed to the inherent complexity of the addressed problem. These difficulties were primarily a result of the resource-intensive nature of neural network verification. As we reflect on the research and development process, the following challenges have been identified:

- **Dealing with (diverse) variants of NN architectures:** The field of neural networks has witnessed rapid advancements, leading to the creation of diverse architectures, activation functions, optimization techniques, and training algorithms. Each variation may be tailored to specific tasks and domains, resulting in a highly complex landscape of neural network configurations. Addressing this challenge requires a flexible and adaptable approach to verification and analysis. The verification methodologies must be capable of handling different network structures, ranging from simple feed-forward networks to more intricate recurrent neural networks and transformers. Moreover, considering the presence of various activation functions, such as sigmoid, tanh, ReLU, and beyond, further adds to the complexity.

- **Various neural network frameworks:** The existence of different implementations of neural networks, such as TensorFlow [PNW20] and PyTorch [KMKM21], presents a significant challenge in this project. These frameworks offer various functionalities, optimizations, and programming paradigms, making it essential to ensure consistency and compatibility in the verification process. The verification techniques and tools developed for one framework may not be directly transferable to another due to differences in their underlying architecture and computation models. This demands adaptation and possibly re-implementation of verification algorithms to accommodate the specific characteristics of each framework. Furthermore, different implementations may have varying levels of support for formal verification libraries

and solvers, requiring careful consideration when integrating verification tools into each framework. The availability and compatibility of third-party libraries, as well as their support for nonlinear activation functions and recurrent connections, may vary between frameworks.

- **Reachability analysis techniques in RNNs:** Reachability refers to the ability to determine whether a certain state or condition can be reached from a given starting point within a system. The absence of suitable reachability analysis techniques for RNNs limits the ability to benchmark and compare the performance of the proposed verification method. Without established baselines and benchmarks, it becomes challenging to evaluate the efficiency and effectiveness of the new approach objectively.

1.4 Structure

The rest of the thesis is organized as follows: Chapter 2 provides necessary mathematical and algorithmic background information, including an overview of neural networks, Boolean satisfiability problems, and SMT solvers, with a detailed exploration of iSAT [FHT+07] and dReal [GKC13]. This chapter lays the foundation for the subsequent chapters by introducing the fundamental concepts and tools essential for understanding the neural network verification methodologies presented in this thesis. Chapter 3 will explore the various aspects and objectives that neural network verification aims to address. Chapter 4 focuses on the verification of neural networks, discussing the translation of neural network models into iSAT constraint problems, the methodology used for verification, and various optimization techniques to enhance the verification process. Chapter 5 delves into the evaluation of the proposed methods, presenting metrics for assessment and discussing benchmarks involving data sets such as MNIST [LCB10], NGSIM [Adm07], and satellite collision detection [Eur]. Finally, in Chapter 6, the thesis concludes by summarizing the findings and contributions of the research, along with potential future directions in the field of neural network verification.

Chapter 2

Background

The beginning of Chapter 2 takes a review of neural networks, which is a crucial aspect of our research. Our goal is to explain these concepts with clarity, offering a comprehensive overview of the fundamentals that underlie the functioning of neural networks. As we uncover the core elements of their complex structures, we explore the layers of connected neurons. Additionally, we shed light on the important role that activation functions play in influencing the behavior and results of the network.

Moreover, we explore the impact of training algorithms, serving as crucial agents in refining the network's numerous parameters to achieve desired outcomes. Within the expansive domain of neural networks, we place special focus on two distinct yet influential network topologies: feed-forward neural networks and recurrent neural networks. The former, recognized for its prevalence and straightforward information flow, contrasts with the latter, which displays dynamic temporal behavior and cyclic connections. This unique characteristic makes recurrent neural networks adept at processing sequential data, enhancing their suitability for various applications. Our thorough examination of both network types yields profound insights into their distinctive traits and complexities, proving invaluable in addressing intricate verification challenges that arise in safety-critical contexts.

In Section 2.2.3 following that, we introduced iSAT, an SMT solver serving as the verification backend utilized in this thesis. This solver plays a pivotal role in understanding and implementing the verification techniques central to our project. We embark on a detailed journey through the underlying principles of SMT solvers, unveiling their remark-

able effectiveness in addressing complex Boolean combinations of linear and non-linear constraint formulas. Specifically, we shed light on iSAT's exceptional ability to handle transcendental functions, including the sigmoidal and hyperbolic tangent activation functions commonly encountered in the context of neural networks. This makes iSAT's capabilities particularly relevant to the objectives of our project.

2.1 Neural Networks

In this overview, we aim to provide a thorough understanding of the fundamental principles that govern neural networks' operations. By elucidating the intricate interplay of neurons, layers, and activation functions, we aim to establish a clear mental model of how these networks process information and extract meaningful insights from data. As we progress from this broad understanding, we seamlessly transition to the core focal points of our investigation. The first of these is the feed-forward neural network, a prevalent architecture characterized by its unidirectional flow of information. Here, our intention is to unravel the architecture's components, its information propagation mechanisms, and its utilization in various real-world applications.

Simultaneously, our exploration ventures into the realm of recurrent neural networks, a distinct type known for its capacity to effectively handle sequential and time-series data. As we delve into the intricacies of recurrent connections and cyclic behavior, we endeavor to provide a comprehensive perspective on the unique strengths and challenges posed by these networks.

By combining this foundational understanding of neural networks with our focused insights into the distinctive qualities of feed-forward and recurrent neural networks, we set the stage for the innovative verification approach that constitutes the essence of our research.

2.1.1 Overview

In the realm of computational architectures, artificial neural networks, also commonly referred to as connector systems, showcase the emulation of intricate operations found in biological neural networks. Drawing inspiration from their biological counterparts, these networks orchestrate the amalgamation of tens of thousands, and at times millions, of artificial neurons.

A neural network [AJO+18] is composed of fundamental processing components known as units, nodes, or neurons. These units are meticulously organized into multiple layers, each layer housing a multitude of nodes. While the nodes within a specific layer do not

share direct connections amongst themselves, they establish vital connections with nodes residing in adjacent layers. This connectivity pattern forms the backbone that facilitates the seamless transmission of information across the network. Precisely, nodes within a certain layer acquire input signals from nodes in the preceding layer and propagate their output to nodes located in subsequent layers, which exist downstream in the network's progression.

There are two primary classes of network architectures based on the connections between neurons: "feed-forward neural networks" and "recurrent neural networks." A network is classified as a feed-forward neural network when there is no feedback loop from the neuron outputs to the inputs within the network. On the other hand, if such feedback connections exist, where outputs connect to inputs, whether their own or those of other neurons, the network is labeled a recurrent neural network.

2.1.2 Feed-Forward Neural Networks

A feed-forward NN is a neural network where the inner architecture is organized in subsequent layers of neurons and every neuron of a layer is connected to the neurons of the subsequent layer. The dynamicity of the design is another key aspect. Because no memory is permitted in an FFNN, the network can only be used to represent static functions which refer to functions that do not incorporate memory or time-dependent elements. In the context of a feed-forward neural network, where no memory or delay is allowed, the network is limited to representing static functions. This means that the FFNN is suitable for tasks where the input and output are not dependent on previous states or changes over time, making it well-suited for certain types of pattern recognition and classification tasks [HSW90].

Typically, neural networks are organized in layers. Within the category of feed-forward neural networks, there are two subtypes based on the number of layers: "single-layer" and "multi-layer."

Figure 2.1 illustrates a single-layer feed-forward neural network with full connectivity. This structure comprises two layers, but the input layer is not considered as it does not perform any computations. Instead, it simply transmits input signals to the output layer, where the output signals are computed by the neurons using the weights. The term "weights" denotes the parameters of the neural network that determine the magnitude of influence or significance of each input feature on the output of the neural network. These weights essentially signify the importance or impact of each input feature on the network's decision-making process.

A multi-layer feed-forward neural network can be broadly categorized into three dis-

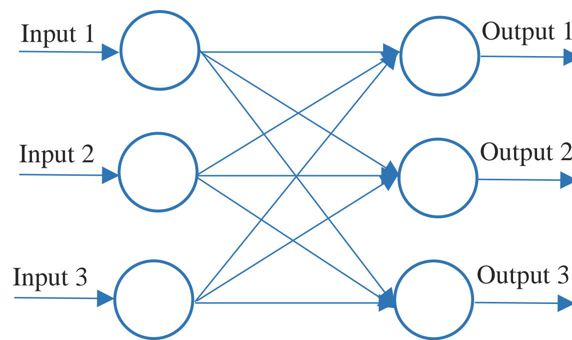


Figure 2.1: A single layer feed-forward neural network

tinct types of units, each with its defined role:

- **Input units**: The initial layer of the network comprises these units. They are responsible for receiving diverse types of information, which the network aims to process, learn, and ultimately identify.
- **Hidden units**: They are situated within intermediate layers, and these internal units play a pivotal role in the network's processing operations. The output often denoted as activation, of a hidden layer node, emerges from either forming a linear combination of its input values or by applying a particular activation function to a solitary input value.
- **Output units**: They are occupying the final layer. These units carry the outcomes of the network's processing tasks. Their role is to provide meaningful results based on the processing performed by the network's hidden layers.

To provide a visual representation, consider a multi-layer feed-forward network depicted in Figure 2.2. This specific example showcases an architecture with n input nodes, 2 output nodes, and 2 hidden layers. The primary goal of the network is to process diverse types of information, aiming to understand, recognize, and analyze this data.

The starting point for this data is the input units, marking the beginning of its journey through the network. As data traverses through the network, hidden layers come into play, imbuing the network with the capability to decipher intricate relationships inherent within the data.

The transition between the network's input and output nodes is facilitated by these concealed layers. Within these layers, a node's output, often referred to as its activation, is governed either by performing an affine combination of its input values or by applying a designated activation function to an individual input value.

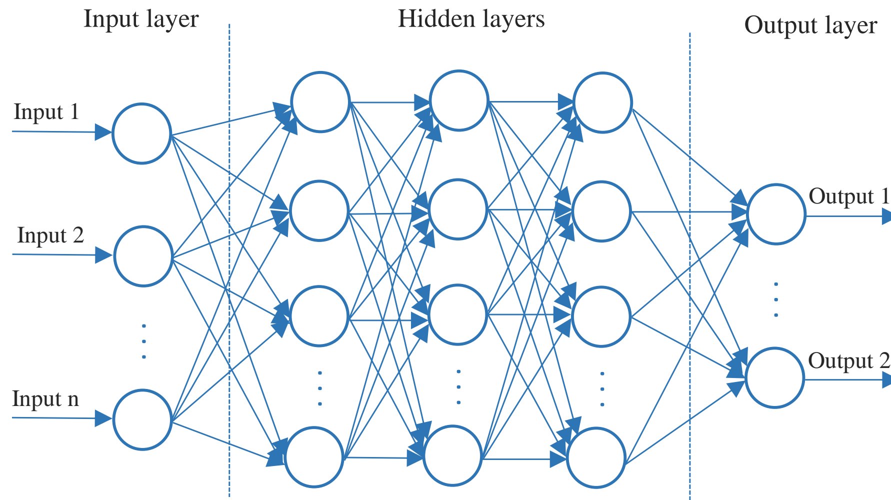


Figure 2.2: A FFNN with n input nodes, 2 output nodes, and 3 hidden layers.

Data is transmitted between neurons through connections, and these connections possess weights that can be either excitatory or inhibitory. In a neural network, layers typically refer to groups of neurons that are organized in a hierarchical manner. Each layer can be connected to the layers before and after it, forming a network.

Definition (feed-forward neural network):

A tuple $(N, V, w, b, F_{\text{act}})$ is a feed-forward neural network, where:

- N: Is the set of neurons.
- V: Is the set of connections between neurons of the form (i, j), where $i, j \in N$, ensuring that the neural network topology represented by (N, V) forms an acyclic graph. This acyclic property is crucial for feed-forward neural networks to ensure that information flows in one direction, from input to output, without any cycles or feedback loops.
- $w : V \to \mathbb{R}$ is a function that specifies the weight associated with each connection. These weights determine the strength of influence one neuron has on another in the network.
- $b : N \to \mathbb{R}$ is a function defining the bias of a neuron.
- $F_{\text{act}} : \{f_{\text{act}_1}, f_{\text{act}_2}, \ldots, f_{\text{act}_m}\} \to \mathbb{R}$ is the activation function, where m is the number of activation functions in the network.

The equation 2.1 depicts an affine transformation with bias. It arises from the definition that, for a given neuron j, there could exist a collection of neurons $I = \{i_1, i_2, \ldots, i_p\}$ for

which $(i, j) \in V$ for every $t \in 1, \ldots, p$. The network input of neuron j, represented as net_j, is defined in equation 2.1 [Kri07].

$$\text{Affine:} \quad net_j = \sum_{t=1}^{p} o_t \cdot w_{t,j} + b_j \tag{2.1}$$

net_j represents the affine transformation of the inputs to neuron j, where o_i is the output of neuron i and $w_{i,j}$ is the weight corresponding to the connection (i, j) in the network structure V. This affine transformation sums the products of the outputs o_i and their corresponding weights $w_{i,j}$ for all neurons i connected to neuron j.

The output of each neuron in the network, such as o_i, is connected to the network structure V. In a feed-forward neural network, this connection structure V ensures that the output of each neuron in a given layer becomes the input to the neurons in the subsequent layer. Specifically, if neuron i is connected to neuron j (i.e., (i, j) is a connection in V), then the output of neuron i contributes to the calculation of the network input net_j for neuron j. The weight $w_{i,j}$ associated with the connection (i, j) determines the extent to which the output of neuron i influences the network input of neuron j. Thus, the connection structure defined by V ensures that information flows through the network during forward propagation.

Equation 2.2 shows the computational process of a feedforward neural network for a single node. net_j represents the affine transformation of the inputs to neuron j and y_j denotes the output of neuron j.

$$y_j = f_{\text{act}}(net_j) = f_{\text{act}}(\sum_{t=1}^{p} o_t \cdot w_{t,j} + b_j) \tag{2.2}$$

Furthermore, Equation 2.3 provides an overview of a feedforward neural network with multiple layers and diverse activation functions. In this equation, y_k represents the output of the network for a specific neuron k. The expression demonstrates the iterative nature of information flow through the network, with each layer applying its respective activation function f_{act_m}.

The equation encapsulates the transformation of inputs o_l through each layer's weights $w_{l,j}$, culminating in the final output y_k. Additionally, bias terms b_{m-1} and b_m are incorporated to adjust the outputs of each layer.

This formulation provides a comprehensive depiction of the network's computation, emphasizing the sequential processing of information across its multiple layers.

$$y_k = f_{\text{act}_m}\left(\sum_{j=0}^{g} f_{\text{act}_{m-1}}\left(\sum_{i=0}^{z} f_{\text{act}_{m-2}}\left(\dots\left(\sum_{l=0}^{s} o_l \cdot w_{l,j}\right) \cdot w_{j,k}\right) \cdot w_{k,n} + b_{m-1}\right) + b_m\right) \tag{2.3}$$

Different characteristics are assigned to the network depending on how these functions are defined. For instance, if the problem relates to classification, a sigmoid function might be appropriate. The hidden number of nodes in the network is denoted by the constant M. The input layer's dimensions and the input vector's size are determined by the constant N. The overall output of the entire network is produced by combining all K output values y_k into a vector.

The activation of a neuron is another factor to consider. A non-linear activation function, often also simply referred to as an activation function, may be present in neurons to non-linearly transform input impulses into output signals. In other words, the activation of a neuron in a neural network depends on its input values as well as its activation function. The activation function's goal is to transform a node's input signal into an output signal in the range, often 0 to 1 or -1 to 1. The subsequent layer receives this output as an input. Our neural network would not be able to categorize non-convex sets or learn non-monotonic functions, which are required to model images and videos, without the use of non-linear activation functions. However, the presence of this non-linear component in ANNs' structure accounts for a large portion of the difficulty in determining the features of these networks. The dynamics of training and task performance in neural networks are significantly influenced by the choice of activation functions. Generally, there are three groups of activation functions. Here we consider value net_j as an input of the activation function (equation 2.1). For simplicity, in the following formula, we assume that x represents the value of net_j.

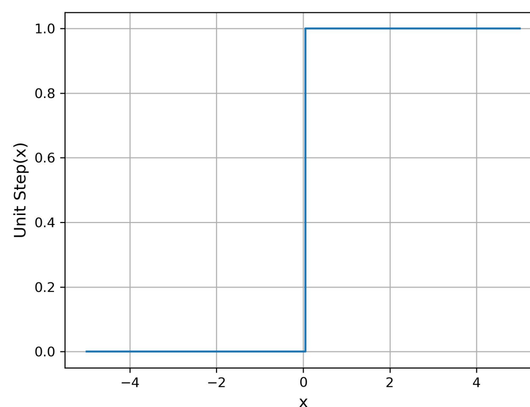


Figure 2.3: Unit step or binary step activation function

Unit Step or Binary Step Function

Figure 2.3 illustrates the unit step or binary step activation function. Unit step is a threshold-based activation function that generates output at two discrete levels (equation 2.4), returning one of the two levels depending on whether the total input is larger or smaller than a given threshold.

$$f_{act}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.4}$$

Identity Activation Functions

It produces an output signal that is proportional to the input by multiplying the inputs by the weights assigned to each neuron. Equation 2.5 illustrates an identity function that is different from a step function in one sense since it allows for several outputs rather than simply yes or no.

$$f_{act}(x) = x \tag{2.5}$$

However, there are several problems. For example, we cannot train the model using gradient descent (backpropagation). This is because the function's derivative is a constant that has no bearing on the input value. Therefore, it is impossible to go back and determine which input neuron weights can produce a better forecast.

Non-Linear Activation Functions

Non-linear activation functions are used in contemporary neural network models. They give the model the ability to construct intricate mappings between the network's inputs and outputs, which are crucial for understanding and simulating complicated data, such as pictures, video, audio, and non-linear or highly dimensional data sets. In this section, we will briefly refer to the most common and popular nonlinear activation functions.

- **Sigmoid (Logistic) function**: A popular activation function transforming its input into an output value between 0 and 1. The smooth gradient of the sigmoid activation function prevents jumps in output values.

$$f_{act}(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

- **Hyperbolic Tangent function (Tanh)**: This function is similar to sigmoid, but its output is zero-centered. The gradient of the tanh is steeper than the sigmoid activation function.

$$f_{act}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.7}$$

- **Rectified Linear Unit (ReLU)**: This function is piecewise linear. ReLU is the identity for all positive input values and yields zero on all negative inputs.

$$f_{act}(x) = max(0, x) \tag{2.8}$$

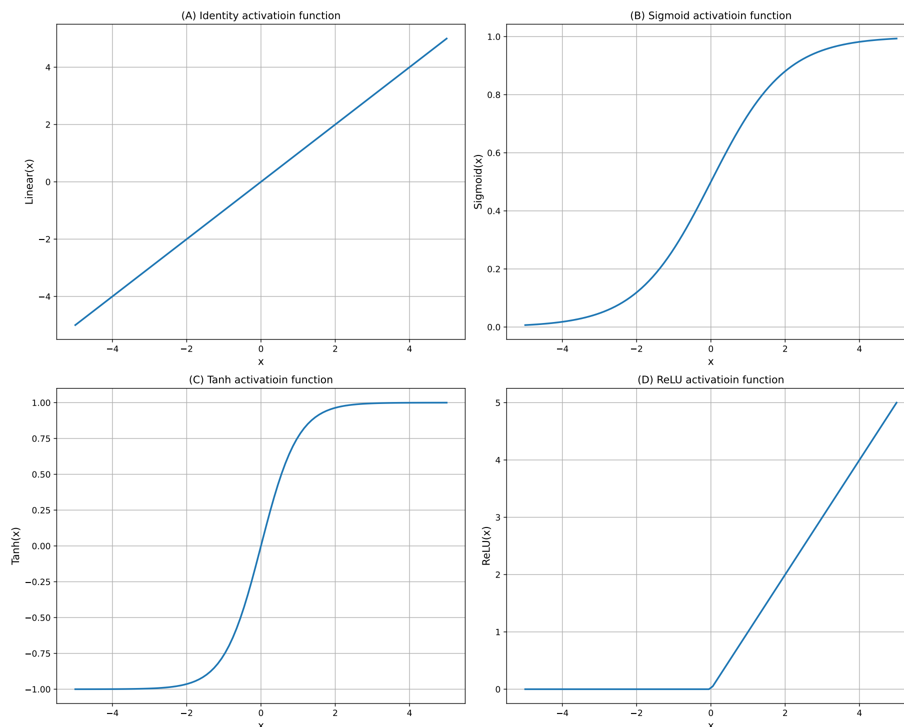


Figure 2.4: A spectrum of activation functions shaping neural network behavior, from (A) Identity to (D) ReLU, (B) Sigmoid, and (C) Tanh.

Figure 2.4 presents a graphical representation showcasing the characteristics of the linear, ReLU, sigmoid, and tanh activation functions. This chart offers insight into how these functions shape the behavior of neural networks, highlighting their distinct non-linearities and linearities. When training ReLU networks, it can be difficult since the gradient of the function becomes 0 when inputs are close to zero or negative, making it impossible for the network to do backpropagation—that is, to change the weights on the input to the ReLU node—and possibly failing to learn. To tackle this issue, Google proposed the Swish activation function in [RZL17], which shares a similar form with ReLU. However, it incor-

porates a smoothness factor due to the sigmoid function. Figure 2.5 illustrates the Swish activation function, which is defined in equation 2.9, where β is a parameter controlling the function's shape. The plot demonstrates the Swish function's characteristic S-shaped curve, showcasing its smooth and differentiable nature.

$$f_{act}(x) = \frac{x}{1 + e^{-\beta x}} \tag{2.9}$$

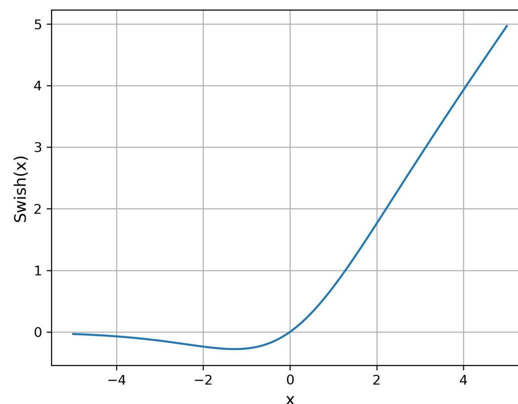


Figure 2.5: Swish activation functions

A network must begin learning how to produce the right result given specific inputs after it has an established design. The following learning paradigms can be distinguished in this process, depending on the type of training set being used: supervised, unsupervised, and reinforcement learning [Kri07]. Different topologies have been developed to address different categories of tasks by utilizing the fundamental ideas of a neural network. Two main types of networks are examined in great depth for the project's purposes: recurrent neural networks and feed-forward neural networks, as already introduced.

2.1.3 Recurrent Neural Networks

Recurrent neural networks represent a class of neural networks renowned for their remarkable ability to process sequential data effectively. Unlike feed-forward networks, RNNs possess a distinctive feature – the capacity to maintain an internal state or memory, enabling them to store and utilize information related to previous inputs. This critical attribute is particularly valuable in various applications that involve uncovering underlying structures in sequential or time-ordered data.

RNNs find versatile applications across domains like time series analysis, natural language processing, speech recognition, financial data modeling, audio processing, video analysis, and more. These data structures inherently possess temporal dependencies, where

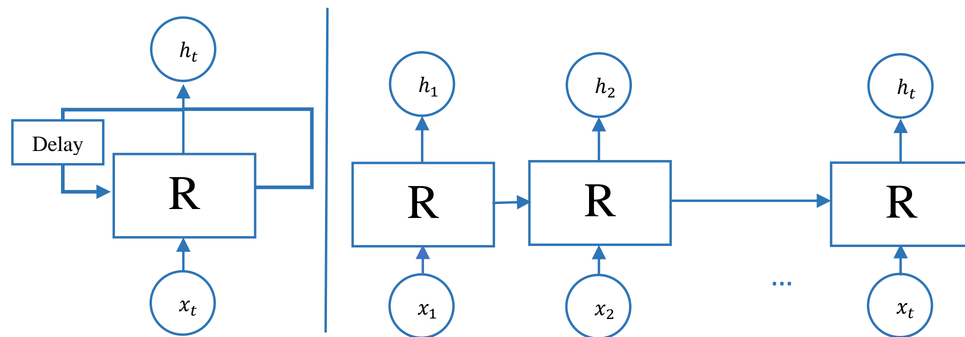


Figure 2.6: Recurrent neural network element (left). The behavior at any time point is given by the unfolding of its time-discrete feedback behavior (right).

each data point relies on its preceding ones. Herein lies the strength of recurrent neural networks, as they can leverage their internal memory to retain context and capture long-term dependencies across the sequence. The organization of this memory is crucial to understanding how RNNs manage information:

- **Memory Update**: It is in RNN updated at each time step as new input is processed. The update involves incorporating information from the current input and the previous state of the memory. The update process typically follows a formulation such as:

  $$h_t = \text{RNNCell}(x_t, h_{t-1}),$$

  where h_t is the updated memory state at time t, x_t is the input at time t, and h_{t-1} is the previous memory state.

- **Memory Read**: The memory read is utilized to influence the prediction or computation at each time step. The output or prediction at a given time step is often a function of the current input and the current state of the memory. Mathematically, this can be expressed as:

  $$y_t = \text{OutputFunction}(x_t, h_t),$$

  where y_t is the output at time t, x_t is the input at time t, and h_t is the memory state at time t.

This capacity to process sequential data and maintain context sets RNNs apart from feedforward networks, which lack the ability to retain and process non-local information. For instance, while a feedforward network can only learn bounded-range addition, an RNN excels at tasks requiring the processing of sequences of arbitrary length. The core structure

of an RNN is illustrated in Figure 2.6, where each cell in the network takes an input, x_0, and produces an output, h_0. This output, together with the next input, x_1, forms the basis for computation in the subsequent time step. The internal state, represented as h_{t-1}, plays a pivotal role in the computation at time step t, creating a recursive process that facilitates the flow of information across the sequence. By building upon this recursive computation scheme, RNNs can effectively process sequential data, uncover patterns, and make predictions based on historical context.

