Contributions to the Verification and Monitoring of Neural Network Systems

Fateh Boudardara

Université de Lille - Sciences et Technologies
Université Gustave Eiffel

THÈSE

présentée en vue d'obtenir le grade de

DOCTEUR

en

Automatique, Génie Informatique

par

Fateh BOUDARDARA

Doctorat délivré par l'université de Gustave Eiffel

Titre de la thèse :

Contributions to the Verification and Monitoring of Neural Network Systems

Contributions à la Vérification et au Monitoring des Systèmes de Réseaux de Neurones

Soutenue le 23/05/2024 devant le jury d'examen :

Rapporteur	Nesrine Zoghlami	Professeur d'université	Université de Tunis El-Manar
Rapporteur	Mohamed Sallak	Maître de conférence / HDR	UTC de Compiègne - HEUDIASYC
Directeur de thèse	Mohamed Ghazel	Directeur de recherche	Université Gustave Eiffel - COSYS/ESTAS
Encadrant de thèse	Abderraouf Boussif	Chargé de recherche	Université Gustave Eiffel - COSYS/ESTAS

Thèse préparée au Laboratoire d'Évaluation des Systèmes de Transports
Automatisés et de leur Sécurité
Université Gustave Eiffel, COSYS/ESTAS, Villeneuve d'Ascq
École Doctorale MADIS-631 - Université Lille Nord de France

Contributions to the verification and monitoring of neural network systems

Abstract

The evaluation and verification of neural networks (NNs), as a part of their safe design and deployment, becomes a hot research topic, particularly with the recent studies showing their sensitivity and vulnerability to operational conditions (adversarial attacks, environment conditions, etc.). Despite its importance in ensuring the accuracy and the reliability of NNs, test-based approaches for NNs evaluation suffer from several limitations that may impact their effectiveness. To overcome the limitation of NN testing, researchers are exploring formal verification as complementary activity to enhance the reliability and safety of NN-based systems. Indeed, while testing can illustrate the ability of a system to maintain its level of performance under varying conditions, proving this requires some form of formal analysis. NN verification aims to provide formal guarantees regarding the behavior and properties of NNs. It involves analyzing the model's inputs, outputs, and internal computations to ensure that the network behaves correctly and meets the desired specifications. While NN verification is applied before the deployment of the network, NN runtime verification (or monitoring) is used to continuously check and assess the correct behavior of the network during runtime. Broadly speaking, NN monitoring consists in building a monitor that runs in parallel to the network in order to supervise its behavior and decision. If the monitor detects a malfunctioning of the network, or some abnormal behavior, it raises alarms demanding an examination of the current decision. Although these techniques have been successfully applied in solving certain properties of NN, NN verification and monitoring remains challenging, particularly when it comes to verifying large networks with practical interests. This is mainly due to the complexity and the non-linearity of NN models, and the limitations of the traditional formal methods to scale up to large real-world models.

In this dissertation, we propose two main contributions to address these challenges, namely (*i*) NN abstraction (model reduction) for verification purposes and (*ii*) NN runtime monitoring. In order to enhance the scalability of NN verification, we propose two approaches involving the merging of neurons within the same hidden layer of a network to reduce its size. Both approaches ensure that the resulting reduced model over-approximates the original network. The over-approximation relation is crucial, as it guarantees that any verified property on the reduced model remains valid on the original network. Our proposed approaches rely on mathematical formulas to formally establish and ensure this over-approximation relationship. Additionally, we provide formal proofs of this relation for each approach, thus ensuring the rigor and reliability of our methods. The second contribution involves the development of a monitoring system specifically designed for NNs used in image classification tasks. The key idea behind this approach is to identify and extract relevant path activations that are referred to as *NAPath*. The computation of NAPaths is performed for each class of images using the training set. Each NAPath serves as a reference pattern that captures the essential characteristics of the associated class. During the runtime, the monitoring system compares the network's classification results to the most similar NAPath. This comparison analysis enables the monitor to evaluate the consistency of the network's classification decisions and detect any potential deviations or misclassification. To evaluate the proposed approaches, we have implemented them as Python tools and carried out a set of experiments on well-known NN benchmarks and/ or railway use cases.

Keywords: railway system safety; safe artificial intelligence (safe ai); formal verification; neural networks; neural networks verification; neural networks abstraction; neural networks monitoring

Résumé

L'évaluation et la vérification des réseaux neuronaux (NN), lors de leur conception et de leur déploiement sécurisé, suscitent un vif intérêt de recherche, en particulier à la lumière d'études récentes mettant en évidence leur sensibilité et leur vulnérabilité face à diverses conditions opérationnelles telles que les attaques adverses et les variations environnementales. Bien que leur importance soit cruciale dans l'assurance de la précision et de la fiabilité des NN, les approches fondées sur les tests pour l'évaluation des NN sont entravées par plusieurs limitations qui peuvent compromettre leur efficacité. Afin de remédier à ces limites, des travaux de recherche explorent la vérification formelle comme approche complémentaire pour vérifier la fiabilité et la sécurité des NN. Si les tests peuvent démontrer la capacité d'un système à maintenir ses performances dans des conditions variables, démontrer cela nécessite une analyse formelle approfondie. La vérification des NN vise ainsi à fournir des garanties formelles quant au comportement et aux propriétés des NN. Cela implique une analyse minutieuse des entrées, des sorties et des calculs internes du modèle pour s'assurer que le réseau se comporte correctement vis-à-vis des spécifications requises. Alors que la vérification des NN est, d'une manière générale, effectuée lors de la conception des systèmes, le monitoring des NN, quant à lui, est cruciale pour garantir l'exécution du bon comportement lors de la phase opérationnelle. Pour ce faire, un système de monitoring fonctionne en parallèle du NN, détectant toute anomalie ou comportement inattendu et déclenchant des alertes en cas de besoin. Bien que des succès aient été rencontrés dans certaines applications, la vérification et le monitoring des NN restent des défis, notamment lorsqu'il s'agit de NN complexes ou de taille importante. Cette difficulté découle en grande partie de la complexité et de la non-linéarité des modèles NN, ainsi que des limites des méthodes formelles existantes pour s'adapter à des modèles réels de grande taille.
Dans cette thèse, nous proposons deux contributions principales pour répondre aux deux défis susmentionnés. Concrètement, nous proposons des techniques d'abstraction des NN (réduction de modèle) à des fins de vérification, ainsi qu'une approche de monitoring des NN en temps réel spécifiquement conçu pour les tâches de classification d'images. Ces approches d'abstraction de NN visent à améliorer la scalabilité et l'efficacité de la vérification des NN. En se basant sur des formulations mathématiques, nous garantissons que les modèles abstraits résultants conservent les propriétés essentielles (e.g., la sur-approximation) des réseaux d'origine, assurant ainsi la validité des résultats de vérification. Pour la partie monitoring, nous développons un système de surveillance qui identifie et évalue en temps réel les décisions de classification des réseaux par rapport à des modèles de référence spécifiques (motives de surveillance). Enfin, nous validons nos approches à travers des expérimentations menées sur des benchmarks académiques, ainsi qu'un cas d'étude spécifique au domaine ferroviaire.

Mots clés : sécurité des systèmes ferroviaires ; sécurité de l'intelligence artificielle ; réseaux de neurones ; vérification formelle, vérification des réseaux de neurones ; abstraction des réseaux de neurones, monitoring des réseaux de neurones

Acknowledgment

This thesis manuscript cannot be considered complete without expressing my heartfelt thanks to all the persons who supported and encouraged me throughout my PhD journey. I thank all those who have contributed in any way to the completion of this thesis.

To my family.
To my mother & my father.
To my wife & my prince Amir.

Contents

List of Tables

List of Figures

List of Acronyms

ACAS Xu	Airborne Collision Avoidance System Xu
AI	Artificial Intelligence
BaB	Branch and Bound
CNN	Convolutional Neural Networks
DL	Deep Learning
FM	Formal Methods
FFNN	Feed-Forward Neural Networks
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
NN	Neural Network
ReLU	Rectified Linear Unit
RNN	Reccurent Neural Networks
ReLU	Rectified Linear Unit
SAT	Satisfiability problem
SMT	Satisfiability Modulo Theories
SVM	Support Vector Machine
Tanh	Hyperbolic Tangent
UNSAT	Unsatisfiable

Chapter 1

Introduction

Outline of the current chapter

1.1 General Context

Today, Artificial Intelligence (AI) has become an ubiquitous term, reflecting its broad range of applications and its growing impact across various domains. Indeed, AI has been successfully employed to handle various tasks that are playing a significant role in our daily lives, ranging from daily applications like voice assistants, predictive text auto-completion and personalized recommendations for articles and videos, to complex applications in healthcare, finance, and autonomous transportation systems (Zhang and Lu 2021).

Despite its current prominence, AI is not a novel concept. The first use of the term AI can be traced back to the 1950s. At the Dartmouth Conference in 1956, John McCarthy defined the AI problem as follows: *"For the present purpose, the artificial intelligence problem is taken to be that of making a machine behaves in ways that would be called intelligent if a human were so behaving"* (McCarthy et al. 1955). Nowadays, there exist many definitions of AI. (Rich 1983) stated that: *"AI is the study of how to make computers do things at which, at the moment, people are better."*. The European Commission Joint Research Centre[1] defines AI as *"any machine or algorithm that is capable of observing its environment, learning, and based on the knowledge and experience gained, take intelligent actions or propose decisions"* (Nativi et al. 2019). Despite this number of definitions, the majority of them technically agree that AI includes a set of techniques and methods designed to mimic the intelligent behaviors of the living beings, including human brain, insects, etc (Sheikh et al. 2023). Algorithms such as Artificial Bee Colony (Karaboga et al. 2014), Ant Colony Optimization (Dorigo et al. 2019), and Genetic Algorithm (Lambora et al. 2019) are widely applied to solve several optimization problems.

In the last decade, artificial neural networks (or shortly Neural Networks (NNs)) have witnessed huge success stories. NNs rely on imitating the learning behavior of the human brain from past experiences and its ability to analyze data. NNs are widely used in solving complex problems in various domains: natural language processing, computer vision, healthcare, etc. (Liu, Wang, et al. 2017). These successful applications motivated people to deploy AI in some safety-critical systems such as transportation systems. Indeed, AI can be applied in different development stages and operational phases of transportation systems, helping them to be safer and more comfortable. For instance, AI can be applied to identify risks, avoid traffic congestion, reduce CO_2 emission, and optimize the utilization of equipment and infrastructure (Tang, De Donato, et al. 2022b;

[1] https://publications.jrc.ec.europa.eu/

Wäschle et al. 2022).

The accomplishments of AI dealing with tasks that typically require human intelligence motivated people to integrate these techniques in safety-critical systems, such as transportation systems. The railway domain is no exception, as AI has found utility in numerous tasks, including maintenance and inspection, traffic management, passenger mobility, autonomous driving and control, just to name a few (Tang, De Donato, et al. 2022a). The purpose of applying AI in the railway domain is to efficiently exploit the railway infrastructure leading to an improved performance and capacity. AI can also be used to help build sustainable transport by reducing the energy consumption and CO_2 emissions. All this should enhance the services' quality for a better and safer travelling experience (Trentesaux et al. 2018a). One highly illustrative instance of AI application in railway remains the pivotal role in the automation and digitalization of train operations. The underlying goal is toincrease punctuality, expand mobility, enhance safety, reliability, and ultimately improve the overall capacity of existing rail networks.

From the AI technology viewpoint, numerous techniques and tools have the potential to contribute significantly in improving the performance of different Railway modules. Techniques such as Support Vector Machine (Gibert et al. 2015; Hua et al. 2020) and Particle Swarm Optimization (PSO) (Pu et al. 2019), Ant Colony Optimization (ACO) (Zhang, Yuan, et al. 2018), and Genetic Algorithm (GA) (Brenna et al. 2016; Hickish et al. 2020) have been employed in different Railway activities such as, for instance, traffic and planning management, energy management, scheduling, maintenance and infrastructure monitoring, etc. (Tang, De Donato, et al. 2022a). However, the major research efforts focus on applying computer vision and pattern recognition methods, specifically, neural networks (NNs) due to their fast improvements and their achievements in dealing with large image-data (Ristić-Durrant et al. 2021). Indeed, NN-based techniques are applied for Railway maintenance and inspection (Gibert et al. 2017; Yang, Wang, et al. 2022), environment monitoring and autonomous driving (Hadded et al. 2022; Mahtani et al. 2020), and on-board events' monitoring (Laurendin et al. 2021; Velastin et al. 2017).

This surge of interest in deploying AI for automating trains has also been witnessed by numerous international and national projects, where most of them involve collaboration between academia and industry. Prominent initiatives

like X2Rail-4[2], TAURO[3], SafeTrAIn[4], and R2DATO[5] are few examples showing the huge interest of different actors in applying AI for the digitalization and automation of the railway domain. In France, the ambition of autonomous trains targets several improvements in the railway system, including the overall safety level as presented in (Lagay et al. 2018). In early 2017, the Directorate of Railway Systems at SNCF (*French Public Sector Undertaking for National Railways*) initiated a technological program called Tech4Rail to build the fundamentals of future autonomous railway systems and to prepare the deployment of the safe autonomous and semi-autonomous trains (see (Lagay et al. 2018; Masson et al. 2019; Trentesaux et al. 2018b)). Within the framework of the Autonomous Train program at the Railway Technological Research Institute – Railenium (with industrial and academic partners), two consortia have launched three major projects in this direction, setting their sight to develop, respectively, a remote driving train (TC-Rail project), an autonomous freight train (TFA project), and an autonomous passenger train (TASV).

While artificial intelligence and neural networks have the ability to become an overwhelmingly beneficial utility for railway transportation and mobility, yet their trustworthiness and safety assurance require a particular attention. Indeed, the use of AI in general, and in railway particularly, gives rise to numerous challenges and multidisciplinary issues, encompassing ethical, social, economic, and technical dimensions (De Donato et al. 2022). From the safety viewpoint, the main technical issues that are limiting the safe integration of AI (and particularly NNs) in the context of autonomous trains are related to their vulnerability and sensitivity (*robustness*), as well as to their black-box nature (*explainability*). While robustness[6] is used to describe the ultimate ability of an AI system to maintain its level of performance under any circumstances including external interference (e.g., adversarial attacks) or harsh environmental conditions (ISO/IEC TR 24028 2020), explainability (known as XAI) is presented as an attribute of the transparency of an AI system, and represents the degree to which a stakeholder with defined needs can understand the reasons of an AI model's output (DIN-SPEC 92001 2020).

As safety is a major requirement in railways, as well as in other critical

[2] https://projects.shift2rail.org/s2r_ip2_n.aspx?p=X2RAIL-4
[3] https://cordis.europa.eu/project/id/101014984
[4] https://safetrain-projekt.de/en/
[5] https://projects.rail-research.europa.eu/eurail-fp2/
[6] Another definition of robustness is the capability of an AI module to cope with erroneous, noisy, unknown, or adversarially constructed date (DIN SPEC 92001-2:2020).

transport systems, sophisticated risk management and safety assurance processes must be developed to integrate AI techniques in the whole life cycle of such systems (De Donato et al. 2022). In the the last few years, several research works, R&D projects, and standardization initiative are proposed to set up processes, approaches and tools for the safety assurance and certification of systems integrating AI, particularly NNs (Hawkins et al. 2021; Koopman et al. 2019; Mamalet et al. 2021; VDE-AR-E 2842-61 2021).

1.2 Problem Statement

The evaluation and verification of NNs, as part of their safe design and deployment, becomes a hot topic, especially with the recent studies showing their sensitivity and vulnerability to operational conditions (adversarial attacks, environment conditions, etc.). Test-based verification (testing for short), formal verification, and runtime monitoring are three key verification activities for assuring the safe and reliable deployment of neural network systems. Testing utilises the verification data to demonstrate that the model generalises to cases not present in the model learning; concretely, it involves evaluating the trained model through the process of comparing the predictions made by the model on various data sets to the actual labels in the data, with respect to given performance metric (Goodfellow, Bengio, et al. 2016). Formal verification involves using mathematical techniques to prove that the learnt model satisfies formally-specified properties (Gehr et al. 2018; Katz, Barrett, et al. 2017). When formal verification is applied, counter-examples are typically generated which demonstrate the properties that are violated (Hawkins et al. 2021). While testing and formal verification are performed during the system design phase, monitoring is an ongoing runtime activity performed during the system operational phase. It involves real-time tracking of inputs, the performance and the behavior of the model (Cheng, Nührenberg, and Yasuoka 2019). For instance, inputs can be monitored with appropriate statistical techniques to make sure that they are close to the training data distributions (Hawkins et al. 2021).

Despite its importance in ensuring the accuracy and the reliability of NN, test-based approaches for NN verification suffer from several limitations that may impact their effectiveness. These approaches rely on a set of data, commonly referred to as the test set, which does not cover all possible cases the system may encounter during operation its operational phase. Additionally, although testing may offer insights into the network's performance, such performance depends

highly on the used test set, and the obtained results remain valid only on this set. Consequently, a high accuracy achieved by a network on a specific test set does not provide any formal guarantees regarding its behavior when processing new, unseen inputs. This issue becomes more obvious with adversarial attacks, where even slight modifications to inputs can deceive the network to generate wrong and unpredictable outputs (Xu, Ma, et al. 2020).

To overcome the limitation of NN testing, researchers are exploring formal verification as complementary activity to enhance the reliability and safety of neural network systems. Indeed, while testing can enhance the ability of a system to maintain its level of performance under varying conditions, proving it requires some form of formal analysis.

Conventional approach to using formal methods consists in three main steps: (i) the system to be analysed is formally defined in a model that precisely captures all possible behaviors of the system, (ii) the property of the requirement to be verified is defined or specified using a formal language; and finally, (iii) a formal technique, such as solver, abstract interpretation or model checking, is used to assess whether the system meets the given property, yielding either a proof or a counterexample. Standard ISO/IEC 24029-2:2022 provides a methodology, including recommendation and requirements, on the use of formal methods to assess properties related to the robustness of neural networks during their life cycle.

In the research literature, many works have been proposed to adapt formal verification methods to evaluate the safety and the correctness of NNs (Urban et al. 2021). The first attempts toward verifying NNs were based on transforming the NN verification problem into a Satifiability problem (SAT) and use solvers on the shelf to check the desired properties (Ehlers 2017; Huang, Kwiatkowska, et al. 2017; Katz, Huang, et al. 2019). Another similar works consider encoding the verification problem as a Linear Programming(LP) problem and then use Mixed-Integer Linear Programming (MILP) (Cheng, Nührenberg, and Ruess 2017; Dutta et al. 2018; Lomuscio et al. 2017; Tjeng et al. 2019). These methods (SAT/SMT MILP) provide a definitive answer on the compliance of the NN to the property of interest, at the price of a greatly increased computational complexity (i.e., state explosion problem that hamper them from scaling well to verify large networks). As a result, other research were directed toward abstracting and over-approximating the behavior of the network using techniques like Abstract Interpretation and bound propagation to provide an answer subject to a degree of uncertainty (Eramo et al. 2022). The abstraction of the network's behavior

helps making the verification process more scalable, and thus, can be used to verify larger networks (Ashok et al. 2020; Boudardara, Boussif, Meyer, et al. 2022; Elboher et al. 2020; Gehr et al. 2018; Prabhakar and Rahimi Afzal 2019; Wang et al. 2018a,b; Xiang et al. 2020).

Another significant aspect of NN evaluation and verification is pertaining to monitoring NN during its operational phase. Runtime monitoring provides a more realistic and applicable alternative to verification in the setting of real neural networks used in industry (Hashemi, Křetínskỳ, Rieder, et al. 2023). This involves the construction of a separate model, namely the monitor, that operates alongside the network to oversee its behavior and performance. The construction of the monitor is an important step, as it directly influences the efficacy of the whole monitoring process. For NN, the idea is to build a monitor that captures some of the features learned by the network. Then, this monitor is used to continuously observe and supervise the network and compare its behavior against the expected one. It is designed to identify deviations, anomalies, or any degradation in its performance that may indicate a malfunction or a breach of the desired behavior. Additionally, it can be particularly useful for detecting out-of-distribution (OOD) inputs, for which the network was not trained and can yield erroneous results (Hashemi, Křetínskỳ, Rieder, et al. 2023). Concretely, when the monitor identifies a misbehavior, such as misclassificaiton or unusual patterns, it promptly raises an alarm to inform the control unit (responsible individual or another systems). These alarms serve as notifications, alerting of the existence of a malfunctioning or an undesired behavior.

Although such evaluation techniques exist, NN evaluation presents significant challenges. Namely, it suffers from the lack of formal verification tools suitable to check properties on the generated networks. This is mainly related to the complexity of the NN models, e.g., NNs used in autonomous trains. The complexity is mainly related to the NNs architectures, the large number of parameters, the non-linearity of functions, and the size of the networks in terms of the number of layers and neurons. Furthermore, the available NN verification methods face issues when it comes to scaling up and handling real-world sized networks. The complexity and the non-linearity of NN models make the verification process computationally expensive and resource-demanding. As a result, the scalability of these methods becomes a significant concern.