2.1.4 LSTM Neural Networks

Long short-term memory (LSTM) networks [HS97] represent an improved version of basic recurrent neural networks, addressing the problem of vanishing gradients during training. One of the key features of LSTM networks is their ability to actively control when to remember past data, achieved through a fixed memory structure [HS97]. This crucial enhancement makes LSTMs particularly suitable for tasks involving time series data with time delays of unknown durations, as they can effectively classify, process, and predict such sequences. The network is trained using the back-propagation algorithm.

A fundamental difference between LSTM networks and basic RNNs lies in the inclusion of so-called *gates* within LSTMs. These gates play a vital role in actively regulating the flow of information between memory and computational units. By doing so, LSTMs gain the ability to actively control the lifespan of stored information, which overcomes the limitations of basic RNNs in remembering properties across long sequences and retaining information for extended periods.

In essence, LSTM networks can be seen as a special type of recursive neural network, as they possess the unique capability to learn and capture long-term dependencies within sequential data. This advancement in memory and control mechanisms makes LSTMs a powerful tool for a wide range of applications, particularly in tasks that involve complex temporal dependencies and require the ability to retain and recall important information over extended periods. Through their architecture, LSTM networks have significantly improved the capacity of recurrent neural networks, enabling them to tackle more challenging and real-world sequential data analysis tasks with enhanced accuracy and efficiency.

LSTM Architecture

Figure 2.7 depicts the architecture of an LSTM cell, showcasing its unique design with a series of gates that govern the flow of information within the memory component of the network. These gates play a critical role in controlling how data in a sequence is received, stored, and transmitted within the LSTM cell. Although there are variations of LSTM

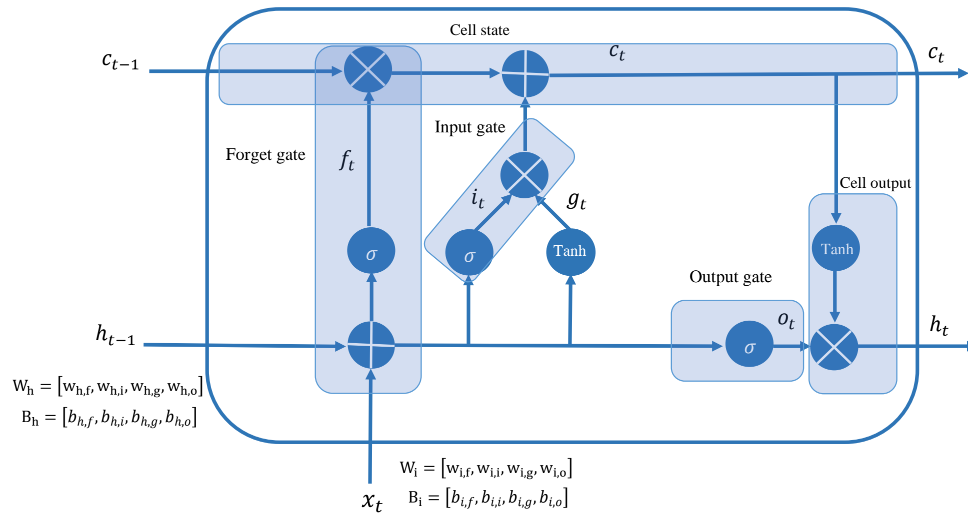


Figure 2.7: LSTM structure. [YSHZ19]

structures, a standard LSTM cell usually consists of three fundamental gates: *forget gate*, *input gate*, and *output gate*. These gates serve as filters, regulating the flow of information throughout the LSTM cell.

Various components of the LSTM, including input vectors, hidden state vectors, cell state vectors, weight vectors, bias vectors, and gate outputs, involve the use of vectors.

- **Input gate** (i_t): The input gate determines which new information from the current time step (x_t) and the previous hidden state (h_{t-1}) should be stored in the cell state (c_t). It computes the gate value using a weighted sum of the input and hidden state, passed through the sigmoid function. A gate value close to 1 indicates that information should be stored, while a value close to 0 means the information is discarded. The characteristic equation for the input gate is given by:

$$i_t = \sigma(w_{h,i}h_{t-1} + w_{i,i}x_t + b_{h,i} + b_{i,i}) \tag{2.10}$$

- **Cell state** (c_t): The cell state stores and retains relevant information over time. It is updated using the output of the input gate (i_t), combined with the output of a tanh function that normalizes the input and hidden state values into the range -1 to 1. The forget gate output (f_t) from the previous time step also influences the updating of the

cell state. The characteristic equations for the cell state are:

$$g_t = \tanh(w_{h,g}h_{t-1} + w_{i,g}x_t + b_{h,g} + b_{i,g}) \quad (2.11)$$

$$c_t = i_t \odot g_t + f_t \odot c_{t-1} \quad (2.12)$$

- **Forget gate (f_t)**: The forget gate determines whether to retain or forget information stored in the cell state. It computes the gate value by considering the input (x_t) and hidden state (h_{t-1}) and passing them through the sigmoid function. The gate output, which ranges from 0 to 1, decides how much of the previous cell state should be retained. The characteristic equation for the forget gate is given by:

$$f_t = \sigma(w_{h,f}h_{t-1} + w_{i,f}x_t + b_{h,f} + b_{i,f}) \quad (2.13)$$

- **Output gate (o_t)**: The output gate prepares the relevant information to be output from the LSTM cell. It computes the gate value by considering the input (x_t) and hidden state (h_{t-1}) and passing them through the sigmoid function. The updated cell state value (c_t) is then passed through the tanh function. The output of the sigmoid and tanh functions become combined linearly to determine the hidden state (h_t) that will be passed to the next time step. The characteristic equation for the output gate is:

$$o_t = \sigma(w_{h,o}h_{t-1} + w_{i,o}x_t + b_{h,o} + b_{i,o}) \quad (2.14)$$

- **Cell output (h_t)**: The cell output (h_t) is the final output of the LSTM cell, which represents the hidden state of the cell. It is computed as the element-wise multiplication of the output gate (o_t) and the hyperbolic tangent of the updated cell state (c_t). This combination ensures that the relevant information is carried forward to the next time step. The characteristic equation for the cell output is:

$$h_t = y_t = o_t \odot \tanh(c_t) \quad (2.15)$$

2.1.5 Frequently Encountered Issues

In the realm of training and working with neural networks, several challenges are often encountered, each of which has a profound impact on the model's performance. These challenges are inherent to neural networks and affect their performance in various tasks,

including optimization. They influence the model's ability to learn and generalize, impacting optimization by posing challenges in updating model parameters effectively. Addressing these challenges often involves selecting suitable architectures, activation functions, regularization methods, and optimization strategies to enhance overall performance.

One of these common issues is the *vanishing gradient problem* [Hoc98], where gradients become extremely small during back-propagation [RDGC13], hindering the model's training process. *Overfitting* [Yin19], on the other hand, arises when a model becomes excessively complex and starts fitting the training data too closely, leading to poor generalization. Conversely, *underfitting* [Koe18] occurs when a model is too simplistic to capture the underlying patterns in the data, resulting in suboptimal performance. Addressing these challenges is paramount in the pursuit of effective and robust neural network solutions.

- **Vanishing gradient problem** : This problem can manifest during network training when utilizing common gradient-based learning techniques. This issue results in suboptimal network accuracy and, at its worst, an inability to complete the training process altogether. The training process relies on backpropagation, which calculates gradients by applying the chain rule to the partial derivatives of activation functions. In each training iteration, weights in the network are adjusted based on the partial derivative of the error function with respect to the current weight. However, when dealing with functions like the sigmoid, which have derivatives within the range of 0 to 0.25, the repetitive multiplication of such small values leads to a rapid decline in gradient magnitudes, particularly in the early layers. This hinders training progress and effectively stalls weight updates, thereby impeding the network's ability to enhance output accuracy.

- **Overfitting**: A network is deemed overfitted when it closely mimics the output of the training data, a situation often stemming from an excessive number of adjustable parameters or the adoption of an unnecessarily complex approach. Overfitting leads to the network factoring in extraneous details from the training data, incorporating "noise," rather than restricting its considerations to the minimal information required for accurate results across all input data sets. While unnecessary complexity diminishes the network's overall performance, the incorporation of noise-based information from test data can substantially distort outcomes for general input data. Determining whether a network exhibits signs of overfitting is a pertinent aspect of its validation. Utilizing a range of values instead of exact inputs permits an examination of the extent to which noise can increase before it exerts an impact on the output.

- **Underfitting**: A network is considered to be underfitted when it inadequately cap-

tures the nuances of the training data, resulting in a lack of fidelity between the network's output and the actual data. This typically arises from a model that is too simplistic or has too few adjustable parameters, causing it to oversimplify the underlying patterns. Underfitting leads to the network failing to take into account relevant information from the training data, thereby producing inaccurate results and disregarding essential details necessary for accurate predictions. While a simpler model may be computationally efficient, it struggles to generalize to new data, and this deficiency in complexity impairs its performance. In summary, underfitting is the opposite of overfitting, where the model's simplicity compromises its ability to accurately represent and predict the data.

2.1.6 Frameworks and Libraries

A deep learning and neural network library is a software framework or toolkit designed to facilitate the development, training, and deployment of neural networks. These libraries provide a wide range of tools, functions, and pre-implemented algorithms that streamline the process of building and working with complex neural networks. In this section, we will explore three of the most renowned open-source libraries that facilitate the development of neural networks, making the process more convenient and efficient: PyTorch [KMKM21], TensorFlow [PNW20], and Keras [KK17].

- **Tensorflow**: It is an open-source library for high-speed numerical computing that was introduced in 2017 and has since grown in popularity for deep learning and machine learning [Dev22]. Due to its adaptable architecture, which permits deployment on many platforms like CPUs, GPUs, and TPUs, its use in industry has increased. More developers are also using the platform thanks to its thorough documentation and effective debugging system.

- **Keras**: With its Python-based API, Keras focuses on providing simple, dependable methods for creating and training neural networks. It is highly customizable. Usability is increased by its user-friendly wrappers, while efficiency is ensured by assigning demanding computations to other specialized frameworks like TensorFlow, Theano, or CNTK [KK17].

- **PyTorch**: A machine learning framework built upon the Torch library, PyTorch is open-source and facilitates the smooth transition from research prototypes to deployment [KMKM21]. PyTorch incorporates a significant innovation from Chainer [TOHC15] known as reverse-mode automated differentiation. This mechanism operates akin to a tape recorder, computing gradients by replaying previously executed

operations. As a result, PyTorch offers ease of debugging and is well-suited for various applications, particularly dynamic neural networks. Its adaptability to varying iterations makes it popular for prototyping purposes.

2.2 Verification

In this section, we delve into various facets of verification, exploring fundamental concepts and key techniques that underpin this essential discipline. Beginning with an examination of foundational problems such as the Boolean Satisfiability Problem, we progress towards more sophisticated approaches including Satisfiability Modulo Theories (SMT). By comprehensively exploring these topics, we aim to establish a fundamental understanding of verification methodologies, laying the groundwork for the subsequent analysis and discussion in this thesis.

Algorithm 1 The DPLL algorithm

Input: A set of clauses ϕ
Output: the integer fraction that represents t
Function DPLL(ϕ)**:**
  $T^* \longleftarrow F$;
  while *there is a unit clause* ξ *in* ϕ **do**
    $\phi \longleftarrow unit - propagate(\xi, \phi)$;
  end
  while *a pure literal* ξ *that appears in* ϕ **do**
    $\phi \longleftarrow pure - literal - assign(\xi, \phi)$;
  end
  if ϕ *is empty* **then**
    return *true;*
  end
  if ϕ *encloses an empty clause* **then**
    return *false;*
  end
  $\xi \longleftarrow select - literal(\phi)$;
  return *DPLL($\phi \land \xi$) or DPLL($\phi \land not(\xi)$);*

2.2.1 Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) [AMM22] in computer science is the issue of figuring out whether there is a particular Boolean interpretation that satisfies a given Boolean formula. SAT attempts to determine whether a feasible set of assignments of Boolean variables exists such that the formula evaluates to true, in which case the formula is satisfiable, or whether there is no feasible set of assignments that satisfies the formula such that it always evaluates to false and, as a result, the formula is unsatisfiable. One of

the earliest problems that was demonstrated to be NP-complete was SAT, which is also essential to the development of artificial intelligence, algorithms, and hardware. The *Davis-Putnam-Logemann-Loveland method* (DPLL) [DLL62] is one remarkable algorithm that serves as the foundation for numerous SAT solvers. The DPLL algorithm can be summarized in the following pseudocode 1, where ϕ is the *conjunctive normal form* (CNF) of a formula [DLL62]:

The Algorithm 1 takes a set of clauses, denoted as ϕ, as input and aims to determine the truth value of the given formula, represented as a binary value (0 or 1). In other words, the algorithm seeks to find a truth assignment to the variables in ϕ that makes the entire formula true. A clause is a disjunction of literals, where a literal is either a propositional variable or its negation. These clauses collectively form the propositional formula, and the DPLL algorithm works to find a satisfying assignment of truth values to the variables that make the entire formula true. In Algorithm 1, three key functions play crucial roles in the process of solving the Boolean Satisfiability Problem. Unit-propagate(ξ, ϕ) deals with unit clauses, which are clauses containing only one literal within the formula ϕ. In this step, the algorithm propagates the truth value of the single literal ξ throughout the formula ϕ. By assigning it a truth value consistent with the clause's polarity, the function simplifies ϕ accordingly. Pure-literal-assign(ξ, ϕ) are literals that consistently appear with the same polarity throughout the formula ϕ. This function identifies and assigns truth values to such pure literals. By doing so, it simplifies ϕ further, reducing redundancy and facilitating the search for a satisfying assignment. Select-literal(ϕ) is responsible for selecting the next literal to explore in the search tree. It determines which literal ξ to choose next based on a heuristic strategy. This selection process is crucial for guiding the search towards a satisfying assignment efficiently. These functions collectively contribute to the effectiveness of the DPLL algorithm in solving SAT instances by systematically simplifying the formula and guiding the search for a satisfying assignment. The algorithm begins by initializing a temporary working set, T^*, with the same clauses as the input formula. It then iteratively performs several key steps. First, it checks for unit clauses and employs unit propagation to simplify ϕ. Unit propagation determines the truth value of the single literal in a unit clause and propagates it throughout ϕ, simplifying the formula accordingly. Next, it identifies and assigns pure literals, further simplifying ϕ by reducing redundancy. If the formula becomes empty during these steps, it is deemed satisfiable, and the algorithm returns true. This occurs as the algorithm removes satisfied clauses and eliminates redundant literals. Conversely, if the algorithm encounters an empty clause (indicating a contradiction), it returns false. Otherwise, it selects a literal ξ and recursively explores two branches of the search tree. By employing a divide-and-conquer approach, the DPLL algorithm efficiently explores possible assignments and ultimately determines

the satisfiability of the input formula. It plays a crucial role in propositional logic and Boolean satisfiability, serving as a valuable tool in various areas of computer science and artificial intelligence.

2.2.2 Satisfiability Modulo Theories

A decision problem known as the *Satisfiability Modulo Theories* (SMT) [BT18] problem asks for an algorithmic determination of the satisfiability of a Boolean combination of quantifier-free claims over a theory or combination of theories in a form that is mostly CNF. Equivalence reasoning over uninterpreted function symbols, linear or polynomial arithmetic, fixed-size bit-vectors, arrays, and other practical first-order theories, as well as combinations thereof, are common examples of such theories [DMB08]. In most practical solvers for SMT problems, an underlying SAT solver determines the satisfiability of a propositional logic formula obtained by replacing the theory atoms by propositions, while a coupled theory solver determines the satisfiability of conjunctions of theory atoms from the particular first-order theory T embedded into the particular SMT fragment [SKB13]. Propositional logic's connectives are used in the language of SMT problems, but complicated expressions containing constants, functions, and predicate symbols are used in place of propositional variables. These include the signature $\sum$, functions, and predicates which define the vocabulary and fundamental building blocks of the logical language.

- **Signature** Σ: The signature defines the set of symbols used in the logical language, including function symbols, predicate symbols, and constants. It essentially characterizes the vocabulary of the logical theories involved.
- **Functions and predicates**: Functions and predicates are fundamental building blocks of the logical language. Functions represent operations that take input values and produce output values, while predicates express relations between values. These symbols play a central role in expressing constraints and relationships within the theories.
- **Propositional logic connectives**: The language of SMT problems utilizes the connectives of propositional logic, such as `AND`, `OR`, `NOT`, implying the combination of simpler formulas to form more complex ones.

Definition 1. *(Formula) For a given signature $\sum$ and theory T over this signature, a $\sum$formula φ is an expression formed using symbols and operations from the signature $\sum$, adhering to the syntax rules defined by the formal language of T. A formula φ is satisfiable if there exists an assignment ξ to its free variables that render it true. It is called*

unsatisfiable if no such satisfying assignment exists. Let D be the domain over which the variables in φ are defined. The assignment ξ provides values to the free variables in φ from D.

For example, considering $\sum = \{0, 1, \ldots, +, \geq, \leq, \ldots\}$ and $D = \mathbb{Z}$, the formula $3x+y \geq 8$ under the model $\{\mathrm{x} \rightarrow 2, \mathrm{y} \rightarrow 3\}$ is satisfiable, as there exists an assignment ξ to its free variables (x and y) that renders it true. On the other hand, the formula $2x + 3y \geq 0 \wedge (-1 \geq y) \wedge (-1 \geq x)$ is unsatisfiable, as no such satisfying assignment exists.

According to Section 2.1, since neural networks have activation functions, their SMT-based verification requires SMT-solvers to incorporate theory solvers that can handle non-linear and transcendental functions. iSAT [FHT+07, SKB13] and dReal [GAC12] are two solvers that can handle these types of functions.

2.2.3 iSAT

The iSAT [FHT+07, SKB13] algorithm combines SAT-solving techniques with *Interval-based Arithmetic Constraint Solving* (ICP), enabling it to reason about non-linear arithmetic constraints. iSAT (up to its second release iSAT2) does not include a theory solver to which the SAT-solver delegated the decisions over theory predicates, in contrast to the techniques for SMT solution that were previously discussed. Instead, to achieve a tight integration of the two, iSAT takes advantage of similarities between interval-based arithmetic constraint propagation and DPLL-style SAT-solving approaches. As a result, the constraint propagation within theory atoms can be directly controlled by the DPLL-style solver. ICP is an incomplete decision procedure that effectively reduces the domain of a set of variables, covering transcendental functions, etc. It is one of the subtopics in the field of constraint programming.

The ICP step for the primitive constraint $y = e^x$ is depicted in Figure 2.8. Assume that the current intervals are $x \in [-5, 2]$ and $y \in [1.0, 2.7]$. The current interval of variable x is taken as input and computes using interval arithmetic that variable y is in $[0.0, 7.38]$. This indicates that ICP was successful in obtaining a more robust lower bound for the y-interval. As the maximum value of y is currently 7.38 we derive $e = 2.7$ as a new stronger upper bound of the interval of x.

There are three iSAT algorithm implementations. HySAT [Her11], the original implementation, is no longer supported. The second implementation, now known as iSAT, is currently being developed on. To more clearly distinguish between the iSAT algorithm and the second implementation of iSAT, in the following, we refer to this version as iSAT2. The most recent iSAT version is named iSAT3 [isa10] since it is the third implementation of the iSAT algorithm. In contrast to HySAT and iSAT2, which both act on simple bounds

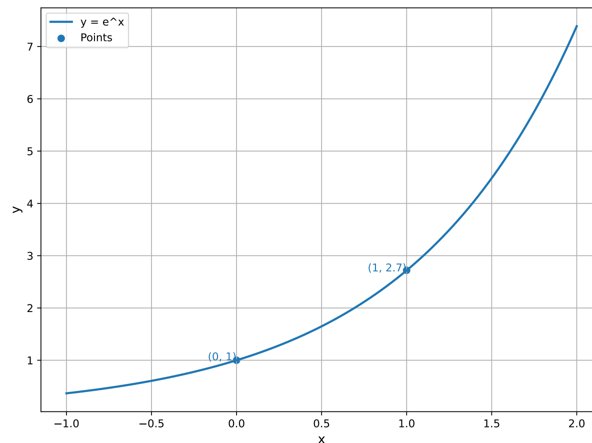


Figure 2.8: Interval constraint propagation (ICP) performed on constraint e^x

directly, iSAT3 reverts to a little more lazy approach and explicitly maps each simple constraint to a literal once again.

The iSAT command-line tool offers two distinct modes of operation: Firstly, it serves as a satisfiability checker for individual formulas. Secondly, it functions as a tool for tracing the behavior of a hybrid system through bounded model checking (BMC).

- **Single formula mode**: In the single formula mode, the input file comprises two main sections: The initial section, marked by the keyword DECL, includes declarations for all variables present in the formula under consideration. The subsequent section, initiated by the keyword EXPR, encompasses the formula itself, typically consisting of a single arithmetic constraint.

- **Bounded model checking mode**

  - **DECL**: Variables used in the model should be declared, representing the state of the system or any relevant quantities to be tracked. Symbolic constants, which are values remaining constant throughout the model's execution, can also be defined.

  - **INIT**: This part specifies the initial conditions or states of the system. It sets the starting values for all variables declared in the DECLARATION section, representing the system's state at the beginning of the verification.

  - **TRANS**: In this part, the evolution of the system over time is defined, describing the relationship between the current state of the system and its next state. Variables may appear in primed (x') or unprimed (x) form, where primed variables represent their values in the subsequent time step, after the transition.

```
DECL
    boole jump;                -- true iff ball bounces
    int [0, 15] n;             -- bounce counter
    float [0.0, 10000] t;      -- global time
    float [-100.0, 100.0] xv;  -- vertical position of the ball
    float [-100.0, 100.0] yv;  -- vertical velocity of the ball
    float [-100.0, 100.0] xh;  -- horizontal position of the ball
    float [-100.0, 100.0] yh;  -- horizontal velocity of the ball

    define dt = 0.2;           -- discrete time adances in steps of size dt
    define g  = 9.8;           -- gravitational acceleration on earth
    define lv = 0.3;           -- loss of speed in vertical direction
    define lh = 0.1;           -- loss of speed in horizontal direction
    define xv_limit = 15.0;    -- upper bound of the initial hight
    define vh_limit = 5.0;     -- upper bound of the initial horizontal speed
    define xh_hole  = 40.0;    -- position of the hole

INIT
    -- Conditions at the moment when ball is dropped.
    xv <= xv_limit;
    yv = 0.0;

    xh = 0.0;
    yh <= vh_limit;

    n = 0;
    t = 0.0;

TRANS
    -- A jump occurs if the heigth of the ball is zero and the velocity
    -- is directed towards the ground.
    jump <-> (xv <= 0.0 and yv < 0.0);


    -- Continuous part of the dynamics.
    !jump ->
        (xv' = xv + dt * yv  - 0.5 * g * dt^2 and
         yv' = yv - dt * g and
         xh' = xh + dt * yh and
         yh' = yh and
         n' = n and
         t' = t + dt);

    -- Discrete part of the dynamics.
    jump ->
        (xv' = 0.0 and
         yv' = -(1.0 - lv) * yv and
         xh' = xh and
         yh' = (1.0 - lh) * yh and
         n' = n + 1 and
         t' = t);

TARGET
    -- The ball shall hit the hole on ground level.
    xh = xh_hole and xv = 0.0;
```

Figure 2.9: iSAT code of a bouncing ball [FHT+07, SKB13]

Table 2.1: Arithmetic operators available in iSAT. Operators marked with an * are only available in iSAT3 [isa10]

Operator	Args	Meaning	
+	infix	1 or 2	unary ’plus’ and addition
-	infix	1 or 2	unary ’minus’ and subtraction
*	infix	2	multiplication
abs	prefix	2	absolute value
min	prefix	2	minimum
max	prefix	2	maximum
ite*	prefix	3	If-then-else
exp	prefix	1	exponential function regarding base e
exp2*	prefix	1	exponential function regarding base 2
exp10*	prefix	1	exponential function regarding base 10
log	prefix	1	logarithmic function regarding base e
log2*	prefix	1	logarithmic function regarding base 2
log10*	prefix	1	logarithmic function regarding base 10
sin	prefix	1	sine (unit: radian)
cos	prefix	1	cosine (unit: radian)
pow	prefix	2	(2nd argument) has to be an integer, $n \geq 0$
^	infix	2	(2nd argument) has to be an integer, $n \geq 0$
nrt	infix	2	nth root, (2nd argument) has to be an integer, $n \geq 1$

– **TARGET**: This section specifies the property or condition desired to be verified about the system, defining the state(s) aimed to be reached or avoided during the verification.

Figure 2.9 illustrates an example of an iSAT code translating the motion of a bouncing ball. The ball is initially dropped from a certain height and given an initial velocity in the horizontal direction. As the ball descends, it experiences acceleration due to gravity in the vertical direction. When the ball hits the ground, this event is treated as a discrete transition (’jump’), during which the ball loses a fraction of its energy. The objective of the solver is to determine the initial height and velocity required for the ball to hit a hole located at a specified distance from the starting point. To avoid trivial solutions where the ball does not bounce at all, upper bounds are set for the initial height and velocity.

iSAT Natively Supports Non-Linear Arithmetic

iSAT is specifically designed for solving Boolean combinations of linear and non-linear (not limited to polynomial, but in contrast to most of its competitors also including transcendental functions) constraint formulas. It extends the classical *conflict-driven clause learning* (CDCL) [Sol21] framework by interval constraint propagation. ICP enables iSAT to reason about the linear and non-linear constraints with the help of interval arith-

Table 2.2: Recently released activation functions

Activation Function	Formula
swish	$\frac{x}{1+e^{-x}}$
soft plus	$\ln(1+e^x)$
ELU	$\begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$
SELU	$\begin{cases} \lambda\alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$

metic.

Therefore, we propose iSAT as a suitable tool to verify neural networks containing activation functions with a more complex arithmetic base than just piecewise linear, such as sigmoid or tanh activation functions, which are part of complex neural networks. The mathematical operators in iSAT are shown in Table 2.1, allowing us to encode nonlinear activation functions that comprise transcendental arithmetic. For instance, exponential function terms e^x form the basis of activation functions like sigmoid, tanh, and recent variants like swish [RZL17], ELU, SELU [RAS20] as shown in Table 2.2 and Figure 2.10.