Additionally, the challenges faced by existing NN verification techniques include difficulties in formally expressing certain specifications of interest on NNs. This lack of formal specifications becomes particularly crucial when

dealing with perception modules (Leucker 2020). For instance, ensuring that all images of a stop sign are correctly classified as a stop sign by the network is a desirable verification specification. However, such a specification cannot be formally expressed. This is due to the inherent difficulty in mathematically representing all the possible forms of an image of a stop sign.

Finally, one can point out that even NN verification remains insufficient for the assurance of safe operations of the NN models; and thus, runtime verification and monitoring are needed to supervise the NN during the operation phase.

1.3 Main Contributions

In this section, we present a general overview of our contributions addressing the aforementioned challenges and issues related to both the deployment of NN in safety-critical systems and the evaluation of such models (networks). More precisely, our contributions mainly focus on NN abstraction and NN monitoring.

1.3.1 Neural Networks Abstraction

To tackle the non-scalability issue of the existing NN verification methods, we propose two abstraction approaches based on model reduction. The main idea of the approaches is to abstract the network by reducing its size, allowing for a fast and more scalable verification process. This is achieved by merging its neurons while maintaining an over-approximation relationship with the original network. This relationship helps to straightforwardly lift the correctness proof from the reduced to the original network. To mitigate the loss of precision resulting from the abstraction, we propose a heuristic for nodes selection. Integrating this heuristic with the proposed approaches aim to enhance the precision of the abstract network, and yields tighter output range bounds (on the abstract network).

The first proposed approach is called *INNAbstract*. It is based on Interval Neural Networks (INN) and aims to ensure that the output of the original network is within the obtained abstract network's output, i.e., $N(x) \subseteq \overline{N}(x)$. INNAbstract can be applied on NN with numerous activation functions, including monotone odd activation functions and the ReLU function. To construct such abstract networks, INNAbstract explores the signs of the weights in the original NN and represents the weights of the abstract NN as intervals. Specifically, the abstract outgoing weights are the sum of the absolute values of

the corresponding weights in the original network, and the incoming weights are intervals represented by the minimum and the maximum of the signs of outgoing weights multiplied by the corresponding incoming ones in the original network.

Our second model reduction approach supports NNs with non-negative activation functions, such as ReLU and Sigmoid. The abstraction process is performed in two main steps: removing negative weights and then merging neurons. The first step involves building an initial abstract network by eliminating edges that have negative weights. The second step involves merging the selected set of neurons and replacing it with a single abstract neuron. Upon neurons' merging, the weights of the obtained abstract neuron is calculated as follows: the outgoing weights correspond to the sum of the absolute values of the outgoing weights of the selected neurons, and the incoming weights are the maximum values over the incoming weights. This guarantees that the output of the abstract network is always greater or equal to the original network's output.

The proposed approaches are implemented as Python software tool, which is used to perform some experimental studies. The experiments are used to evaluate the efficiently and the scalability of our approaches, firstly on randomly generated NNs and then on two well-known benchmarks, namely MNIST (LeCun 1998) and ACAS Xu (Kochenderfer 2015). Additionally, a comparison study between our approaches and other selected approaches from the literature is performed. The obtained results through this series of experiments show that our approaches efficiently outcome the existing approaches on the literature. Furthermore, the experiments also illustrates that the proposed heuristics for INNAbstract can effectively improve its precision, i.e., INNAbstract with the proposed heuristics generates more precise abstract NNs.

1.3.2 Neural Networks Monitoring

The second part of contributions lies in a monitoring approach for NNs used in image classification. Since formally verifying such models remains an open challenge, their monitoring become an important alternative. For instance, when it is not possible to formally express properties on NN like in image classification, the NN monitoring can be used to provide some guarantees, and complement the verification process. In this regard, the proposed approach is typically used as a complementary to formal verification. The later is used before deploying the network, while monitoring is meant to be run during runtime, in parallel to

the network execution.

In this topic, we propose an approach for monitoring NN with ReLU activation function used for image classification. The approach is mainly used to supervise the network's outputs with respect to the input images. It relies on extracting (learned) features by hidden neurons for each class of images, and uses these features to supervise the network's classification decision during runtime. In our work, the features are represented as paths connecting the input layer to the output layer of the network; this allows for maintaining the dependency relationship between different layers of the network.

The NN monitor is synthesized through a paths' extraction (NAPathing) process. The objective of the NAPathing process is to extract patterns (represented as paths) for each class. To do so, we divide the training set of the network into subsets, where each subsets contains only images of the same class. Then, for a single subset, the network is fed with all images to extract neurons' activations (active or inactive). Subsequently, neurons of two adjacent layers whose activations are similar are connected to build paths. Hence, we obtain a set of active paths (containing only active neurons) and a set of inactive paths (containing only inactive neurons) for the class at hand. These two sets constitute the NAPath of this class. This process is repeated for each class, to finally obtain a set of NAPaths, where each NAPath is associated with a class of images. Once the NAPathing phase is completed and the generated set of NAPaths is validated, these NAPaths are used for monitoring the network during runtime. The monitor supervises the classification decision made by the network; it examines the consistency of the network's decision with the extracted patterns. Alarms are triggered by the monitor whenever it detects any abnormal behavior in the network.

To evaluate the effectiveness of our approach, we performed a series of experiments on NNs trained for digit classification on the MNIST dataset (LeCun 1998). The experimental results illustrate how the proposed approach enhances efficiently the reliability and accuracy of the classification decision made by the network. Additionally, we conducted a comparative analysis between our approach and the closest one from the literature (Cheng, Nührenberg, and Yasuoka 2019). According to the obtained results, our monitoring approach demonstrates interesting performances.

1.4 Outlines

The dissertation consists of four main parts, organized as follows:

PART I: this part contains two chapters. In the first chapter (Chapter 2) we provide necessary preliminaries and background that will be used throughout this dissertation. The chapter also includes a detailed overview of the context of this thesis: NN evaluation approaches, particularly NN verification and monitoring. Additionally, issues and challenges related to NN evaluation are discussed in the same chapter. An extended literature review of model reduction techniques applied to NNs (or NN abstraction) for verification purposes is presented in the second chapter of this part (Chapter 3).

PART II: this part is dedicated to present our contributions regarding NN model reduction. This part is composed of two chapters:

- *Chapter 4:* in this chapter we focus on the theoretical aspects of the proposed approaches. First, we provide some relevant notations before getting into the details of the approaches. Since we have two contributions regarding model reduction, each contribution is presented in a separate section, namely, Section 4.3 and Section 4.4. The former section introduces INNAbstract, a general model reduction approach based on Interval Neural Networks (INN) that supports a wide range of activation functions. Section 4.4 discusses our second model reduction method for NNs with non-negative functions. At the end of this chapter, we discuss relevant related model reduction works.

- *Chapter 5:* this chapter is specifically dedicated to present the evaluation study of our contributions, presented in the previous chapter. We evaluate the two approaches on randomly generated networks with different sizes. Additionally, we investigate the applicability of the approaches on two well-known benchmarks, namely ACAS XU (Julian et al. 2016) and MNIST (LeCun 1998). Furthermore, a comparison study is conducted on both randomly generated NNs and on the selected benchmarks with relevant model reduction methods from the literature.

PART III: in this part, we discuss our contribution related to NN monitoring. This part contains two chapters:

- *Chapter 6:* this chapter discusses our contribution related to NN monitoring. We begin by introducing a new concept, namely NAPath, to represent hidden neurons' activation. This concept is then used to build the monitoring system for NN used in image classification. A detailed explanation of the whole monitoring process is provided in this chapter. To evaluate our approach, an experimental study is conducted on the MNIST benchmark. Furthermore, in the same chapter, we illustrate a real application of our monitoring approach on weather conditions' recognition networks developed within the TASV project.

PART IV: this part includes Chapter 7 that present some concluding remarks regarding the dissertation contributions, and outlines potential avenues for future research and perspectives relevant to the thesis's topic.

1.5 List of Publications

Journals

1) Boudardara Fateh, Boussif Abderraouf, Meyer Pierre-Jean, and Ghazel Mohamed (2023). **INNAbstract: an INN-based abstraction method for large-scale neural network verification**. In: IEEE Transactions On Neural Networks And Learning Systems (TNNLS), **IF=10.4**. DOI=10.1109/tnnls.2023.3316551.

2) Boudardara Fateh, Boussif Abderraouf, Meyer Pierre-Jean, and Ghazel Mohamed (2023). **A review of abstraction methods towards verifying neural networks**. In: ACM Transactions on Embedded Computing Systems (ACM TECS), **IF=2.25**. DOI=10.1145/3617508.

Conferences & Workshops

3) Boudardara Fateh, Boussif Abderraouf, Meyer Pierre-Jean, and Ghazel Mohamed (2022). **Interval Weight-Based Abstraction for Neural Network Verification**. In: Computer Safety, Reliability, and Security. SAFECOMP 2022 Workshops. DOI=10.1007/978-3-031-14862-0_24

4) Boudardara Fateh, Boussif Abderraouf, Meyer Pierre-Jean, and Ghazel Mohamed (2022). **Deep Neural Networks Abstraction using An Interval Weights Based Approach**. In: Confiance.ai Days 2022 Workshop. https://hal.science/hal-04024209/ (page 52).

5) Boudardara Fateh, Boussif Abderraouf, and Ghazel Mohamed (2024). **A sound abstraction method towards efficient neural networks verification**. In: In: Ben Hedia, B., Maleh, Y., Krichen, M. (eds) Verification and Evaluation of Computer and Communication Systems. VECoS 2023. Lecture Notes in Computer Science, vol 14368. Springer, Cham. DOI=10.1007/978-3-031-49737-7_6

Part I

Background & Literature Review

Chapter 2

Neural Networks and Formal Verification

Outline of the current chapter

2.1 Introduction

In this chapter, we recall essential foundations pertaining to the formal verification of neural networks as well as a literature review on the works related to the NN verification and monitoring. We begin by examining the fundamental principles of artificial intelligence (AI), with a specific focus on neural networks (NNs). Then, we delve into the fundamental concepts of formal methods, encompassing verification. Subsequently, we explore the application of formal verification techniques to examine some features in the neural networks. Concretely, this chapter is organized as follows:

1. AI & NN: in Section 2.2, we present the general concepts of artificial intelligence with a main focus on neural networks. An important part of this section will be dedicated to present the essential notations and mathematical formulations, offering a deeper insight into the inner workings of neural networks.

2. Formal verification & formal methods: Section 2.3 discusses the basics of formal methods and the various verification techniques used to provide formal guarantees on software and hardware systems.

3. Formal verification on NN: finally, in Section 2.4, we discuss the bridges between the two research fields by introducing neural network verification. In particular, we present a detailed formulation of the problem of verifying neural networks. A literature review of NN verification and NN monitoring is also provided in this section.

In each section, we provide necessary notations and definitions that will be used through this dissertation. We provide a brief literature review of relevant techniques within each section. Given that the main focus of this dissertation is pertaining to NN evaluation, a detailed and comprehensive literature review of NN evaluation methods, including NN verification and monitoring, is provided in Section 2.4.1 and Section 2.4.2, respectively.

2.2 Artificial Intelligence & Neural Networks

Artificial intelligence (AI) is a specialized field within computer science aiming to develop machines that are capable to perform tasks that typically require

some human intelligence (Zhang and Lu 2021). AI systems achieve this by either imitating certain intelligent behaviors observed in humans, or by drawing inspiration from the processes of biological evolution. There exist many definitions of the AI; however, with respect to the scope of this dissertation, we opt for the definition presented by Besinovic et al. (2022): *AI is the discipline gathering all the aspects that allow an entity to determine how to perform a task and/or make a decision based on the experience matured by observing samples and/or by interacting with an environment, possibly competing against or cooperating with other entities.*

Machine learning (ML) as a sub-field of AI allows machines (or programs) to learn and perform specific tasks from experiences without the need for explicit rule-based programming. These learning experiences are derived from a set of data called the training dataset. Based on their way of learning, ML techniques are divided mainly into four categories: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning (Mahesh 2020). For supervised learning, the training dataset is a collection of samples organized as input-output pairs. The inputs are the features or attributes that describe the sample, and the outputs represent the desired target or label associated with that input. The purpose of an ML algorithm is to find a mapping function, also called a model, that can accurately associate each input with the corresponding desired output. When the targeted outputs are unknown, the term unsupervised learning is often used. The goal of this sub-category of ML is to group inputs sharing similar features into clusters (Berry et al. 2019).

An ML algorithm iterates over the training dataset and adjusts the model's parameters with the objective of minimizing the error between the desired outputs (ground truth) and the predicted output by the model. This error is mainly measured using *a loss function*. One of the core capabilities of ML methods is their ability to generalize the learned input-output relationships (the mapping function), allowing them to make predictions on unseen data. Notice that there exist several ML techniques such as linear regression, decision trees, clustering, support vector machines, and neural networks, etc. The selection of the learning algorithm, loss function, and model's architecture and parameters depends on the ML technique and the application domain (Mahesh 2020).

The performance of a model is often measured by calculating its *accuracy* in predicting unseen samples, which are referred to as the test set. This accuracy measurements involves computing the rate (percentage) of the correct predictions made by the ML model on the test set. Once the model achieves an

acceptable accuracy, e.g., exceeds a predefined threshold, it is considered suitable for deployment to reason on unseen samples. An illustration of the learning process of ML model is presented in Figure 2.1.

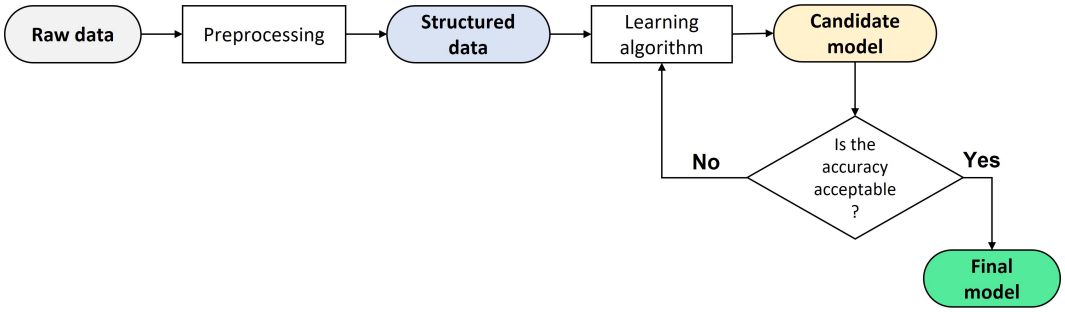


Figure 2.1: A general learning process of AI algorithms.

Artificial neural networks, often abbreviated as neural networks (NN), stand as one of the most widely used ML techniques. This technique has witnessed a remarkable success in dealing with complex problems, like object detection and recognition, natural language processing, etc (Liu, Wang, et al. 2017). Given the extensive deployment of NN in AI-based systems applied in the railway domain (Tang, De Donato, et al. 2022a; Yang, Wang, et al. 2022), we focus more on this techniques in the subsequent sections of the manuscript. Firstly, we provide a comprehensive background of NNs in the following section.

2.2.1 Basic Concepts

Neural networks have become a significant game changer in modern ML techniques. NNs are known for their efficient capabilities of learning and pattern identification, their high classification accuracy for testing datasets, their ability to generalize from the training data to unseen examples, and their adaptability to re-learn new patterns (transfer learning). As a results, NNs have found widespread and successful application to solve different classification and decision-making problems (Liu, Wang, et al. 2017).

NN are inspired by the way that the human brain works. Initially, the brain captures signals from the surrounding environment by means of the sensory organs likes eye, hand, tongue, etc. These signals are then propagated through a network of connected cells, namely neurons, enables the brain to process, interpret and output meaningful response about the received signals. Similarly, an NN has an input layer to receive data (using captures), process these data by propagating them through a set of connected computation units, namely nodes

or neurons, and returns the final result via an output layer.

While there are numerous types and architectures of NNs, our primarily focus is on Feedforward-Neural Networks (FFNN) (Bebis et al. 1994). This is because other types of NN are often derived from FFNN for specific application domains. In this dissertation, when there is no ambiguity, unless otherwise specified, we will use the term NN to refer to FFNN.

A FFNN, or shortly NN, is a set of connected layers $\{l_0, l_1, \ldots, l_n\}$, where l_0 is the input layer, l_n is the output layer, and $l_i : i \in \{1, 2, \ldots, n-1\}$ represents the set of hidden layers. Each layer $l_i : i \in \{0, 1, \ldots, n\}$ contains a set of nodes S_i. Thus, the set of nodes in the input and the output layers (vectors) are denoted by S_0 and S_n, respectively. The number of nodes within S_i is represented by $|S_i|$. We denote by s_{ij} the j^{th} node within the ordered set S_i.

For $i \in \{1, \ldots, n\}$, two successive layers l_{i-1} and l_i are connected via a weight-matrix W^i for all $0 \leq i \leq n$, such that: $w^i_{jk} = w(s_{i-1,k}, s_{i,j})$ is the weight value of the edge connecting the node $s_{i-1,k} \in S_{i-1}$ to the node $s_{i,j} \in S_i$, where $k \in \{1, 2, \ldots, |S_{i-1}|\}$ and $j \in \{1, 2, \ldots, |S_i|\}$ are indexes . A hidden layer l_i is associated with a bias vector b_i, such that b_{ij} is the bias of the node $s_{ij} \in S_i$. Figure 2.2 depicts an example of an NN of with three-dimensional inputs, two-dimensional outputs, and two hidden layers. In this example, the two hidden layers consist of four and three hidden neurons, respectively.

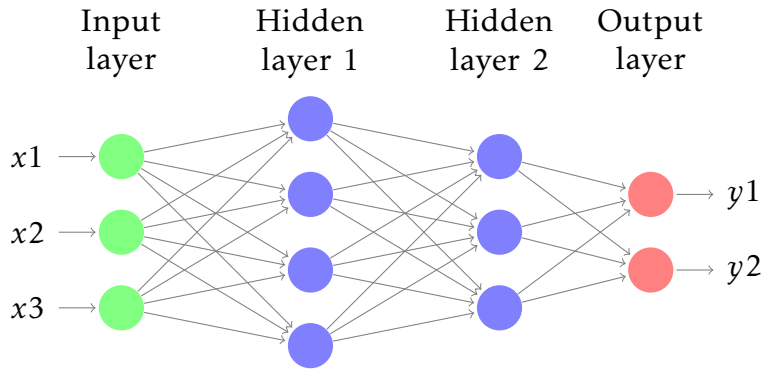


Figure 2.2: Example of a neural network.

Formally, a neural network of n layers can be seen as a function $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}^{|S_n|}$, where $|S_0|$ (resp. $S_n|$) represents the size of its input layer (resp. output layer). N is a composition of vector functions $z_i : i \in \{1, 2, \ldots, n\}$, such that: for an input $x \in \mathbb{R}^{|S_0|}$, $N(x) = z_n(z_{n-1}(\ldots(z_1(x))))$, where $z_i : \mathbb{R}^{|S_{i-1}|} \rightarrow \mathbb{R}^{|S_i|}$ is the

associated function of layer l_i, defined as:

$$z_i(x) = \alpha(\hat{z}_i) \tag{2.1}$$

where:

$$\hat{z}_i(x) = W^i \hat{z}_{i-1} + b_i \tag{2.2}$$

In Equation (2.1), the function α is called the activation function, and thus z_i and $\hat{z}_i$ are called activated and pre-activated values of x, respectively. In other words, for $i \in \{1, 2, \ldots, n\}$, N can be defined recursively as follows:

$$\begin{cases} z_0 = x \\ \hat{z}_i = W_i \times z_{i-1} + b_i \\ z_i = \alpha(\hat{z}_i) \\ N(x) = z_n \end{cases} \tag{2.3}$$

Similarly, we define $z_{i,j} = \alpha(\hat{z}_{i,j})$ as the activation value of a neuron $s_{i,j}$ (i.e., the j^{th} neuron of the i^{th} layer), where $\hat{z}_{i,j}$ is its corresponding pre-activation value. Figure 2.3 illustrates the operations performed by the hidden node s_{i1} of a layer l_i.

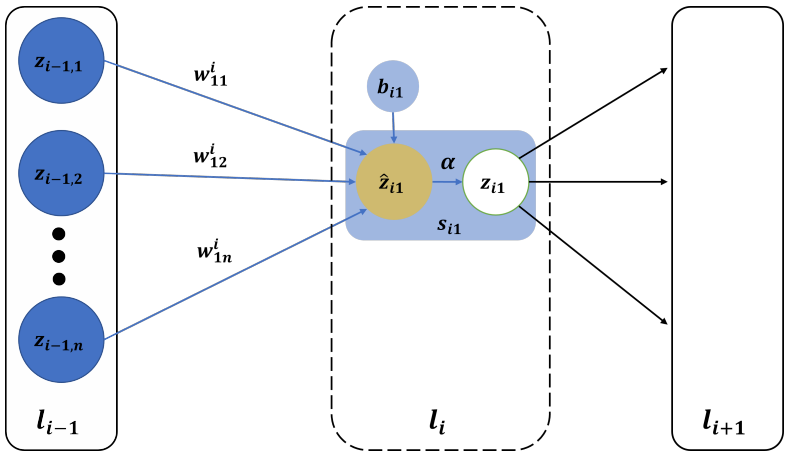


Figure 2.3: An illustration of the connection of a layer l_i to its preceding and succeeding layers.

There exist many used activation functions, and they are usually non-linear. In the following we provide the definitions of some of these functions (Nwankpa et al. 2018):

1. ***The rectified linear activation unit (ReLU)***:

$$\begin{aligned} &relu : \mathbb{R} \rightarrow \mathbb{R}^{+} \\ &relu(x) = max(0, x) \end{aligned} \tag{2.4}$$

2. ***The hyperbolic tangent activation function (Tanh)***:

$$\begin{aligned} &tanh : \mathbb{R} \rightarrow [-1, 1] \\ &tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \end{aligned} \tag{2.5}$$

3. ***The Sigmoid activation function (σ)***:

$$\begin{aligned} &\sigma : \mathbb{R} \rightarrow [0, 1] \\ &\sigma(x) = \frac{1}{1 + e^{-x}} \end{aligned} \tag{2.6}$$

As an ML technique, an NN learns to identify the mapping function that associates inputs with the corresponding outputs from the training data and aims to generalize this mapping relationship to make accurate predictions on unseen data. The training process of NNs involves several key steps. First and before starting the learning procedure, a set of parameters, namely hyper-parameters, must be determined. This includes defining the dimensions of the input and the output layers, determining the number of hidden layers and the number of nodes within each layer, choosing an appropriate activation function, and setting an initialization strategy to generate the initial weights matrices and biases vectors of the network. Next, an iterative learning procedure is performed to continuously fine-tune the weights and biases of the network. Within this procedure, the following steps are executed:

1. **Forward propagation:** the network receives an input from the training dataset and propagates it forward through the network's layers. The value of a neuron is computed based on this input data and the current weights and biases of the network. This process is repeatedly performed on each layer till the output layer.

2. **Loss function calculation:** the loss function calculates an error (also called loss) value measuring the difference between the predicted output by the network and the expected output (ground-truth). The goal of training a network is to minimize this error.

3. **Back-propagation:** to optimize the error calculated using a defined loss function, an optimization algorithm is used to fine-tune the weights and the biases of the network. Among the available algorithms, gradient descent optimization or its variants are often selected. During this phase, the gradient descent of the loss function with respect to each weight and bias is calculated. These weights and biases are updated in a way that reduces the error value.

The optimization process aims to find the optimal configuration of weights and biases that minimizes the loss value and allows the neural network to accurately generalize to unseen data. Once the network is trained, it is tested on a new data (the test dataset) to evaluate its performance and assess its ability to make accurate predictions on these samples. After testing, if the network achieves the desired performance on the test set, it is deployed to be used in real systems. For the verification phase, we assume that the network has been sufficiently trained and its parameters, specifically weights and biases, are fixed.

It is worth mentioning here that there exist various architectures and techniques that are derived from NNs, for instance, Convolution Neural Networks (CNN), Recurrent Neural Networks (RNN), Residual Neural Network (ResNet). (Liu, Wang, et al. 2017). In the following, we briefly present CNN and RNN, respectively.

1- Convolution Neural Networks (CNN)

CNN are designed to efficiently process data presented in matrices, making them well-suited in dealing with images, such as object detection and recognition (Li, Liu, Yang, Peng, et al. 2022).

A CNN is composed of multiple layers. Generally, it contains a sequence of *convolutional* layers followed by one or more dense layers (or fully-connected layers). While both the convolutional and dense layers play crucial roles in the CNN, the convolutional layers stand out as the fundamental building blocks of this architecture (Wu 2017). In fact, convolutional layers are designed specifically for processing data with grid-like structures (multidimensional matrices), such as images. They work by using kernels (also known as filters) that move systematically across the input data allowing for recognizing its (relevant) features and patterns. Similar to FFNN, a convolutional layer may have some biases. A convolutional layer is generally followed by a pooling layer (max-pooling, average-pooling, etc.). The purpose of introducing such operation is to reduce the spatial dimensions of the data while retaining its

essential features. Finally, the fully-connected layers process the high-level features extracted by the convolutional layers and learn to make predictions based on them. The output from these layers leads to the final classification or regression results. Figure 2.4 depicts an example of a CNN.

Notice that during training, the kernels' values of the convolutional layers, the weights of the fully-connected part of the network, and the existing biases are adjusted (Wu 2017).

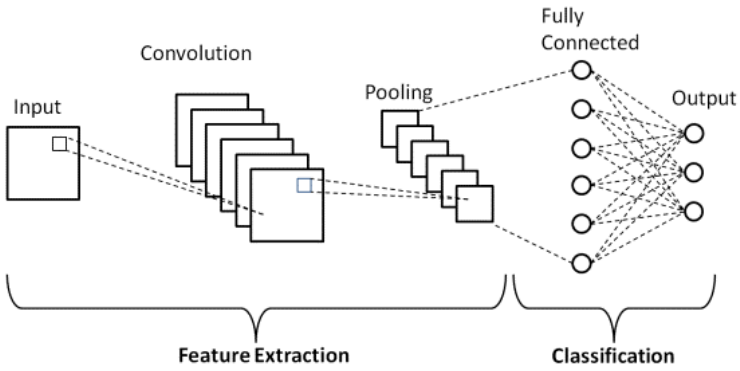


Figure 2.4: Example of a CNN.

2- Recurrent Neural Networks (RNN)

RNN are built in a way to save some information from previous layers for later use (memory-based NN). This is achieved via recurrent connections, which allow for creating a form of memory or temporal dependence. Additionally, and unlike traditional FFNNs, RNNs are capable of handling sequences of data with varying lengths, which makes this technique achieves astounding results across various domains such as speech recognition, machine translation, speech-to-text transcription, etc. (Lipton et al. 2015).

As depicted in Figure 2.5, an individual neuron within an RNN holds an internal memory, often referred to as the hidden state. This state is designed to store past values, which are subsequently employed in the computation of the current output. The process of an RNN begins with the initialization of the hidden state for each neuron.

When an RNN receives an input, presented as a sequence of data, it processes this data one element at a time. For instance, in natural language processing, a sentence is handled word by word. Upon processing each element within the input sequence, the hidden states of the neurons are updated based on the newly acquired information. Using these updated hidden states, the network

generates an output relevant to the current input element. In the context of natural language processing, this output might be a prediction of the next word in the sentence.

These fundamental steps are reiterated for each element of the input sequence. The hidden states are continually updated during each iteration and are persistently employed in subsequent iterations. Ultimately, after processing all elements in the sequence, the network produces its final output, which depends on the specific task at hand (Sherstinsky 2020).

It is important to note that these steps provide a high-level overview of how RNNs process sequential data. In practice, variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been introduced. More details about RNNs and its variants can be found in the survey of Lipton et al. (2015).

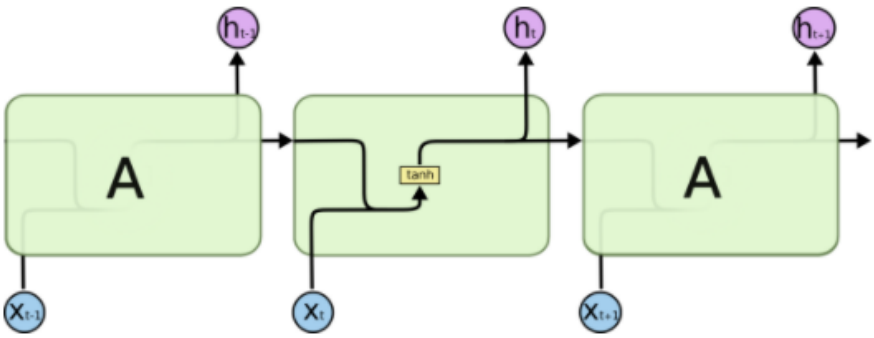


Figure 2.5: Example of an RNN.

NNs are widely used in different domains. For more details about different architectures of NNs and their applications, the reader can refer to Liu, Wang, et al. (2017). For instance, in image classification, the network is often called *image classifier*. A definition of an image classifier is provided in Definition 2.1.

Definition 2.1 *An image classifier is an NN used to classify images into predefined classes. Similar to NN, an image classifier is defined as a function* $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}^{|S_n|}$ *that maps an input image* $x \in \mathbb{R}^{|S_0|}$ *to classification label* $c_l = \underset{1 \leq j \leq |S_n|}{argmax}(N_j(x))$*, where* $N_j : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}$ *is the score function for the* j^{th} *class.*

Here, *argmax* stands for the "argument of the maxima" function. Such a function is used to find the class-index $j*$ that has the highest score N_j*, i.e., for the given input x, $N_j*(x) > N_j(x)$, $\forall j \in \{1, 2, \ldots, |S_n|\} \wedge j \neq j*$. In Definition 2.1, the input data (images) are in the form of multi-dimensional matrices. However, when

working with FFNN, a process called vectorization is performed to transform these multi-dimensional matrix inputs into equivalent vector inputs.

2.2.2 Sensitivity & Vulnerability of Neural Networks

Despite their successful achievements in solving various complex problems, recent studies have demonstrated that NNs are vulnerable, and a small perturbation on the inputs can lead the network to make incorrect and unpredictable decisions, even for an NN with high accuracy metrics (Szegedy et al. 2014). For instance, given an image classifier $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}^{|S_n|}$ and an input-image $\hat{x}$ from class c_l (its real class). Although the classifier N has correctly classified $\hat{x}$ as class c_l, i.e., $\underset{1 \leq j \leq |S_n|}{argmax}(N_i(\hat{x})) = c_l$, it may be easily fooled to misclassify a perturbed image of $\hat{x}$ by introducing a small and imperceptible perturbation ϵ (Kurakin et al. 2018; Xu, Ma, et al. 2020). This can be mathematically expressed as:

$$\exists x : \|x - \hat{x}\|_p \leq \epsilon \wedge \underset{1 \leq j \leq |S_n|}{argmax}(N_i(\hat{x})) = c_l \wedge \underset{1 \leq j \leq |S_n|}{argmax}(N_i(x)) \neq c_l \tag{2.7}$$

where $\|.\|_p$ refers to the distance measurement using an L_p norm[1], and $\epsilon \in \mathbb{R}^+$ is the perturbation's threshold.

Due to the introduced perturbations, the obtained inputs are called adversarial examples. Such inputs can occur naturally as a result of various factors such as weather conditions, occlusions, or distortions in the environment, etc. For example, in the context of object detection systems, if a camera is partially obstructed or affected by adverse weather conditions like rain or fog, the resulting images may contain perturbations that can lead to misclassifications and false decision. Similarly, in autonomous driving scenarios, environmental factors like shadows, reflections, or lighting conditions can introduce perturbations that impact the performance of the object detection system. In addition to this natural perturbation, there are also deliberate adversarial examples carried out by human-beings. This type of adversarial examples is often referred to as adversarial attacks. Such attacks involve intentionally manipulating the input data to exploit the vulnerabilities of the NN and deceive it into making incorrect predictions. Adversarial attacks can be

[1] An L_p norm measures the distance between to entities using the p-norm distance. For instance L_1 and L_2 refers to as the Manhattan and the Euclidean distance measurements, respectively.

applied to various domains, such as image classification, speech recognition, or natural language processing (Szegedy et al. 2014; Xu, Ma, et al. 2020).

The sensitivity and vulnerability of neural networks raise concerns about the trustworthiness of these models when deployed in real-world systems. This issue becomes particularly crucial in the context of safety-critical systems, such as autonomous driving (Besinovic et al. 2022) and medical diagnosis (Litjens et al. 2017), where the correct functioning of NN is not only desirable but imperative. To address this issue, several approaches have been proposed by adapting formal verification methods to provide more safety assurance of NN (Huang, Kroening, et al. 2020; Urban et al. 2021).

In the subsequent sections, we will first discuss formal verification methods in a general context, and then we will focus on how these techniques can be adapted for verifying NN.

2.3 Formal Methods & Formal Verification

Formal Methods (FMs) is a scientific discipline that includes a set of rigorous mathematical techniques and approaches used to specify, model, design and verify software and hardware systems. FMs can be used across all stages of a system life-cycle, including requirements engineering, architecture, design, implementation, testing, and maintenance (Clarke and Wing 1996). With the growing utilization and increasing complexity of both hardware and software systems, the probability of encountering subtle errors has significantly increased. Moreover, such errors may be the source of money and time losses, and more importantly they can even affect human lives such as in safety-critical systems (SCS)[2] (Grimm et al. 2018). Indeed, the design, development and operation of safety-critical systems are subject to rigorous safety standards and regulations. Such standards and regulations highly recommend (sometimes require) the use of FMs to minimize the risk of failures that could lead to catastrophic losses (CENELEC-EN50128 2011). Consequently, FMs have extensively applied to increase the confidence in the correctness of SCS. For instance, FMs are commonly used to ensure the safety of railway systems (Ferrari et al. 2022), nuclear power plant control systems (Lawford et al. 2012), and automotive and aviation systems (Woodcock et al. 2009).

Formal verification methods are used to formally check whether a system

[2]Safety-critical systems are systems in which the consequences of a failure or a malfunctioning can harm human lives.

(represented by its model) satisfies a given specification or property. This is achieved by either providing a proof of correctness of the property, or by generating a counterexample witnessing the violation of the property at hand. To this end, the formal verification procedure of a system often includes: modeling the system's behavior, formalizing the desired specification, and applying a verification technique or engine to check the property on the system's model (Clarke, Henzinger, et al. 2018).

1. **System modeling:** this step involves building a model M which accurately describes the system's behavior. The model should capture the system's structure and the interactions between its components in a formal and precise manner. Various modeling languages can be employed in formal methods, depending on the nature of the system, the type of properties to be verified, and the level of required details. A system can be expressed using Finite-state machines (FSMs), Petri nets, or programming languages with formal semantics. The choice of the modeling language is crucial, and it should be adequate to describe the system's behavior, and formally express the specifications to be verified.

2. **Specification formalization:** this step consists in formalizing the desired specification as mathematical property p. The specifications are expressed using appropriate mathematical notations, such as temporal logics (e.g., LTL, CTL, etc.).

3. **Verification algorithm:** formal verification algorithms (or techniques) are applied once the system model and the property are formally expressed. Such algorithms aim to check whether the system model M satisfies some property p:

$$M \models_? p \tag{2.8}$$

   The verification technique often prove that the property p holds on M, i.e., $M \models p$, or provides a counterexample (i.e., a scenario in the system behavior) demonstrating that $M \not\models p$.

The current state of the art in the field of formal verification, encompassing both academic and industrial sectors, comprises a multitude of verification techniques that are supported by software tools, including formal verification techniques (or algorithms). For instance, Model Checking (MC), Theorem Proving, Satisfiability Modulo Theories (SMT), and abstract interpretation are

successfully applied to verify many complex systems (Fantechi et al. 2012; Garavel 2012). In the sequel, we briefly present each of these techniques.

1. **Model Checking (MC):** is a widely used formal verification technique that involves exhaustively exploring the state space of the system model to check whether the investigated properties hold in all possible states. It provides a systematic way to verify properties such as safety and liveness in a finite-state or temporal logic framework (Clarke, Henzinger, et al. 2018).

2. **Theorem Proving:** is another formal method that employs mathematical logic and reasoning to construct formal proofs. It allows for the establishment of system properties by using formal deduction rules and logical inference (Davis et al. 1962).

3. **Satisfiability Modulo Theories (SMT):** is a problem-solving technique used to determine the satisfiability of logical formulas. SMT is a generaliztion of the boolean Satisfiability probelm solving (SAT) by handling multiple theories and formulas, involving real numbers, integers, etc. SAT/SMT solvers play a crucial role in formal methods, especially in the context of automated reasoning and verification, where they are used to find solutions or counterexamples to logical formulae (Biere et al. 2009).

4. **Abstraction interpretation:** is a technique that aims to analyze systems by abstracting certain details or aspects of the system while preserving important properties. It allows for the efficient analysis of complex systems by considering a simplified or abstract representation (Cousot et al. 1977).

The depicted diagram in Figure 2.6 outlines the general procedure of applying formal verification to check whether a system satisfies a set of specifications[3]. If $M \models P$, the procedure outputs a correctness proof of p on M, otherwise, a counterexample is (often) generated as a demonstration that the property does not hold. In the case of MC, the system's model is often a transition system (i.e., a finite state automaton), and the properties are formulated using temporal logic. The verification algorithm exhaustively explores all states to check if the property holds. On the other hand, in Theorem Proving, both the model and the property are expressed using formal logic. Instead of exploring states and transitions as in MC, Theorem Proving aims to construct a formal proof using logical inference (Ouimet et al. 2007).

[3]As mentioned, the figure represents a general procedure; additional details can be incorporated for each category of FMs.

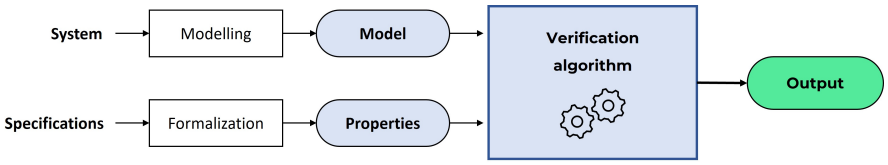


Figure 2.6: Illustration of the overall process of formal verification methods.

Depending on the provided level of certainty and exhaustiveness in determining whether a model satisfies a given property, formal verification methods can be grouped into two main categories: complete methods and incomplete methods.

1. **Complete methods:** this category of verification methods aims to provide a deterministic answer (i.e., property is satisfied or not) by modeling the exact and complete behavior of the system (i.e., without abstraction). When a complete method indicates that a property is valid, it means that this property holds under all possible scenarios (exhaustiveness). Similarly, if the method returns that the property does not hold, it provides an actual counterexample that demonstrates the property violation on at least a path or a scenario in the model. A general process of complete verification methods is illustrated in Figure 2.7. These methods exhaustively analyze all possible states or behaviors of a system to prove whether the property is satisfied or not. Thus, they are computationally expensive and may not be feasible for large or complex systems (due to combinatorial explosion).

2. **Incomplete methods:** these methods are designed to offer improved computational efficiency and scalability compared to complete methods, addressing the resource-intensive nature of the latter. They often involve the generation of an abstract model (i.e., reduced model), which represents an abstraction of the system's behavior. These incomplete methods guarantee that whenever a property holds on the abstract model, it must hold on the initial original model. However, owing to the inherent approximation aspects, if an incomplete method indicates that a property does not hold on the abstract model, it does not necessarily mean that the property is false in the original model. Indeed, the generated counterexample when the property does not hold may be spurious and not a real counterexample in the original model (Cousot et al. 1977). As a result, a refinement step is typically integrated into these methods. This additional step attempts to reduce the abstraction level, thereby reducing

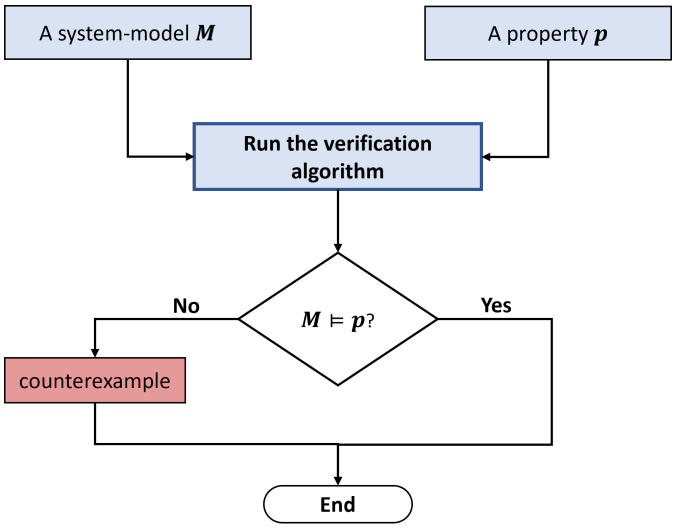


Figure 2.7: Complete formal verification - General overview.