Regarding scalability, iSAT has routinely solved arithmetic problems involving transcendental functions as well as polynomials of degrees in the hundreds, and problem sizes of hundreds of thousands of variables, leading us to expect it to scale well to neural networks with comparable numbers of neurons.

iSAT enhances the performance of arithmetic constraint reasoning by use of verified floating-point arithmetic

When working with SMT formulas that involve real-valued variables, an important consideration is how real numbers are represented. To avoid round-off errors and ensure soundness, most SMT solvers use precise arithmetic, which is different from approximate floating-point arithmetic. However, precise computation can significantly slow down the computational speed due to the need for extremely high precision, which tends to increase over the course of computation. Moreover, when dealing with transcendental functions, exact arithmetic is not available or feasible on digital computers. To reason about real numbers, iSAT has adopted a different approach since its early versions. It employs verified floating-point arithmetic, following the principles of ICP based on safe, i.e. downward for lower interval bounds and upward for upper bounds, rounding. This technique provides

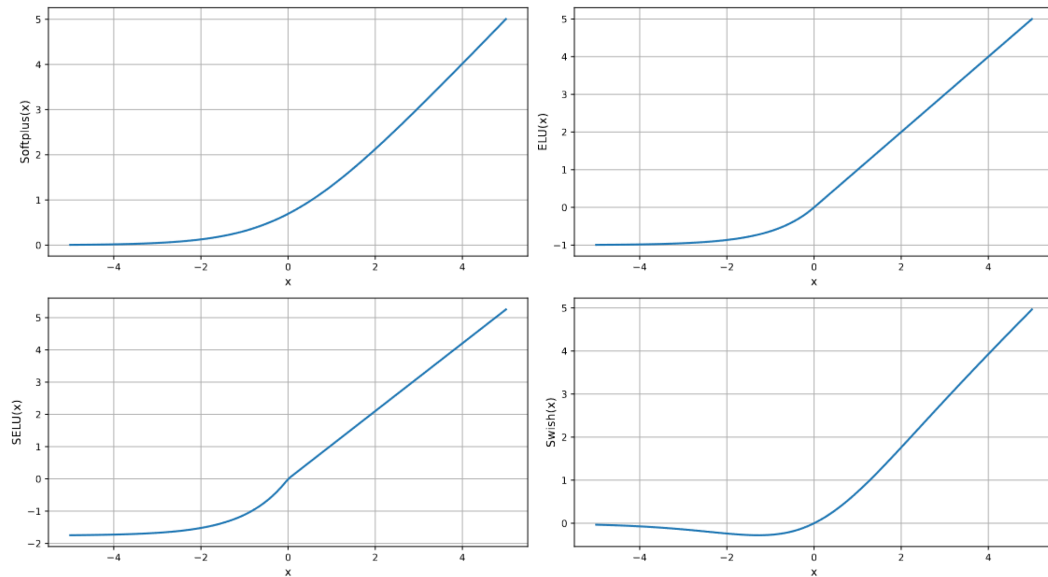


Figure 2.10: Activation function showcase: Softplus, ELU, SELU, and Swish. Dive into the diverse curves shaping neural network behavior.

a balance between accuracy and efficiency, ensuring reliable results when handling real numbers in SMT formulas.

2.2.4 dReal

In Chapter 5, the study will involve a comprehensive comparison between our proposed approach in iSAT and the dReal solver which is an SMT-Solver-based tool capable of handling the non-linearity of the activation functions. Like iSAT, it concentrates on solving existential first-order logic formulas over real numbers [GAC12] and follows a more traditional lazy approach to SMT and blends DPLL-style SAT solving methods with interval-based arithmetic constraint propagation. In contrast to iSAT, the dReal chose to promote the SMT-LIB initiative rather than offering its own input language. In order to develop the field of SMT systems, the SMT-LIB initiative offers a standardized format and a library of benchmarks. This allows SMT-solvers that support the SMT-LIB format to be benchmarked against other SMT-LIB compatible solvers.

Figure 2.11 and Table 2.3 illustrate the available commands and a snippet of dReal code within the SMT-LIB 2.0 format. This format differs from iSAT in that it does not offer differentiated sections in input files referring to the elements of symbolic transition systems, which also implies that dReal does not natively support bounded model-checking. Instead, it allows variables and expressions to be defined anywhere in the file, as long as they are defined before they are used. A valid SMT-LIB 2.0 file contains a series of commands that can either interact with the solver (such as modifying parameters) or describe

Table 2.3: commands available in the SMT-LIB 2.0 format [BST+10]

```
(command)::=(set-logic (symbol))
          | (set-option (option))
          | (set-info (attribute))
          | (declare-sort (symbol) (numeral))
          | (define-sort (symbol)((symbol)*) (sort))
          | (declare-fun (symbol)((sort)*) (sort))
          | (define-fun (symbol)((sorted_var)*)(sort)(term))
          | (push (numeral))
          | (pop (numeral))
          | (assert (term))
          | (check-sat)
          | (get-assertions)
          | (get-proof)
          | (get-unsat-core)
          | (get-value ((term)+ ))
          | (get-assignment)
          | (get-option (keyword))
          | (get-info (info_flag))
          | (exit )
```

```
1  (set-logic QF_NRA)
2  (declare-fun x1 () Real)
3  (declare-fun x2 () Real)
4  (assert (<= 3.0 x1))
5  (assert (<= x1 3.14))
6  (assert (<= -7.0 x2))
7  (assert (<= x2 5.0))
8  (assert (<= (- (* 2.0 3.14159265) (* 2.0 (* x1 (arcsin (* (cos 0.797) (sin (/ 3.14159265 x1)))))))
9              (+ (- (- 0.591) (* 0.0331 x2)) (+ 0.506 1.0))))
10 (check-sat)
11 (exit)
```

Figure 2.11: A snippet of dReal represented in the SMT-LIB 2.0 format [GAC12].

the problem. The command syntax in SMT-LIB 2.0 is akin to LISP, with each command enclosed in parentheses and written in prefix notation. When it comes to verifying neural networks, the initial essential command is "set-logic" which this command in SMT-LIB 2.0 is used to specify the logic for addressing the problem. Commonly used logics include:

- `QF_LIA`: Quantifier-Free Linear Integer Arithmetic
- `QF_LRA`: Quantifier-Free Linear Real Arithmetic
- `QF_UFLIA`: Quantifier-Free Uninterpreted Functions with Linear Integer Arithmetic
- `QF_UFLRA`: Quantifier-Free Uninterpreted Functions with Linear Real Arithmetic
- `BV`: Bit-Vectors
- `AUFLIA`: Arrays, Uninterpreted Functions, and Linear Integer Arithmetic

Depending on the problem's nature and verification task requirements, the logic that best represents the system's constraints and properties would be chosen.

Chapter 3

Rationale for Verifying Neural Networks

In this chapter, we will delve into the task of verifying neural networks and uncover the reasons behind their necessity. We will examine why ensuring the accuracy and reliability of these networks is essential in today's computing landscape. Additionally, in the following sections, we will explore the various aspects and objectives that neural network verification aims to address. In 1949, Alan Turing wrote a paper titled "Checking a Large Routine" [Tur89] which looked ahead in addressing a key question: How do we confirm that our programs do what they are meant to do? In the paper, he went on to provide correctness proof for a program implementing the factorial function. In Turing's proof for the correctness of his factorial program, he addressed a concern about potential errors in programming computers for mathematical operations. His proof demonstrated that his factorial implementation aligns with the mathematical definition, highlighting the importance of functional correctness. This concept ensures that a program faithfully represents a specific mathematical function, a critical aspect given the potentially severe consequences of errors in applications like cryptographic primitives or aircraft controllers.

3.1 Why is Verification of Neural Network Important?

The progress in neural networks and deep learning has revolutionized our perception of software, its capabilities, and the methods employed in its construction. Contemporary

software is evolving into a hybrid landscape, incorporating both conventional, manually crafted code and dynamically trained neural networks that often engage in continuous learning. However, the inherent fragility of deep neural networks and their potential for yielding unforeseen outcomes underscore the importance of cautious integration, especially in critical applications like autonomous vehicles.

Formal methods, essential mathematical techniques utilized in the specification, development, and validation of software and hardware systems, have been widely adopted in various fields (for an early survey, see [CW96]). These methods rely on mathematical representations, often termed formal specification languages, to precisely articulate system properties. Formal methods offer valuable benefits across multiple development phases for a diverse array of systems. They facilitate thorough analysis and verification, tackling crucial elements like safety, security, reliability, and correctness with precision. As time progresses, formal methods have garnered increasing recognition and are now being integrated into industry standards. This signifies a growing acknowledgment and utilization of formal methods as an indispensable asset in augmenting the development and assurance of complex systems. Through the utilization of formal methods, developers and engineers gain the capacity to engage in mathematical reasoning concerning system behavior, detect potential flaws or errors at the outset of the design process, and establish formal assurances regarding system correctness. Consequently, this results in the creation of software and hardware systems that are more resilient and trustworthy. In this context, this thesis aims to leverage the advantages of formal methods over testing specifically on neural networks.

3.2 Different Types of Formal Methods

In this section, we offer a brief overview of the primary types of formal methods documented in the literature [Kri24]. These formal methods encompass a diverse array of mathematical and logical techniques employed in various stages of system development and verification. By understanding and leveraging these methods, developers, and engineers can enhance the rigor and reliability of their software and hardware systems.

- **Abstract interpretation**: Abstract interpretation is a method that strives to validate a system at a higher level of abstraction, disregarding insignificant intricacies. This technique formalizes the concept that an over-approximation of the system's behavior can offer valuable insights into its properties. By abstracting away non-essential details, abstract interpretation enables the analysis of complex systems more efficiently, aiding in the identification of key properties and potential issues. Symbolic execution over an abstract domain lies at the core of this approach, with summaries at join points of the control flow facilitated by the complete-lattice struc-

ture of the abstract domain. This approach is particularly useful in scenarios where precise analysis of every system aspect is impractical or unnecessary, allowing for a more manageable and insightful examination of system behavior and characteristics [Cou01].

- **Model checking**: Model checking involves the verification of specifications on textual models that represent different types of systems. This technique employs exhaustive exploration of the model to verify properties and detect potential errors or violations. By systematically examining all possible states of the model, model checking ensures that the system meets specified requirements and adheres to desired properties. This approach provides a rigorous and automated method for verifying system correctness and identifying flaws early in the development process. Additionally, model checking is particularly effective for systems with complex behaviors or intricate interactions between components, making it an invaluable tool for ensuring the reliability and safety of software and hardware systems [Cla97].

- **Semantic static analysis**: Semantic static analysis refers to a thorough and automated examination of a program's source code without executing it. By scrutinizing the structure and semantics of the code, static program analyzers can pinpoint potential errors or vulnerabilities. This process enables developers to detect issues early in the development cycle, reducing the likelihood of encountering critical issues during runtime. Additionally, semantic static analysis aids in improving code quality and reliability by enforcing coding standards and best practices. By leveraging semantic static analysis tools, developers can enhance the robustness and security of their software applications while streamlining the development process [GS15].

- **Proof Assistants**: Proof assistants are interactive tools utilized for constructing and verifying formal proofs of theorems. Employed in both mathematics and computer science domains, these tools play a crucial role in ensuring the correctness of formal specifications and reasoning processes. By providing a platform for users to interactively develop and validate proofs, proof assistants enhance the reliability and rigor of mathematical and computational reasoning. These tools enable users to formally verify the correctness of complex algorithms, system designs, and mathematical conjectures, thereby bolstering confidence in the validity of the outcomes. Additionally, proof assistants facilitate collaboration and peer review by allowing users to share and scrutinize formal proofs, fostering a culture of transparency and rigor in academic and research communities [Geu09].

- **Deductive verification**: Deductive verification encompasses the process of validating a program against a formal specification. This method involves providing both

the program and a precisely defined specification, which serves as a set of desired properties or behaviors that the program should exhibit. Through techniques such as weakest preconditions and symbolic execution, the specification is systematically propagated through the program's source code. This propagation process aims to ensure that the program aligns with the specified requirements and adheres to the desired properties. Deductive verification provides a rigorous approach to software validation, enabling developers to formally establish the correctness and reliability of their programs through logical reasoning and systematic analysis. By leveraging deductive verification techniques, developers can mitigate the risk of errors and ensure the robustness of their software systems [PRZ01].

It is important to note that the categories described are not entirely disjoint. Proof assistants implement and support deductive verification, offering some level of automation to the process. Symbolic model-checkers also pursue a specialized form of deductive verification, as does abstract interpretation.

- **Design by refinement**: Design by refinement is a methodological approach that entails iteratively refining a sequence of textual presentations of computational processes and their specifications by automata or temporal logics, beginning with an abstract model and gradually evolving toward a concrete implementation. This iterative refinement process involves successive stages of development, where each refinement step adds more detail and specificity to the design. The goal of design by refinement is to ensure that the final implementation meets the desired properties specified at the outset. By systematically refining the design in this manner, developers can incrementally transform abstract concepts into concrete solutions while maintaining alignment with the original specifications. This approach fosters a structured and systematic development process, enabling the creation of robust and reliable systems through a series of well-defined refinement steps [BPSV03].

- **Model-based testing**: Model-based testing (MBT) is a dynamic methodology focused on deriving test scenarios directly from a system's formal specification. These scenarios are then executed to validate the system's behavior against its formal requirements, ensuring alignment with its intended functionality as outlined in the specification. The advantages of MBT are manifold, including increased test coverage, heightened reliability of test cases, and improved traceability between requirements and tests [Kri10, Kri18].

  However, it is crucial to recognize that, in contrast to the aforementioned benefits, MBT is an inherently incomplete procedure. While it outperforms non-model-based testing approaches, it falls short of guaranteeing absolute alignment between the sys-

tem's behavior and the specification due to incomplete probing inherent in testing methodologies. While it does provide valuable evidence of alignment, it does not offer rigorous and exhaustive proof, unlike proof-based processes that ensure comprehensive coverage. It is essential to acknowledge the partial coverage nature of MBT and understand that proof-based methods excel in providing exhaustive coverage.

3.3 Verification Conditions Applicable to Neural Networks

The field of neural network verification encompasses a wide range of conditions and techniques aimed at ensuring correctness. This correctness manifests in two primary forms: adherence to specifications and optimization robustness. Additionally, verification efforts extend to ensuring the stability and safety of these intricate systems. Let us delve into the various aspects and verification conditions applied to neural networks, breaking down the key concepts for individual systems, pairs of neural networks, and their joint dynamic behavior with the physical context.

3.3.1 Verification Conditions for Individual Systems and Neural Networks

In the realm of neural network verification, ensuring the correctness and reliability of individual systems requires meticulous attention to various verification conditions. These conditions encompass a spectrum of aspects, each crucial for maintaining the integrity and consistency of neural network behavior.

- **Invariance properties and Circuit/State Invariants**: Invariance properties specify that observable state variables remain within a defined set. Verifying satisfaction of invariants is crucial for maintaining the integrity and consistency of neural network behavior. This involves monitoring the evolution of state variables over time and verifying that they adhere to specified constraints [JBK20a]. Circuit invariants focus on the validation of input-output relations for feedforward networks, ensuring that the network produces the expected outputs for given inputs. State invariants address the dynamic behavior of recurrent networks, ensuring that the network's state trajectories remain within designated sets of states.

- **Stabilization**: Neural networks often operate in dynamic environments where in-

puts and conditions may change over time and are subject to disturbances. Verification of stabilization entails confirming that observables converge towards a specific equilibrium state, potentially at a certain rate, ensuring the network's or a network-controlled system's responsiveness and reliability under varying and partially unforeseeable conditions [Kor22].

- **Temporal Logic Properties**: Temporal logic properties are useful for capturing and constraining the time-dependent behavior of recurrent neural networks or of neural networks coupled to dynamic systems. Verifying temporal logic properties involves ensuring that traces or time series generated by the network conform to specified safety and liveness properties over time, guaranteeing the system's temporal correctness and robustness [WYW+19].

- **Hyperproperties**: Hyperproperties extend traditional temporal properties to encompass ensembles of possible traces rather than individual traces. Verifying hyperproperties involves analyzing the collective behavior of neural networks and assessing their adherence to higher-level properties that govern system-wide behavior and interactions [AMTZ21].

3.3.2 Verification Conditions for Pairs of Neural Networks

Comparing the behavior of two neural networks enables us to ascertain their similarity or dissimilarity, which is crucial for various applications such as ensemble learning, model compression, and system integration. Verification conditions for pairs of neural networks involve establishing equivalence or almost equivalence, where the goal is to determine whether the behavior of one network matches that of another within a specified margin of error [EKKT22]. Equivalence or almost equivalence verification serves as a means to validate the implementation correctness of neural networks, aiding in identifying and rectifying discrepancies, and ensuring the reliability and consistency of network interactions within complex systems. "Implementation correctness" refers to verifying that the implementation of a neural network accurately reflects its intended behavior, ensuring that it operates as expected and produces correct outputs for given inputs.

3.3.3 Verification Conditions for Joint Dynamic Behavior of an NN and its Physical Context

Understanding the dynamic interaction between neural networks and their physical context is paramount to ensuring the stability and reliability of integrated systems [LLL+20]. In scenarios where neural networks serve as controllers within feedback loops or interact with dynamic physical environments, verifying their joint dynamic behavior becomes imperative. This verification process entails scrutinizing how neural networks influence and respond to changes in the physical context, thereby ensuring the overall system operates safely and effectively. It involves scrutinizing safety invariants and stabilization mechanisms to guarantee the stability and reliability of the overall control system.

Chapter 4

Verification of Neural Networks

Our research is centered around the automatic functional verification of nonlinear artificial neural networks through the utilization of the SMT solver iSAT. Following a comprehensive review of the theoretical components associated with the problem we are addressing, we have organized our efforts into two primary categories: verification of feed-forward and LSTM neural networks.

Our research journey commences with a succinct explanation of the data acquisition process, where we obtain neural network information, namely the structure and parameters of a trained network, using the ONNX format. We then delve into the intricacies of translating these data into iSAT expressions, with a particular focus on the activation functions. Each step of the translation process for various activation functions is meticulously detailed. Furthermore, we explore optimizations that can be applied during the conversion of neural networks into iSAT expressions to enhance verification efficiency. One of the most pivotal optimizations involves establishing precise bounds on all variables essential for mapping the neural network functionality into iSAT. This not only reduces the search space that iSAT needs to navigate but also eliminates extraneous expressions that do not contribute to the accurate evaluation of the network's output. In essence, our research aims to aid the field of automatic functional verification for nonlinear ANNs by offering a comprehensive framework that enhances the reliability and efficiency of this critical process. Through our systematic approach, we are advancing the capabilities of SMT solver iSAT and contributing to the broader landscape of artificial intelligence research.

Table 4.1: File format in different frameworks

Framework	File Format	File extension
TensorFlow	Protobuf	.pd
Keras	H5	.h5
Caffe	Caffe model	.caffemodel
PySpark	MLeap	.zip
PyTorch	Torch Script	.pt
Scikit-learn	Pickled Python	.pkl
iOS Core ML	Apple ML Model	.mlmodel

4.1 Methodology

The proposed method consists of two steps: First, we develop a neural network as usual, which involves training and testing the desired neural network with nonlinear activation functions. We then automatically extract all the learned features and proceed to automated functional verification. This requires exporting all the weights W and biases b_i that have been learned during the training process using optimization techniques like backpropagation.

4.2 Translating to iSAT

Converting neural networks into alternative formats suitable for formal verification presents a formidable challenge. This difficulty arises from several factors, including the network's size, the diverse range of library types, and more. The presence of numerous distinct frameworks for neural network definition, training, and the creation of other machine learning applications, each with its unique approach to saving trained networks, adds to the complexity.

Consequently, the development of a way that can handle the myriad variations across such a wide spectrum of formats poses a substantial and intricate challenge. Several formats are employed by different frameworks to store models. For instance, TensorFlow [Dev22] and Keras [KK17] utilize formats like Protobuffers or HDF5. In contrast, PyTorch [KMKM21] serializes the data and stores it within compressed zip files.

We designed a system for the purpose of transforming neural networks, which can be generated using virtually any framework, into representations that can be evaluated and processed by iSAT. Additionally, our designed system must be able to easily undergo continuous maintenance and updates whenever a framework undergoes minor format changes or when a new framework is introduced. To address this, an intermediate step is introduced, wherein the different formats are initially converted into a standardized universal

Table 4.2: Properties of the graph object in ONNX [onn]

Name	Type	Description
name	string	The name of the model graph
node	Node[]	A list of nodes, forming a partially ordered computation graph based on input/output data dependencies.
initializer	Tensor[]	A list of named tensor values. When an initializer has the same name as a graph input, it specifies a default value for that input. When an initializer has a name different from all graph inputs, it specifies a constant value. The order of the list is unspecified.
doc-string	string	Human-readable documentation for this model. Markdown is allowed.
input	ValueInfo[]	The input parameters of the graph, possibly initialized by a default value found in 'initializer.'
output	ValueInfo	The output parameters of the graph. Once all output parameters have been written to by a graph execution, the execution is complete.
value-info	ValueInfo[]	used to store the type and shape information of values that are not inputs or outputs.

format. This approach allows our system to exclusively support this universal format, eliminating the need to recognize and accommodate the specific framework used for training the network that needs to be converted [Fib22].

Open neural network exchange (ONNX) [onn] is a project with the goal of enhancing the interoperability of neural network frameworks. To accomplish this interoperability, ONNX introduces an open format that specifies the structures for storing machine learning models, along with techniques for major frameworks to store and retrieve models in the ONNX format. Because the ONNX format offers both an open structure and a comprehensive toolset that encompasses all crucial interactions required by major network frameworks, it represents an ideal choice as a universal intermediary format that can serve as standardized input for our designed system.

For frameworks that are not directly supported by the ONNX project, ONNX provides libraries for various programming languages and corresponding documentation, enabling developers of such frameworks or tools to independently integrate support for the ONNX format into their systems. Although our designed system is not a machine learning framework itself, we can leverage the ONNX library to load models in the ONNX format and extract the essential data.

The $ModelProto$ structure serves as the top-level object in the ONNX format. It holds versioning details and includes the *GraphProto* container. Table 4.2 shows the structure of the *GraphProto* datatype. This container, which is a specific ONNX data type, charac-

terizes the semantics of the trained networks and encompasses all the data required by our designed system. Within the node parameter, there exists a list of nodes that collectively form a structured acyclic graph. Each entry in this list represents a node identified by its op_{type}, which specifies the ONNX operator employed for the node's computation. Furthermore, each node contains references to the data structures used as inputs by the operator, as well as references utilized by other nodes to access the output of the current node. This graph of ONNX operators defines the structure of the expression.

All static values, i.e. parameters referenced by the operators are stored in the *initializer* list. Each *initializer* stores information such as the name used as a reference for the data, the dimensions of the data, the raw data containing binary information for all values, and the data type necessary for reconstructing the values from the raw data. In conjunction with the *initializer* data, the operator graph provides all the essential information required. Additionally, the input and output parameters define the input and output dimensions of the model. When a model is stored in the ONNX format, it undergoes a transformation into a collection of standardized ONNX operators (opset) that constitute the ONNX graph. In

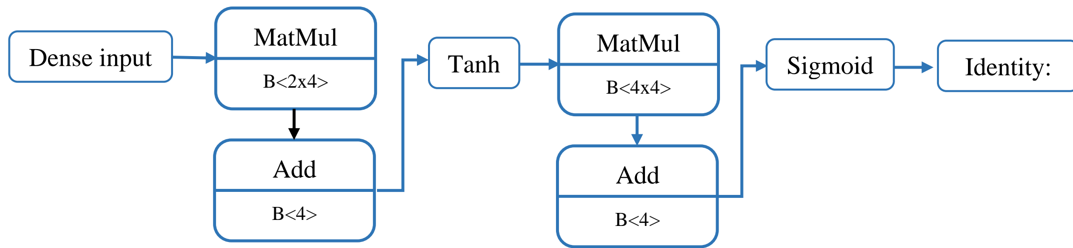


Figure 4.1: ONNX Operator graph of a neural network

equations (4.1) to (4.5), where defined inputs i_1 and i_2 are utilized, a mathematical abstraction is presented, illustrating operations that involve matrix multiplication (MatMul) and element-wise addition (Add). This formula elucidates the manipulation of input variables, weights, and biases to generate the resulting output tensors.

$$\text{input} : 0 = \begin{bmatrix} i_1 & i_2 \end{bmatrix} \tag{4.1}$$

$$\text{MatMul/1} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix} \tag{4.2}$$

$$\text{Add/1} = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix} \tag{4.3}$$

$$\begin{gathered} \text{MatMul}(input : 0, \text{MatMul/1}) = \\ \begin{bmatrix} i_1 w_{1,1} + i_2 w_{2,1} & i_1 w_{1,2} + i_2 w_{2,2} & i_1 w_{1,3} + w_{2,3} & i_1 w_{1,4} + i_2 w_{2,4} \end{bmatrix} \end{gathered} \tag{4.4}$$

$$\begin{aligned} &\text{Add}(\text{MatMul}(input: 0, \text{MatMul}/1), \text{Add}/1) = \\ &\begin{bmatrix} i_1 w_{1,1} + i_2 w_{2,1} + b_1 & i_1 w_{1,2} + i_2 w_{2,2} + b_2 & i_1 w_{1,3} + w_{2,3} + b_3 & i_1 w_{1,4} + i_2 w_{2,4} + b_4 \end{bmatrix} \end{aligned} \tag{4.5}$$

Each operator includes all the parameters needed for its function and references to the operator's inputs. These references can either point to the output of another operator or a static data matrix (which is also provided by the ONNX model file). When examining a simple three-layer network graph (refer to Figure 4.1), we can make two observations. Firstly, ONNX operators apply their functions to matrices where the first dimension corresponds to the number of neurons in a layer. Consequently, they represent layers of the network rather than individual neurons. Secondly, a layer is not defined by a single operator but is instead composed of a multiplication operation (MatMul), an addition operation (Add), and optionally an operator for the activation function.