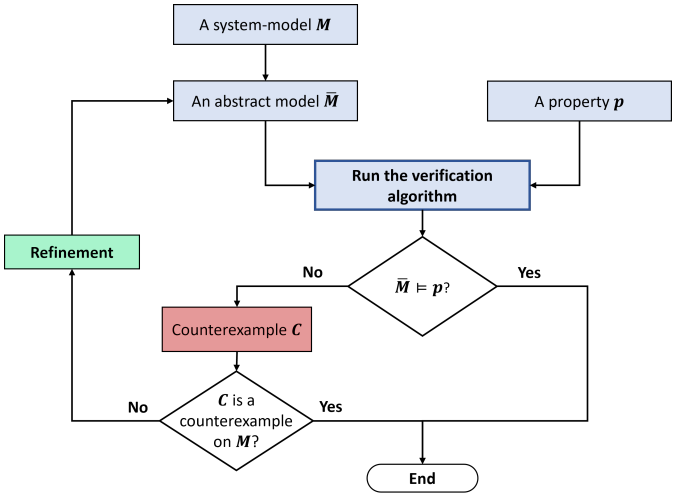


Figure 2.8: Incomplete formal verification - General overview.

the generated spurious counterexamples and enhancing the precision of the generated abstract model. Figure 2.8 depicts the general procedure of incomplete verification methods.

In the subsequent section, we will explore how these formal verification methods are adopted to assess different properties of neural networks.

2.4 Verification and Monitoring of Neural Networks

Despite the widespread use of NNs in various domains, deploying them in safety-critical systems raises many concerns due to the high safety and dependability requirements of these systems. Thus, there is a growing demand

on developing rigorous methods for verifying and evaluating NNs. This demand becomes particularly crucial given the vulnerability and the sensitivity of NN to adversarial attacks.

In this context, several techniques have been suggested in order to ensure the safety and the dependability of NN. These techniques can be classified, with respect to the system life-cycle, into two categories, which are:

1) **Offline evaluation (during the design phase):** this category includes testing and formal verification of NNs. Testing involves checking the performance of the trained network on a set of data known as test set. On the other hand, NN verification consists of applying formal verification methods to formally prove (or disapprove) the correctness of a set of properties on the network. Both testing and formal verification are performed before the deployment of the network. Indeed, the network must have an acceptable accuracy (the percentage of outputs that match the predicted ones), and verify the defined formal properties before its deployment (Huang, Kroening, et al. 2020).

2) **Online evaluation (during the runtime phase):** techniques that belong to this category are often called monitoring techniques. Monitoring allows for real-time tracking of the behavior and the performance of the network during runtime. The purpose of monitoring is to identify any abnormal behavior or degradation in performance exhibited by the network. A monitoring system typically consists of a pre-processing phase and an execution phase. During the former phase, a monitor module is developed. This module defines the metrics, properties, or criteria that will be used to supervise the network's behavior and performance. It establishes a set of guidelines (rules) to distinguish normal or expected behaviors of the network from those that are considered outliers or abnormal. In the execution phase, the monitor module is executed in parallel with the network. It continuously observes the inputs, outputs and internal states of the network, and then analyzes them in real-time. The monitoring system compares the observed behavior against the predefined metrics or criteria to assess the network's performance. If any anomaly or abnormal behavior occurs, the monitoring system raises warnings to immediately notify the decision and control unit (Cheng, Nührenberg, and Yasuoka 2019).

After this brief introduction to NN evaluation, the following sections provide

further details about these approaches and review the related scientific literature, with a main focus on NN verification and NN monitoring.

2.4.1 Neural Networks Verification

The evaluation of NNs has primarily centered around traditional testing. NN testing consists in evaluating the performance and the behavior of a trained network using a set of data samples. These data, referred to as the "test dataset", consist of new samples that the network has not encountered during its training phase. During testing, the network takes inputs from the test dataset iteratively and produces corresponding outputs (predictions or classifications). These outputs are then compared to the actual/expected outputs (ground-truth) of the test dataset. Finally, after processing all the input data, the accuracy of the network is calculated to measure its performance. This metric is a rate (percentage) calculated by dividing the number of correct predictions by the total number of samples (or the size of the test dataset) (LeCun et al. 2015).

Another interesting set of methods focuses on testing NNs against corner cases inputs. The generation of such inputs, namely adversarial example (Section 2.2.2), is based on sophisticated algorithms such as Fast Gradient Sign Method (FGSM) (Goodfellow, Shlens, et al. 2014), Basic Iterative Method (BIM) (Kurakin et al. 2018), Projected Gradient Descent (PGD) (Madry et al. 2017), Carlini and Wagner (CW) attacks (Carlini et al. 2017), and others (Akhtar et al. 2021; Chakraborty et al. 2021). Similarly to traditional testing, the goal of this family of techniques is to check the performance of the network on these generated samples.

While both testing on a data set and against adversarial examples remain important and essential steps in evaluating NN, they both fail to provide formal guarantees about the absence of inputs that may fool the network. Indeed, testing does not cover the whole behavior of the network since it is restricted to a limited test set (lack of exhaustiveness). Indeed, a tiny perturbation of inputs can lead the network to drastically change the corresponding output. For instance, adding a small imperceptible perturbation to an image may lead the network to misclassify the perturbed image. Therefore, formal verification of NN has become crucial to complete and support the traditional testing in providing confident guarantees about the NN performances and behaviors. Indeed, these methods aim to obtain formal guarantees about the correctness of the model with respect to a set of desired properties.

Generally, formal verification consists of checking whether a model M of the considered system satisfies a defined property p.

$$M \models_? p \tag{2.9}$$

As discussed in the previous section, this involves three main steps: (*i*) modeling the system, (*ii*) formalizing the desired specification, and finally (*iii*) applying an adequate verification algorithm to check whether or not the system model M satisfies the property p.

Similarity, NN verification consists of modeling the network's behavior, expressing the properties using mathematical notations (and languages), and then applying a verification engine to check whether the property p holds on the network's model N, or not. This is formulated as:

$$N \models_? p \tag{2.10}$$

In the sequel, we outline various formulations of NN properties, and then we discuss different approaches to solve and verify them.

2.4.1.1 Expressing NN Properties for Verification

As highlighted in the survey conducted by Leofante et al. (2018), the works and studies regarding the verification of NNs have mainly focused on three fundamental types of properties:

1. **Invertibility:** this property considers one single network N, an input and an output regions defined by their respective constraints pre and $post$. For any given output $y = N(x)$ that satisfies the constraints $post$, the property expresses whether the corresponding input x satisfies the set of constraints pre. This property is formulated as:

$$\forall y, \exists x : (post(y) \wedge y = N(x)) \implies pre(x) \tag{2.11}$$

2. **Invariance:** this property also focuses on verifying a single network N with respect to a set of constraints pre and $post$ that define an input region and an output region, respectively. The objective of this property is to ensure that for all inputs that satisfy the constraints pre, their corresponding outputs generated by N also satisfy the constraints $post$. Equation (2.12)

provides the formal definition of the invariance property.

$$\forall x,\ \forall y : (pre(x) \wedge y = N(x)) \implies post(y) \tag{2.12}$$

3. **Equivalence:** while the invertibility and the invariance properties consider only one network, the equivalence property involves two networks N_1 and N_2. The aim of this property is to check whether these networks are equivalent with respect to some defined input and output regions. For an input region defined by a set of constraints pre, and an output region defined by a set of constraints $post$, we seek to check if: for all inputs satisfying the input-constraints pre, and their corresponding outputs obtained using N_1 and N_2 satisfy the output-constraints $post$, are the outputs of N_1 and N_2 equal? In other words:

$$\begin{aligned} &\forall x,\ \forall y_1,\ \forall y_2 : \\ &(pre(x) \wedge y_1 = N_1(x) \wedge post(y_1)) \\ &\wedge y_2 = N_2(x) \wedge post(y_2)) \implies y_1 = y_2 \end{aligned} \tag{2.13}$$

It is important to note that the invariance property, also known as input-output relation property, is the most used in NN verification. This property is used to verify safety property such as properties on Aircraft Collision Avoidance System (ACAS Xu) system (Katz, Barrett, et al. 2017; Kochenderfer 2015), as well as to check the robustness of image classification networks (Meng et al. 2022). The set of constraints pre and $post$ can take various forms. However, due to the current limitations of available verification tools, these constraints are often expressed through first-order logic formulas. Thus, the regions defined by the set of constraints pre and $post$ have often polytopic forms[4].

For image classifiers, that are networks used in classifying images into some pre-defined classes, the robustness property is widely investigated to verify that the network is robust against adversarial examples (see Equation (2.7)). A classifier N is considered robust for an input image x_0 of a class c_i, if it consistently assigns the label c_i to all inputs within a small region surrounding

[4]A polytope is a generalization of a polygon (2D) and a polyhedron (3D) to higher dimensions. It is defined by a finite set of linear inequalities or equations that specify its shape.

x_0. The robustness verification problem can be formulated as:

$$\begin{cases} For \;\; x_0 \in D_x : N(x_0) = c_i \\ pre(x) : \|x - x_0\|_p \leq \epsilon \implies post(y) : y = c_i \end{cases} \tag{2.14}$$

In Equation (2.14), $\epsilon \in \mathbb{R}^+$ represents the intensity of the perturbation, and $\|.\|_p$ is the used norm for calculating the distance between x_0 and x, e.g., the infinity norm ($\|.\|_\infty$) is defined as: $\|X\|_\infty = \max_{x \in X}(|x|)$ (Huang, Kroening, et al. 2020).

In contrast to testing using adversarial examples, robustness verification offers more formal assurance of the network's stability in classifying inputs within the region surrounding a given input $x_0 \in D_x$. This region, also called the robustness region of x_0, is controlled by two parameters: the $p-$norm and the magnitude ϵ.

2.4.1.2 NN Modeling for Verification

The verification of a system relies on how accurately the used model represents its behavior. In other words, using model-based techniques, the verification is only as good as the modeling of the system (Grimm et al. 2018). In NN verification, and depending on whether we exactly or approximately model the behavior of the network, there exist two main categories of NN verifiers. A *complete (exact)* verifier encodes the exact behavior of the network, and thus can solve exactly the verification problem by returning either a confirmation of the correctness of the property, or a counterexample witnessing its violation (Figure 2.9). On the other hand, an *incomplete* verifier soundly over-approximates (or abstracts) the NN behavior by relaxing and/or linearizing the NN operations. The over-approximation guarantees that the property holds on the original network whenever it holds on the abstract one. However, when the property does not hold on the abstract network, an incomplete verifier may not be able to conclude if the property is truly violated on the original network. Thus, incomplete verification methods have generally an additional step, called refinement, which allows for refining the abstraction and re-generate a more precise abstract network (Figure 2.10).

Since ReLU has been the main focus of most of the existing NN verification methods, in this dissertation we focus on ReLU-NNs. Given a ReLU neuron $s_{ij} \in S_i$, where $l_{ij} \leq \hat{z}_{i,j} \leq u_{ij}]$ is its pre-activation value, the activation status of the neuron s_{ij} can be either stable or unstable, such that:

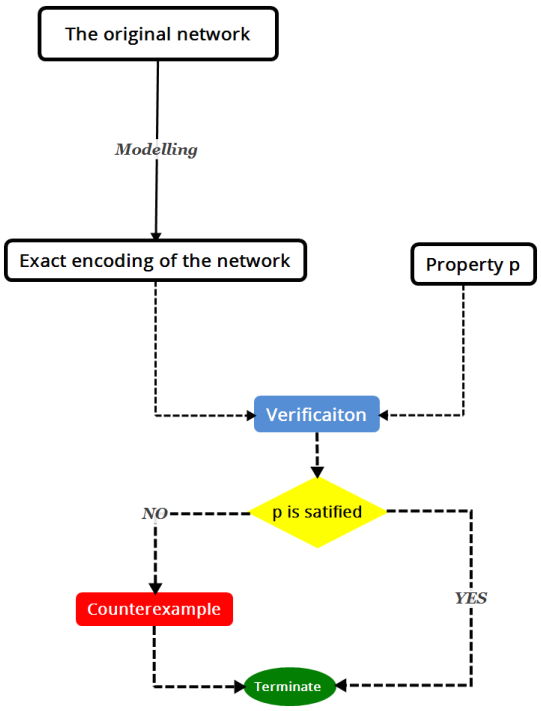


Figure 2.9: Complete formal verification of neural networks.

1. **Stable**: if $l_{ij} \geq 0$ or $u_{ij} < 0$, the activated value of s_{ij} is either $z_{ij} = \hat{z}_{ij}$, or $z_{ij} = 0$, respectively. In both cases, the ReLU function can be safely replaced by the respective linear formula, namely $ReLU(\hat{z}_{ij}) = \hat{z}_{ij}$ or $ReLU(\hat{z}_{ij}) = 0$, respectively.

2. **Unstable:** if $l_{ij} \leq 0$ and $u_{ij} \geq 0$. In this case no exact linear formula can be derived to replace the ReLU function; and thus at least two linear formulas are required to represent the ReLU function.

In ReLU-based NN verification, the completeness of the verification process is generally assessed based on how the verifier handles the stability or not of the ReLU neurons. A comprehensive discussion about complete and incomplete verification of NNs is given below.

a) Complete methods:

Modeling an NN was initially based on the encoding of its behavior as a system of constraints (i.e., linear programming) and then using off-the-shelf tools for verification. As explained in Section 2.2, each neuron $s_{ij} \in S_i$ has two different operations:

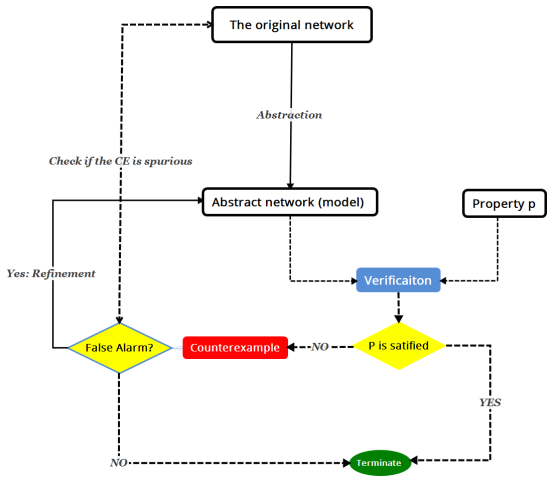


Figure 2.10: Incomplete formal verification of neural networks.

- The linear affine transformation: $\hat{z}_{ij} = \sum_{s_{i-1,k} \in S_{i-1}} (w(s_{i-1,k}, s_{ij}) \times v(s_{i-1,k})) + b_{ij}$.
- The activation function: $z_{ij} = \alpha(\hat{z}_{ij})$.

While the linear affine transformation can be directly encoded, the challenge of encoding a network as a linear programming problem is mainly related to the non-linearity of the used activation function α.

SAT/SMT-based NN verifiers: Historically, the interest in verifying NNs dates back to 2000s (Taylor 2006); however, the first complete and concrete work in this area was proposed in 2010 (Pulina et al. 2010). In this work, the authors introduced a method based on SAT/SMT for NNs with *Sigmoid* activation function. The method relies on splitting the *Sigmoid* function in linear segments, while ensuring that the actual values of the *Sigmoid* function lay within these segments. Each segment is considered as a linear problem that is solved using a SAT/SMT solver. Following that work, the authors conduced a comparison study of different SMT solvers for their approach on a set of networks with different sizes (Pulina et al. 2012). They concluded that the splitting of the *Sigmoid* function leads to the generation of numerous linear sub-problems. The number of linear problems to be solved by SAT/SMT increases exponentially with the number of hidden neurons, which raises the scalability issue of this approach and limits its application to relatively small NNs.

Since ReLU is a linear piece-wise function[5], most of works focused their studies on verifying NN with this activation functions. Modeling a ReLU-

[5]A piece-wise function is function defined on several pieces or segments of intervals or subdomains. When each "piece" or segment of the function is a linear equation, the function is called linear piece-wise.

NN relied on splitting each ReLU neuron into two linear constraints that correspond to the two linear regions of ReLU, and then employ an adequate solver for verification. For instance, the application of SAT/SMT solvers has been extensively studied (Huang, Kroening, et al. 2020). An example of encoding a ReLU-NN of n layers $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}$ as a SAT/SMT problem is presented in Equation (2.15) (Bunel et al. 2018). In this example, the objective is to verify that the output of the network is always positive while considering some constraints on the inputs.

$$lb_i \leq x_i \leq ub_i \quad \forall i \in \{1, \ldots, |S_0|\} \tag{2.15a}$$

$$\hat{z}_i = W_i \times z_{i-1} + b_i \quad \forall i \in \{1, n-1\} \tag{2.15b}$$

$$z_i = \max(0, \hat{z}_i) \quad \forall i \in \{1, \ldots, n-1\} \tag{2.15c}$$

$$z_n < 0 \tag{2.15d}$$

Equation (2.15a) and Equation (2.15d) represent the input constraints (*pre*) and the negation of the output constraints (*post*), respectively. Equation (2.15b) encodes the affine transformation of the network, and Equation (2.15c) encodes the ReLU activation function. An assignment that satisfies all the problem's constraints represents a valid counterexample witnessing the violation of the property. In case the problem is UNSAT, we can conclude that the property holds on N. Note that, in the worst case, Equation (2.15c) leads to split every ReLU neuron into two linear constraints. Consequently, the complexity of the verification problem grows exponentially with the number of ReLU neurons within the network, which makes the verification of ReLU-NN an NP-complete problem. Figure 2.11 depicts the problem of ReLU-NN verification state explosion.

Katz, Barrett, et al. (2017) introduced Reluplex as a verification method for ReLU-NNs. Reluplex is an adjusted version of the Simplex optimization method to handle the ReLU constraints. Moreover, Reluplex applies a SAT solver for splitting the ReLU neurons into linear constraints. An improved version of Reluplex is proposed by Katz, Huang, et al. (2019), namely *Marabou*. By introducing input region's splitting and developing their own SMT solver instead of using an external SMT solver, the authors claim that Marabou is faster and can be applied to verify larger networks. Huang, Kwiatkowska, et al. (2017) proposed an SMT-based method to prove that a classifier is robust and no adversarial example exists within the neighborhood of the given input. The approach relies on discretizing the infinite neighborhood region into a reduced

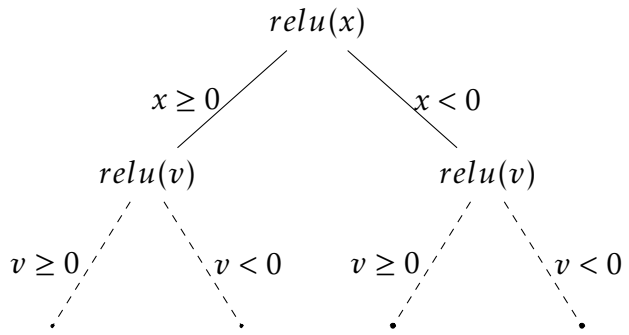


Figure 2.11: An example showing the state-space explosion of ReLU-NN verification.

finite set of points. Then, by means of an SMT solver, the set of constraints linking the representative points between layers is propagated through the network, to finally check that the network provides the same output (the same class). Ehlers (2017) introduced "*Planet*", a verification method for ReLU-NN, which encodes the network's behavior as a combination of linear constraints. The ReLU constraints are approximated using three linear constraints and the verification problem is transformed into a satisfiability problem by taking the negation of the desired safety property. In one of the most recent works, Duong et al. (2023) presents NeuralSAT, an SMT framework for NN verification. In this work, the DPLL(T)[6] algorithm (Davis et al. 1962), which is widely used in modern SAT/SMT solvers, is redesigned to be more appropriate for NN verification. The framework includes clause learning, abstraction, and theory solving. NeuralSAT is claimed to be sound and complete, and it supports piece-wise linear activation functions and different types of neural networks: FFNN, CNN and residual networks (He et al. 2016).

While the above mentioned SAT/SMT-based methods generally deal with FFNN and CNN with pieace-wise functions (e.g., ReLU and maxpoling), Narodytska et al. (2018) introduced an exact encoding of Binary Neural Networks[7] (BNNs) as Boolean formulas. Then, the verification problem is handled by means of a SAT solver. The approach is limited to BNN with only binarized components. Recently, Amir et al. (2021) presented an extension of the

[6]DPLL stands for "Davis-Putnam-Logemann-Loveland", which is a widely used algorithm for solving the Boolean satisfiability problem (SAT).

[7]Briefly, a Binary Neural Network (BNN) is a type of neural network in which the network's weights and activations are binary values. Please check the survey of Yuan et al. (2023) for more details about BNNs.

Reluplex method (Katz, Barrett, et al. 2017) that can handle both binarized and non-binarized components of BNN. A parallelization of the verification process is also proposed to enhance the scalability of the approach.

MILP-based NN verifiers: MILP-based approaches transform the network behavior and the property to be verified to a mixed integer linear problem. The encoding of the investigated property (see Equation (2.10)) is straightforward. However, the challenging part is related to the encoding of the network's behavior, particularly the activation function behavior. Several encodings have been proposed for the ReLU activation function.

Let us consider a network $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}$ of n layers and one output (see Equation (2.3)), and we assume that we want to verify that for a bounded input, the output must be positive. Tjeng et al. (2019) introduced a MILP encoding for ReLU-NN using binary variables to handle the ReLU constraints:

$$lb_i \leq x_i \leq ub_i \quad \forall i \in \{1,\ldots,|S_0|\} \tag{2.16a}$$

$$\hat{z}_i = W_i \times z_{i-1} + b_i \quad \forall i \in \{1, n-1\} \tag{2.16b}$$

$$\delta_i \in \{0,1\}^{|z_i|}, \quad 0 \leq z_i \leq u_i \times \delta_i, \quad \hat{z}_i \leq z_i \leq \hat{z} - l_i \times (1-\delta_i) \quad \forall i \in \{1,\ldots,n-1\} \tag{2.16c}$$

$$\min(z_n) \tag{2.16d}$$

Equation (2.16a) denotes the input constraints, where lb_i and ub_i is the lower and upper bounds of the i^{th} element of the input x. Equation (2.16b) represents the affine transformation performed by N. Equation (2.16c) represents the encoding of the ReLU function using binary variables (δ_i). The variables u_i and l_i represent an estimation of the upper and lower bounds of z_i, respectively. Interval or symbolic bounds propagation algorithms (Wang et al. 2018b; Xiang et al. 2020) can be used to estimate their values. Finally, Equation (2.16d) is the objective function of the MILP optimization problem. If the solver returns a negative solution to this problem, i.e., $\min(x_n) \leq 0$, we can conclude that the property does not hold, and the corresponding input is a valid counterexample. Otherwise, we conclude that the given property does hold on the network N.

Cheng, Nührenberg, and Ruess (2017) proposed an alternative MILP encoding of the ReLU-NN verification, known as Big-M encoding. The developed approach is then used to solve the global robustness of the given network: the approach seeks to find the largest neighborhood of an input where no adversarial example exists. The encoding of the input constraints, the affine transformations and the objective function are the same as presented in Formula (2.16a), (2.16b), and (2.16d), respectively. However, the Equation (2.16c) was updated as

presented in Equation (2.17).

$$\delta_i \in \{0,1\}^{|z_i|},\ \ 0 \leq z_i \leq M_i \times \delta_i,\ \ \hat{z}_i \leq z_i \leq \hat{z} - M_i \times (1-\delta_i)\ \ \forall i \in \{1,\ldots,n-1\} \quad (2.17a)$$

where $M_i = \max(u_i, -l_i)$

These two formulations of the NN verification problem are then adapted to check local robustness property, adversarial examples identification (Fischetti et al. 2018; Tjeng et al. 2019), and reachability analysis (Dutta et al. 2018; Lomuscio et al. 2017). Similarly to SAT/SMT based verification, the straightforward application of MILP is computationally expensive. Thus, some works address this issue by applying optimization algorithms to reduce the number of generated linear instances (Fischetti et al. 2018; Lomuscio et al. 2017; Tjeng et al. 2019). For instance, Botoeva et al. (2020) analysed the dependency relation between the ReLU neurons to reduce the MILP search space. Concretely, the dependency analysis allows for fixing the state of ReLU neurons (positive or negative), and thus help pruning the search tree generated by MILP. The performance of the approach is further improved when combined with input domain splitting and symbolic interval propagation (Wang et al. 2018b).

Branch and Bound (BaB) based methods were also proposed to accelerate the verification process of NNs. BaB is adapted for ReLU-NN by using Linear Programming (LP) to establish bounds, and ReLU neuron split refinement for branching (Bunel et al. 2018; Hashemi, Kouvaros, et al. 2021; Henriksen et al. 2021). Moreover, some approaches exploring branching over the input domain (Wang et al. 2018a).

These methods rely on extensively explore the search space, which can limit their scalability when applied to large networks. To address this issue, incomplete methods that offer a trade-off between precision and scalability are often employed. In the following section, we will review and discuss these alternative methods.

b) Incomplete methods:

As discussed above, splitting each neuron into linear regions is computationally expensive and increases the complexity of the verification problem exponentially with the number of neurons to split. For example, for a ReLU-NN of n neurons (considering all hidden layers), splitting all its hidden neurons leads to generate 2^n sub-problems (see Section 3 for more details). One way to overcome this

issue is to resort to approximation and abstraction methods. There are two main categories of methods: abstraction of the activation function, and model reduction. The main idea of the former is to use linear constraints instead of the exact activation function to avoid the splitting. In order to ensure the soundness of these approaches, the linear constraints must over-approximate the activation function at hand (Huang, Kroening, et al. 2020; Urban et al. 2021). For instance, techniques based on abstract interpretation (Gehr et al. 2018; Singh, Gehr, Mirman, et al. 2018) and reachability analysis (Wang et al. 2018a,b; Xiang et al. 2020) are successfully applied to verify some NN models.

The second sub-category of abstraction methods consists in reducing the NN size by merging its neurons. These methods are generally referred to as model reduction, and their goal is to construct an abstract (reduced) network that constitutes a sound over-approximation of the original one (Boudardara, Boussif, Meyer, et al. 2023a). This helps verifying properties on the reduced model in less computation time, and infer their correctness on the original one.

In Chapter 3, we will review in more details abstraction methods, including both activation function abstraction and model reduction methods.

2.4.2 Neural Networks Monitoring

While verification focuses on providing some guarantees on a network prior to its deployments, monitoring tends to continuously track the NN behavior and performance during runtime. The primary goal of monitoring is to detect potential errors, deviations, or incorrect behavior exhibited by the network in real-time, where ground-truth labels are not available. Before diving into the application of runtime monitoring for NNs, let first introduce runtime monitoring in a general context.

Runtime monitoring is an old topic. It may be presented with various alternative names, such as runtime analysis, runtime verification, or runtime checking (Watterson et al. 2007). Runtime monitoring has been introduced as a tradeoff between formal verification and testing. Indeed, formal verification methods are known to be computationally demanding, which makes them hard to apply on large-scale applications. Conversely, testing, typically reliant on a set of test cases, proves to be a time-consuming process, compounded by the limitation that not all errors are detectable during the development phase. Consequently, runtime monitoring is often employed to complement formal verification and testing, aiming to increase the safety assurance by continuously

tracking the behavior and the performance of the system (Leucker and Schallhart 2009).

An online monitoring system relies on two main phases, (*i*)building the monitor and then (*ii*) using the monitor in real-time.

1. **Building the monitor (offline):** this phase is performed during the development of the monitoring system. It consists in defining properties and measurement metrics that are used to implement the core (algorithms) of the monitor. This enables the monitor to observe and analyze the runtime behavior of the system.

2. **Using the monitor in real-time (online):** after building and validating the monitor, it is used in real-time to observe and analyse the behavior of the system. The monitor is executed in parallel with the system allowing for raising alarms for potential misbehavior, errors of the system decisions, or performance insufficiency.

Figure 2.12 illustrates the general process of monitoring, including the two main phases presented above. In some cases, further information may be needed in order to build the monitor, e.g., details about the architecture and the internal state of the system. The output of the monitor can be a confirmation that the system's behavior meets the defined requirements and measures, an alarm of a mis-behavior, or a feedback to the system itself and to the controller suggesting possible improvements.

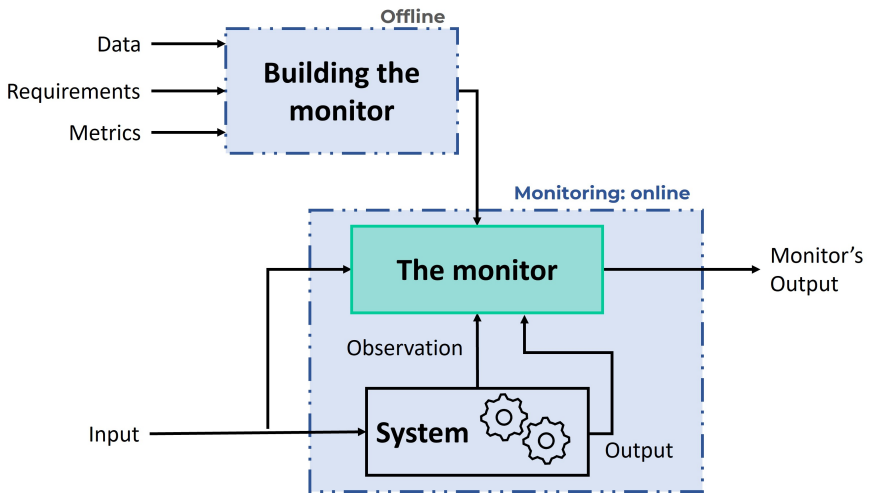


Figure 2.12: Main phases and general process of the runtime monitoring.

Similar to the general runtime monitoring process, NN monitoring consists of two main phases: buidling the monitor, and using the monitor to supervise

the NN during runtime. In the context of NN, the monitor building phase relies on extracting learned patterns from the network, using some data. Next, the monitor represented by these patterns is used to supervise the network's decisions during runtime. The former phase is essential, since it results in extracting relevant patterns, which constitute the core of the monitor. Although it is hard to formally prove that the used patterns represent exactly the learned patterns by the networks, researchers attempt to provide empirical guarantees by assessing the efficiency of the proposed monitoring system on different data sets and scenarios (Cheng, Nührenberg, and Yasuoka 2019).

Most of NN monitoring research works seeks to analyze the inner activations of the network. The broad idea is to feed the network N with a set of labeled data, e.g., the training set. Next, the values of hidden neurons of N with respect to the set of data is calculated. These hidden neurons' values are referred to as hidden neurons activations. The monitor is then constructed using these activations, where each set of activations represents a pattern, such as in an image classification problem (Cheng, Nührenberg, and Yasuoka 2019; Geng et al. 2022; Henzinger et al. 2020).

In the context of NN monitoring, Cheng, Nührenberg, and Yasuoka (2019) proposed a runtime monitoring system based on activation patterns. First, they use the training set to calculate hidden activations. Then, they build an activation pattern for each class and save it by means of Binary Decision Diagrams (BDD). The set of activation patterns is called Neural Activation Patterns (NAPs). During runtime, in addition to making a classification decision using the network, the monitoring system checks whether the activation patterns of the input are similar to one of the pre-built NAPs. In their work, they use the Hamming distance to measure the similarity between NAPs. If no similar NAP is found, the system raises a warning notifying that the classification decision should be re-checked. The effectiveness of this approach is constrained by the performance of BDDs, which have several limitations, including restrictions on the number of variables that can be handled. Recently, Geng et al. (2022) proposed a new type of specifications called *neural representation as specification*. They introduced a new formula of the robustness property by adding a new constraint using NAPs. The specification states that all inputs following a NAP shall never be misclassified. This means that, for an input-image of class c that follows a NAP NAP_c, if the perturbed image also follow NAP_c, the network has to classify this image as class c. The authors updated the Marabou verifier (Katz, Huang, et al. 2019) to support the new formula of robustness. Additionally, in case the

property is violated, the authors claim that the generated counter-examples are more realistic.

The problem of detecting novel inputs, which is called novelty or out-of-distribution problem, has been studied for many years, and many methods have been proposed (Pimentel et al. 2014; Yang, Zhou, et al. 2021). One of the most promising approaches involves analyzing the activation patterns of hidden neurons. Henzinger et al. (2020) proposed a monitoring method for neural networks based on abstraction. The method involves constructing box abstractions by over-approximating the output of selected hidden layers using the training data. At the end of this phase, the hidden neurons of the selected layers are associated with a box that over-approximates their values in response to the fed inputs. During the monitoring phase, the system checks whether the values of these layers lie within the range of the calculated intervals or boxes. If they are outside the calculated boxes, the corresponding input is considered as novel, which results in triggering a warning. This approach has been refined in a subsequent work by Wu et al. (2023). The enhancement involves incorporating both ”good” (correct decisions) and ”bad” (incorrect decisions) behaviors of the network as references to construct the box abstractions. Incorporating both types of references introduces an uncertainty verdict, enabling the identification of suspicious regions when their abstractions overlap. Additionally, the paper introduces the box resolution, which enables controlling the precision of the box and measuring how coarse is the abstraction. Hashemi, Křetínskỳ, Mohr, et al. (2021) introduced a method by modelling the neuron's activations as a Gaussian model. During runtime, this model is used as an out-of-distribution detector. In a recent work, Olber et al. (2022) extended NAPs to extract activation patterns on convolution layers, and then used these patterns to detect out-of-distribution image samples on CNNs.

NAPs have also been applied in various studies for the purposes of explainability and interpretability. For example, Bäuerle et al. (2022) leveraged the activation values of neurons to analyze and extract learned features. Then, they identified groups of similar NAPs that could be used to visually interpret the learned features within a layer. While Krug et al. (2018) utilized NAPs for interpreting CNN models used in speech recognition, Stano et al. (2020) proposed a method that involves encoding the behavior of neurons using a Gaussian Mixture Model (GMM) when exposed to a set of inputs from the same class. The resulting GMM model is then used to explain the classification decision made by the network.

2.5 Conclusion

In this chapter, we firstly provided preliminaries and foundations related to AI, its evaluation, focusing specifically on NN verification. Concretely, we presented a quite comprehensive explanation of the functioning of NNs, along with necessary notations and foundations that will be used throughout this dissertation. Then, we presented the basis of formal verification, existing verification techniques, and their applications to ensure the correctness of safety-critical systems. Next, we explained how formal verification methods are adapted to formally verify properties on NNs. The chapter also provided a detailed literature review of verification techniques applied on NNs, highlighting their corresponding issues when dealing with such models. The main focus of this chapter was on complete NN verification methods, since the following chapter will deeply discuss the incomplete ones.

Chapter 3

A Review on Abstraction Methods for NN Verification

Outline of the current chapter

3.1 Introduction

As discussed in the previous chapter, NN verification has gained significant attention in recent years, driven by the rapid advancements in NN and their intensive deployment across various domains, including safety-critical systems (Julian et al. 2016; Urban et al. 2021). This heightened attention towards NN verification is witnessed by the amount of developed methods, and published works and surveys (Huang, Kroening, et al. 2020; Leofante et al. 2018; Liu, Arnon, et al. 2021; Tran, Xiang, et al. 2020). The research community roughly agrees that one of the major drawbacks of the existing methods is their applicability and scalability when it comes to deal with large networks. Abstraction methods are seen as a potential solution to overcome this issue. In this chapter, we will provide a literature review of the existing NN abstraction approaches.

In this chapter, we conduct an in-depth examination of existing activation function abstraction and model reduction methods from the literature, that are employed for the purpose of NN verification. Concretely, we perform a critical analysis of each presented technique, outlining its advantages and drawbacks, and discussing its related formal guarantees. For model reduction techniques, we particularly highlight the impact of each method on the verification process, and we discuss further research directions related to these techniques. Although the main focus of the considered works is on feed-forward NNs, we also provide some perspectives on how these abstraction methods can be adjusted to support other types of NNs.

It is worthwhile to notice that the core of this chapter relies on our recently published survey paper concerning abstraction methods for the purpose of NN verification (Boudardara, Boussif, Meyer, et al. 2023a). To the best of our knowledge, our work represents the first comprehensive overview within this specific sub-field of NN verification.

3.2 Abstraction Approaches for NN Verification

Recently, many approaches enabling to check properties on NNs have been proposed (Huang, Kroening, et al. 2020; Liu, Arnon, et al. 2021). The straightforward verification way consists in encoding the NN behavior, as well as the property to be checked, as a system of linear equations, and then using an appropriate engine to perform the verification process. For instance, SAT/SMT

and MILP encoding are widely used to verify NN properties (Cheng, Nührenberg, and Ruess 2017; Dutta et al. 2018; Huang, Kwiatkowska, et al. 2017; Katz, Barrett, et al. 2017; Katz, Huang, et al. 2019; Lomuscio et al. 2017; Tjeng et al. 2019). These methods are also called *complete* because they encode the exact behavior of the network. However, since most of the common activation functions are nonlinear, this kind of verification methods does not scale in the case of large neural networks, and suffers from state-space explosion. For example, for the piece-wise linear activation function *ReLU*, each *ReLU* node has to be split into two linear constraints, i.e.: if $y = relu(x)$, then $y = 0$ when $x < 0$ and $y = x$ when x is positive. Therefore, solving a verification problem on a network of n *ReLU* nodes requires solving 2^n linear sub-problems as illustrated in Figure 3.1. To address this issue, several approaches based on abstraction have been proposed. The next section provides more details about this category of techniques.

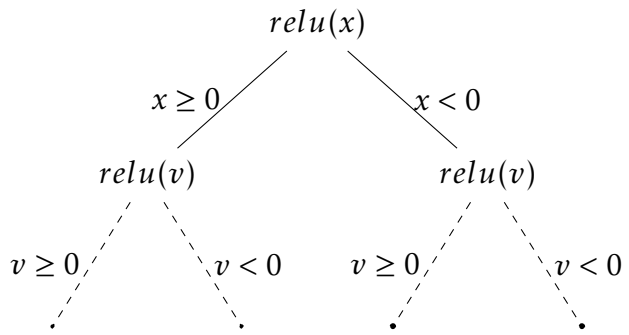


Figure 3.1: Illustration of the state-space explosion: for two *ReLU* nodes, case splitting leads to four linear subproblems.

3.2.1 NN Abstraction Principle

In order to overcome the drawbacks of complete verification methods for NN, some abstraction approaches are proposed. The main idea behind these approaches consists in generating an abstract model from the original network ensuring that whenever the property $\boldsymbol{p}$ holds on the abstract model $\overline{\boldsymbol{N}}$, it necessarily holds on the original one $\boldsymbol{N}$, i.e.,:

$$\overline{N} \models p \implies N \models p. \tag{3.1}$$

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

Concretely, the abstraction of NN can be performed in two different manners:

- *Activation function abstraction*: to ease the verification process, non-linear activation functions of the NN are over-approximated by a set of linear constraints.
- *NN model reduction*: abstracting the network model by merging some nodes in order to reduce the size of the network, and thus improve the scalability of existing verification techniques.

In is worth noticing that the above two manners can be combined in order to generate the abstract models. A detailed survey of these methods is given in Sections 3.2.2 and 3.2.3, respectively.

Remark 3.1 (Refinement) *Some works consider improving the incomplete verification methods by ruling out as many spurious CE as possible by introducing a refinement phase. During this phase, the verification method refines the abstract model iteratively until we can prove either the property holds or the generated CE exhibits a real behavior on the original model (Ehlers 2017; Tran, Bak, et al. 2020; Tran, Manzanas Lopez, et al. 2019; Wang et al. 2018a,b).*

3.2.2 Abstraction of the Activation Function

The key challenge of NN verification pertains to the non-linearity of activation functions. Activation function-based abstraction approaches are applied to handle this issue by over-approximating the activation functions with linear constraints.

The earliest work dealing with NN abstraction for verification purposes was introduced by Pulina et al. (2010). In their work, the authors divide the nonlinear $Sigmoid$ activation function into small regions, then a linear over-approximation is computed for each region, as shown in Figure 3.2.

With the same spirit, Ehlers (2017) proposed a precise $ReLU$-abstraction technique where $ReLU$ is replaced by a system of linear constraints (see

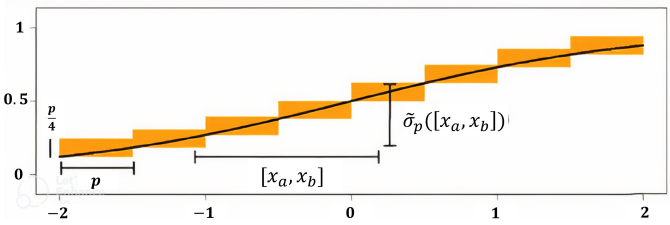


Figure 3.2: The activation function Sigmoid (σ) and its abstraction on $x \in [-2, 2]$. The solid line represents $y = \sigma(x)$ and each small region (yellow rectangles) is an over-approximation of y (Pulina et al. 2010).

Figure 3.3a) and hence the verification problem of NNs is reformulated as a linear programming (LP) problem that can be solved using classic LP solvers. The approach in (Ehlers 2017) was implemented in a tool called Planet[1] and brings the LP toolkit GLPK[2] into play along with the Minisat solver[3] for verification.