By combining the MatMul, Add, and the optional activation function operator, our designed system possesses all the essential information needed to reconstruct the formula for individual neurons. Through the use of the ONNX operator graph of a network, the system reconstructs both the overall layer structure and the individual neurons. This reconstruction of each neuron introduces the possibility of optimizing the neural network graph.

4.2.1 Encoding a Feed-Forward Neural Network into iSAT

Building upon the definition provided in Chapter 2 (see definition 2.1.2), we characterize the artificial neural network as follows: we assume that our network is a classifier ANN with k disjoint classes. Let m be the number of layers in the network, denoted as $L = L_0, L_1, \ldots, L_m$. The input vectors corresponding to x network input values are represented as $X = [x_1, x_2, x_3, \ldots, x_{n-1}, x_n]$, where $x_i \in [L_x, U_x] \subseteq \mathbb{R}$, and L_i represents the lower limit, and U_i signifies the upper limit of the input value range. Similarly, W is a weight vector represented as $W = [w_{1,1}, w_{1,2}, w_{1,3}, \ldots, w_{m,n-1}, w_{m,n}]$, and $B = [b_{1,1}, b_{1,2}, b_{1,3}, \ldots, b_{m,n-1}, b_{m,n}]$, where w and b represent the weights and biases, respectively. We consider T_o and T_z as the dimensions of the input and output, respectively. The network output Y is represented as $y \in [\mathrm{L}_y, U_y] \subseteq \mathbb{R}$ and is generally obtained by applying the activation function to the result of multiplying the weights of each layer by the inputs and adding the bias.

4.2.2 Declaration of Input Data into iSAT

Figure 4.2 and 4.3 display the declaration of the input neurons and the first layer within the translated network in iSAT. All variables occurring in the formula to be solved need to be declared [MF21]. The types supported by iSAT are float (computational reals according to the IEEE standard), int (bounded integers), real (subranges of the mathematical reals), and Boolean. Translation rules transforming the network node by node follow: the input values of the neural network are in a specified range depending on their type. Let $x_{i,j}$ be the j-th node in the i-th layer. An iSAT variable reflecting its value can be defined by the following declaration:

$$x_{i,j} \in \{[l_x, u_x] \mid l_x, u_x \subseteq \mathbb{R}\} \tag{4.6}$$

```
DECL
    -- Input neurons
    float [0, 0] x_0_0;
    float [733949/4194304, 4417397/16777216] x_0_1;
    float [0, 0] x_0_2;
    float [0, 0] x_0_3;
    float [15718143/67108864, 8644571/33554432] x_0_4;
    float [2269289/16777216, 3917081/8388608] x_0_5;
    float [9681549/134217728, 7033605/67108864] x_0_6;
```

Figure 4.2: Declaration of the Input neurons of the translated network, showcasing 7 neurons in iSAT

The nodes in subsequent layers are obtained by applying the activation function to the result of multiplying the weights of each layer by the inputs and adding the bias. Ultimately, all auxiliary variables contribute to the calculation, and the result is stored in the node $x_{i,j}$ for the next calculation. Weight and bias values, unlike input and output values, need to be defined as constants and do not change during the verification phase. They are known before the verification process. The iSAT formula is recursively defined over the number of layers and the activation functions, and it encodes the activation function itself.

Definition 2. *As shown in equation 4.7, for simplicity, let $\Sigma_{j=1}^{n} x_{i-1,j} w_{i-1,j} + b_i$ denote the node value before the activation function. Let $x_{i,j}$ be the value of the j-th node in the i-th layer L_i, and $x_{i-1,j}$, $w_{i,j}$, and $b_{i,j}$ be the input, weight, and bias from the i-th layer.*

$$x_{i,j} = \Sigma_{j=1}^{n} x_{i-1,j} w_{i-1,j} + b_i \tag{4.7}$$

In the subsequent section, we present iSAT code translations for several activation functions, including ReLU, sigmoid, tanh, and swish. The corresponding iSAT code translations for each function are represented by Equations 4.10, 4.12, 4.14, and 4.32.

```
-- Layer 1 neurons
float [0, 1] x_1_0, x_1_0_0, x_1_0_1, x_1_0_2, x_1_0_3, x_1_0_4, x_1_0_5;
float [0, 1] x_1_1, x_1_1_0, x_1_1_1, x_1_1_2, x_1_1_3, x_1_1_4, x_1_1_5;
float [0, 1] x_1_2, x_1_2_0, x_1_2_1, x_1_2_2, x_1_2_3, x_1_2_4, x_1_2_5;
float [0, 1] x_1_3, x_1_3_0, x_1_3_1, x_1_3_2, x_1_3_3, x_1_3_4, x_1_3_5;
float [0, 1] x_1_4, x_1_4_0, x_1_4_1, x_1_4_2, x_1_4_3, x_1_4_4, x_1_4_5;
float [0, 1] x_1_5, x_1_5_0, x_1_5_1, x_1_5_2, x_1_5_3, x_1_5_4, x_1_5_5;
float [0, 1] x_1_6, x_1_6_0, x_1_6_1, x_1_6_2, x_1_6_3, x_1_6_4, x_1_6_5;
float [0, 1] x_1_7, x_1_7_0, x_1_7_1, x_1_7_2, x_1_7_3, x_1_7_4, x_1_7_5;
float [0, 1] x_1_3, x_1_8_0, x_1_8_1, x_1_8_2, x_1_8_3, x_1_8_4, x_1_8_5;
```

Figure 4.3: Declaration of the first layer of the translated network, showcasing 7 neurons in iSAT

- **ReLU activation function**: For the maximum operation, an operator is available in iSAT2, which can be used to convert this function.

$$f_{\text{ReLU}}(x_{i-1,j}) = \max(0, x_{i-1,j}) \tag{4.8}$$

Alternatively, in iSAT3, the maximum operation can be created using the iSAT operator ite (if-then-else).

$$f_{\text{ReLU}}(x_{i-1,j}) = \begin{cases} 0 & \text{if } x_{i-1,j} < 0 \\ x_{i-1,j} & \text{if } x_{i-1,j} \geq 0 \end{cases} \tag{4.9}$$

$$f_{\text{ReLU}}(x_{i-1,j}) = \text{ite}(x_{i-1,j} < 0, 0, x_{i-1,j}) \tag{4.10}$$

- **Sigmoid activation functions**: Sigmoid employs division, addition, and exponential operations. iSAT provides operators for addition (+) and exponential (exp()), but there is no specific operator for division. However, iSAT does not require explicit value assignments (e.g., x = 1 + 2), as it calculates variable values from available expressions. Hence, iSAT can directly deduce the value of x from the expression x - 2 = 1, resulting in x = 3. To address the absence of a division operator, iSAT accomplishes division by multiplying with the reciprocal.

$$x_{i,j,k} = f_{\text{sigmoid}}(x_{i-1,j}) = \frac{1}{1 + e^{-x_{i-1,j}}} \tag{4.11}$$

$$x_{i,j,k} \cdot (1 + \exp(-x_{i-1,j})) = 1 \tag{4.12}$$

- **Tanh activation functions**: By leveraging iSAT operators and cleverly handling divisions as multiplications by their inverses, the given function can be transformed

into an iSAT expression.

$$x_{i,j,k} = f_{\tanh}(x_{i-1,j}) = \frac{\exp(x_{i-1,j}) - \exp(-x_{i-1,j})}{\exp(x_{i-1,j}) + \exp(-x_{i-1,j})} \tag{4.13}$$

$$x_{i,j,k} \cdot (\exp(x_{i-1,j}) + \exp(-x_{i-1,j})) = \exp(x_{i-1,j}) - \exp(-x_{i-1,j}) \tag{4.14}$$

However, the objective is to minimize computational complexity by encoding each formula with as few expensive iSAT operators (such as exp) as possible. To achieve this, we can cleverly factor out either e^x or e^{-x}, leading to the elimination of two exp operators. Moreover, the formula can be further simplified by factoring out $e^{2x} + 1$ or $e^{-2x} + 1$ (Formula 4.26). This approach effectively reduces the number of costly operations involved in the computation.

First approach of factoring out e^{-x} from Tanh activation function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.15}$$

$$= \frac{e^{-x}(e^{2x} - 1)}{e^{-x}(e^{2x} + 1)} \tag{4.16}$$

$$= \frac{e^{2x} - 1}{e^{2x} + 1} \tag{4.17}$$

$$= \frac{e^{2x} - 1 + 2 - 2}{e^{2x} + 1} = \frac{(e^{2x} + 1) - 2}{e^{2x} + 1} \tag{4.18}$$

$$= \frac{e^{2x} + 1}{e^{2x} + 1} - \frac{2}{e^{2x} + 1} \tag{4.19}$$

$$= 1 - \frac{2}{e^{2x} + 1} \tag{4.20}$$

Second approach of factoring out e^{-x} from Tanh activation function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.21}$$

$$= \frac{e^x(1 - e^{-2x})}{e^x(1 + e^{-2x})} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{4.22}$$

$$= -1\frac{-1(1 - e^{-2x})}{1 + e^{-2x}} = -1\frac{-1 + e^{-2x}}{1 + e^{-2x}} \tag{4.23}$$

$$= -1\frac{-1 + e^{-2x} + 2 - 2}{1 + e^{-2x}} = -1\frac{(1 + e^{-2x}) - 2}{1 + e^{-2x}} \tag{4.24}$$

$$= -1\left(\frac{1 + e^{-2x}}{1 + e^{-2x}} - \frac{2}{1 + e^{-2x}}\right) = -1\left(1 - \frac{2}{1 + e^{-2x}}\right) \tag{4.25}$$

$$= -1 + \frac{2}{1 + e^{-2x}} \tag{4.26}$$

If the hyperbolic tangent is simplified by factoring out e^{-x}, the resulting expression bears a strong resemblance to the sigmoid function. In fact, upon substituting 4.26 into 4.27, it becomes evident that the hyperbolic tangent is essentially a sigmoid function that has been both shifted and rescaled, as indicated in equation 4.29 [Fib22].

$$\tanh(x) = \frac{-1+2}{1+e^{-2x}} \tag{4.27}$$

$$\text{sigmoid}(2x) = \frac{1}{1+e^{-2x}} \tag{4.28}$$

$$\Rightarrow \tanh(x) = -1 + 2\text{sigmoid}(2x) \tag{4.29}$$

Then in the encoding Tanh from the equation 4.29 to the iSAT we have:

$$x_{i,j,k} = 1 + 2f_{sigmoid}(2x_{i-1,j}) \tag{4.30}$$

- **Swish activation function:**

$$x_{i,j,k} = f_{\text{swish}}(x_{i-1,j}) = \frac{x_{i-1,j}}{1+e^{-x_{i-1,j}}} \tag{4.31}$$

$$x_{i,j,k} \cdot (1 + \exp(x_{i-1,j})) = x_{i-1,j} \tag{4.32}$$

4.2.3 Encoding LSTMs into iSAT

In order to obtain a recurrent neural network to be verified, we train a neural network containing LSTM nodes [MFF23]. As usual, this training factually alternates between training and testing phases until the network is empirically found to be well-behaved. We then proceed to verification by first extracting all parametric features that have been adjusted, i.e., learned, during the training. This means we have to export all weights W and biases B for every gate in the LSTM that have been determined during the training by the optimization procedure of back-propagation. Assume a layer in the LSTM ν that receives n inputs forming an input vector $X = [x_0, x_1, x_2, ..., x_n]$ ranging over $x_m \in [L_x, U_x] \subset \mathbb{R}$, where L_m and U_m are the lower bound and upper bound, respectively, defining an interval where input values in this range are acceptable.

These inputs can be the outputs from another neural network layer, such as a fully connected layer, or inputs to the overall network.

The neural network layer then receives input sequences over time, where $x_{m,p}$ is the m-th input at time-step p. Similarly, the layer features outputs $x_{m,p}$ with an output interval $y_{m,p} \in [L_y, U_y] \subseteq \mathbb{R}$. As the LSTM is a stateful network, there are also states $H = [h_{0,0}, h_{0,1}, \ldots, h_{n,t}]$ where $h_{m,p}$ represents the hidden state for the m-th feature at time-

step p. Similarly, the cell states across the features and time-steps are described by $C = [c_{0,0}, c_{0,1}, \dots, c_{n,t}]$ where $c_{m,p}$ denotes the cell state for the m-th feature at time-step p capturing the long-term memory of the LSTM for each feature and time-step. The outputs of the LSTM are collected in the matrix $O = [o_{0,0}, o_{0,1}, \dots, o_{n,t}]$ where $o_{m,p}$ refers to the output for the m-th at time-step p. This matrix represents the final output of the entire LSTM, with each $o_{m,p}$ being the processed result at each time-step for the corresponding feature.

In our subsequent discussion of an LSTM network, all units are pooled into vectors: g represents the vector of cell input, i represents the vector of input gates, f represents the vector of forgetting gates, c represents the vector of memory cell states, o represents the vector of output gates, and y represents the vector of cell outputs.

Unlike input and output values, weight and bias values are constants that are determined during the training phase and are known prior to the verification. W represents the overall weight vector. For an LSTM cell, the weights $W_i = [w_{i,f}, w_{i,i}, w_{i,g}, w_{i,o}]$ and $W_h = [w_{h,g}, w_{h,i}, w_{h,f}, w_{h,o}]$ correspond to the weights of the connections between inputs and the cell input, input gate, forget gate, and output gate, respectively. The vectors $B = [b_g, b_i, b_f, b_o]$ represent the bias vectors of the cell inputs, input gates, forget gates, and output gates, respectively.

The LSTM also generates the signals c_t and h_t, providing the state output to the next time step and forming a special case of a general time-discrete recursive neural network. We use the variable μ to count the time steps across the recursive network evaluation process. We represent the dynamic behavior of the LSTM as a symbolic transition system, using the primed-variable notation of iSAT to encode h' and c' as the next-state values of the LSTM cell state c_{t+1} and hidden state h_{t+1}, respectively. This is achieved by employing the prime notation in *Bounded Model Checking* (BMC) with iSAT.

In iSAT, all variables present in the formula to be solved must be declared [isa10]. The following equations (4.33)–(4.43) illustrate the translation of an LSTM node with $\mu = t$ to iSAT. For instance, equation (4.35) is a direct translation of equation (2.10). The variable μ is employed to keep track of the progression of time steps by BMC in iSAT throughout the recursive network evaluation procedure (Figure 4.5).

As illustrated in Figure 4.4 and 4.6 for simplicity, we denote auxiliary variables associated with input gates, forget gates, cell states, output gates, and cell outputs as $x_{i,j}_it$, $x_{i,j}_ft$, $x_{i,j}_gt$, $x_{i,j}_ct$, and $x_{i,j}_ot$, respectively. In the nomenclature of these variables, $x_{i,j}$ indicates the LSTM node to which these auxiliary variables belong. In the iSAT code, the input gate of an LSTM cell is translated into Equation 4.35. Equation 4.33 calculates the input gate activation (x_{i,j_it}) using the sigmoid activation function. This line of code computes a weighted sum of the previous hidden state (h'), the current input (x_t), and bias

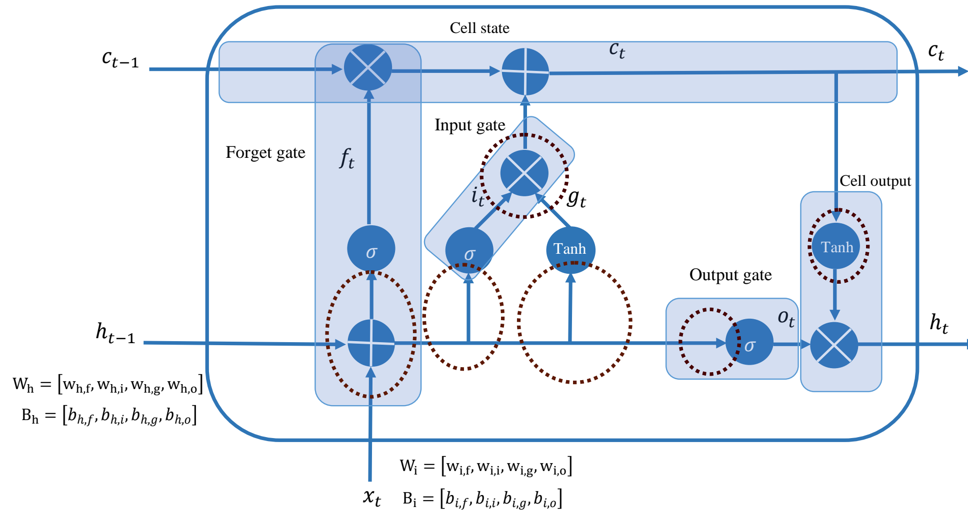


Figure 4.4: Structure of the LSTM with dashed circles representing spaces reserved for auxiliary variables exclusively needed for iSAT encoding [YSHZ19].

terms ($b_{h,i}$ and $b_{i,i}$), applying the sigmoid function to squash the result between 0 and 1. Equation 4.34, 4.36, 4.38, 4.38, and 4.43 represent the translation of LSTM gates in the iSAT code.

Translation of the input gate of an LSTM cell:

$$x_{i,j}_it = \sigma(w_{h,i}h' + w_{i,i}x_t + b_{h,i} + b_{i,i}) \tag{4.33}$$

$$x_{i,j}_it * (1 + exp10(w_{h,i}h' + w_{i,i}x_t + b_{h,i} + b_{i,i})) = 1 \tag{4.34}$$

Translation of the forget gate:

$$x_{i,j}_ft = w_{h,f}h' + w_{i,f}x_t + b_{h,f} + b_{i,f} \tag{4.35}$$

$$x_{i,j}_ft * (1 + exp10(w_{h,f}h' + w_{i,f}x_t + b_{h,f} + b_{i,f})) = 1 \tag{4.36}$$

Translation of the cell gate:

$$x_{i,j}_gt = tanh(w_{h,g}h' + w_{i,g}x_t + b_{h,g} + b_{i,g}) \tag{4.37}$$

```
TRANS
    step' = step + 1;
    step = 0 -> x_0_0' = 0;
    step = 1 -> x_0_0' = 33490438/1761602;
    step = 2 -> x_0_0' = 33554420/1677721;
    step = 3 -> x_0_0' = 66437676/3019898;
    step = 0 -> x_0_1' = 33730304/243648;
    step = 1 -> x_0_1' = 33813248/236544;
    step = 2 -> x_0_1' = 33630400/231520;
    step = 3 -> x_0_1' = 49656576/3355443;
    step = 0 -> x_0_2' = 4322285/16777216;
    step = 1 -> x_0_2' = 16686701/6710886;
    step = 2 -> x_0_2' = 6139911/2621442;
    step = 3 -> x_0_2' = 87789335/567137;
    step = 0 -> x_0_3' = 0;
    step = 1 -> x_0_3' = 33554616/11184872;
    step = 2 -> x_0_3' = 33554432/16777216;
    step = 3 -> x_0_3' = 33554432/33554432;
    step = 0 -> x_0_4' = 33554400/3355441;
    step = 1 -> x_0_4' = 12329495/134217728;
    step = 2 -> x_0_4' = 14067209/134217728;
    step = 3 -> x_0_4' = 33385204/0276824;
```

Figure 4.5: iSAT code snapshot: X_0_0 to X_0_4 indicate the input values in every time steps which here $\mu = 3$

$$x_{i,j}_g_t * \left(\exp 10\left(w_{h,g}h' + w_{i,g}x_t + b_{h,g} + b_{i,g}\right) + \exp 10\left(-(w_{h,g}h' + w_{i,g}x_t + b_{h,g} + b_{i,g})\right)\right)$$
$$= \exp 10\left(w_{h,g}h' + w_{i,g}x_t + b_{h,g} + b_{i,g}\right) - \exp 10\left(-(w_{h,g}h' + w_{i,g}x_t + b_{h,g} + b_{i,g})\right) \quad (4.38)$$

$$x_{i,j}_ct = x_{i,j}_it \odot x_{i,j}_gt + x_{i,j}_ft \odot x_{i,j}_ct' \quad (4.39)$$

Translation of the output gate:

$$x_{i,j}_ot = \sigma(w_{h,o}h' + w_{i,o}x_t + b_{h,o} + b_{i,o}) \quad (4.40)$$

$$x_{i,j}_ot * (1 + exp10(w_{h,o}h' + w_{i,o}x_t + b_{h,o} + b_{i,o})) = 1 \quad (4.41)$$

Translation of cell output:

$$x_{i,j}_yt = x_{i,j}_ot \odot \tanh(x_{i,j}_ct) \quad (4.42)$$

$$x_{i,j}_yt * (exp10(x_{i,j}_ct) + exp10(-(x_{i,j}_ct))) = (x_{i,j}_ct) - exp10(-(x_{i,j}_ct))) \odot x_{i,j}_ot \quad (4.43)$$

```
Expression for Layer: 1
 x_1_0_0t * (1 + exp(-1*((x_0_0 * (-6422877/8388608)) + (x_0_1 * (-11004421/67108864)) + (x_0_2 * (8168739/8388608))
 x_1_0_ft * (1 + exp(-1*((x_0_0 * (-11618519/8388608)) + (x_0_1 * (5099405/8388608)) + (x_0_2 * (3724805/8388608)) +
 x_1_0_ct * (1 + exp(-1*((x_0_0 * (-4204209/4194304)) + (x_0_1 * (-7182545/33554432)) + (x_0_2 * (12308467/8388608))
 x_1_0_Ct_0 = x_1_0_ft * x_1_0_Ct_0 + x_1_0_0t * x_1_0_ct;
 x_1_0_ot * (1 + exp(-1*((x_0_0 * (-5197209/4194304)) + (x_0_1 * (7018609/2147483648)) + (x_0_2 * (2314727/4194304))
 x_1_0_v0 * (1 + exp(-1*(x_1_0_Ct_0))) = 1;
 x_1_0 = x_1_0_ot * x_1_0_v0;
 x_1_1_0t * (1 + exp(-1*((x_0_0 * (-6973781/4194304)) + (x_0_1 * (-10460509/134217728)) + (x_0_2 * (14886939/6710886
 x_1_1_ft * (1 + exp(-1*((x_0_0 * (-7899085/2097152)) + (x_0_1 * (-4360107/16777216)) + (x_0_2 * (-5411075/16777216)
 x_1_1_ct * (1 + exp(-1*((x_0_0 * (-12567883/8388608)) + (x_0_1 * (7119309/16777216)) + (x_0_2 * (3584707/8388608))
 x_1_1_Ct_1 = x_1_1_ft * x_1_1_Ct_1 + x_1_1_0t * x_1_1_ct;
 x_1_1_ot * (1 + exp(-1*((x_0_0 * (-2200577/1048576)) + (x_0_1 * (-5915275/8388608)) + (x_0_2 * (10903299/268435456)
 x_1_1_v0 * (1 + exp(-1*(x_1_1_Ct_1))) = 1;
 x_1_1 = x_1_1_ot * x_1_1_v0;
 x_1_2_0t * (1 + exp(-1*((x_0_0 * (-12824547/8388608)) + (x_0_1 * (-7296031/33554432)) + (x_0_2 * (8978415/8388608))
 x_1_2_ft * (1 + exp(-1*((x_0_0 * (-6685751/2097152)) + (x_0_1 * (-11106181/16777216)) + (x_0_2 * (-4240509/33554432
 x_1_2_ct * (1 + exp(-1*((x_0_0 * (-13511789/8388608)) + (x_0_1 * (-12237831/33554432)) + (x_0_2 * (14446111/6710886
 x_1_2_Ct_2 = x_1_2_ft * x_1_2_Ct_2 + x_1_2_0t * x_1_2_ct;
 x_1_2_ot * (1 + exp(-1*((x_0_0 * (-5723157/2097152)) + (x_0_1 * (-1918503/4194304)) + (x_0_2 * (-8479371/33554432))
 x_1_2_v0 * (1 + exp(-1*(x_1_2_Ct_2))) = 1;
 x_1_2 = x_1_2_ot * x_1_2_v0;
 x_1_3_0t * (1 + exp(-1*((x_0_0 * (15278149/4194304)) + (x_0_1 * (4739825/8388608)) + (x_0_2 * (6671833/33554432)) +
 x_1_3_ft * (1 + exp(-1*((x_0_0 * (3349667/1048576)) + (x_0_1 * (12343481/33554432)) + (x_0_2 * (11459973/8388608))
 x_1_3_ct * (1 + exp(-1*((x_0_0 * (12480247/4194304)) + (x_0_1 * (14387561/33554432)) + (x_0_2 * (1601767/4194304))
 x_1_3_Ct_3 = x_1_3_ft * x_1_3_Ct_3 + x_1_3_0t * x_1_3_ct;
 x_1_3_ot * (1 + exp(-1*((x_0_0 * (6252679/2097152)) + (x_0_1 * (372807/1048576)) + (x_0_2 * (14909615/16777216)) +
 x_1_3_v0 * (1 + exp(-1*(x_1_3_Ct_3))) = 1;
 x_1_3 = x_1_3_ot * x_1_3_v0;
```

Figure 4.6: iSAT code snapshot: first layer of translated LSTM neurons with auxiliary variables. X_0_0 to X_0_9 indicate the input variables.

The $\odot$ symbol between two variables indicates the element-wise multiplication of two input vectors with k elements. The translation scheme described above offers a compositional translation of an LSTM into a symbolic transition system, which results in a transition system of linear size with respect to the size of the LSTM.

4.3 Property Definition

To facilitate iSAT in verifying the properties of a particular network, it is not enough to just provide the network itself in a format that iSAT can handle. Equally crucial is the definition of the verification goal outlining how iSAT should evaluate the network. These criteria need to be specified through property definitions. Property definitions encompass two types of constraints. The first type constrains the range of inputs to the neural network, representing the scenario against which the property's compliance should be assessed. The second type of constraint outlines the desired properties that the network's output must satisfy. These constraints describe a target condition for which iSAT checks whether it's achievable based on the given input scenario. Since iSAT relies on interval constraint arithmetic, the information within the property can be used to optimize both variables and expressions further. Although it's possible to manually include the target conditions into the iSAT expressions from the network translation a posteriori, given the large number of expressions and the potential for additional optimizations, it's preferable to convert neural networks with a specific property in mind [Fib22].

The input: Constraints define the situation against which the property's adherence will be assessed. These input constraints can vary in scope, from restricting input values

to a specific range to specifying an exact input value. In iSAT, an input is straightforwardly portrayed as a floating-point variable with either an interval or a designated value. At a minimum, the property definition must specify the input interval. The assignment can take the form of ">", ">", "=", "<," or "<=" along with the corresponding assigned expression. If the assignment matches the input interval of the property, the assignment specifications in the property definition can be omitted. In this case, iSAT will automatically consider the input interval as the assignment definition. However, the property is not restricted to a single assignment; it should include as many assignments as necessary to accurately define the input scenario.

$$v_i \in N_{\text{inputs}}, \odot \in \{>=, >, =, <, <=\}, v_{\text{val}} \in \mathbb{R} : v_i \odot v_{\text{val}} \tag{4.44}$$

Output and Target: Similar to how our system automatically assigns names to the network's inputs, it also provides automatic generation for naming each output. This enhances usability by enabling the formulation of targets using these names instead of the default "*n_o<number >*". By utilizing the required outputs as parameters in the constraint, the target is then defined as one or a sequence of equations, from which a truth value is derived.

$$v_o \in \mathbb{N}_{\text{outputs}}, \odot \in \{>=, >, =, <, <=\}, v_{\text{val}} \in \mathbb{R} \cup \{N_{\text{outputs}} \setminus v_o\} \tag{4.45}$$

4.4 Optimization

This section of the thesis embarks on an exploration of optimization techniques to enhance the capabilities of iSAT in the verification of neural networks. Over the course of the following sections, we explore how these optimization methods play a pivotal role in augmenting the performance and efficiency of iSAT, ultimately contributing to the advancement of the formal verification process.

4.4.1 Enhancing Boundaries

iSAT employs interval arithmetic for determining satisfiability, necessitating predefined boundaries for each variable used in iSAT expressions. These boundaries serve to restrict the search space that iSAT needs to assess. While the standard boundaries for the output of activation functions are well-known and can be used during conversion, optimizing boundaries further can significantly enhance iSAT's performance. For instance, consider a neuron with an input range of [-12, 5] using the ReLU activation function, which typically

has a general range of $[0, \infty]$. However, a more detailed examination of this function concerning the input reveals that an evaluation with the optimized range of [0, 5] would suffice. Hence, our designed system offers the capability to explore potential range optimizations for each variable and provide these optimizations to iSAT in the form of refined range definitions for iSAT variables. This empowers iSAT to confine the satisfiability check to a narrower search space, ultimately boosting its performance.

To enable our designed system to ascertain optimized ranges, it's essential to establish the ranges for both the transfer function and the activation function. The transfer function operates by multiplying each input by its predefined weight and then adding the results to the bias. The range encompassing all potential outcomes of the transfer function can be derived by executing all operations of the transfer function using the input ranges rather than their actual values [Fib22].

However, it is crucial to note that the choice of boundaries for evaluation depends on the sign of the weights. Specifically, when determining the upper (lower) bound on the output range, we consider the upper (lower) bounds of all inputs with positive weights. Conversely, for inputs with negative weights, we take the lower (upper) bounds. This distinction is essential, as the sign of the weight influences the direction of the impact on the output range. Careful consideration of these factors ensures an accurate and meaningful determination of optimized ranges for the designed system.

Determining the boundaries of the activation function is a more intricate task, as it can assume various forms. As the formula is dissected into its constituent operators during this process, its boundaries can also be examined. The formula's monotonicity plays a pivotal role in determining the extent of scrutiny required for its boundaries analysis.

For activation functions that exhibit monotonic behavior, it is satisfactory to evaluate the function's outcome at both the lower and upper boundaries of its input. This is because the monotonicity property guarantees that the lower and upper boundaries consistently correspond to the minimum and maximum values within this input range.

As an example, let's consider the sigmoid activation function, defined in this function has a range between 0 and 1, and it monotonically increases from 0 to 1 as the input x increases from negative infinity to positive infinity.

Let us evaluate the sigmoid function as described in equation 2.6 for an input interval of $[11, 14.5]$:

- For $x = 11$:

$$\sigma(11) = \frac{1}{1 + e^{-11}} = 0.999983$$

- For $x = 14.5$:

$$\sigma(14.5) = \frac{1}{1 + e^{-14.5}} = 0.99999969$$

When evaluating the outcome of the sigmoid function at both the lower and upper boundaries of its input interval, we observe that it approaches 1 in both cases. This aligns with our understanding that the sigmoid function tends towards 1 as the input increases.

Furthermore, it is important to note the behavior of the function in extreme cases:

- **Lower boundary**: As x approaches negative infinity, e^{-x} approaches infinity, and thus the denominator $1 + e^{-x}$ approaches infinity as well. Consequently, $\sigma(x)$ approaches 0.

- **Upper boundary**: As x approaches positive infinity, e^{-x} approaches 0, and thus the denominator $1 + e^{-x}$ approaches 1. Consequently, $\sigma(x)$ approaches 1.

Therefore, by evaluating the sigmoid function at both the concrete input interval and the boundaries, we can understand its behavior comprehensively and ensure that it behaves as expected in various scenarios.

It is worth noting that the majority of common activation functions, such as the sigmoid, tanh, and ReLU, exhibit monotonic behavior. These functions are widely used in neural networks. In contrast, non-monotonic activation functions, including the *Gaussian Error Linear Unit* (GELU), Swish, HardSwish, *Rectified Exponential Unit* (REU), and Mish have been introduced to leverage different characteristics for improved learning capabilities in certain scenarios [ZMW+21].

Let us consider ReLU as a concrete example of an activation function exhibiting monotonic behavior. For ReLU activation as outlined in equation 2.8, it is straightforward to observe that the ReLU function is monotonic because, for any two inputs x_1 and x_2 where $x_1 < x_2$, the corresponding function values $f(x_1)$ and $f(x_2)$ also follow the inequality $f(x_1) \leq f(x_2)$. Now, let us consider evaluating the ReLU function at both the lower and upper boundaries of its input range:

- **Lower boundary**: $x = -\infty$

$$f(-\infty) = \max(0, -\infty) = 0$$

- **Upper boundary**: $x = +\infty$

$$f(+\infty) = \max(0, +\infty) = +\infty$$

In this case, the monotonic behavior of the ReLU function ensures that the lower boundary corresponds to the minimum value (0), and the upper boundary corresponds to the maximum value ($+\infty$) within the input range. This example illustrates the suitability of

evaluating the function at the input boundaries due to the monotonicity property, providing a clear understanding of the function's behavior across its entire range.

In the case of non-monotonic activation functions, the initial boundary calculations follow the same approach as for monotonic functions. However, for these non-monotonic functions, it becomes necessary to identify the local maxima and minima during the function parsing process. The GELU activation function [HG16] is defined as:

$$\text{GELU}(x) = 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \tag{4.46}$$

The derivative of GELU with respect to x is given by equation 4.47:

$$\begin{aligned}\text{GELU}'(x) = \frac{1}{2} + \frac{1}{2}\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \\ \times\left(1 - \tanh^2\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)\left(0.5\sqrt{\frac{2}{\pi}}(1 + 0.134145x^2)\right)\end{aligned} \tag{4.47}$$

In a subsequent step, the optimized upper and lower boundaries are determined by selecting the maximum and minimum values from the initial boundaries and all local maxima and minima found within the input interval. This procedure ensures that any extrema exceeding or falling short of the initial boundaries are accurately incorporated into the determined boundaries of the function for that specific input range. In scenarios where an activation function is decomposed into multiple iSAT expressions, it becomes necessary to establish the boundaries for each individual sub-function.

Concurrently, during the translation of activation functions into iSAT expressions (as discussed in Section 4.2), not only are the iSAT expressions themselves generated, but also the corresponding mathematical formula and its derivatives. These formulae and their derivatives serve as essential components for function analysis. Specifically, identifying local extrema involves identifying values of x where the first derivative $f'(x)$ equals zero, as such points signify either minima, maxima, or saddle points in the original function [Fib22].

$$f'(x) = 0 \Rightarrow \text{implies a critical point} \tag{4.48}$$

$$f'(x) = 0 \text{ and } f''(x) < 0 \Rightarrow \text{indicate a local maximum} \tag{4.49}$$

$$f'(x) = 0 \text{ and } f''(x) > 0 \Rightarrow \text{suggest a local minimum} \tag{4.50}$$

$$f'(x) = 0 \text{ and } f''(x) = 0 \Rightarrow \text{signify a saddle point} \tag{4.51}$$

For a comprehensive analysis of the function $f(x)$, the second derivative, $f''(x)$, is uti-

lized to ascertain whether any value of x that satisfies $f'(x) = 0$ corresponds to a minimum, maximum, or saddle point. However, when assessing the boundaries of the function, this distinction becomes non-essential, and merely recognizing the presence of such points is sufficient. Having information about all extrema and the capacity to compute the outcome of an activation function's formula, it becomes possible to define optimized boundaries in relation to the input interval, as previously mentioned. The outcomes of the formula for both the upper and lower input boundaries are calculated and saved in a temporary list, which also encompasses the extrema within the input boundaries. Following this, the minimum and maximum values from this list are furnished to iSAT as optimized boundaries for the given input scenario. For the initial layer, this process yields accurate boundaries based on the input scenario. However, this level of accuracy cannot be guaranteed for subsequent layers, where we in general obtain overapproximations of the actual value range due to the well-known dependency problem of interval arithmetic [Krä06].

In the training process of neural networks, individual neurons emphasize specific input features, the selection of which is often not understandable to humans. Consequently, it is highly probable that one neuron may approach its upper boundary whenever certain others gravitate toward the lower boundary. The boundary calculation, however, does not account for these relationships between neurons in the previous layer. Instead, it uses the previous layer's boundaries as the input intervals for the current layer's boundary calculations. Consequently, this approach to boundary generation is likely to produce overestimated boundaries.

Figure 4.7 illustrates an example network in which the first-layer neurons are weighted in opposing ways, ensuring that when one neuron has a positive output, the other has an output of zero, and vice versa. While computing the range of the output neuron based on the boundaries of each first-layer neuron suggests a range of [0,4], the actual range is [0,2] due to the interdependence between the first-layer neurons.

Computing boundaries while taking into account the dependencies between neurons would necessitate analyzing all potential input combinations for each neuron, including those in previous layers leading to the target neuron. This computational process would be highly resource-intensive and economically impractical, especially considering the marginal benefits of testing narrower intervals. Although the suggested approach may not yield the narrowest possible boundaries, it still identifies reasonably narrow intervals. Please note that the correctness of the verification verdicts generated by iSAT does not depend on the tightness of these value ranges, as long as they are overapproximating the true ranges. Tightening the ranges only reduces iSAT's search space and thereby enhances verification performance.

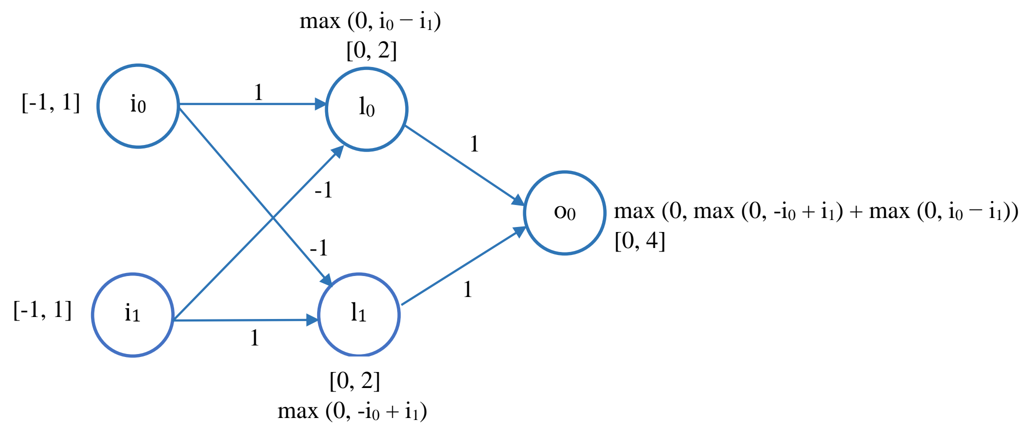


Figure 4.7: An illustrative depiction of the connections between neurons within a layer [Fib22].

4.4.2 Managing the Numerical Error During Boundary Calculation

Every computation involving real numbers in a computer relies on the approximation of real values into floating-point numbers. In computer science, floating-point numbers are a standardized data type [IEE19] that typically exists in five fundamental formats: binary 32-bit, 64-bit, and 128-bit, as well as decimal 64-bit and 128-bit formats. The IEEE-754-defined formats are known as dyadic rationals, representing a subset of rational numbers where the denominator is a power of two in equation 4.52.

$$\text{float} = \frac{a}{2^b} \tag{4.52}$$

Numbers that cannot be represented accurately must be approximated to a representable floating-point number. This approximation is a common practice for many rational numbers. By default, the approximation with the smallest numerical error is used to represent the number, and this practice introduces rounding nearest. This error can be further magnified in subsequent calculations. Any software in which the result of the calculation is mission-critical must take this error into account. Therefore, the SMT-solver iSAT uses interval arithmetic, which allows any number that cannot be exactly represented to be represented by the interval enclosed by the values of both rounding modes. Since both the computation of neuron outputs in the actual execution of neuronal networks as well as the generation of tight boundaries for iSAT variables their SMT encoding are subject to such numerical error, all calculations are performed with interval arithmetic [Fib22].

Let F be the set of floating-point numbers. The expression $z = \odot([x_l, x_u])$ represents

the interval result of applying the operation $\odot$ to the argument interval $[x_l, x_u]$, accounting for numerical errors due to floating-point representations.

- **Monotonic and antitonic operators**

  For monotonic operators (e.g., addition, and multiplication in each of its arguments individually), the operator preserves or inverts the ordering of the interval depending on whether the operation is monotonic (increasing) or antitonic (decreasing). In the context of binary operations, the result can be determined by evaluating the operation at the interval endpoints, but care must be taken regarding the nature of the operation.

  Thus, for a binary operator $\odot$, applied to intervals $[x_l, x_u]$ and $[y_l, y_u]$, we derive the following result in equation 4.53:

$$\begin{aligned}\odot([x_l, x_u], [y_l, y_u]) = \big[&\min\{\odot(x_l, y_l), \odot(x_l, y_u), \odot(x_u, y_l), \odot(x_u, y_u)\},\\ &\max\{\odot(x_l, y_l), \odot(x_l, y_u), \odot(x_u, y_l), \odot(x_u, y_u)\}\big]\,.\end{aligned} \tag{4.53}$$

  For multiplication, the behavior depends on the signs of the interval bounds. Multiplication is monotonic if both numbers are non-negative or both are non-positive, but it becomes antitonic if one number is positive and the other is negative. Therefore, to compute the correct interval, we must evaluate all four combinations of the interval endpoints: $(x_l, y_l), (x_l, y_u), (x_u, y_l), (x_u, y_u)$. For example, consider multiplying two intervals $[x_l, x_u] \times [y_l, y_u]$. The result is given by equation 4.54:

$$\begin{aligned}[x_l, x_u] \times [y_l, y_u] = \big[&\min(x_l \times y_l, x_l \times y_u, x_u \times y_l, x_u \times y_u),\\ &\max(x_l \times y_l, x_l \times y_u, x_u \times y_l, x_u \times y_u)\big]\,.\end{aligned} \tag{4.54}$$

  This ensures that all possible combinations are considered, even when intervals contain negative values or zero, which affect the sign of the product and may invert the ordering.

- **Non-monotonic operators**

  For non-monotonic operators (e.g., squaring, sine), the minimum and maximum may not occur at the interval endpoints. Therefore, we must evaluate the operator at points within the interval, considering critical points where the function's behavior changes.

  The general form for a unary non-monotonic operator $\oplus$ applied to an interval $[x_l, x_u]$ is presented in equation 4.55:

$$\oplus([x_l, x_u]) = \big[\min\{\oplus(y) \mid y \in [x_l, x_u]\}, \max\{\oplus(y) \mid y \in [x_l, x_u]\}\big]\,. \tag{4.55}$$

For instance, squaring the interval $[-2, 2]$ yields $[0, 4]$, because the minimum value occurs at $x = 0$, and the maximum occurs at $x = \pm 2$.

- **Generalizing to n-ary operators**

  To cover n-ary operators more generally, where we have multiple input intervals $[x_{l_1}, x_{u_1}], [x_{l_2}, x_{u_2}], \ldots, [x_{l_n}, x_{u_n}]$, the interval extension is given by equation 4.56:

$$\odot([x_{l_1}, x_{u_1}], [x_{l_2}, x_{u_2}], \ldots, [x_{l_n}, x_{u_n}]) = \big[\min\{\odot(y_1, y_2, \ldots, y_n) \mid y_i \in [x_{l_i}, x_{u_i}]\}, \max\{\odot(y_1, y_2, \ldots, y_n) \mid y_i \in [x_{l_i}, x_{u_i}]\}\big]\,. \tag{4.56}$$

  For binary operations like addition or multiplication, the formula simplifies to checking the extreme combinations of the interval endpoints, as shown in the multiplication example above.

The resulting interval from an operation is established by selecting the smallest and largest outcomes from the sub-operations to serve as the lower and upper boundaries. In the event that one of the sub-operation results is also not exactly representable, the interval for the exact value is utilized, and its boundaries are employed for the selection process. The application of interval arithmetic ensures that the outcome of each calculation is an interval encompassing its precise result, with sufficient width to accommodate the total numerical error. This ensures that the boundary created for each variable is at least as large as the boundary used by iSAT (equation 4.57).

$$\forall x \in V : B_{x,\mathrm{conv}} \supseteq B_{x,\mathrm{iSAT}} \tag{4.57}$$

4.4.3 Detecting Constant Sub-Expressions and Optimizing their Encoding

In addition to variable boundaries, another area where optimization is possible is in the analysis of expressions and their inputs. Since boundaries are already generated for all variables, they can be leveraged to optimize expressions. One significant way to achieve this optimization is through the evaluation of individual neurons. When the boundary for a neuron's output is a single point, the expression that computes the neuron's value can be simplified to a direct assignment of the value determined by that boundary. This simplification is possible because the generated boundaries are guaranteed to be equal to or larger than the exact boundaries of the calculations.

The boundary information can also play a role in optimizing the evaluation of neuron transfer functions. The transfer function involves multiplying each input by its weight

and summing the results. By examining the individual multiplications within the transfer function for point intervals, we can simplify them by assigning specific values. The input boundary is computed by the converter, whereas the weight is supplied by the neural network framework as an exact 32-bit floating-point number (e.g., TensorFlow stores weights and biases as 32-bit floating-point numbers). Analyzing the multiplications in the transfer function is simplified when dealing with exactly representable numbers since their intervals are inherently point intervals. In reference 4.59, the condition $(x_u - x_l) = 0$ indicates that the difference between the upper and lower bounds of the input interval collapses to zero, implying that the interval itself collapses to a single point. Consequently, the resulting interval of the multiplication operation also becomes a point interval.

The expression $(x_u - x_l) = 0 \vee w = 0$ in equation 4.61 underscores that either the difference between the upper and lower bounds of the input interval is zero, leading to a collapsed interval, or the weight is zero. In either case, the multiplication operation results in a zero-width interval. Hence, it is demonstrated that when an interval is multiplied by a point interval, the outcome will be a point interval if either both intervals are point intervals or the weight is zero.

$$[x_l, x_u] \cdot [w, w] = \begin{cases} [x_l \cdot w, x_u \cdot w] & \text{if } w \geq 0 \\ [x_u \cdot w, x_l \cdot w] & \text{if } w < 0 \end{cases} \tag{4.58}$$

$$x_u \cdot w - x_l \cdot w = 0 \tag{4.59}$$

$$(x_u - x_l) \cdot w = 0 \tag{4.60}$$

$$\Rightarrow (x_u - x_l) = 0 \vee w = 0 \tag{4.61}$$

$$\Rightarrow \text{width}([x_l, x_u]) = 0 \vee w = 0 \tag{4.62}$$

Another approach to optimize the expressions involves tracking the dependencies of each neuron in our approach. A neuron is considered dependent on another neuron if the multiplication of its weight with the input value range, provided by the other neuron, results in an interval with a non-zero width. This mechanism allows the converter to assess whether a neuron is genuinely essential for the network's evaluation. When it is determined that no other neuron depends on a particular neuron anymore, that neuron is marked as irrelevant, and all its associated expressions and variables can be omitted. Upon detecting an irrelevant neuron, all neurons in the previous layer need to be re-evaluated to check if any of them still rely on the now-irrelevant neuron after its dependency has ceased to exist. Additionally, any output neuron not utilized in the evaluation, based on the target, is also labeled as irrelevant. This ensures that only the expressions required for neurons essential to determine the satisfiability of the target are provided to the SMT solver. The

algorithm outlined for eliminating unnecessary expressions during network evaluation can take advantage of boundary optimizations in two ways. The first approach, which is likely more beneficial for network verification, involves employing constrained input boundaries determined by the property.

In this context, the suggested approach will exclusively generate the expressions that are genuinely needed for testing the specific scenario aligned with the property. Any determinations about irrelevant neurons are specific to the given scenario and cannot be extrapolated to other properties. The second approach involves employing the complete spectrum of boundaries for each input of the network. This enables edge optimization alongside the elimination of expressions resulting in point intervals, helping identify sections of neural networks that may go largely unused in solving the problem they were trained for due to the training process. These redundant neurons can be pruned from the neural network without affecting its output accuracy. These two approaches are illustrated in Figure 4.7, where l_g employs a general boundary based on the entire input range, potentially relevant for output, and l_p uses a property-based boundary yielding a point interval and therefore can be omitted [Fib22].

4.5 Converter

To facilitate the implementation of the methods we have described so far, We have employed a converter that comprises supplied translators from different neural network packages to the ONNX intermediate format, along with our self-developed translation from ONNX to iSAT2, iSAT3, and dReal [Fib22]. This converter serves the purpose of automatically transforming trained neural networks, even those with millions of nodes and originating from different frameworks such as TensorFlow, into the solver expressions format. It offers multiple levels of optimization, allowing us to explore automated functional verification of both feed-forward and LSTM networks. This automation greatly streamlines the process of preparing neural networks for analysis, making it feasible to conduct extensive verification studies efficiently. The converter's objective is to transform networks constructed with any framework into iSAT-evaluable statements.

It would take a lot of work to implement every single one of the frameworks' many different direct support forms. Additionally, the converter would need to be regularly updated and maintained if a framework launched a new framework or made a minor change to the existing format. Due to this, an intermediate step is added in which the various formats are first converted into a universal format, removing the need for the converter to be able to recognize and support the particular framework used to train the network to be converted and instead requiring support for this universal format.

Chapter 5

Evaluation

In Section 5, we provide detailed descriptions of three specific examples that were employed to test the effectiveness of the neural network verification techniques discussed earlier. These examples were carefully selected to demonstrate the robustness and accuracy of the proposed methods in handling different scenarios and complex systems. Each example serves as a test case to evaluate the performance and reliability of the implemented approach, showcasing its capability to analyze and verify the feedforward and recurrent neural networks in practical applications. By examining these concrete examples, we gain valuable insights into the capabilities and limitations of the proposed techniques, contributing to a comprehensive understanding of their applicability and potential in various real-world settings.

- **Benchmark 1**: involves a thorough scalability assessment using the *Modified National Institute of Standards and Technology database* (MNIST) dataset [LCB10]. It encompasses a collection of customizable neural networks specifically trained on the MNIST dataset. The primary goal here was to showcase the scalability of the proposed method in networks with feedforward structures. The networks used in this benchmark were highly customizable, enabling a wide range of problem solutions to be represented for the same task. By leveraging iSAT2 and iSAT3, the scalability of the approach in handling diverse neural network evaluations was thoroughly evaluated.

- **Benchmark 2** : involves training an LSTM neural network on a dataset of recorded traffic from the *Next Generation Simulation* (NGSIM project) [Adm07]. The pri-

mary proof obligation for iSAT was to verify whether the trained network accurately detected near collisions. The verification process was essential to ensure the robustness and reliability of the LSTM network's collision detection capabilities also outside the concrete training and test points used during network construction.

- **Benchmark 3**: focuses on investigating the scalability of iSAT when applied to LSTM neural networks. This study involved analyzing satellites and objects orbiting in space [Eur], which presents a complex scenario. By exploring this intricate system, we gained valuable insights into the scalability of our proposed method, shedding light on its efficiency in handling challenging scenarios with LSTM networks.

We measured the performance of each solver and the impact of the optimization with respect to the following four metrics:

- **Solver time**: Solver time signifies the duration that each solver employed during the verification process to establish satisfiability. While iSAT2 and dReal4 provide explicit information regarding the solving duration, iSAT3 lacks this feature, necessitating the utilization of the Linux command time to gauge the time consumption. Nevertheless, this command's measurements encompass not only the solver and preprocessing times but also a minor interval between result return and process termination. To ensure equitable comparisons, the internal solver time data from the other two solvers were disregarded. Instead, the time command was uniformly applied to gauge time for all solvers. By deducting the reported preprocessing time, a balanced approximation of solver time was derived, encompassing uniform imprecision inherent to all operating-system-based timing measurements.

- **Memory usage**: The amount of memory allocated for the verification process is acquired from the SLURM workload management system, which was employed to orchestrate each verification task. The values presented are slightly inflated due to the inherent limitations of the workload management system. It is capable of gauging the extent of memory allocation to the process, yet it lacks the capacity to furnish details about the precise memory consumption during the verification process itself.