Gehr et al. (2018) applied an *abstract interpretation* method (Cousot et al. 1977) on NN for the first time. They proposed a framework called AI^2 (Abstract Interpretation for Artificial Intelligence) that soundly over-approximates NN operations by means of *zonotope* abstract domain. The approach can be extended to support other abstract domains. AI^2 can handle feed-forward and convolutional neural networks (CNN) with *ReLU* and *max-pooling* functions. The approach in (Gehr et al. 2018) was extended by Singh, Gehr, Mirman, et al. (2018) to support the *Sigmoid* and *Tanh* activation functions. This is accomplished by means of abstract transformers based on zonotopes for each function. As an example, the abstraction of *ReLU* is given in Figure 3.3b

Definition 3.1 *An abstract domain is a set of logical constraints that define a geometric shape. The most popular abstract domains are: box (or Interval), zonotope and polyhedra. For example, a zonotope abstract domain (Ghorbal et al. 2009) Z is defined by a set of constraints* z_i, *s.t:* $z_i = a_i + \sum_{j=1}^{m} b_{ij}\epsilon_j$, *where* $\epsilon_j \in [l_i, u_i]$ *is an error term and* a_i, b_{ij} *are constants.*

Furthermore, Singh, Gehr, Püschel, et al. (2019a) proposed a new method, called *DeepPoly*, based on Abstract Interpretation by introducing a new abstract domain. DeepPoly combines floating point polyhedra and intervals. Each neuron is represented by its concrete and symbolic upper and lower bounds. Moreover, the authors introduced abstract transformers for popular NN operations: affine

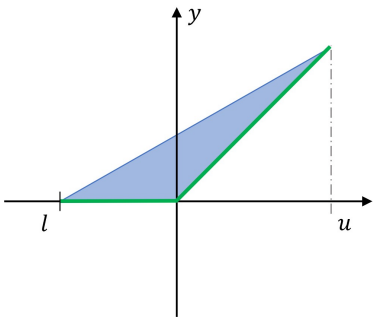
[1]The Planet verifier is available here: https://github.com/progirep/planet
[2]https://www.gnu.org/software/glpk/
[3]http://minisat.se/

transformation, *ReLU*, *Sigmoid*, *Tanh* and *Max-pooling* to propagate the inputs successively through the layers of the network. For *ReLU*, two different abstractions are proposed, as shown in Figures 3.3c and 3.3d. It is worthwhile to mention that the approach supports both feed-forward and CNN.

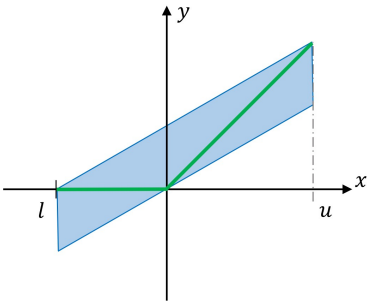
While the previous works consider only a single neuron, some others try to define sound approximations of a set of neurons, jointly. Singh, Ganvir, et al. (2019) introduced a new method that provides an approximation of k *ReLU* nodes (in the same layer) at a time in order to capture dependencies of the *ReLU* inputs. First, the k nodes are selected and then the convex relaxation of the group of nodes is calculated. The framework has a parameter k which represents the number of *ReLU* nodes to be considered together. Based on the work of Singh, Ganvir, et al. (2019), a more general framework was recently proposed by Müller et al. (2022). The framework, called *PRIMA* (PRecIse Multi-neuron Abstraction), computes the convex over-approximation of a set of k outputs of an arbitrary activation function, including *ReLU*, *Sigmoid* and *Tanh*. The approach decomposes the set of neurons in each hidden layer into overlapping groups of size k, then calculates the convex approximation of the octahedral over-approximation for each group. Finally, it takes the union of all the obtained output constraints. These constraints combined with the encoding of the whole NN are used for verification.

Other techniques based on symbolic propagation are proposed in (Li, Liu, Yang, Chen, et al. 2019; Yang, Li, et al. 2021) to enhance the precision of abstract interpretation-based approaches. In symbolic propagation every neuron is associated with a formula expressed using the activations of neurons in its previous layers. In (Singh, Gehr, Püschel, et al. 2019b), a combination of over-approximation techniques with linear relaxation methods is proposed so as to gain more precision of over-approximation techniques and the scalability of complete methods.

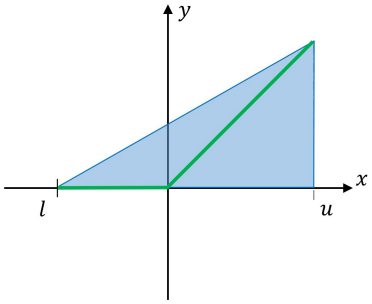
Finally, we should notice that the above-mentioned techniques can be adapted to support further types of NNs. For instance, one way to deal with recurrent NNs (RNN) is to generate an equivalent FFNN and then apply the abstraction method (Akintunde et al. 2019; Jacoby et al. 2020). For CNN, most of the techniques are straightforwardly applicable with the only restriction that the activation function of the convolution layer has to be *ReLU* or other supported functions such as Sigmoid and Tanh (Singh, Gehr, Püschel, et al. 2019a).



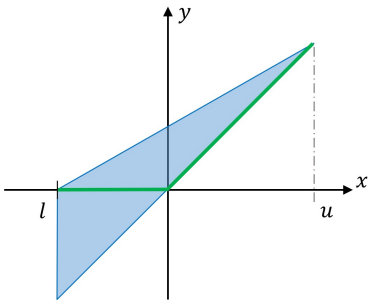
(a) The abstraction of the *ReLU* activation function proposed in (Ehlers 2017).



(b) The abstraction of the *ReLU* activation function using zonotopes (Gehr et al. 2018; Singh, Gehr, Mirman, et al. 2018).

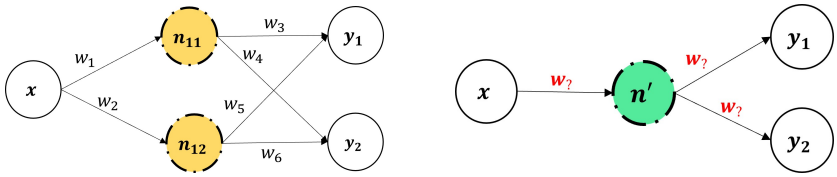


(c) An abstraction of the *ReLU* activation function proposed in (Singh, Gehr, Püschel, et al. 2019a).



(d) An abstraction of the *ReLU* activation function proposed in (Singh, Gehr, Püschel, et al. 2019a).

Figure 3.3: ReLU activation function abstractions using different abstract domains. The *ReLU* ($y = relu(x)$) is represented by the green line and its over-approximation on the range $x \in [l, u]$ by the blue filled area.



(a) An NN before abstraction (the concrete NN model).

(b) The NN after model reduction (the two hidden nodes are merged).

Figure 3.4: Model reduction of a small neural network.

3.2.3 Neural Networks' Model Reduction

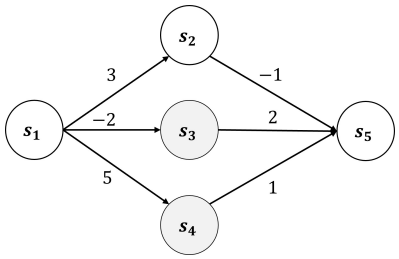
The main objective of NN model reduction is to reduce the size of the NN model while guaranteeing and preserving some behavioral relationship, mainly, some desired property $\boldsymbol{p}$ that holds on the original model $\boldsymbol{N}$ remains fulfilled in the reduced model $\overline{\boldsymbol{N}}$ as defined in (3.1). Figure 3.4 provides an illustrative example of the main idea behind model reduction applied on a small neural network.

Such a behavioral relation is obtained by ensuring that $\overline{N}$ is an over-approximation of N. Therefore, the reduction process must carefully select the set of neurons to be merged (or removed), and determine how to calculate the weights of the new edges.

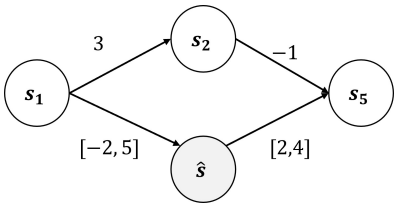
Prabhakar and Rahimi Afzal (2019) proposed a method based on Interval Neural Networks (INN) for output range analysis. In this method, the nodes of the same layer are merged while replacing the weights of their input edges by the interval hull of the incoming edges. In other words, the weights of incoming edges are replaced by $[\min(W_{in}), \max(W_{in})]$, where W_{in} are the values of the incoming weights to the nodes to be merged. The weights of the outgoing edges from these nodes (W_{out}) are replaced by the interval hull multiplied by the number of merged nodes n, i.e., $n \times [\min(W_{out}), \max(W_{out})]$. Figure 3.5 depicts an application example of this method on a toy network.

For the verification part, Prabhakar and Rahimi Afzal (2019) adapted INN to MILP big-M encoding (Cheng, Nührenberg, and Ruess 2017) and used the Gurobi MILP solver[4] for verification. The performance of this method is evaluated on the airborne collision avoidance ACAS Xu benchmark (Julian et al. 2016; Katz, Barrett, et al. 2017). The authors claim that the abstraction enhances the verification process. Namely, Gurobi was not able to verify a number of properties on the original model (no return - out of time), while the

[4] https://www.gurobi.com/



(a) The original network. For $v(s_1) = 1$, we have $v(s_5) = 2$.



(b) The obtained reduced network.

Figure 3.5: The abstract network using INN method (Prabhakar and Rahimi Afzal 2019) and ANN (Sotoudeh et al. 2020). For $v(s_1) = 1$, $\hat{v}(s_5) = [0, 17]$, and we have $v(s_5) \in \hat{v}(s_5)$.

same properties have been successfully checked when Gurobi was applied on the abstract model.

By introducing the notion of Abstract Neural Networks (ANN), Sotoudeh et al. (2020) provided a formalization of a general abstraction approach. In ANN, the weights are represented using abstract domains (see Definition 3.1). Accordingly, the approach proposed by Prabhakar and Rahimi Afzal (2019) can be considered as a particular instantiation of this approach using the interval abstract domain. Notice that the proposed approach supports a wide range of activation functions. Moreover, it can be instantiated using other convex abstract domains and it is not restricted to intervals as used in INN (Prabhakar and Rahimi Afzal 2019). The approach provides a generic formula to calculate the weight merging matrix $\overline{W}$ from the original weight matrix W and the partitions P^{in} and P^{out} of two successive abstract layers l_i and l_{i+1}, respectively. A partition P_i is a rearrangement of a set S_i of neurons, i.e., if $S_i = \{n_{i1}, n_{i2}, n_{i3}\}$, a possible partition of S_i would be $P_i = \{\{n_{i1}, n_{i2}\}, \{n_{i3}\}\}$, which means that n_{i1} and n_{i2} will be merged in the abstract network. $\overline{W}$ is the convex combination (calculated by a function g) of the partitioning combination matrix of P^{in} and P^{out}, denoted by C and D, respectively, and the weight matrix W, i.e., $\overline{W} = g(D, W, C)$. Next, the abstract weight matrix, denoted by W_{abs}, is built by applying a convex abstract domain α_A on the obtained $\overline{W}$: $W_{abs} = \alpha_A(\overline{W})$. The reduced model is obtained by applying the same procedure to every layer, iteratively. Therefore, the obtained reduced model is an over-approximation for any non-negative activation function that satisfies the Weakened Intermediate Value Property (WIVP)[5]. Although

[5]As defined in (Sotoudeh et al. 2020), a function $f : \mathbb{R} \to \mathbb{R}$ satisfies the Weakened Intermediate Value Property if, for every $a_1 \leq a_2 \leq \cdots \leq a_n \in \mathbb{R}$, there exists some $b \in [a_1, a_2]$ such that $f(b) = \sum_{i=1}^{i=n} \frac{f(a_i)}{n}$.

some activation functions can have negative values and others are not continuous (thus not WIVP), the authors of (Sotoudeh et al. 2020) claim that there is always a way to overcome these problems, as they showed for Leaky *ReLU* and the *threshold* activation functions.

In (Ashok et al. 2020), the authors apply K-means clustering algorithms to partition each hidden layer l_i into k_i subgroups, such that $k_i \leq |S_i|$, then replace each subgroup with its representative neuron. The abstraction method, called DeepAbstract, has three parameters: the original network N, a finite set of input-points X and a vector K_L which contains the number of nodes on each abstract layer. For each hidden layer l_i, the following steps are performed:

1. For every $x \in X$, calculate the value $v_{ij}(x)$ of each neuron in S_i,
2. Apply K-means to split each layer l_i into k_i clusters. Let C_{l_i} denote the set of clusters of l_i,
3. For each cluster $C \in C_{l_i}$:
   (a) Determine the representative neuron rep_C,
   (b) Calculate the corresponding outgoing weights of rep_C:

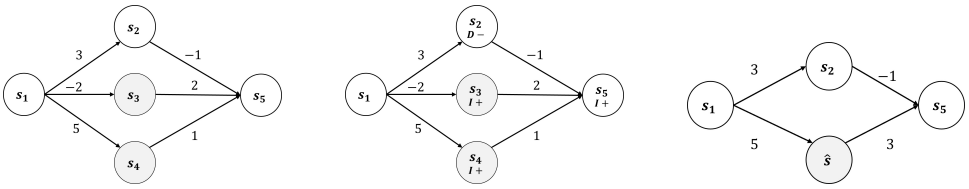
$$\overline{W}^i_*, rep_C = \sum_{n_{ij} \in C} W^i_*, n_{ij}$$

   (c) Replace all the neurons in C with rep_C.

Note that the representative neuron rep_C of a cluster C is the nearest neuron to the centroid of C; thus, the incoming weights of rep_C remain the same as the corresponding neuron before abstraction. All the other neurons from cluster C are omitted with their incoming edges.

In addition, Ashok et al. (2020) provide a method to lift the verification results from the abstract model to the original one. The idea is to calculate the accumulated error induced by replacing a cluster of neurons by its representative for each image x in X, and then propagate this error through the successive layers using the DeepPoly verification Algorithm[6]. A set of experiments were conducted to check the performance of DeepAbstract. Local robustness of some MNIST[7] images was checked and the authors claim that the verification time was reduced by 25% when DeepPoly is combined with DeepAbstract.

[6] https://github.com/eth-sri/ERAN
[7] http://yann.lecun.com/exdb/mnist/



(a) The original network. For $v(s_1) = 1$, we have $v(s_5) = 2$.

(b) Neurons' classification: positive /negative and increasing / decreasing (Elboher et al. 2020).

(c) The obtained abstract network.

Figure 3.6: The abstract network using the method of Elboher et al. (2020). For $v(s_1) = 1$, $v(s_5) = -2$, $\hat{v}(s_5) = 12$, and we have $v(s_5) \leq \hat{v}(s_5)$.

Elboher et al. (2020) proposed an abstraction approach based on merging neurons of the same *category* (see hereafter) to build a smaller model so as to enhance the scalability of the existing verification tools. Regarding the verification property, which has the form: $p : \forall x \in pre(x) \implies y \leq c$, the aim of this approach is to build a reduced model $\overline{N}$, such that: $\forall x \in D_x$, $\overline{N}(x) \geq N(x)$. Therefore, $N \models p$ whenever $\overline{N} \models p$ (i.e., $\overline{N}(x) \leq c$). To obtain the neurons categories, each neuron is firstly labelled according to the sign of its outgoing weights. A neuron is split if it has positive and negative outgoing weights at the same time. Next, to guarantee that $\overline{N}$ is an over-approximation of N, the proposed method seeks to increase the output of the abstract model by classifying each neuron as I or D. The class I means the output will increase by increasing the value of the neuron at hand, while a neuron is marked as D if decreasing its value leads to increasing the output's value. Finally, the nodes of the same layer and the same category can be merged by summing up the weights of their outgoing edges and taking the *min* value of the the weights of their incoming edges if they are marked as D, or the *max* value for any I group of nodes (Figure 3.6 illustrates these steps on a small network). Moreover, some heuristics are proposed in (Elboher et al. 2020) to enhance the abstraction process. The proposed method is applied on ACAS Xu networks while Marabou tool (Katz, Huang, et al. 2019) is used as a back-end verification tool. A comparison study between the abstraction method combined with Marabou and the vanilla version of Marabou was conducted, and the results show that the abstraction method allows Marabou to verify more properties in less execution time.

A novel approach based on Bisimulation (Larsen et al. 1991) is proposed by Prabhakar (2022). The generated abstract neural network is equivalent, or

bisimilar, to the original one. To guarantee the equivalence between N and $\overline{N}$, two neurons n_{ij} and n_{ik} to be merged must have the same activation function, the same bias value ($b_{ij} = b_{ik}$) and the same weights for each incoming edge respectively, i.e., $\forall n' \in S_{i-1}, w(n', n_{ij}) = w(n', n_{ik})$. Due to the strict conditions that, generally, do not hold in most of real networks, Prabhakar (2022) extends the NN bisimulation to a more feasible relaxed method, called NN δ-bisimulation. Using NN δ-bisimulation ($\delta \in \mathbb{R}^+$), two nodes n_{ij} and n_{ik} in S_i can be merged if the following conditions are satisfied:

1. n_{ij} and n_{ik} have the same activation function
2. $|b_{ij} - b_{ik}| \leq \delta$, where $\delta \geq 0$
3. $\forall n' \in S_{i-1}, |w(n', n_{ij}) - w(n', n_{ik})| \leq \delta$

It is important to note that the obtained network $\overline{N}$ is no longer (exactly) equivalent to the original network N, but the two networks are called δ-bisimilar.

Taking advantages of code refactoring (Fowler 2018), Shriver et al. (2019) introduced the concept of refactoring neural networks to restructure the initial model and preserve its accuracy to enhance further operations on it, for instance verification. Concretely, NN refactoring consists of two steps: architecture transformation and distillation. The former applies some changes on the network's architecture by dropping or changing some layers and/or their types that are not supported by verification tools (e.g. residual blocks and convolutional layers). The latter updates the model's parameters: weights and biases, while preserving the original model's behavior, which is captured by its accuracy and test error according to Shriver et al. (2019). A tool called R4V (Refactoring for Verification) was developed from this approach. R4V was tested on DAVE-2 (Bojarski et al. 2016) and DroNet (Loquercio et al. 2018) networks.

The used verification tools are presented in Table 3.1. The results show that applying the verification tools on the refactored model improves their scalability. For example, Planet (Ehlers 2017) fails to check any property on DroNet within 24 hours. However, after refactoring the network, Planet was able to verify three out of the ten selected properties. The main features of the above discussed neural networks reduction techniques are summarized in Table 3.1. The last two columns of the table contain verification methods and the data sets used during the evaluation of the abstraction method. Verification methods are those used during the evaluation of the abstraction in the original paper; notice that other methods can be used to verify the obtained abstract model.

Method	Pub. Year	Supported AFs	Verification methods	Evaluation on	Guarantees of the reduced model
R4V (Shriver et al. 2019)	2019	*ReLU*	ReLUpex(Katz, Barrett, et al. 2017), ERAN(Singh, Gehr, Mirman, et al. 2018), Neurify(Wang et al. 2018a), Planet(Ehlers 2017)	DAVE-2(Bojarski et al. 2016), DroNet(Loquercio et al. 2018)	None
INN (Prabhakar and Rahimi Afzal 2019)	2019	*ReLU*	MILP (Lomuscio et al. 2017)	ACAS Xu (Katz, Barrett, et al. 2017)	$N(x) \in \overline{N}(x)$
ANN (Sotoudeh et al. 2020)	2020	*ReLU*, *Leaky ReLU*[8]	-	-	$N(x) \in \overline{N}(x)$
DeepAbstract (Ashok et al. 2020)	2020	*ReLU*	ERAN	MNIST(LeCun 1998)	Depends on the data set
Elboher et al. (2020)	2020	*ReLU*	Marabou(Katz, Huang, et al. 2019)	ACAS Xu Katz, Barrett, et al. (2017)	$N(x) \leq \overline{N}(x)$
Bisimulation (Prabhakar 2022)	2021	*ReLU*	-	-	$N \equiv \overline{N}$, s.t. $\equiv$ represents the equivalent relation[9]

Table 3.1: A list of NN model reduction methods used for verification. The underscore symbol "–" is used to denote that no information is provided in the corresponding original paper.

[8] The authors claim that the method can be adjusted to support other activation function

[9] The abstract network is equivalent to the original one when bisimulation is used which is not the case for δ-bisimulation

Figures 3.5 and 3.6 illustrate the application of (Prabhakar and Rahimi Afzal 2019) and (Elboher et al. 2020), respectively. Notice that the abstract network using the ANN method (Sotoudeh et al. 2020) with the box (or interval) abstract domain is the same as the abstract network obtained using the method of INN (Prabhakar and Rahimi Afzal 2019) (see Figure 3.5). Due to the need of supplementary details to apply the other methods, we did not include them in this illustrative example. For instance, DeepAbstract (Ashok et al. 2020) needs a data set for the clustering algorithm.

Note that the figures show a segment of a *ReLU*-NN, i.e, s_1 is an arbitrary neuron of some hidden layer and not necessarily the input of the network, and all nodes are assigned a *ReLU* activation function. We apply abstraction (using the selected methods) to merge the two nodes s_3 and s_4, while assuming that $v(s_1) = 1$, and we calculate the values[10] of s_5, $v(s_5)$ and $\hat{v}(s_5)$ on the original and the abstract networks, respectively. While model reduction methods (Prabhakar and Rahimi Afzal 2019; Sotoudeh et al. 2020) (Figure 3.5) ensure that the output of the original network is within the range of the output of the abstract network, i.e.: $v(s_5) \in \hat{v}(s_5)$, the method introduced in (Elboher et al. 2020) (Figure 3.6) guarantees that the output of the obtained abstract network is always higher than the output's value of the original network, i.e.: $v(s_5) \leq \hat{v}(s_5)$.