5.1 Benchmark 1: MNIST Dataset

The MNIST dataset [LCB10], frequently employed for training image recognition methods, consists of handwritten numbers. These images are 28 by 28 pixels in size and display

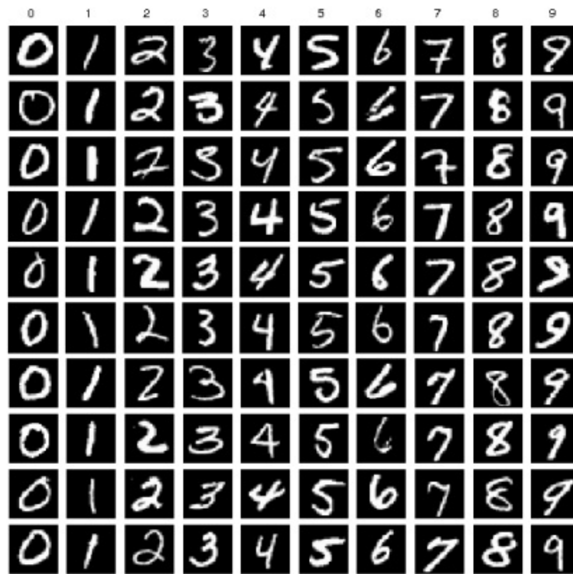


Figure 5.1: Instances of handwritten numbers of MNIST dataset [LCB10].

grayscale shades from 0 to 255. Each dataset entry contains the pixel information of the image, maintaining consistent dimensions where the digit was positioned and standardized. It also includes a class label that identifies the depicted digit in Figure 5.1. By training various networks using the MNIST dataset, they can be uniformly assessed due to identical input and output conditions. The sole variables at play are the network configurations and the precise weights and biases acquired during training. Although it is not feasible to entirely exclude weights and biases as factors influencing evaluation, their impact can be lessened by training numerous networks for each configuration and subsequently averaging outcomes. This strategy ultimately narrows down the performance influence to just the network configuration.

The configurations exhibit variations in terms of the number of layers and neurons within each layer, as well as the selection of activation functions employed. The evaluated number of layers spanned from one to six, and the allocation of neurons to these layers ranged from 250 to 1500, increasing in increments of 250. Among the choices were sigmoid and hyperbolic tangent as activation functions. To establish configurations, all possible combinations of 1 to 6 layers, 250 to 1500 neurons, and the two activation functions were utilized for training, with each configuration being applied to train the networks.

The neurons were distributed across the layers to resemble an exponential distribution as closely as feasible. In instances where the final layer would have received fewer than ten neurons, an adjustment was made: initially, all layers were equipped with ten neurons, and any remaining neurons were subsequently allocated to all layers except the last one. It is important to note that all networks underwent training with an output layer containing

ten neurons and utilizing sigmoid activation.

- **Property 1**: The initial property assessed on the MNIST networks involved an instance from the MNIST dataset portraying the digit seven. In this scenario, the solvers scrutinize whether there exists a possibility for the network's output, indicating the classification of the input as "seven," to fall below a threshold of 0.8. In Equation 5.1, the verification target is depicted, where n_o_0 represents the output of the neural network [Fib22].

$$\texttt{Target} : n_o_0 <= 0.8 \tag{5.1}$$

  The input remained devoid of any noise, thereby constraining the solvers to address only a single, precise value assignment of each input. In theory, the solution process should be straightforward due to the simplicity of the task. This particular *property* serves as a baseline, strategically chosen to demonstrate solver performance in the context of a highly uncomplicated criterion.

The experiment involved evaluating the $property1$ across five distinct optimization levels for a set of six networks, each characterized by varying node counts: 250, 500, 750, 1000, 1250, and 1500.

- **b0**: unoptimized boundaries. Since iSAT requires boundaries for all variables, infinite values were replaced with 245,000,000.
- **b1**: boundaries are based on the input ranges of the network.
- **fb1**: boundaries are based on the input ranges of the network. Irrelevant expressions were removed.
- **b2**: boundaries are based on the input ranges determined by the *property* to be checked.
- **fb2**: boundaries are based on the input ranges determined by the *property* to be checked. Irrelevant expressions were removed.

All verification processes were run on the CARL cluster which is a multi-purpose cluster designed to meet the needs of compute-intensive and data-driven research projects in Oldenburg University [vOU24] using nodes equipped with "Intel Xeon E5-2650 v4 12C" CPUs at 2.2GHz and 256 GB RAM consisting of 8x 32GB TruDDR4 modules at 2400MHz.

Performance on networks with sigmoidal activation functions

In order to assess the efficiency of iSAT2 and iSAT3 concerning networks employing the sigmoid activation function, a consistent function encoding was adopted across all converted networks. Table 5.1, 5.2, 5.3, and Figure 5.2 offer a comprehensive comparison of solver times for $property1$, focusing on networks utilizing the sigmoid activation function with iSAT2, iSAT3, and dReal solvers.

When examining solver times across different optimization levels, a notable trend becomes apparent. Notably, transitioning from boundaries based on the input interval $b1$ to boundaries determined by the $property1$, $b2$ brings about a significant reduction in solver time for iSAT2. Furthermore, the comparison between optimization strategies $fb1$ and $fb2$ with the other approaches is illuminating. $fb1$ demonstrates enhancement in solver performance while utilizing boundaries based on the input ranges of the network. In addition, $fb2$ demonstrates a significant decrease in solver time, reflecting consistent performance enhancements. This underscores the effectiveness and adaptability of these optimization strategies across different solver platforms.

Moreover, the results demonstrate that the proposed optimization strategies, $b2$, and $fb2$ yield comparable enhancements in solver performance across the iSAT2 solver. Within the context of the defined networks, iSAT2 demonstrated robust verification across all five levels of optimization parameters: b0, b1, $b2$, $fb1$, and $fb2$, without encountering any timeouts. However, it is worth noting an observation with iSAT3, where timeouts occurred during verification processes for $b0$, $b1$, and $fb1$ levels. Further analysis of solver times revealed intriguing disparities between iSAT2 and iSAT3, particularly evident in the $b2$ and $fb2$ optimization levels. Notably, solver times in these specific levels were noticeably shorter in iSAT2 compared to iSAT3. Both iSAT2 and iSAT3 showcased superior performance compared to dReal, particularly evident in scenarios involving networks with over 250 nodes. In contrast, dReal encountered timeouts under similar conditions, indicating potential scalability limitations.

The primary distinction between iSAT2 and iSAT3 lies in their treatment of boundaries. iSAT2 directly handles the boundaries, whereas iSAT3 represents these bounds as literals that require an initial evaluation. The variation in the time it takes to solve a problem may suggest differences in performance due to the approach used for integrating satisfiability checks with the theory solver. A more detailed examination of memory usage results provides additional evidence to support this hypothesis. Both iSAT3 and dReal assess the boundaries as boolean constraints, enabling them to terminate promptly since these boundaries directly contradict the target. In contrast, iSAT2 must first initialize all the necessary variables and then assess the boundaries before it can determine the satisfiability of the target.

Table 5.1: Solver time of iSAT2 applied to *property*1 of the network on the MNIST dataset using the sigmoid function (in seconds)

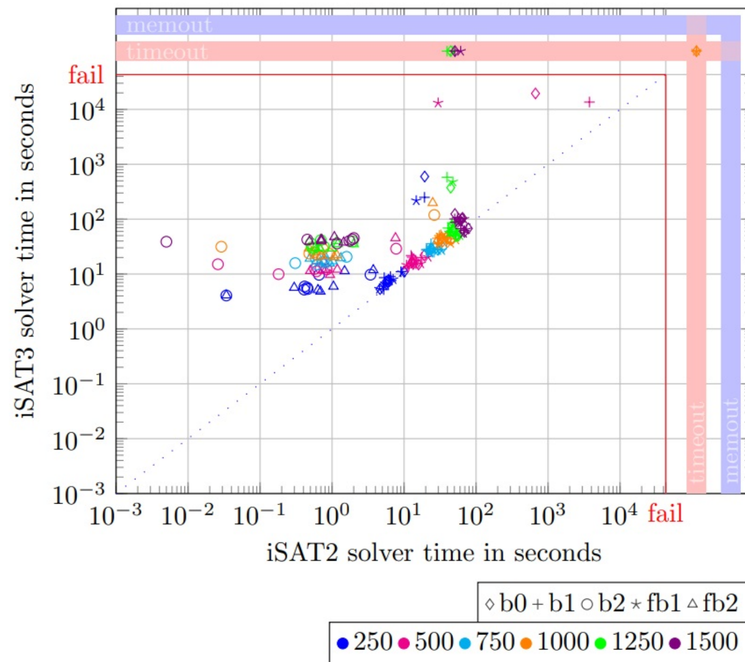
Nodes	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	78.70	81.25	0.63	79.10	0.59
250	2	90.82	82.55	0.81	80.20	0.60
250	3	89.62	88.68	0.83	87.93	0.77
250	4	82.67	82.85	1.35	80.87	1.12
250	5	81.00	85.83	2.90	89.00	1.60
250	6	78.70	87.00	3.73	91.50	1.87
500	1	202.97	182.25	0.92	181.62	0.90
500	2	203.25	182.47	0.93	183.75	0.23
500	3	204.32	183.87	1.22	184.77	1.12
500	4	205.35	183.78	1.87	181.78	1.55
500	5	215.33	186.78	3.87	183.90	2.03
500	6	223.00	194.03	3.73	191.03	2.95
750	1	301.45	300.25	1.83	289.52	0.91
750	2	350.00	350.63	1.00	349.63	0.95
750	3	354.40	369.52	1.75	340.35	1.05
750	4	354.07	349.85	2.97	349.72	1.90
750	5	367.58	368.68	4.40	350.42	3.17
750	6	390.25	481.58	4.50	369.85	3.33
1000	1	504.08	500.82	1.35	489.78	0.99
1000	2	512.72	500.08	1.45	480.68	1.02
1000	3	517.38	500.40	1.90	491.97	1.12
1000	4	519.60	511.32	1.95	503.68	1.14
1000	5	522.07	525.32	8.07	506.93	5.27
1000	6	556.53	568.98	10.53	512.72	6.83
1250	1	721.23	711.88	1.65	659.92	1.17
1250	2	724.32	713.17	2.20	649.35	1.20
1250	3	727.10	719.17	3.18	638.23	2.08
1250	4	730.33	715.30	3.37	690.35	2.27
1250	5	741.17	723.65	4.02	704.98	3.08
1250	6	762.50	728.82	12.83	705.03	8.35
1500	1	1019.20	1000.53	3.37	901.17	1.87
1500	2	1020.63	1011.90	4.32	1004.27	2.03
1500	3	1023.77	1014.27	4.58	1006.42	2.37
1500	4	1062.33	1016.07	5.23	1010.73	3.10
1500	5	1046.00	1025.82	7.70	1011.10	8.50
1500	6	1388.05	1132.93	12.43	1001.25	9.52

Table 5.2: Solver time of iSAT3 applied to *property*1 of the network on the MNIST dataset using the sigmoid function (in seconds)

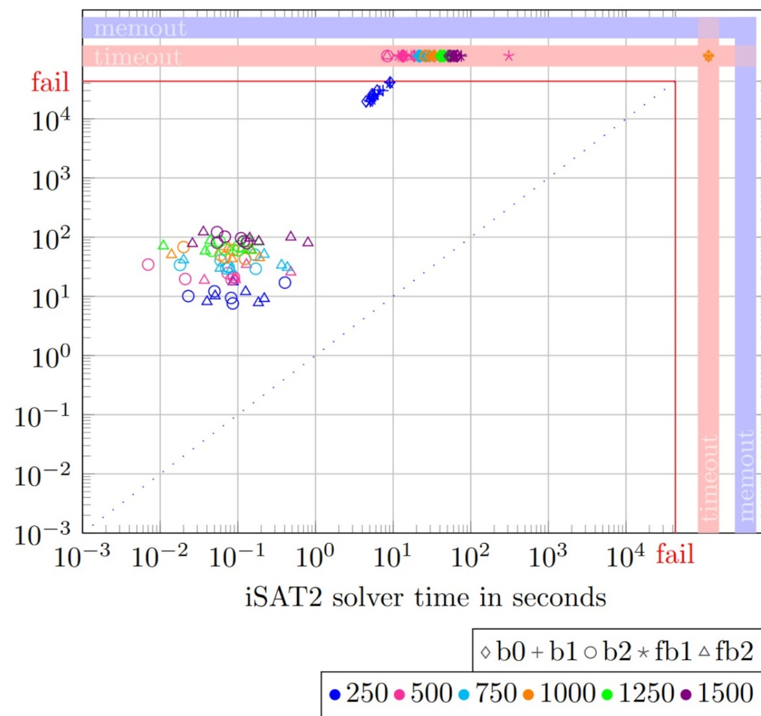
Nodes	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	128.38	124.68	76.82	118.95	66.92
250	2	129.10	126.17	78.70	119.97	67.48
250	3	129.78	126.93	85.03	126.15	68.40
250	4	134.25	127.27	85.22	91.20	69.13
250	5	137.58	129.50	86.73	94.37	72.42
250	6	148.50	144.67	89.60	100.17	82.63
500	1	237.92	220.00	168.12	211.87	142.58
500	2	240.48	239.60	170.38	217.88	143.13
500	3	261.40	246.20	189.05	232.07	150.52
500	4	268.65	246.83	190.38	235.18	161.07
500	5	276.28	251.65	192.08	245.53	171.35
500	6	290.63	262.57	196.25	248.70	183.22
750	1	451.60	440.75	260.78	438.42	199.68
750	2	452.67	443.95	261.75	440.67	200.40
750	3	467.70	456.43	265.73	446.82	204.33
750	4	471.78	458.63	281.68	450.20	217.62
750	5	474.65	462.78	285.67	452.92	223.35
750	6	497.70	466.57	291.95	458.38	229.40
1000	1	665.13	644.82	352.52	635.08	343.33
1000	2	669.77	645.47	353.30	636.52	345.42
1000	3	691.87	673.97	366.03	678.73	350.17
1000	4	693.00	683.75	369.78	679.13	356.38
1000	5	701.95	690.54	371.70	681.13	358.77
1000	6	732.63	694.80	386.35	684.75	372.67
1250	1	796.05	783.18	508.27	784.50	488.78
1250	2	839.33	825.55	510.05	818.42	490.83
1250	3	1055.58	1024.35	555.95	1014.57	531.25
1250	4	1064.10	1045.48	570.62	1037.30	549.25
1250	5	1085.75	1069.78	583.55	1052.92	554.78
1250	6	Time-out	1197.22	591.42	1135.92	562.77
1500	1	899.60	890.88	676.10	882.90	624.48
1500	2	1008.15	1000.98	682.35	891.53	635.90
1500	3	1568.17	1513.78	722.70	1485.20	699.60
1500	4	1574.07	1562.32	731.13	1520.57	701.47
1500	5	1591.43	1582.47	746.10	1569.82	711.94
1500	6	Time-out	Time-out	774.90	Time-out	732.48

Table 5.3: Solver time of dReal applied to *property*1 of the network on the MNIST dataset using the sigmoid function (in seconds)

Nodes	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	326722.27	326115.17	168.23	326023.10	129.92
250	2	354753.28	354705.13	169.15	354635.53	130.7
250	3	400887.75	400533.82	202.23	400481.15	169.2
250	4	4326906.6	4326332.82	250.37	4326298.83	235.92
250	5	4989821.6	4989652.13	264.1	4989608.03	243.47
250	6	6856343.42	6856257.38	272.42	6856212.37	250.92
500	1	Time-out	Time-out	Time-out	Time-out	Time-out
500	2	Time-out	Time-out	Time-out	Time-out	Time-out
500	3	Time-out	Time-out	Time-out	Time-out	Time-out
500	4	Time-out	Time-out	Time-out	Time-out	Time-out
500	5	Time-out	Time-out	Time-out	Time-out	Time-out
500	6	Time-out	Time-out	Time-out	Time-out	Time-out
750	1	Time-out	Time-out	Time-out	Time-out	Time-out
750	2	Time-out	Time-out	Time-out	Time-out	Time-out
750	3	Time-out	Time-out	Time-out	Time-out	Time-out
750	4	Time-out	Time-out	Time-out	Time-out	Time-out
750	5	Time-out	Time-out	Time-out	Time-out	Time-out
750	6	Time-out	Time-out	Time-out	Time-out	Time-out
1000	1	Time-out	Time-out	Time-out	Time-out	Time-out
1000	2	Time-out	Time-out	Time-out	Time-out	Time-out
1000	3	Time-out	Time-out	Time-out	Time-out	Time-out
1000	4	Time-out	Time-out	Time-out	Time-out	Time-out
1000	5	Time-out	Time-out	Time-out	Time-out	Time-out
1000	6	Time-out	Time-out	Time-out	Time-out	Time-out
1250	1	Time-out	Time-out	Time-out	Time-out	Time-out
1250	2	Time-out	Time-out	Time-out	Time-out	Time-out
1250	3	Time-out	Time-out	Time-out	Time-out	Time-out
1250	4	Time-out	Time-out	Time-out	Time-out	Time-out
1250	5	Time-out	Time-out	Time-out	Time-out	Time-out
1250	6	Time-out	Time-out	Time-out	Time-out	Time-out
1500	1	Time-out	Time-out	Time-out	Time-out	Time-out
1500	2	Time-out	Time-out	Time-out	Time-out	Time-out
1500	3	Time-out	Time-out	Time-out	Time-out	Time-out
1500	4	Time-out	Time-out	Time-out	Time-out	Time-out
1500	5	Time-out	Time-out	Time-out	Time-out	Time-out
1500	6	Time-out	Time-out	Time-out	Time-out	Time-out



(a) iSAT2 compared to iSAT3



(b) iSAT2 compared to dReal

Figure 5.2: Comparison of the solver time of iSAT2, iSAT3, and dReal4 for the verification process of *property*1 of networks using the sigmoid function.
Red lines indicate a 12-hour time limit, and the blue dotted line shows where solver times converge. A logarithmic scale on both axes enhances the visibility of swift resolutions.

Table 5.4: Percentage of aborted verification runs for networks using the sigmoid function grouped by the solver and optimization level for *property*1.

Solver	b0	b1	b2	fb1	fb2
iSAT2	0%	0%	0%	0%	0%
iSAT3	5.56%	2.78%	0%	2.78%	0%
dReal4	83.33%	83.33%	83.33%	83.33%	83.33%

Table 5.5: The highest number of layers for which each solver successfully obtained results while assessing *property*1 employing the sigmoid function.

Nodes	iSAT2	iSAT3	dReal4
250	6	6	6
500	6	6	T
750	6	6	T
1000	6	6	T
1250	6	6	T
1500	6	6	T

Table 5.4 provides a detailed breakdown of the percentage of aborted verification runs for neural networks utilizing the sigmoid function, drawing from the results presented in Tables 5.1, 5.2, and 5.3. The data is organized by solver type and optimization level, offering insights into the robustness of each solver under various scenarios. Notably, iSAT2 exhibits remarkable stability with minimal aborted runs across all optimization levels.

Additionally, Table 5.5 depicts the maximum number of layers for which each solver successfully evaluated the characteristics of networks using the sigmoid function, along with explanations for why additional layers could not be analyzed. In this context, T signifies that the solving process was halted due to a timeout.

Hyperbolic Tangent performance

To evaluate the performance of iSAT2,iSAT3, and dReal on networks utilizing the Tanh activation function, a standardized function encoding was applied to all converted networks. Tables 5.6, 5.7, and 5.8 and Figure 5.3 provide an extensive comparison of solver times for *property*1, emphasizing networks employing the Tanh activation function with iSAT2, iSAT3, and dReal solvers. The results from the defined networks reveal that iSAT2 successfully verified all five networks across the five optimization levels ($b0$, $b1$, $b2$, $fb1$, $fb2$), experiencing timeouts only in networks with 5 and 6 layers in b0, b1, and fb1. Conversely, iSAT3 exhibited timeouts in networks with similar configurations, albeit fewer than iSAT2. Additionally, iSAT2 demonstrated shorter solver times in optimization levels b and $fb2$ compared to iSAT3. A comparison of solver times among iSAT2, iSAT3, and dReal indicates that iSAT2 and iSAT3 perform better overall, while dReal encountered

Table 5.6: Solver time of iSAT2 applied to *property*1 of the network on the MNIST dataset using the Tanh activation function (in seconds)

Nodes	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	178.62	176.85	0.43	174.50	0.35
250	2	180.95	177.57	1.53	176.00	0.75
250	3	182.63	179.03	1.65	199.48	0.80
250	4	2556.77	22843.40	1.68	21955.93	1.30
250	5	Time-out	Time-out	1.72	Time-out	1.51
250	6	Timeout	Time-out	2.55	Time-out	1.82
500	1	335.60	340.99	1.00	317.47	0.47
500	2	351.30	348.52	1.37	357.67	0.82
500	3	60003.65	54629.72	1.58	54510.07	1.20
500	4	Time-out	Time-out	1.62	Time-out	1.52
500	5	Time-out	Time-out	2.23	Time-out	2.01
500	6	Time-out	Time-out	3.10	Time-out	2.88
750	1	615.88	568.07	1.47	646.92	1.18
750	2	668.50	660.80	2.17	648.37	1.92
750	3	Time-out	Time-out	2.32	Time-out	1.80
750	4	Time-out	Time-out	2.43	Time-out	2.38
750	5	Time-out	Time-out	2.63	Time-out	2.51
750	6	Time-out	Time-out	10.00	Time-out	2.95
1000	1	893.53	879.18	1.80	871.35	1.32
1000	2	1376.10	1355.25	2.52	1349.20	1.98
1000	3	Time-out	Time-out	3.41	Time-out	2.58
1000	4	Time-out	Time-out	4.77	Time-out	3.81
1000	5	Time-out	Time-out	6.05	Time-out	4.72
1000	6	Time-out	Time-out	10.33	Time-out	5.98
1250	1	1145.27	1181.13	1.98	1089.05	1.58
1250	2	2762.18	2594.58	2.80	2017.52	1.97
1250	3	Time-out	Time-out	5.50	Time-out	3.65
1250	4	Time-out	Time-out	6.32	Time-out	3.85
1250	5	Time-out	Time-out	6.57	Time-out	4.33
1250	6	Time-out	Time-out	10.97	Time-out	6.72
1500	1	1351.62	1368.48	2.00	1301.95	1.83
1500	2	3105.12	2200.23	3.02	2107.57	2.50
1500	3	Time-out	Time-out	7.43	Time-out	5.95
1500	4	Time-out	Time-out	10.10	Time-out	8.53
1500	5	Time-out	Time-out	13.90	Time-out	12.17
1500	6	Time-out	Time-out	16.60	Time-out	14.67

Table 5.7: Solver time of iSAT3 applied to *property*1 of the network on the MNIST dataset using the Tanh activation function (in seconds)

Nodes	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	107.62	94.80	80.22	90.65	73.80
250	2	113.00	112.20	82.03	110.37	74.12
250	3	123.62	121.75	84.48	115.60	77.10
250	4	125.18	122.27	85.02	116.18	79.60
250	5	129.54	124.80	92.90	119.30	87.07
250	6	Time-out	Time-out	143.60	Time-out	106.12
500	1	252.73	248.97	190.45	241.23	172.13
500	2	256.77	249.42	200.78	242.17	178.32
500	3	258.22	250.35	202.93	248.08	185.32
500	4	263.65	259.33	208.85	253.67	186.35
500	5	319.70	298.95	225.21	288.93	189.10
500	6	Time-out	Time-out	231.70	Time-out	193.32
750	1	441.68	438.78	267.47	430.13	260.72
750	2	454.28	448.50	279.88	439.17	269.48
750	3	481.18	472.15	300.03	463.32	292.43
750	4	487.25	481.38	300.27	474.33	293.93
750	5	544.15	531.31	330.93	520.12	319.48
750	6	Time-out	Time-out	347.70	Time-out	336.72
1000	1	632.37	626.93	317.33	621.27	304.13
1000	2	640.30	636.93	365.42	625.48	340.68
1000	3	668.78	661.52	369.42	655.22	336.03
1000	4	687.88	682.67	417.40	671.87	391.08
1000	5	Time-out	Time-out	450.05	Time-out	431.12
1000	6	Time-out	Time-out	458.22	Time-out	440.82
1250	1	807.25	803.97	518.80	797.18	537.07
1250	2	849.50	845.85	519.10	839.42	559.62
1250	3	1192.80	1188.40	569.60	1179.62	575.43
1250	4	1213.32	1205.63	588.83	1189.53	587.43
1250	5	Time-out	Time-out	602.17	Time-out	620.78
1250	6	Time-out	Time-out	605.32	Time-out	629.18
1500	1	907.60	896.92	633.73	870.17	621.27
1500	2	1038.23	1020.17	648.62	929.18	632.27
1500	3	1746.80	1495.35	675.08	1038.23	665.40
1500	4	1746.80	1703.57	676.20	1641.72	664.10
1500	5	Time-out	Time-out	788.62	Time-out	771.58
1500	6	Time-out	Time-out	793.12	Time-out	770.12

Table 5.8: Solver time of dReal for *Property* 1 of networks (in second)

Neurons	Layers	b0 (s)	b1 (s)	b2 (s)	fb1 (s)	fb2 (s)
250	1	40705.9	41277.44	293.93	39843.65	295.53
250	2	30467.75	30038.56	216.88	29407.6	199.97
250	3	26024.89	25987.39	177.7	25500.39	178.05
250	4	24140.65	24151.99	169.83	22993.99	171.48
250	5	21383.56	21356.44	154.57	20954.2	133.12
250	6	19689.56	19669.43	136.97	19318.6	128.12
500	1	Time-out	Time-out	Time-out	Time-out	Time-out
500	2	Time-out	Time-out	Time-out	Time-out	Time-out
500	3	Time-out	Time-out	Time-out	Time-out	Time-out
500	4	Time-out	Time-out	Time-out	Time-out	Time-out
500	5	Time-out	Time-out	Time-out	Time-out	Time-out
500	6	Time-out	Time-out	Time-out	Time-out	Time-out
750	1	Time-out	Time-out	Time-out	Time-out	Time-out
750	2	Time-out	Time-out	Time-out	Time-out	Time-out
750	3	Time-out	Time-out	Time-out	Time-out	Time-out
750	4	Time-out	Time-out	Time-out	Time-out	Time-out
750	5	Time-out	Time-out	Time-out	Time-out	Time-out
750	6	Time-out	Time-out	Time-out	Time-out	Time-out
1000	1	Time-out	Time-out	Time-out	Time-out	Time-out
1000	2	Time-out	Time-out	Time-out	Time-out	Time-out
1000	3	Time-out	Time-out	Time-out	Time-out	Time-out
1000	4	Time-out	Time-out	Time-out	Time-out	Time-out
1000	5	Time-out	Time-out	Time-out	Time-out	Time-out
1000	6	Time-out	Time-out	Time-out	Time-out	Time-out
1250	1	Time-out	Time-out	Time-out	Time-out	Time-out
1250	2	Time-out	Time-out	Time-out	Time-out	Time-out
1250	3	Time-out	Time-out	Time-out	Time-out	Time-out
1250	4	Time-out	Time-out	Time-out	Time-out	Time-out
1250	5	Time-out	Time-out	Time-out	Time-out	Time-out
1250	6	Time-out	Time-out	Time-out	Time-out	Time-out
1500	1	Time-out	Time-out	Time-out	Time-out	Time-out
1500	2	Time-out	Time-out	Time-out	Time-out	Time-out
1500	3	Time-out	Time-out	Time-out	Time-out	Time-out
1500	4	Time-out	Time-out	Time-out	Time-out	Time-out
1500	5	Time-out	Time-out	Time-out	Time-out	Time-out
1500	6	Time-out	Time-out	Time-out	Time-out	Time-out

timeouts for networks with more than 250 nodes.

Table 5.9 offers a comprehensive breakdown of the percentage of aborted verification runs for neural networks employing the Tanh function, sourced from Tables 5.6, 5.7, and 5.8. This data is structured by solver type and optimization level, providing valuable insights into the robustness of each solver across different scenarios. Notably, iSAT2 demonstrates remarkable stability with minimal aborted runs observed across all optimization levels.

Furthermore, Table 5.10 presents the maximum number of layers for which each solver successfully evaluated the characteristics of networks using the Tanh function, accompanied by explanations for why additional layers could not be analyzed. Here, instances marked with T denote halts in the solving process due to timeouts.

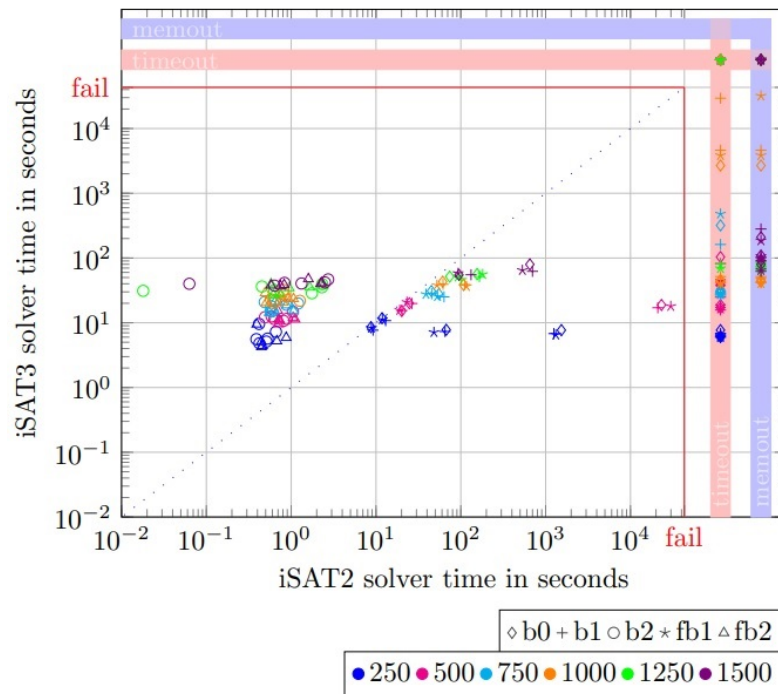
Table 5.9: Percentage of aborted verification runs of the $property1$ for networks using the hyperbolic tangent function grouped by solver and optimization level

Solver	b0	b1	b2	fb1	fb2
iSAT2	58.33%	58.33%	0%	58.33%	0%
iSAT3	25%	25%	0%	25%	0%
dReal4	83.33%	83.33%	83.33%	83.33%	83.33%

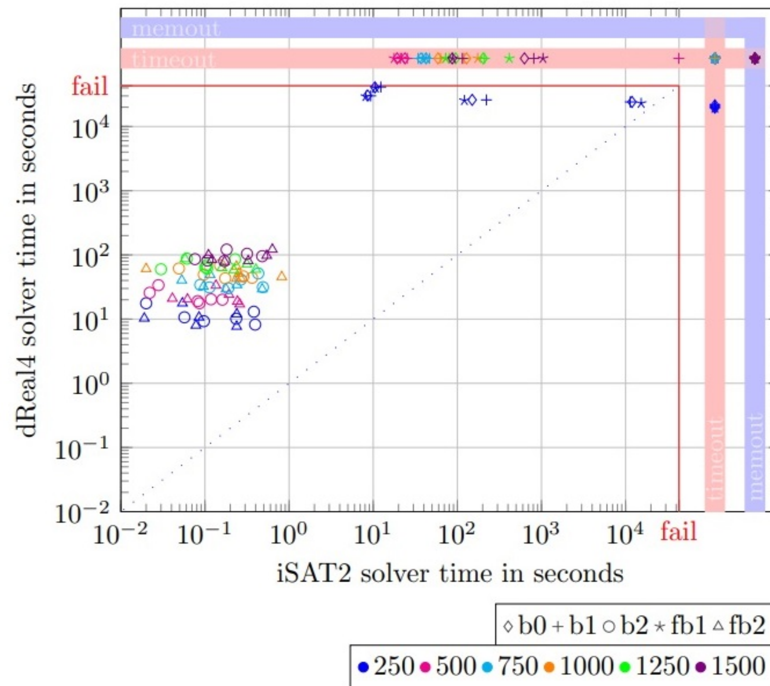
Table 5.10: The highest count of network layers that each solver successfully addressed for $property1$ using the hyperbolic tangent function. T signifies the termination of the solver process due to a timeout.

	iSAT2	**iSAT3**	**dReal4**
250	6	6	6
500	6	6	T
750	6	6	T
1000	6	6	T
1250	6	6	T
1500	6	6	T

The optimization levels have different effects on the solvers. For iSAT2 and iSAT3, the basic optimization levels ($b0$, $b1$, and $fb1$) do not make much difference, as they have similar percentages of aborted verification runs. However, the fine-grained optimization levels ($b2$ and $fb2$) reduce the percentage of aborted verification runs for iSAT2 and iSAT3. This suggests that the fine-grained optimization levels are more suitable for iSAT2 and iSAT3. For dReal4, the optimization levels do not seem to matter, as it has the same percentage of aborted verification runs for all levels when the number of nodes in the networks is bigger than 250.



(a) iSAT2 compared to iSAT3



(b) iSAT2 compared to dReal

Figure 5.3: Comparison of the solver time of iSAT2, iSAT3, and dReal4 for the verification process of *property*1 of networks using the Tanh function.
Red lines indicate a 12-hour time limit, and the blue dotted line shows where solver times converge. A logarithmic scale on both axes enhances the visibility of swift resolutions.

Table 5.11: NGSIM database [Adm07]

V_ID	F_ID	Local_X	Local_Y	Length	Width	Vel	Acc	Lane_ID	Leading
288	1570	50.366	267.751	14	6.5	18.74	9.79	4	292
288	1757	51.159	314.237	14	6.5	0	-1.78	4	292
288	1948	24.405	699.007	14.5	7.4	51.31	11.25	1	291
288	1911	52.399	313.651	14	6.5	0.04	0	4	292
289	2086	19.455	2154.879	15.5	5.9	2	50	-5.33	300
289	903	41.523	1045.145	17	8.4	5	2.89	3	291

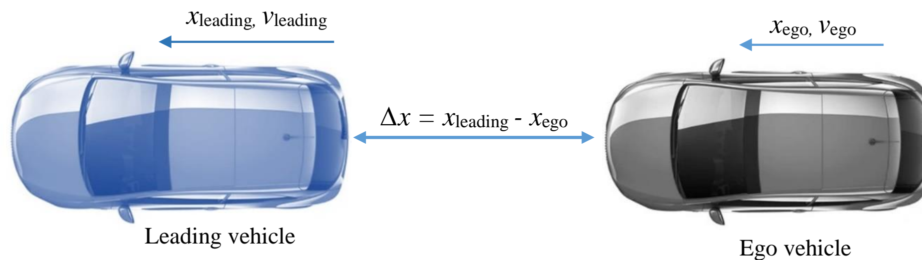


Figure 5.4: Visualization of the distance between cars

5.2 Benchmark 2: NGSIM

In the second benchmark of our evaluation to assess the effectiveness of our proposed verification method for neural networks, with a particular emphasis on recurrent neural networks, we carried out experiments on a trained LSTM model [MFF23]. This LSTM network was trained using a dataset derived from the NGSIM project, which recorded traffic scenarios to learn an auto-braking maneuver. The NGSIM dataset contains comprehensive information about the positions, speeds, accelerations, and lanes utilized by cars traveling on US Highway 101 between 7:50 am and 8:35 am [Adm07]. The study area covers a length of 640 meters and is comprised of five lanes. A portion of this dataset is illustrated in Table 5.11. Each entry corresponds to a specific vehicle, and they are uniquely identified by a Vehicle_ID number. This identifier allows individual cars to be tracked over an extended period by filtering rows with the respective identifier, thereby generating a time series of movements for each Vehicle_ID.

To address the classification task of predicting near collisions, a classifier network can be trained on this dataset. Since the data represents a time series of state snapshots and involves recovering dynamics by correlating these snapshots, an LSTM model was selected for this classification task. LSTM is well-suited for capturing dependencies and patterns

over time, making it a suitable choice to analyze the dynamic movements of vehicles and predict potential near collisions based on the time series of state snapshots. The primary proof obligation, which was handled by iSAT, involved verifying whether the trained network accurately detects near collisions.

The visualization depicted in Figure 5.4 illustrates the spacing between vehicles. According to the Vienna Convention, the safe distance between two vehicles is defined as a "sufficient distance [...] to avoid a collision if the vehicle in front should suddenly slow down or stop" [ECfE]. This safety distance is crucial in ensuring overall safety and preventing potential accidents on the road. The trained network's ability to identify and predict near collisions aligns with the concept of maintaining a safe distance between vehicles to reduce the risk of collisions. Indeed, maintaining a safe distance between the ego vehicle and the leading vehicle is of paramount importance. This safe distance should be large enough to allow for a prompt and safe avoidance of collision, especially in the event of an emergency deceleration by the leading vehicle. By ensuring an adequate following distance, drivers can have sufficient time and space to react to sudden changes in the traffic flow, preventing potential collisions and promoting overall road safety. Adhering to the concept of a safe distance plays a crucial role in mitigating risks and enhancing the safety of road users.

Two vehicles, initially separated by a distance Δx, could potentially collide if the leading vehicle, referred to as $vehicle_{leading}$, suddenly applies brakes or comes to a complete stop. To prevent such collisions, it is imperative to ensure that the distance between $vehicle_{ego}$ and $vehicle_{leading}$ is always kept sufficiently large to allow for sudden and maximum deceleration.

To establish a corresponding distance requirement, one can leverage principles from Newtonian mechanics, consider the physical characteristics of the vehicles, and account for human reaction times. By incorporating these factors, it becomes feasible to calculate the necessary safe distance, thereby enabling the formulation of a comprehensive safety specification that serves as the foundation for formal verification.

The future position of a vehicle at any given time $t \geq 0$ is governed by the laws of accelerated movement, as described by equations of motion. Equation 5.2 represents the position of a vehicle at any time $t \geq 0$, where $x(t)$ denotes the position of the vehicle at time t, x_0 is the initial position, v is the initial velocity, a is the acceleration, and t is time.

Utilizing these principles, it is possible to determine the vehicle's trajectory over time, facilitating the computation of the required distance that ensures a secure following between $vehicle_{ego}$ and $vehicle_{leading}$. This safety distance guarantees that $vehicle_{ego}$ can respond effectively to any sudden deceleration by $vehicle_{leading}$, effectively reducing the risk of potential collisions.

By adhering to the prescribed safe distance and conducting thorough formal verification, road safety can be reinforced, and the likelihood of accidents resulting from abrupt changes in traffic conditions can be mitigated.

$$x(t) = x_0 + vt + \frac{1}{2}at^2 \tag{5.2}$$

A collision between the ego vehicle and the leading vehicle occurs when their positions are equal at some future time $t \geq 0$. Mathematically, a collision is possible if the following condition holds:

$$\exists t \geq 0 : x_{ego(t)} = x_{leading(t)} \tag{5.3}$$

Based on equations (5.2) and (5.3), a potential for an unavoidable collision exists if the following constraint system is satisfiable:

$$v_{ego}t + \frac{1}{2}at^2 = \Delta x + v_{leading}t + \frac{1}{2}at^2, \quad 0 \leq t \leq \frac{v_{ego}}{a} \tag{5.4}$$

In equation (5.4), the variable Δx represents the initial distance between the ego vehicle and the leading vehicle. Here, v_{ego} and $v_{leading}$ represent the velocities of the ego vehicle and the leading vehicle, respectively, while a corresponds to the acceleration. The satisfaction of this constraint system indicates the possibility of an unavoidable collision scenario within the specified time frame. The maximum deceleration a (which is assumed to be the same for both vehicles for simplicity) plays a crucial role in determining the braking demand. To establish a physically justified braking requirement, we start with equation (5.4), where the identical expression$\frac{1}{2}at^2$ occurs on both sides. By removing this common term and simplifying, we derive equation (5.5), which specifies whether braking is necessary:

$$\text{statebrake} \iff 0 \leq \Delta x \cdot a \leq v_{ego} \cdot (v_{ego} - v_{leading}) \tag{5.5}$$

In this context, a represents the maximum deceleration of the ego vehicle. The implication in equation (5.5) indicates that the ego vehicle may encroach upon the safety envelope of the leading vehicle or even collide with it if the right-hand side of the implication becomes true. Therefore, activating the brakes upon a positive edge of *statebrake* will keep us outside or at the surface of the risk region. This is because the lead car will decelerate with a maximum deceleration of a, ensuring a safe distance between the two vehicles. By adhering to this requirement, the ego vehicle ensures that it can respond appropriately to

changes in the leading vehicle's motion, thereby preventing potential collisions and maintaining road safety.

As a result, equation (5.5) offers the precise definition for the braking advisories, which serves a dual purpose in supervised learning. It functions as labels for training cases and as a foundation for verification. It is crucial to note that despite using the same predicate as a label in supervised learning, the verification process remains essential. This is because neural networks are in general not capable of achieving a loss (i.e., error rate) of 0 across all training points, and even if they were, their generalization between training points — i.e., to almost all instances to be expected in the latter application — remains unpredictable. Hence, the verification step is necessary to ensure the neural network's adherence to the specified braking advisories and to verify its ability to maintain safe distances between vehicles effectively and unfallibly. By combining supervised learning and formal verification, a comprehensive approach is adopted, addressing complexities and uncertainties while reinforcing safety requirements and enhancing the reliability of the collision avoidance system.

Our classification model consisted of 50 LSTM nodes, 4-time steps, and 3 fully connected layers. During training, we assigned safe and unsafe labels to NGSIM data using equation (5.5), creating a supervised learning problem. The features frame_ID, lane_ID, velocity, acceleration, and positions of the ego and leading vehicles were extracted from 1,048,576 records of the NGSIM database, which we divided into training and test datasets. Remarkably, the training process achieved an accuracy of approximately 98% across the test dataset. For training a neural network to provide emergency braking advisories, we conducted experiments using the following computer configuration: Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz, 64 GB Memory, and Ubuntu OS.

After training, we converted the LSTM network into iSAT constraint format [FHT+07, isa10] using the translation rules presented in Chapter 4. The resulting iSAT constraint formula, denoted as ϕ, effectively captured the behavior of the full LSTM network and served as a representation of its functionality. The LSTM network possesses two float-valued outputs, denoted as γ_{unsafe} and γ_{safe}, which determine the classification based on the stronger evidence between them. Specifically, when $\gamma_{unsafe} > \gamma_{safe}$ holds true, the network detects a critical condition, thereby generating a braking advisory.

The verification process aims to demonstrate that the LSTM outputs, as defined by the LSTM's structure and weights, or equivalently by its logical encoding ϕ, consistently align with the output required by equation (5.5). We can recover the LSTM's advisory from its logical encoding ϕ using the following expression:

$$state_{nn_brake} \iff \phi \land (\gamma_{safe} < \gamma_{unsafe}) \tag{5.6}$$

The functional verification of the LSTM involves ensuring that equations (5.5) and (5.6) always remain consistent, irrespective of the actual input to the network. In other words, the LSTM should never, under any input sequence, reach a state where its output $state_{nn_brake}$, defined by constraint system equation (5.6), differs from the expected label $state_{brake}$ derived from equation (5.5). This verification process ensures the reliability and accuracy of the LSTM in generating appropriate braking advisories under various conditions. Of particular interest are situations where, as per the requirement defined in formula (5.5), an emergency braking maneuver is required, but the neural network based on condition in equation (5.6) fails to issue an emergency braking advisory. This verification obligation aims to investigate whether the neural network could potentially fail to provide the required braking advice, i.e., whether the condition in equation (5.6) may yield a no-braking advisory when the condition in equation (5.5) determines emergency braking as necessary.

To examine this scenario, we introduce the verification target in equation (5.7), which allows us to employ iSAT to check the satisfiability of this condition. If iSAT2 finds a satisfying assignment, it implies the presence of a counterexample to the safety of the LSTM, thus highlighting situations where the neural network will not generate the necessary emergency braking advisory despite the situation requiring it. This verification target is represented by the following iSAT expression for iSAT's bounded model-checking mode:

$$\texttt{Target} : state_{brake} \wedge \neg state_{nn_brake} \tag{5.7}$$

When iSAT is tasked with solving the conjunctively combined system of equations (5.5) and (5.6), along with the reachability target equation (5.7), it provides a candidate solution as a result, highlighting a trace leading to satisfaction of the target. The fourth time step input data that iSAT gives us as a candidate solution as shown in Table 5.12. In this particular case, when the position of the ego vehicle is at 148 m with a speed of 22 m/s, and simultaneously, the leading vehicle is at 155 m with a speed of 1 m/s, it becomes evident that emergency braking would be necessary. However, the trained LSTM makes an erroneous generalization from its training points, incorrectly reporting $\neg state_{nn_brake}$. This is a highly critical issue that needs to be addressed.

This example underscores the importance of formal verification in identifying potential shortcomings or misgeneralizations of the LSTM neural network. By revealing such discrepancies, the verification process plays a vital role in ensuring the reliability and safety of the collision avoidance system.

In the converse case where we examined the possibility of the LSTM providing a braking advisory when it should not be necessary (verification target: $\texttt{Target} : \neg state_{brake} \wedge$

Table 5.12: The result of verifying the NGSIM case study

Target	p_{ego}	v_{ego}	$p_{leading}$	$v_{leading}$	$P_{time-second}$	State
$state_{brake}$ and $!state_{nn_brake}$	148	22	155	1	120.02	candidate solution
$!state_{brake}$ and $state_{nn_brake}$	-	-	-	-	200.41	Unsatisfiable

$state_{nn_brake}$), iSAT's analysis did not yield any satisfying instance. This result conclusively demonstrates that the LSTM is robust in avoiding the issuance of a braking advisory when it is not warranted. The absence of any satisfying instance provides strong evidence that the LSTM's decision-making process is reliable and effective in correctly discerning situations that do not require emergency braking. As a result, we can be confident that the LSTM model is capable of avoiding unnecessary intervention and minimizing the occurrence of false alarms. However, it's important to note that this ability to avoid false alarms comes at the cost of potentially generating false negatives, as demonstrated earlier. This highlights a critical consideration, indicating that the standard training process may be grossly inappropriate for LSTMs in safety-critical applications. In such contexts, false negatives are deemed safety-critical and unacceptable, whereas false positives only impact performance.

5.3 Benchmark 3: Satellite Collision Detection

We explore the scalability of our proposed method through a comprehensive investigation in the third case study, which revolves around gathering crucial information about satellites and objects orbiting in space [MFF23]. In the present scenario, *Low Earth orbit* (LEO) satellites are frequently responsible for emitting numerous hazard alerts each week, indicating close encounters with other space objects, such as satellites or space debris. The *European Space Agency* (ESA) estimated in January 2019 that our planet is encircled by over 34,000 objects larger than 10 cm, out of which 22,300 are actively monitored. The locations of these objects are diligently recorded in a shared global database [Eur]. As depicted in Figure 5.5, the spatial density representation of objects in LEO orbits highlights the complex distribution and potential risks associated with space debris and satellite interactions. Preventing collisions between spacecraft has become a critical aspect of satellite operations. To achieve this, various operators rely on sophisticated and regularly updated calculations to assess collision risks between orbiting objects. Subsequently, they devise risk reduction strategies based on the obtained information. When a possible close encounter with any object is detected, the collected data is consolidated into a *Conjunction*

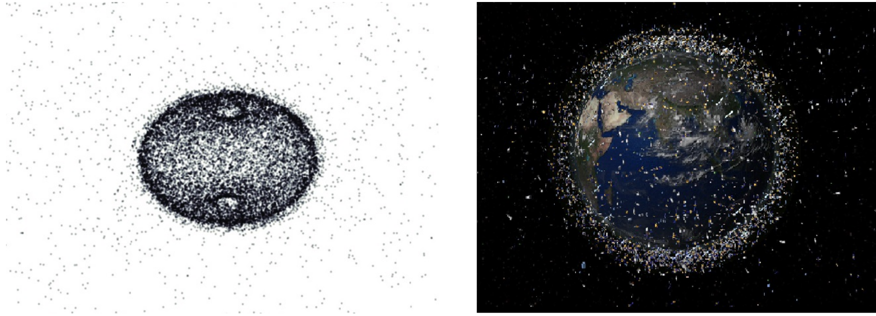


Figure 5.5: Visualization of the density of objects orbiting the low Earth orbit [CFS, Eur]

Data Message (CDM). Each CDM contains essential details about the approach, such as the *Time of Closest Approach* (TCA), the satellite's identity, the type of potential collision object, and the relative position between the chaser and target, among other elements. Additionally, the CDM includes a self-reported risk, computed using various CDM elements.

Typically, three CDMs are recorded daily for each potential near approach over the course of one week. Consequently, a time series of CDMs is established for each event, allowing for a comprehensive analysis of the data and risk assessment over time. This continuous monitoring and reporting of CDMs facilitate effective collision avoidance strategies and contribute to the overall safety and stability of space operations. ESA played a significant role in providing a valuable compiled dataset containing information on close approach events in the form of conjunction data messages spanning from 2015 to 2019. This dataset served as the foundation for the spacecraft collision avoidance challenge, a machine-learning competition where participating teams were tasked with creating models to predict the likelihood of an orbiting object colliding with another [UIS+22]. In this context, an event is categorized as having a high risk if its last recorded risk value is greater than -6.

- **Property 1:** The initial property evaluated in satellite collision detection networks involves an instance from the satellite collision detection dataset depicting risk situation data. In this context, the solvers examine whether there exists a possibility for the network's output, classifying the input as risk data, to fall below a threshold of -6, serving as a counterexample. Equation 5.8 illustrates the verification target, with n_{o0} denoting the output of the neural network.

$$\texttt{Target} : n_o_0 <= -6 \tag{5.8}$$

The input remained devoid of any noise, thereby constraining the solvers to address only a single, precise value assignment of each input. In theory, the solution process should be straightforward due to the simplicity of the task. This particular $property1$

serves as a baseline, strategically chosen to demonstrate solver performance in the context of a highly uncomplicated criterion.

We are interested in ensuring that the last recorded risk value for an event *n_o_0* is equal to or less than −6, categorizing it as high risk. Satisfaction with this property implies that the LSTM-encoded network has successfully captured and classified events based on their associated risks, contributing to the overall reliability of the risk assessment system. The main objective was to check the satisfiability of the equation (5.8). To execute the verification processes, we employed Oldenburg University's high-performance computing cluster, CARL. Each process was limited to a maximum of 300 GB of RAM and 24 hours of processing time.

To facilitate scalability testing, we carefully selected seven features like *time_to_tca*, *max_risk_estimate*,*max_risk_scaling*, *mahalanobis_distance*, *miss_distance*, *c_position*, *covariance_det*, and *c_obs_used* such as the team's final score in the competition [UIS+22]. For the evaluation, our trained models were constructed with varying numbers of LSTM nodes distributed across two layers. Additionally, the last three CDM data were used as 3 time steps, enabling a comprehensive assessment of the model's performance over time. This comprehensive approach allowed us to gauge the scalability and efficiency of the trained models, providing valuable insights into their effectiveness in predicting and addressing collision risks in satellite operations.

To assess the scalability of LSTM verification using iSAT2 and iSAT3, we conducted performance measurements based on two key metrics: *Solver time* and *Memory usage*. These measurements were performed on trained LSTM networks with two different activation functions, namely tanh and sigmoid. *Solver time* refers to the CPU time in seconds that each solver (iSAT2 and iSAT3) consumed during the verification process to determine satisfiability.