We should mention here that these techniques can be adjusted to support further types of NNs. For instance, an RNN can be transformed into an equivalent FFNN (Akintunde et al. 2019; Jacoby et al. 2020), and then model reduction approaches can be applied to generate the abstract network. On the other hand, model reduction can be applied on the fully connected part of CNNs (Ostrovsky et al. 2022; Xu, Li, et al. 2021). Regarding Binarized Neural Networks (BNN), due to their binary behavior and their small size compared to other types of NNs, their verification does not require abstracting their behavior and, generally, exact methods such as SAT and MILP can be applied directly (Jia et al. 2020; Lazarus et al. 2022; Narodytska et al. 2018).

It is also worth noting that another family of techniques based on merging neurons and removing some edges without affecting the accuracy of the model exists in the literature. These techniques are called NN compression and acceleration, and their objective is to build a smaller network with low computational complexity, so that it can be embedded on devices with limited resources and used in real-time applications, while keeping the accuracy as high as possible (Cheng, Wang, et al. 2017; Han et al. 2015; Liang et al. 2021).

[10]The valuation of a neuron is defined in Part II, chapter 4 (Definition 4.2)

Although both NN model reduction and NN compression strive to reduce the number of neurons, NN compression techniques cannot be used for verification, since the generated models do not fulfil the abstraction condition presented in Formula (3.1). In other words, verifying a property p on the compressed network obtained by any compression method does not imply that the property does hold on the original network.

3.2.4 Discussion

This section discusses the aforementioned model reduction methods, while highlighting their strengths and limitations, and suggesting potential areas for enhancement. In order to fairly compare the efficiency of the discussed approaches, we analyze them according to three main criteria (with respect to the available information in the original papers): (*i*) the precision of the over-approximation, (*ii*) the minimal number of neurons that can be obtained when the reduction method is applied until saturation, and (*iii*) the efficiency regarding the verification time and the number of verified properties on the reduced model versus the original one.

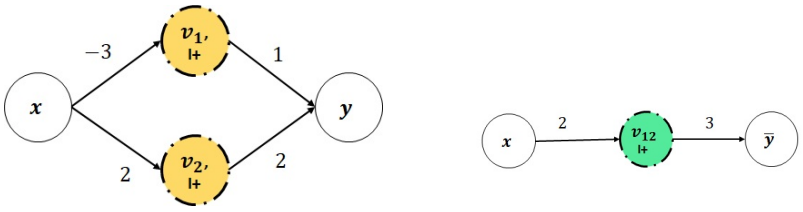
The abstraction method based on INN, proposed by Prabhakar and Rahimi Afzal (2019) seems to be very efficient in terms of output range. An exhaustive application of this method leads to merge all neurons of each hidden layer and replace each layer by one single abstract neuron. The results of their paper show that the precision highly depends on the number and the set of selected nodes to be merged. The method needs some improvements to be more precise, since no study was provided for neuron selection. In addition, operations on intervals may impact the precision of the method. MILP encoding is proposed to address the verification problem on INN, and to the best of our knowledge, no other verification method is proposed to verify INN. Furthermore, the main constraint of this approach lies in its limitation to abstract NNs with non-negative activation functions. Notice that Sotoudeh et al. (2020) have proposed some fundamentals to abstract any NN with different activation functions using any convex abstract domain and which is not limited to intervals. In (Sotoudeh et al. 2020), the authors provide an example of abstraction based on octagons, but no explanation was given of the meaning of using such an abstract domain to represent the merged neurons. Moreover, the work would have been more relevant if it had included an evaluation study to concretely show how the ANN can be extended to deal with other abstract domains.

Regarding the DeepAbstract approch proposed by Ashok et al. (2020), it is mainly parameterized by the number of clusters on each layer; hence, when there are few clusters, the model is more abstract and less precise. In addition, this method relies on the discrete input set X that is used during the clustering phase and can only verify the robustness of the model on points within this set X. Ashok et al. (2020) claim that the verification time is reduced by 25% when DeepAbstract is used along with DeepPoly; however, only 195 out of 200 images could be verified to be robust against 197/200 when DeepPoly is used without abstraction.

The abstraction-refinement approach proposed by Elboher et al. (2020) boosted the `Marabou` verifier (Katz, Huang, et al. 2019) to check more properties (58 out of 90 property vs. 35 of 90). Moreover, the abstraction method reduces the total query median runtime by 60% . As a consequence of the neurons classification, this method can abstract a layer to four neurons at most. This is one of the main drawbacks of the method, since only neurons belonging to the same category can be merged. It should also be mentioned that only properties in the form: $y \leq c$ are considered, although the authors claim that the approach is adaptable to cope with various types of properties by adjusting the output layer. In addition, this method cannot be applied if some neurons have negative values. For instance, this method cannot be applied in hidden layers if the used activation function returns negative values such as sigmoid and Leaky *ReLU*. For the same reason, the first hidden layer cannot be abstracted if the inputs are negative. An example demonstrating this case is given in Figure 3.7, where x is an input, y is the output.

The NN in Figure 3.7.b is generated using the method of (Elboher et al. 2020), which is supposed to be an abstraction of the original model of Figure 3.7.a. Both $\overline{N}$ and N use the *ReLU* activation function on the hidden layer. Although for negative inputs the output of $\overline{N}$ is always zero: $\forall x \leq 0, \overline{y} = 0$, the output of N is always positive, for instance, for $x = -1$, $y = 3$, thus the condition of the over-approximation $\forall x \in D_x : \overline{N}(x) \geq N(x)$ does not hold.

The NN bisimulation method proposed by Prabhakar (2022) guarantees the equivalence between the abstract and the original models; and thus offers an exact abstraction. However, the set of conditions are hard to satisfy on real neural network, especially the condition on weights. On the other hand, the relaxed version, NN-δ-bisimulation, looks more feasible but needs further improvements to keep trace of the verified property on the abstract model and lift it to provide guarantees on the original network.



(a) The concrete model N after the classification of nodes (I+ for increasing positive node).

(b) The abstract model $\overline{N}$.

Figure 3.7: Counterexample of Elboher et al. (2020) abstraction method.

In (Shriver et al. 2019), the authors propose an efficient approach along with a dedicated tool, called R4V, to simplify and compress NN models. The wide experimental study they performed with different verification tools and data sets shows that R4V offers actual benefits to overcome the limitations of some NN verification techniques. However, this method enables to verify properties on the refactored model and proposes no way to lift these guarantees to the original model. In other words, it does not provide any guarantee of whether the property holds on the original model.

To sum up, regarding the challenges of NN verification, we believe that developing a new general approach that overcomes the issues related to the existing abstraction methods mentioned above is necessary. The works (Ashok et al. 2020; Prabhakar 2022; Shriver et al. 2019) could be adopted and combined with some heuristics to select candidate neurons to be merged. For instance, the δ-bisimulation method (Prabhakar 2022) can be used to select similar nodes by analyzing their incoming weights. The approach in (Ashok et al. 2020) can be adapted using discretization of the input region, so that the nodes that are close to each other (in the same cluster) are good candidates for abstraction.

While the technique in (Elboher et al. 2020) ensures that $N(x) \leq \overline{N}(x)$, the three methods presented in (Prabhakar and Rahimi Afzal 2019; Sotoudeh et al. 2020) go further by guaranteeing that the output of the original network is always included within the output range of the obtained abstract network, i.e., $N(x) \in \overline{N}(x)$. However, it is necessary to conduct a comparative study to assess the performance of these methods. On the other hand, an abstract network obtained using DeepAbstract (Ashok et al. 2020) can be used only to verify the robustness of the model on inputs within the set of images X that is used during the clustering phase. The last column of Table 3.1 gives the relationship between the original and the abstract networks using the various considered methods.

3.3 Neural Networks compression

This section is dedicated to introduce NN compression, which is a research topic quite close to NN model reduction. The objective of the developed compression techniques is to enhance operations by reducing the size of the network. Concretely, these techniques focus on providing smaller compressed networks in order to make them more lightweight and computationally efficient, while attempting to retain similar performances, i.e, maintain the accuracy of the compressed network as close as possible to that of the original one (Han et al. 2015). This aspect is particularly critical when dealing with devices that have limited resources, such as low memory and CPU capabilities. This is due to the fact that NN models used in applications like computer vision and natural language processing can contain billions of parameters, including weights and biases. Training and deploying such resource-intensive models often require high-performance computing, making NN compression essential for efficient use in resource-constrained environments (Neill 2020). For instance, NN compression enables the deployment of NNs on mobile phones and embedded systems while maintaining performance levels close to the original large network (Cheng, Wang, et al. 2017).

Several NN compression techniques have been developed in the literature. These techniques include:

1. ***Pruning:*** it is one of the most used NN compression techniques. Prunning involves removing irrelevant connections represented by weights (or kernels in the context of CNN). This can lead to a significant reduction in the network's size and also the computational cost. It is important to note that pruning is not performed randomly; in fact some algorithms for identifying and eliminating irrelevant weights are used in order to keep the compressed network as accurate as possible (Liang et al. 2021).

   Furthermore, many proposed pruning techniques adopt an iterative approach. They repeatedly prune and retrain the network until they achieve a desired trade-off between accuracy and network size. This approach ensures that the network's performance is carefully balanced with the reduction in its size and the computational constraints (Neill 2020).

2. ***Quantization:*** this technique consists in efficiently representing NN parameters, typically weights and biases, with reduced number of bits.

For instance, the values of a network's parameters might be converted from 32-bit floating-point (which is generally used) into a more compact representation such as 8-bit or 4-bit integers. This conversion helps reducing the amount of memory required for storing the parameters and accelerating the training and the execution of inference operations of NN. This process may lead to some loss in terms of model accuracy. However, well-designed quantization techniques aim to minimize this impact. The goal is to ensure that the obtained model retains the capability to make accurate predictions (Gholami et al. 2022).

3. ***Knowledge distillation:*** this NN compression technique involves learning a compact and small network to replicate and inherit the behavior or the "knowledge" of a larger and more complex network. The former network is referred to as *student network* and the latter is referred to as the *teacher network*. To ensure the knowledge transfer, the training of the student network is performed under the supervision of the teacher network. This is achieved by minimizing the entropy, the distance, and the divergence between the probabilistic estimates of the student and teacher networks (Buciluǎ et al. 2006; Neill 2020).

3.4 Conclusion

In this chapter, we reviewed and discussed the state-of-the art related to the abstraction and model reduction of neural networks for verification purposes. We showed that the NN abstraction can be a valuable approach for mitigating the non-linearity and the complexity of the generated networks. Abstraction of NNs can be applied in two levels: abstracting the activation functions and reducing the network's size (model reduction). While the abstraction of activation functions aims to over-approximate the non-linear activation functions with linear constraints, model reduction is used to reduce the number of neurons within the network. Both categories are applied to improve the scalability of the verification process. It is important to note that the abstraction has to be sound, ensuring that any property that holds on the abstract network necessarily holds on the original network.

After discussing the advantages, limitations and the formal guarantees provided by the reviewed reduction methods, we will proceed to present our two main contributions concerning model reduction in the following chapter.

Part II

Neural Networks Abstraction

Chapter 4

Two Model-Reduction Approaches for Efficient NN Verification

Outline of the current chapter

4.1 Introduction

Verifying NNs has become a hot research topic, particularly with the huge interest shown by industry and academia in applying NNs in safety-critical systems. Despite the amount of developed works in this area, the problem of verifying NNs of practical interest remains challenging. As discussed in Chapter 3, the existing NN verification methods face issues when it comes to scaling up and handling real-world sized networks. The complexity and the non-linearity of NN models make the verification process computationally expensive and resource-demanding. As a result, the scalability of these methods becomes a significant concern. To address this issue, model reduction methods, as a part of NN abstraction methods, are seen as a promising research direction to handle larger models.

The main objective of model reduction methods is to reduce the network's size by merging its neurons, while ensuring that the abstract model is an over-approximation of the original one. This process is crucial for enhancing the scalability of the verification process, as it reduces computational complexity. To achieve this objective, model reduction techniques rely on mathematical formulas to calculate the new weights of the reduced network. By establishing an over-approximation relationship between the original network and its abstraction, the desired verification properties can be checked on the (small) abstract network instead of the (larger) original one. By doing so, computational costs are significantly reduced, allowing for more efficient and scalable verification procedures.

In this chapter, we discuss our contributions related to model reduction. Our first approach, called ***INNAbstract*** (Boudardara, Boussif, Meyer, et al. 2022, 2023b), is based on Interval Neural Networks (INN) and ensures that the output of the original network is within the obtained abstract network's output, i.e., $N(x) \subseteq \overline{N}(x)$. INNAbstract can be applied on NNs with numerous activation functions, including monotone odd activation functions and the ReLU function. The broad idea of the approach is to merge neurons belonging to the same layer, and define the new weights. Mathematical formulas and algorithmic procedures to compute the incoming and outgoing weights of abstract net neurons, while preserving the over-approximation relationship, are proposed. Detailed proof of the over-approximation is also presented.

Our second contribution is a model reduction method for NNs with non-negative activation functions (Boudardara, Boussif, and Ghazel 2023). The

proposed method leverages merging neurons whose outgoing weights are positive. However, if the set of neurons to be merged has some negative outgoing weights, a pre-processing step is required. This step involves building an initial abstract network by eliminating edges that have negative weights. Next, the model reduction procedure is applied to merge the preselected set of nodes and computes its corresponding incoming and outgoing weights. This method guarantees that the obtained abstract network's output is always greater than or equal to the original network N, i.e., $\overline{N}(x) \geq N(x)$. Notice that the approach is used for checking properties such that $N(x) \leq c$.

This chapter is dedicated to present our approaches along with the related background, notations, and the corresponding formal proofs. Next, the evaluation of efficiency and scalability of our approaches, including a comparison to some selected relevant related works from the literature, is provided in Chapter 5.

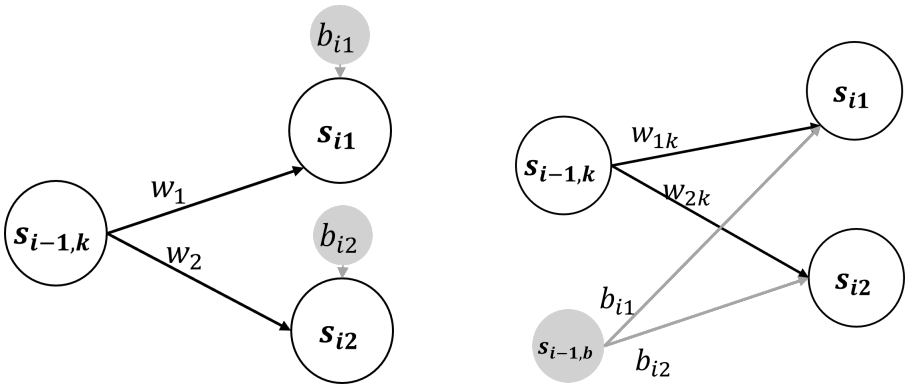
In this chapter, we first present some preliminaries and notations related to our contributions in Section 4.2. Then, Section 4.3 introduces our first contribution related to NN abstraction, namely INNAbstract. In Section 4.4, we provide the theoretical development of the second approach. Finally, in Section 4.5, we present a comprehensive discussion of related works, before concluding this chapter with Section 4.6.

4.2 Preliminaries & Notations

Before delving into the details of the two approaches, it is essential to introduce some key definitions and notations that will be employed throughout this chapter.

4.2.1 Neural Networks Notations

Definition 4.1 *For a NN N, the set of nodes of a layer l_i is represented by S_i. S_0 is the input set, S_n is the output set, and $\forall i \in \{1, 2, \ldots, n-1\}$, S_i represents the set of nodes of the hidden layer l_i. Two successive layers, l_{i-1} and l_i, are connected via a weighted-matrix W_i, such that: for $i \in \{1, 2, \ldots, n\}$, $w^i_{jk} = w(s_{i-1,k}, s_{i,j})$ denotes the weight of the edge connecting $s_{i-1,k} \in S_{i-1}$ to $s_{ij} \in S_i$, for all $k \in \{1, 2, \ldots, |S_{i-1}|\}$ and $j \in \{1, 2, \ldots, |S_i|\}$.*



(a) A network with bias vector.

(b) An equivalent network to the one presented in 4.1a where biases are replaced by weights, s.t. $v(s_{i-1,b}) = 1$.

Figure 4.1: An example explaining how to replace the bias vector by a weight vector.

Definition 4.2 *Given a node $s_{ij} \in S_i$, $i \in \{1, 2, \ldots, n\}$, its valuation $v(s_{ij}) = v_{ij}$ is calculated as follows:*

$$v(s_{ij}) = \alpha\left(\sum_{s_{i-1,k} \in S_{i-1}} w(s_{i-1,k}, s_{ij}) \times v(s_{i-1,k})\right) \tag{4.1}$$

where $\forall s_{0j} \in S_0, j \in \{1, 2, \ldots, |S_0|\}, v(s_{0j}) = x_j$ is the value of the j^{th} element of the input x, and $\alpha : \mathbb{R} \to \mathbb{R}$ is the activation function of the NN (assumed to be the same for every neuron in the NN).

Notice that several types of activation functions can be used with NNs. For instance, Equations (4.2), (4.3), and (4.4) define the *Tanh*, *ReLU* and *Sigmoid* functions, respectively.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}; \qquad x \in \mathbb{R} \tag{4.2}$$

$$ReLU(x) = max(0, x); \qquad x \in \mathbb{R} \tag{4.3}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}}; \qquad x \in \mathbb{R} \tag{4.4}$$

From Definition 4.2, and with a slight abuse of notation, we use $v(S_i)$ or V_i to

denote the valuation set corresponding to all nodes of layer S_i, i.e.,

$$V_i = v(S_i) = [v(s_{i1}), v(s_{i2}), \ldots, v(s_{i|S_i|})]^\tau$$

Definition 4.3 *Given a NN composed of n layers with its associated function* $N : \mathbb{R}^{|S_0|} \rightarrow \mathbb{R}^{|S_n|}$*, its valuation for a given input* x *is* $N(x) = v(S_n)$

In Definition 4.3, the function N is the composition of a set of vector functions f_i where $i \in \{1, 2, ..., n\}$, i.e., $N = f_n \circ f_{n-1} \circ \cdots \circ f_1$, where f_i is the function of layer l_i and is defined as follows:

$$f_i = \alpha\left(W_i \times v(S_{i-1})\right) \tag{4.5}$$

where $W_i \in \mathbb{R}^{|S_i| \times |S_{i-1}|}$ is the weight matrix of layer l_i. In the remainder of this chapter, we exchangeably use w^i_{jk} or $w(s_{i-1,k}, s_{ij})$ to represent the weight of the edge connecting the node $s_{i-1,k} \in S_{i-1}$ to the node $s_{ij} \in S_i$.

Remark 4.1 *For the sake of simplicity of formulas and proofs, we did not include biases explicitly in the definition of NN. In fact, the bias vector* b_i *of a layer* l_i *can be replaced by a weight vector connecting a new node* $s_{i-1,b} \in S_{i-1}$ *to all the nodes* s_{ij} *of* S_i *such that:* $v(s_{i-1,b}) = 1$ *and* $w(s_{i-1,b}, s_{ij}) = b_{ij}$*. An illustrative example is given in Figure 4.1.*

4.2.2 Interval Neural Networks

As presented in chapter 2, the parameters of a NN are scalars, usually of type float ($\in \mathbb{R}$). However, Prabhakar and Rahimi Afzal (2019) have recently introduced a new structure to represent an NN, namely Interval Neural Networks (INN). This structure is initially designed to build an INN that captures and abstracts some behaviors of an original standard NN.

In the context of INNs, the weights (and biases) are represented as intervals. These intervals define the lower and upper bounds of each weight. Formally, a weight w within an INN model is defined as: $w = [w^l, w^u]$, where w^l and w^u are the respective lower and upper bounds, such that $w^l \in \mathbb{R}$, $w^u \in \mathbb{R}$, and $w^l \leq w^u$. Notice that, due to the fact that the weights are intervals, the output of an INN is always an interval, even when the received input is a scalar.