Memory usage data was retrieved from the cluster's workload management system SLURM for each verification process and is reported in megabytes. It's important to note that the provided memory usage values are slightly overestimated because the workload management system can only measure the memory allocated to the process and does not provide information about the memory actually used during the verification process. In Table 5.13, we present the performance comparison between iSAT2 and its successor iSAT3. The columns Sg and Th represent the total number of LSTM nodes in each trained network using the sigmoid and tanh activation functions, respectively. The table provides insights into the efficiency and resource utilization of both solvers when verifying LSTM networks with different activation functions. The results aid in understanding the strengths and weaknesses of each solver, helping in the selection of the most appropriate solver for specific LSTM verification tasks based on computational efficiency and memory usage

Table 5.13: Memory usage and time for solving of iSAT2 and iSAT3 on satellite collision detection data-set for $property1$

Nodes	$variables$	$result$	$iSAT2$ $Memory$	$Time$ $(second)$	$result$	$iSAT3$ $Memory$	$Time$ $(second)$
Sg_{20}	12,768	UNSAT	1.14	1.160	UNSAT	1.14	0.227
Sg_{40}	42,001	UNSAT	1.14	3.152	UNSAT	1.14	0.847
Sg_{60}	88,130	UNSAT	1.14	6.891	UNSAT	1.14	1.963
Sg_{80}	151,054	UNSAT	1.14	11.835	UNSAT	1.14	5.734
Sg_{100}	230,782	UNSAT	1.13	18.412	UNSAT	1.14	6.162
Sg_{120}	327,308	UNSAT	1.14	26.327	UNSAT	1.14	9.933
Sg_{140}	440,628	UNSAT	990.70	35.544	UNSAT	1.14	12.128
Sg_{160} -	570,736	UNSAT	1,562.94	47.017	UNSAT	1.14	15.97
Sg_{180}	717,645	UNSAT	1,562.94	59.463	UNSAT	1.14	21.107
S_{200}	881,349	UNSAT	1,562.94	73.186	UNSAT	8,111.68	54.483
Sg_{400}	3,441,142	UNSAT	1,823.22	290.076	Time-out	-	-
Sg_{600}	7,676,411	UNSAT	1,823.22	669.826	Time-out	-	-
Sg_{800}	9,691,159	UNSAT	283,417.80	1,271.90	ML	-	-
Sg_{1000}	15,085,373	UNSAT	299,361.27	1,993.22	ML	-	-
Th_{20}	12,732	UNSAT	1.14	0.846	UNSAT	1.1367	0.235
Th_{40}	42,122	UNSAT	1.14	3.163	UNSAT	1.14	0.885
Th_{60}	88,311	UNSAT	1.14	6.258	UNSAT	1.14	2.166
Th_{80}	151,298	UNSAT	1.14	12.254	UNSAT	1.14	4.586
Th_{100}	231,088	UNSAT	1.13	22.246	UNSAT	1.14	7.295
Th_{120}	327,672	UNSAT	1.14	24.761	Time-out	-	-
Th_{140}	441,050	UNSAT	990.70	32.305	Time-out	-	-
Th_{160}	571,218	UNSAT	1,562.94	1,102.19	Time-out	-	-
Th_{180}	718,208	Time-out	-	-	Time-out	-	-

considerations. The terms "Time-out" and "ML" signify that the solving process was terminated due to exceeding the time or memory limit, respectively. The findings from the experiments suggest that iSAT2, which employs an embedding of Boolean reasoning into real-valued intervals and interval constraint propagation, outperforms its commercially successful successor iSAT3 in LSTM verification benchmarks for larger instances. iSAT2 achieves this by saving an SAT-modulo-theory style Boolean abstraction of (arithmetic) theory constraints using Boolean trigger variables for theory atoms. In contrast, iSAT3 relies on Boolean literal abstraction for theory atoms.

This observation raises a significant concern that warrants further investigation. The outperformance of iSAT2 over iSAT3 in larger LSTM verification problems could be due to the nature of these problems, which are characterized by an extremely large number of variables spanning a vast search space for constraint solving. For instance, example Sg_{1000} involves around 15 million real-valued variables. In such scenarios, using Boolean literal

abstraction for theory-related facts in iSAT3 might have adverse effects on the solver's efficiency.

This finding should prompt the exploration of dedicated solvers tailored specifically to LSTM verification tasks. The unique characteristics and challenges posed by LSTM verification, namely their immense variable count and constraint complexity, call for specialized approaches that can optimize the solver's performance and overcome the limitations observed in traditional SAT-modulo-theory solvers like iSAT3. Further research in this direction could lead to the development of more effective and efficient solvers for tackling LSTM verification problems.

Chapter 6

Conclusion and Further Research

In the present technological landscape, cyber-physical systems (CPS) represent a key technology integrating and enhancing physical components with computing and communication elements. In CPS, the synergy between the physical and digital domains enables real-time monitoring, control, and coordination, fostering seamless interactions between the physical world and computational processes. Acknowledged as a vital field of contemporary research, CPS are positioned to transform the design and development of future systems. Notably, the functional architecture of emerging systems, such as autonomous vehicles, increasingly incorporates components created through artificial intelligence-driven models like machine learning, interconnected through real-time networks. This departure from traditional engineering methods introduces a diverse combination of components, creating intricate interactions that present substantial challenges in their safety analysis.

Confirming and ensuring the functionality of CPS, particularly in safety-critical sectors, brings forth numerous unresolved challenges. Despite their promise, widely used computational structures like deep neural networks currently lack scalable and automated verification mechanisms. The intrinsic characteristics of artificial neural networks, such as their size, non-linearity, and non-convexity, present considerable difficulties for some existing verification methods like Mixed Integer Linear Programming solvers and Satisfiability Modulo Theories solvers.

6.1 Conclusion

Our research primarily focuses on automating the functional verification of nonlinear artificial neural networks using the SMT solver iSAT. After thoroughly reviewing the theoretical aspects related to our problem, we have structured our work into two main areas: the verification of feed-forward neural networks and the verification of LSTM neural networks, a specific type of recurrent neural network.

In addition, to improve the efficiency of verification by refining the conversion process of neural networks into iSAT expressions, we delved into optimizations. The optimization strategies we employed address key aspects such as managing numerical errors during boundary calculation and enhancing expression efficiency. This not only diminishes the search space required by iSAT but also eliminates unnecessary expressions that do not contribute to the accurate evaluation of the network's output.

Moreover, converting neural networks into alternative formats suitable for formal verification presents a formidable challenge. This difficulty arises from several factors, including the network's size, the diverse range of library types, and more. The presence of numerous distinct frameworks for neural network definition, training, and the creation of other machine learning applications, each with its unique approach to storing trained networks, adds to the complexity. Therefore, we designed a system to transform neural networks, which can be generated using virtually any framework, into representations that can be evaluated and processed by iSAT. Our approach involves utilizing ONNX, a standardized universal format, as an intermediary representation, facilitating the initial conversion from various formats.

To facilitate the implementation of the methods we have described so far, we have employed a tool that comprises supplied translators from different neural network packages to the ONNX intermediate format, along with our self-developed translations from ONNX to iSAT2, iSAT3 and dReal. This converter serves the purpose of automatically transforming trained neural networks, even those with millions of nodes and originating from different frameworks such as TensorFlow, Keras, and Pytorch, automatically into the solver expressions format.

To evaluate our method, we chose three specific benchmarks. These examples were carefully picked to show how well our approach handles different scenarios and complex systems. Each benchmark serves as a test case to assess the performance and reliability of our method. Ultimately, this demonstrates its effectiveness in analyzing and verifying both feed-forward and recurrent neural networks in practical applications.

- Verification of feed-forward In order to evaluate our proposed method in a feed-forward neural network we used MNIST, frequently employed for training image

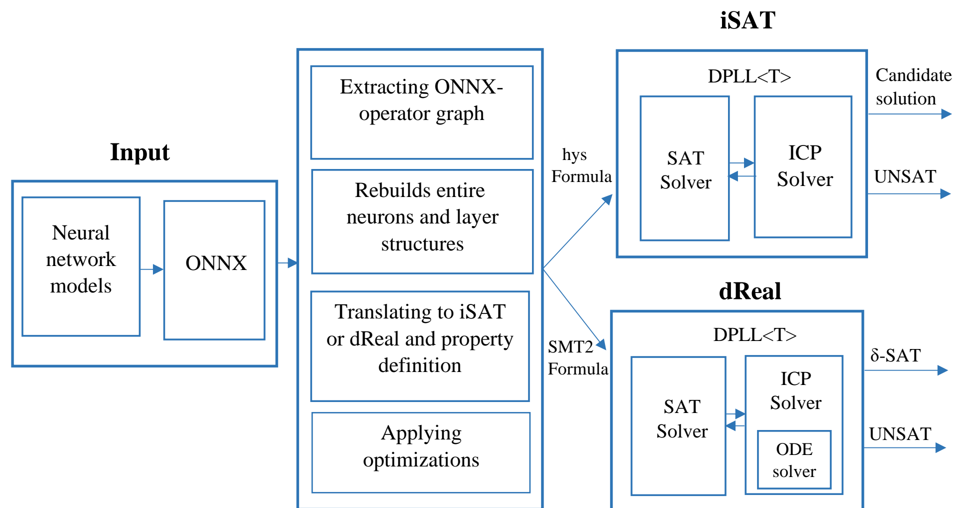


Figure 6.1: Architecture and execution of our study's methodology

recognition methods, consisting of a handwritten numbers dataset. The experiment involved evaluating the $property1$ which is the initial property assessed on the MNIST networks involved an instance from the MNIST dataset portraying the digit seven across five distinct optimization levels, $b0$, $b1$, $b2$, $fb1$, $fb2$ for a set of six networks, each characterized by varying node counts: 250, 500, 750, 1000, 1250, and 1500 employing the sigmoid and Tanh activation functions. In our comparative analysis, we utilized dReal as an alternative method to benchmark against iSAT. The objective was to evaluate the effectiveness of iSAT in the verification of feed-forward neural networks by comparing its results with those obtained using dReal. The findings revealed a notable superiority of iSAT, demonstrating higher efficacy in the verification process. The discernible difference in performance further substantiates the robustness of iSAT in handling feed-forward neural networks. Our investigation demonstrated the effectiveness of both iSAT2 and iSAT3 in solving networks that utilized the sigmoid and Tanh function. Notably, iSAT2 exhibited superior efficiency by requiring less time to solve these networks compared to iSAT3 and dReal.

- **Verification of LSTM**

  We conducted evaluations to assess the effectiveness of our proposed verification method for neural networks, with a particular focus on LSTM neural networks. Our experiments were performed on two distinct datasets: the NGSIM dataset and the satellite collision benchmarks in Chapter 5. First, a network was trained on a data set of recorded traffic from the NGSIM project to learn an auto-braking maneuver.

Second satellite collision avoidance data were used to train a risk assessment system.

In the second benchmark, the data set comprised information about the positions, speeds, accelerations, and lanes used by cars traveling on US Highway 101. We had two iSAT definitions of braking advisories. One formula encodes the physical ground truth of Newtonian mechanics' corresponding distance requirement to avoid the collision, while the other provides the LSTM behavior for giving emergency braking advisories. We were particularly interested in situations where emergency braking is necessary but the neural networks fail to provide the required advisory. These scenarios highlight unsafe conditions where the neural network lacks the capability to offer essential braking advice. By providing that condition as a verification target, iSAT could be successful in checking for the satisfiability of this condition.

We further investigated the scalability of verifying LSTM neural networks using our implemented method in the third case study, which offers insights into satellites and objects orbiting in space. To compare the scalability of LSTM verification using iSAT2 and iSAT3, we measured the performance of each solver with respect to the solver time and also memory usage with tanh and sigmoid activation functions. However, it is noteworthy that, based on these results, we made a strategic decision not to employ dReal for the evaluation of long short-term memory networks. The rationale behind this decision lies in the observed substantial performance gap, where iSAT demonstrated superior results for feed-forward neural networks. Consequently, the focus of our evaluation remained centered on iSAT, recognizing its proficiency and reliability in the verification of neural networks.

In the verification of LSTM networks, our study revealed that iSAT2 outperformed iSAT3 in terms of performance. The inherent complexity of LSTM verification problems stems from the presence of an extensive number of variables, creating a vast search space for constraint solving. For instance, in the case of the Sg1000 example, there were approximately 15 million real-valued variables involved.

The comparison highlighted a notable difference in the approach employed by iSAT2 and iSAT3. iSAT3 utilized an SAT-modulo-theory-style Boolean literal abstraction for theory-related facts, while iSAT2 employed direct interval constraint propagation. In the context of LSTM verification, this distinction seemed to have an impact, with iSAT2's approach proving more efficient. The observation suggests that, in scenarios involving extremely large numbers of variables and intricate search spaces, the direct interval constraint propagation method of iSAT2 may be more advantageous than the abstraction-based approach of iSAT3. This observation serves as

valuable insight for guiding the future development of dedicated LSTM verifiers, indicating the need for tailored approaches that can effectively handle the unique challenges posed by the verification of LSTM neural networks.

6.2 Future Research

To expand upon the findings and potential applications of our verification methods, future research avenues are worth exploring. This section outlines three key areas for future investigation.

- **Handling diverse neural network architectures**: Exploring the applicability of the proposed verification methods to other neural network architectures, including convolutional neural networks, reinforcement learning models, and other diverse architectures.

- **More benchmarks**: To gain further insight into the scale of networks iSAT can successfully check, a set of different benchmarks would allow this conclusion to be further confirmed.

- **Enhance and improve the current procedures**: For future work, it is imperative to refine and optimize the existing procedures while addressing critical questions related to the mutual benefits and drawbacks of reach-set versus satisfiability-based methods for neural network verification. The term 'reach-set' refers to the set of all possible states that a system can reach under given conditions. Beyond mere refinement, an in-depth exploration is required to understand the complementary nature of these methods. A preliminary hypothesis suggests that satisfiability-based methods may excel in falsification scenarios, which warrants further investigation. The insights gained from the comparative analysis of iSAT2 and iSAT3 reveal significant differences in performance. The observed performance bottleneck associated with Boolean abstraction suggests that the use of off-the-shelf SMT solvers might pose challenges for NN verification. Therefore, future research could explore the development of a structural SMT version that traverses the neural network structurally, similar to the relation between structural SAT and CNF SAT in circuit verification [JBH12]. Such an approach holds promise for overcoming current limitations and enhancing the efficiency of NN verification processes.

Bibliography

[Adm07] Federal Highway Administration. US highway 101 dataset NGSIM, 2007. https://www.fhwa.dot.gov/publications/research/operations/07030.

[AJO+18] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11), 2018.

[AKB00] S Agatonovic-Kustrin and Rosemary Beresford. Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research. *Journal of pharmaceutical and biomedical analysis*, 22(5):717–727, 2000.

[AKLP19] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of RNN-based neural agent-environment systems. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 6006–6013. AAAI Press, 2019.

[AMM22] Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. On the structure of the boolean satisfiability problem: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–34, 2022.

[AMTZ21] Mahathi Anand, Vishnu Murali, Ashutosh Trivedi, and Majid Zamani. Verification of hyperproperties for uncertain dynamical systems via barrier certificates. *arXiv preprint arXiv:2105.05493*, 2021.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BFF+19] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. JuliaReach: a toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 39–44, 2019.

[BG94] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4):27–31, 1994.

[BIL+16] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. *Advances in neural information processing systems*, 29, 2016.

[Bje05] Per Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1–es, 2005.

[BPSV03] Jerry R Burch, Roberto Passerone, and Alberto L Sangiovanni-Vincentelli. Modeling techniques in design-by-refinement methodologies. *System Specification & Design Languages: Best of FDL'02*, pages 283–292, 2003.

[BR23] Sumon Biswas and Hridesh Rajan. Fairify: Fairness verification of neural networks. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1546–1558. IEEE, 2023.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB standard: version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[CFS] CFSCC. Space track. `https://www.space-track.org`.

[Cla97] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[CMF21] Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *Uncertainty in Artificial Intelligence*, pages 333–343. PMLR, 2021.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 138–156. Springer, 2001.

[CW96] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[Dev22] TensorFlow Developers. TensorFlow. GitHub: `https://github.com/tensorflow/tensorflow/graphs/contributors`, 2022.

[Dic18] M.R. Dickey. Tesla model X speed up in autopilot mode seconds before fatal crash: according to NTSB. `https://techcrunch.com/story/tesla-model-x-fatal-crash-investigation`, 2018.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DLV11] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2011.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *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 14*, pages 337–340. Springer, 2008.

[DT03] George Bernard Dantzig and Mukund N Thapa. *Linear programming: Theory and extensions*, volume 2. Springer, 2003.

[ECfE] Inland Transport Committee Economic Commission for Europe. Vienna convention on road traffic. http://www.unece.org/fileadmin/DAM/trans/conventn/crt1968e.pdf.

[Ehl17] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*, pages 269–286. Springer, 2017.

[EKKT22] Charis Eleftheriadis, Nikolaos Kekatos, Panagiotis Katsaros, and Stavros Tripakis. On neural network equivalence checking using SMT solvers. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 237–257. Springer, 2022.

[Eur] European Space Agency. Collision detection dataset. https://www.kelvins.esa.int.

[FH07] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30:179–198, 2007.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *J. Satisf. Boolean Model. Comput.*, 1(3-4):209–236, 2007.

[Fib22] Connor Fibich. Towards automated analysis of non-linear neural networks: converting neural networks for verification with iSAT. Master's thesis, Foundations and Applications of Systems of Cyber-Physical Systems Research Group, Carl von Ossietzky Universität Oldenburg, September 2022.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M Clarke. δ-complete decision procedures for satisfiability over the reals. In *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings 6*, pages 286–300. Springer, 2012.

[Geu09] Herman Geuvers. Proof assistants: History, ideas, and future. *Sadhana*, 34(1):3–25, 2009.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated*

Deduction – CADE-24, pages 208–214, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GS15] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*, pages 581–591. Springer, 2015.

[GZZ+23] Xingwu Guo, Ziwei Zhou, Yueling Zhang, Guy Katz, and Min Zhang. OccRob: efficient SMT-based occlusion robustness verification of deep neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 208–226. Springer, 2023.

[Her11] Christian Herde. *Efficient solving of large arithmetic constraint systems with complex Boolean structure*. Springer, 2011.

[HG16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[HKWW17] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 3–29. Springer, 2017.

[Hoc98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[HSW90] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[IEE19] IEEE. IEEE standard for floating-point arithmetic. Technical Report IEEE Std 754-2019 (Revision of IEEE 754-2008), IEEE, 2019.

[isa10] *iSAT Quick Start Guide*. AVACS H1/2 iSAT Developer Team, 2010.

[Jaz14] Nasser Jazdi. Cyber physical systems in the context of Industry 4.0. In *2014 IEEE international conference on automation, quality and testing, robotics*, pages 1–4. IEEE, 2014.

[JBH12] Matti Järvisalo, Armin Biere, and Marijn JH Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012.

[JBK20a] Yuval Jacoby, Clark Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. In *Automated Technology for Verification and Analysis: 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19–23, 2020, Proceedings 18*, pages 57–74. Springer, 2020.

[JBK20b] Yuval Jacoby, Clark W. Barrett, and Guy Katz. Verifying Recurrent Neural Networks Using Invariant Inference. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2020.

[KBD+17a] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 97–117. Springer, 2017.

[KBD+17b] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Towards proving the adversarial robustness of deep neural networks. *arXiv preprint arXiv:1709.02802*, 2017.

[KK17] Nikhil Ketkar and Nikhil Ketkar. Introduction to keras. *Deep learning with python: a hands-on introduction*, pages 97–111, 2017.

[KKLR13] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th international conference on cyber-physical systems*, pages 31–40, 2013.

[KMKM21] Nikhil Ketkar, Jojo Moolayil, Nikhil Ketkar, and Jojo Moolayil. Introduction to pytorch. *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*, pages 27–91, 2021.

[Koe18] Will Koehrsen. Overfitting vs. underfitting: A complete example. *Towards Data Science*, 405, 2018.

[Kor22] Milan Korda. Stability and performance verification of dynamical systems controlled by neural networks: algorithms and complexity. *IEEE Control Systems Letters*, 6:3265–3270, 2022.

[Krä06] Walter Krämer. Generalized intervals and the dependency problem. In *PAMM: proceedings in applied mathematics and mechanics*, volume 6, pages 683–684. Wiley Online Library, 2006.

[Kri07] David Kriesel. *A Brief Introduction to Neural Networks*. dkriesel.com, 2007. eBook: PDF, 244 pages.

[Kri10] Moez Krichen. A formal framework for conformance testing of distributed real-time systems. In *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings 14*, pages 139–142. Springer, 2010.

[Kri18] Moez Krichen. *Contributions to model-based testing of dynamic and distributed real-time systems*. PhD thesis, École Nationale d'Ingénieurs de Sfax (Tunisie), 2018.

[Kri24] Moez Krichen. Exploring the feasibility of formal methods in machine learning and Artificial Intelligence, Jan 2024.

[Lam18] Khoa Lam. Tesla model x on autopilot crashed into california highway barrier, killing driver, 2018. `https://incidentdatabase.ai/cite/321/`.

[LCB10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST Database of Handwritten Digits. `http://yann.lecun.com/exdb/mnist/`, 2010.

[LCTJ23] Diego Manzanas Lopez, Sung Woo Choi, Hoang-Dung Tran, and Taylor T Johnson. NNV 2.0: the neural network verification tool. In *International Conference on Computer Aided Verification*, pages 397–412. Springer, 2023.

[LLL+20] Yu Li, Min Li, Bo Luo, Ye Tian, and Qiang Xu. Deepdyve: Dynamic verification for deep neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–112, 2020.

[LM17] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*, 2017.

[MC01] Danilo Mandic and Jonathon Chambers. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. Wiley, 2001.

[MF21] Farzaneh Moradkhani and Martin Fränzle. Functional verification of cyber-physical systems containing machine-learnt components. *it-Information Technology*, 63(5-6):277–287, 2021.

[MFF23] Farzaneh Moradkhani, Connor Fibich, and Martin Fränzle. Verification of LSTM neural networks with non-linear activation functions. In *NASA Formal Methods Symposium*, pages 1–15. Springer, 2023.

[MPDW21] Sara Mohammadinejad, Brandon Paulsen, Jyotirmoy V. Deshmukh, and Chao Wang. DiffRNN: Differential Verification of Recurrent Neural Networks. In Catalin Dima and Mahsa Shirmohammadi, editors, *Formal Modeling and Analysis of Timed Systems - 19th International Conference, FORMATS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12860 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2021.

[onn] ONNX: open neural network exchange. `https://onnx.ai/`.

[PNW20] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97. Springer, 2001.

[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.

[PT11] Luca Pulina and Armando Tacchella. NeVer: A tool for artificial neural networks verification. *Annals of Mathematics and Artificial Intelligence*, 62:403–425, 2011.

[PT12] Luca Pulina and Armando Tacchella. Challenging SMT solvers to verify neural networks. *Ai Communications*, 25(2):117–135, 2012.

[PYY+19] Zhaoqing Pan, Weijie Yu, Xiaokai Yi, Asifullah Khan, Feng Yuan, and Yuhui Zheng. Recent progress on generative adversarial networks (GANs): A survey. *IEEE access*, 7:36322–36333, 2019.

[RAS20] Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sinčák. A review of activation function for artificial neural network. In *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 281–286. IEEE, 2020.

[RDGC13] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. In *Backpropagation*, pages 1–34. Psychology Press, 2013.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, 7(1):5, 2017.

[She20] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[SKB13] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. Recent Improvements in the SMT Solver iSAT. *MBMV*, 13:231–241, 2013.

[Sol21] SAT Solvers. Conflict-Driven Clause Learning. *Handbook of Satisfiability*, 336:133, 2021.

[SSA17] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[TBXJ20] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T Johnson. Verification of deep convolutional neural networks using imagestars. In *International conference on computer aided verification*, pages 18–42. Springer, 2020.

[TCY+23] Hoang Dung Tran, Sung Woo Choi, Xiaodong Yang, Tomoya Yamaguchi, Bardh Hoxha, and Danil Prokhorov. Verification of recurrent neural networks with star reachability. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–13, 2023.

[TOHC15] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[TPM+21] Hoang-Dung Tran, Neelanjana Pal, Patrick Musau, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Stanley Bak, and Taylor T Johnson. Robustness verification of semantic segmentation neural networks using relaxed reachability. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 263–286. Springer, 2021.

[Tur89] Alan Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72, 1989.

[UIS+22] Thomas Uriot, Dario Izzo, Luís F Simões, Rasit Abay, Nils Einecke, Sven Rebhan, Jose Martinez-Heras, Francesca Letizia, Jan Siminski, and Klaus Merz. Spacecraft collision avoidance challenge: Design and results of a machine learning competition. *Astrodynamics*, 6(2):121–140, 2022.

[vOU24] Carl von Ossietzky University. CARL: center for high-performance computing, 2024. `https://uol.de/fk5/wr/hochleistungsrechnen/hpc-facilities/carl`.

[WYW+19] Xiaobing Wang, Kun Yang, Yanmei Wang, Liang Zhao, and Xinfeng Shu. Towards formal verification of neural networks: a temporal logic based framework. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 73–87. Springer, 2019.

[Yin19] Xue Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.

[YSHZ19] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation*, 31(7):1235–1270, 07 2019.

[ZJ15] Xi Zheng and Christine Julien. Verification and validation in cyber physical systems: Research challenges and a way forward. In *2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 15–18. IEEE, 2015.

[ZMW+21] Meng Zhu, Weidong Min, Qi Wang, Song Zou, and Xinhao Chen. PFLU and FPFLU: Two novel non-monotonic activation functions in convolutional neural networks. *Neurocomputing*, 429:110–117, 2021.

Appendices

Appendix A

Acronyms

This section is dedicated to defining the acronyms and abbreviations utilized in this research.

- **ANNs**: Artificial Neural Networks
- **AF**: Activation Function
- **AV**: Automated Vehicles
- **BMC**: Bounded Model Checking
- **CDCL**: Conflict-Driven Clause Learning
- **CDM**: Conjunction Data Message
- **CNF**: Conjunctive Normal Form
- **CPS**: Cyber-Physical Systems
- **DNN**: Deep Neural Network
- **DPLL**: The Davis-Putnam-Logemann-Loveland Method
- **ESA**: European Space Agency
- **FFNN**: Feedforward Neural Network

- **GAN**: Generative Adversarial Network
- **GELU**: Gaussian Error Linear Unit
- **ICP**: Interval-Based Arithmetic Constraint Solving
- **LEO**: Low Earth orbit
- **LP**: Linear Programming
- **LSTM**: Long Short-Term Memory
- **MBT**: Model Based Testing
- **MNIST**: Modified National Institute of Standards and Technology database
- **MILP**: Mixed Integer Linear Programming Solvers
- **NGSIM**: Next Generation Simulation
- **ONNX**: Open Neural Network Exchange
- **ReLU**: Rectified Linear Unit
- **REU**: Rectified Exponential Unit
- **RNN**: Recurrent Neural Network
- **SAT**: Boolean Satisfiability Problem
- **SMT**: Satisfiability Modulo Theories Solvers
- **Tanh**: Hyperbolic Tangent Function
- **TCA**: Time of Closest Approach
- **XOR**: Exclusive Or