Figure 4.2 depicts an example of an INN. The presented INN has three layers (input, hidden and output layers), where the weights connecting the successive

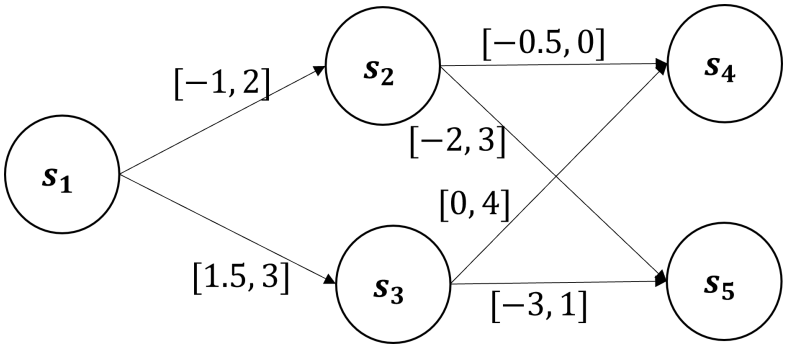
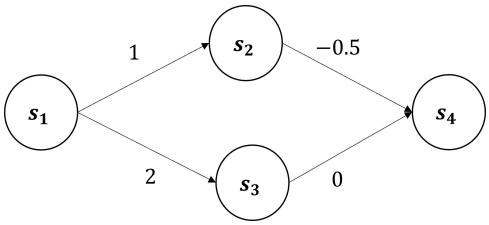


Figure 4.2: An example of an INN of three layers.

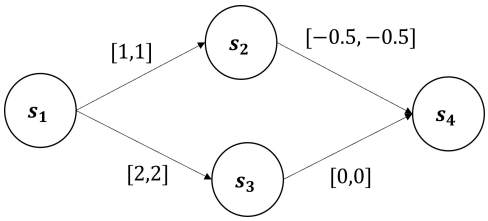
layers are intervals. For $v(s_1) = 1$, the corresponding values of the outputs are: $v(s_4) = [5, 12.5]$ and $v(s_5) = [-13, 9]$.

In fact, INNs can be seen as a general representation of NNs. Notice that any NN can be transformed into an equivalent INN by defining its weights as $w = [w^l, w^u]$, such that $w^l = w^u$. An example illustrating this transformation is presented in Figure 4.3.

In the sequel, and in order to simplify the presentation and the notations, we consider the INN representation of the NN to be abstracted; thus, with a slight abuse of notation, we can write that $N(x) \subseteq \bar{N}(x)$.



(a) A standard NN.

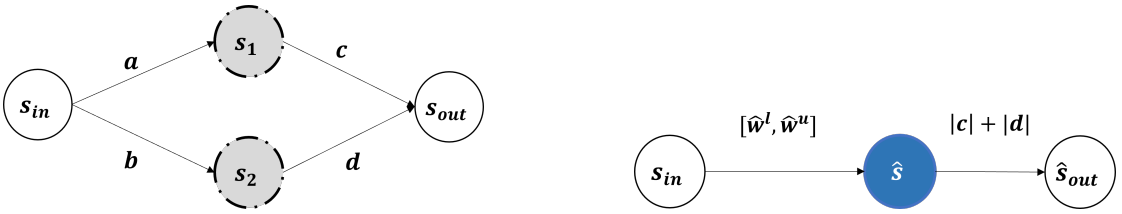


(b) The equivalent INN.

Figure 4.3: An example showing the transformation of an NN (a) to an equivalent INN (b).

4.3 *The INNAbstract* Approach

The objective of our method is to reduce the NN size by merging some sets of nodes. Abstracting a set of nodes requires providing formulas to determine the incoming and outgoing weights of the obtained abstract node, and a formal proof ensuring that the reduced network over-approximates by design the original one. The broad idea of the approach is to define the output weights of the abstract node as the sum of the absolute values over the outgoing weights of the set of



(a) The original network. **(b)** The abstract network.

Figure 4.4: An example explaining the main idea of the proposed approach, where the incoming weight to the abstract node $\hat{s}$ is: $\hat{w}^l = \min\{sign(c)\times a, sign(d)\times b\}$ and $\hat{w}^u = \max\{sign(c)\times a, sign(d)\times b\}$.

merged nodes, and bring back their signs to be used with their incoming weights to calculate the new incoming weights of the abstract node. Figure 4.4 provides an illustration of the abstraction procedure, where Figure 4.4a represents the original network N, and Figure 4.4b the obtained abstract network. Mathematical formulations and examples of the application of the proposed method are presented below. First, we start by giving a general formula for NNs with odd and monotone activation functions, and then provide a modified version to support NNs with the *ReLU* activation function.

4.3.1 Model Reduction for NN with Odd Activation Functions

Before discussing the details of the approach, let us first recall the definitions of an odd function and a monotone function.

Definition 4.4 *A function $g : \mathbb{R} \to \mathbb{R}$ is odd if:*

$$\forall x \in \mathbb{R} : g(-x) = -g(x)$$

Definition 4.5 *A function $g : D_g \subseteq \mathbb{R} \to \mathbb{R}$ is monotone if it is either (exclusively) increasing or decreasing over its entire domain D_g.*

1. *g is monotone increasing if:*

$$\forall x_1, x_2, \in D_g, if\, x_1 \leq x_2 \implies g(x_1) \leq g(x_2)$$

2. *g is monotone decreasing if:*

$$\forall x_1, x_2, \in D_g, if\, x_1 \leq x_2 \implies g(x_1) \geq g(x_2)$$

Examples of some used odd activation functions are *Tanh*, *Bipolar Sigmoid*, *LeCun's Tanh*, the *identity function*, *Hard Tanh*, etc. (Nwankpa et al. 2018).

Our method consists in merging a set of nodes $\hat{S} \subseteq S_i$, where S_i is the set of nodes of hidden layer l_i, and replace the subset $\hat{S}$ by an abstract node $\hat{s}$ which is determined by its weights as follows:

1. Incoming weights:
   $\forall s_{i-1,k} \in S_{i-1}$, $w(s_{i-1,k}, \hat{s}) = [\hat{w}_k^l, \hat{w}_k^u]$, such that:

$$\begin{cases} \hat{w}_k^l = \min\limits_{s_i \in \hat{S}, s_{i+1,j} \in S_{i+1}} \{sign(w(s_i, s_{i+1,j})) \times w(s_{i-1,k}, s_i)\} \\ \hat{w}_k^u = \max\limits_{s_i \in \hat{S}, s_{i+1,j} \in S_{i+1}} \{sign(w(s_i, s_{i+1,j})) \times w(s_{i-1,k}, s_i)\} \end{cases} \tag{4.6}$$

2. Outgoing weights:

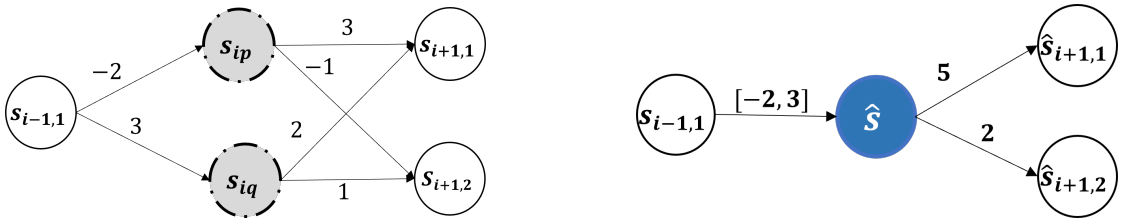
$$\forall s_{i+1,j} \in S_{i+1}, w(\hat{s}, s_{i+1,j}) = \sum_{s_i \in \hat{S}} |w(s_i, s_{i+1,j})| \tag{4.7}$$

where $|.|$ is the absolute value, and $sign$ is the sign function defined as: $sign : \mathbb{R} \to \{-1, 1\}$

$$sign(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{4.8}$$

The selection of nodes of $\hat{S}$ can be done using some nodes selection strategies, such as random selection or heuristics based selection. More details about nodes selection strategies will be discussed later in this chapter.

The main steps of the abstraction method are presented in Algorithm 1. The algorithm takes as input two parameters: the original network N and the nodes selection strategy. First, a copy of the original network, denoted $\bar{N}$, is generated. Next, for each layer, a partition P_i of disjoint subsets of S_i is created using the nodes selection strategy procedure *SelectNodes*. In other words, P_i is a set of subsets $\hat{S}_k$, each contains a number of neurons to be merged. Assume $m \leq |S_i|$ the number of subsets fo neurons to be merged, $P_i = \{\hat{S}_1, \ldots, \hat{S}_m\}$, where $(\hat{S}_k, \hat{S}_{k'}) \subseteq S_i \times S_i$ and $\hat{S}_k \cap \hat{S}_{k'} = \emptyset$, $\forall k, k' \in \{1, \ldots, m\}$. Once *SelectNodes* is completed, the procedure *AbstractOneLayer* presented in Algorithm 2 is applied for merging each subset $\hat{S}$ in the partition P_i and calculating the weights of the obtained abstract nodes. Finally, the algorithm returns the abstract netwrok $\bar{N}$. *SelectNodes* may correspond to a random selection, or may implement some heuristics as



(a) A network N: s_{ip} and s_{iq} are to be merged.

(b) The obtained abstract model $\overline{N}$.

Figure 4.5: An example of the abstraction method applied on two neurons of a hidden layer l_i. For $v(s_{i-1,1}) = 1$, we have $v(s_{i+1,1}) = 0$ and $v(s_{i+1,2}) = 5$, $v(\hat{s}_{i+1,1}) = [-10, 15]$ and $v(\hat{s}_{i+1,2}) = [-4, 6]$. Hence, the over-approximation is fulfilled, since $v(s_{i+1,k}) \in v(\hat{s}_{i+1,k})$ for $k = 1, 2$.

will be discussed in Section 4.3.3.

The execution of Algorithm 1 is illustrated in Figure 4.5. In this example, we assume that the network uses the identity activation function ($\alpha(x) = x$). The application of Algorithm 1 on a network N allows the generation of an abstract network that is an over-approximation of N. Based on Algorithm 1, the following results can be stated.

Algorithm 1 Pseudo-code of INNAbstract

```
1: procedure ABSTRACT(N, selectStrategy)
2:     N̄ ← duplicate(N)                    ▷ Initialize the abstract network
3:     for i ← 1 to n − 1 do                ▷ Explore all layers
4:         P_i ← SelectNodes(N̄, i, selectStrategy)
5:         AbstractOneLayer(N̄, i, P_i)
6:     end for
7:     return N̄
8: end procedure
```

Proposition 4.1 *Let N be a NN. Assume that the activation function of N is odd and monotone. Let $\hat{S}_i = (S_i \backslash \hat{S}) \cup \{\hat{s}\}$ be the set of nodes in layer i after applying the abstraction procedure on this layer. Then, we have:*

$$\forall v(S_{i-1}),\ v(S_{i+1}) \in v(\hat{S}_{i+1})$$

Proposition 4.2 *Let N be a NN. Assume that the activation function of N is odd and monotone. Let $\bar{N}$ be the abstract network of N, obtained using Algorithm 1. Then we have :*

$$\forall x \in \mathbb{R}^{|S_0|},\ N(x) \subseteq \bar{N}(x)$$

Algorithm 2 Pseudo-code of the abstraction method applied on one hidden layer l_i.

```
1: procedure AbstractOneLayer(N, i, P_i ) ▷ P_i a set containing the partitions
2:     for every subset Ŝ in P_i do
3:         create a node ŝ
4:         for s_{i−1,k} ∈ S_{i−1} do
5:             calculate ŵ(s_{i−1,k}, ŝ) using Equation (4.6)
6:         end for
7:         for s_{i+1,j} ∈ S_{i+1} do
8:             calculate ŵ(ŝ, s_{i+1,j}) using Equation (4.7)
9:         end for
10:        for s_i ∈ Ŝ do
11:            remove s_i
12:        end for
13:        add ŝ to S_i
14:    end for
15: end procedure
```

The proof of Proposition 4.1 is given below. The proof of Proposition 4.2 is straightforward and can be derived from the proof of Proposition 4.1 by propagating the abstraction rom layer l_i to l_{i+1}, starting from layer l_1 up to l_{n-1}.

Proof of Proposition 4.1

In Proposition 4.1, $v(S_{i+1}) \in v(\hat{S}_{i+1})$ means that $v(s_{i+1,j}) \in v(\hat{s}_{i+1,j}), \forall j \in \{1, 2, \dots m|\}$, where $m = |S_{i+1}| = |\hat{S}_{i+1}|$. *Proposition 1* considers merging many nodes on a hidden layer l_i, yet for the sake of simplicity, we provide the proof of merging two nodes. The generalization of this proof by considering multiple nodes can be done following the same steps.

Assume that we want to merge two hidden nodes s_1 and s_2 of a hidden layer l_i, such that l_{i-1}, l_i and l_{i+1} all have an odd and monotone activation function, and let $\hat{s}$ denote the obtained abstract node.

We denote by a_k (resp. b_k) the incoming weights of s_1 (resp. s_2) from node $s_{i-1,k} \in S_{i-1}$,

$$\begin{cases} a_k = w(s_{i-1,k}, s_1) \\ b_k = w(s_{i-1,k}, s_2) \end{cases}$$

and we denote by c_j (resp. d_j) the outgoing weight of s_1 (resp. s_2) toward node

$s_{i+1,j} \in S_{i+1}$, i.e.,

$$\begin{cases} c_j = w(s_1, s_{i+1,j}) \\ d_j = w(s_2, s_{i+1,j}) \end{cases}$$

The weights of the remaining edges connecting other nodes than s_1 and s_2 ($s \in S_i \setminus \{s_1, s_2\}$) to each node $s_{i+1,j} \in S_{i+1}$ are defined as follows:

$$w(s, s_{i+1,j}) = w_{s,s_j}$$

We denote by $\hat{s}$ the obtained abstract node upon merging s_1 and s_2 using Algorithm 2. The new abstract layer is denoted as $\hat{S}_i = \hat{s} \cup \{S_i \setminus \{s_1, s_2\}\}$. And for clarity, we use v_k instead of $v_{i-1,k}$. Since the activation function α is monotone and odd, we can state the following:

$$\forall x \in \mathbb{R} : \alpha(-x) = -\alpha(x)$$

and

$$\forall x_1, x_2 \in \mathbb{R} : x_1 \leq x_2 \implies \alpha(x_1) \leq \alpha(x_2)$$

For each $j \in \{1, 2, ..., |S_{i+1}|\}$, we have:

$$v_{i+1,j} = \alpha(z_{i+1,j}) = \alpha\left(c_j \times \alpha(\sum_{k=1}^{m} a_k v_k) + d_j \times \alpha(\sum_{k=1}^{m} b_k v_k) + \sum_{s \in S_i \setminus s1, s2} w_{s,s_j} v(s)\right)$$

$$\hat{v}_{i+1,j} = \alpha(\hat{z}_{i+1,j}) = \alpha\left((|c_j| + |d_j|) \times \alpha(\sum_{k=1}^{m} \hat{w}_k v_k) + \sum_{s \in S_i \setminus s1, s2} w_{s,s_j} v(s)\right)$$

where $z_{i+1,j}$ (resp. $\hat{z}_{i+1,j}$) is the value of $s_{i+1,j}$ (resp. $\hat{s}_{i+1,j}$) before applying the activation function α, $m = |S_{i-1}|$ is the number of nodes of layer l_{i-1} and $\hat{w}_k = [\hat{w}_k^l, \hat{w}_k^u]$ is the abstract weight connecting each node $s_k \in S_{i-1}$ to the abstract node $\hat{s} \in \hat{S}_i$ as defined in Algorithm 2. The main result is to prove that $v_{i+1,j} \in \hat{v}_{i+1,j}$.

By definition of $\hat{w}_k$ we have $\forall k = \{1, 2, ..., m\}, \forall j \in \{1, 2, ..., n\}$:

$$\begin{cases} \hat{w}_k^l \leq sign(c_j) a_k \leq \hat{w}_k^u \\ \hat{w}_k^l \leq sign(d_j) b_k \leq \hat{w}_k^u \end{cases}$$

Thus, we have :

$$\begin{cases} \hat{w}_k^l v_k \leq sign(c_j) a_k v_k \leq \hat{w}_k^u v_k & \text{if } v_k \geq 0 \\ \hat{w}_k^u v_k \leq sign(c_j) a_k v_k \leq \hat{w}_k^l v_k & \text{Otherwise} \end{cases}$$

Two cases need to be considered, $v_k \geq 0$ and $v_k < 0$. We provide a detailed proof for the case $v_k \geq 0$, below. The proof of the second case, i.e., $v_k < 0$, is omitted and can be inferred similarly by just inverting the lower and upper bounds of $\hat{w_k}$ when multiplying it by v_k.

Assume that $v_k \geq 0$, multiplying the aforementioned inequality by $sign(c_j)$, we obtain:

$$\begin{cases} \text{if } c_j \geq 0: \\ \quad sign(c_j)\hat{w}_k^l v_k \leq sign(c_j)sign(c_j)a_k v_k \leq sign(c_j)\hat{w}_k^u v_k \\ \text{Otherwise:} \\ \quad sign(c_j)\hat{w}_k^u v_k \leq sign(c_j)sign(c_j)a_k v_k \leq sign(c_j)\hat{w}_k^l v_k \end{cases}$$

The proof contains two parts: $c_j \geq 0$ and $c_j < 0$. Here we provide details of the proof for the first case: $sign(c_j) \geq 0$. The proof of the other case ($sign(c_j) < 0$) can be performed following the same logic, using the fact that α is odd, and hence $\alpha(sign(c_j) \times x) = sign(c_j) \times \alpha(x)$, and that we have $sign(c_j) \times c_j = |c_j|$

For $sign(c_j) \geq 0$ and by taking the sum over all k we have:

$$\sum_{k=1}^{m} sign(c_j)\hat{w}_k^l v_k \leq \sum_{k=1}^{m} a_k v_k \leq \sum_{k=1}^{m} sign(c_j)\hat{w}_k^u v_k$$

And since α is monotone increasing, we have:

$$\alpha(\sum_{k=1}^{m} sign(c_j)\hat{w}_k^l v_k) \leq \alpha(\sum_{k=1}^{m} a_k v_k) \leq \alpha(\sum_{k=1}^{m} sign(c_j)\hat{w}_k^u v_k)$$

On the other hand, since α is odd, we have $\alpha(sign(c_j) \times x) = sign(c_j) \times \alpha(x)$, thus the previous inequality is equivalent to:

$$sign(c_j)\alpha(\sum_{k=1}^{m} \hat{w}_k^l v_k) \leq \alpha(\sum_{k=1}^{m} a_k v_k) \leq sign(c_j)\alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) \tag{4.9}$$

Multiplying Equality (4.9) by c_j gives:

$$|c_j| \times \alpha(\sum_{k=1}^{m} \hat{w}_k^l v_k) \leq c_j \times \alpha(\sum_{k=1}^{m} a_k v_k) \leq |c_j| \times \alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) \tag{4.10}$$

Recalling that $sign(c_j) \times c_j = |c_j|$. The same reasoning applies when we replace a_k by b_k and c_j by d_j:

$$|d_j| \times \alpha(\sum_{k=1}^{m} \hat{w}_k^l v_k) \leq d_j \times \alpha(\sum_{k=1}^{m} b_k v_k) \leq |d_j| \times \alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) \tag{4.11}$$

Summing inequalities (4.10) and (4.11) with the remaining incoming nodes values to $s_{i+1,j} \in S_{i+1} \setminus \{s_1, s_2\}$ we obtain:

$$\begin{aligned}
&(|c_j| + |d_j|) \times \alpha(\sum_{k=1}^{m} \hat{w}_k^l v_k) + \sum_{s \in S_i \setminus s1,s2} w_{s,s_j} v(s) \\
&\leq c_j \times \alpha(\sum_{k=1}^{m} a_k v_k) + d_j \times \alpha(\sum_{k=1}^{m} b_k v_k) + \sum_{s \in S_i \setminus s1,s2} w_{s,s_j} v(s) \\
&\leq (|c_j| + |d_j|) \times \alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) + \sum_{s \in S_i \setminus s1,s2} w_{s,s_j} v(s)
\end{aligned} \tag{4.12}$$

And since α is monotone, by applying the activation α on the Inequality (4.12), we obtain:

$$\hat{v}(\hat{s}_{i+1,j})^l \leq v(s_{i+1,j}) \leq \hat{v}(\hat{s}_{i+1,j})^u$$

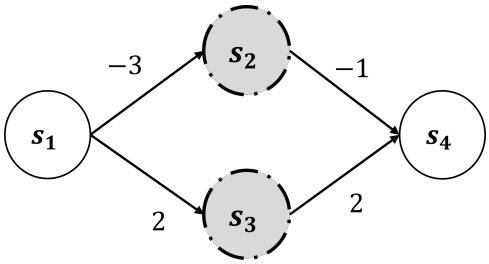
Finally we conclude that:

$$v_{i+1,j} \in \hat{v}_{i+1,j} \ , \ \forall j \in \{1, ..., n\}$$

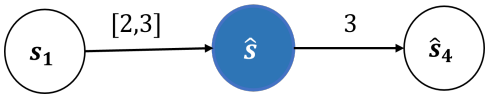
□

4.3.2 Model Reduction Method for *ReLU*-NN

In this section, we present a relaxation of our method presented in Section 4.3.1 in such a way as it can support *ReLU*-NNs. Recall that the piece-wise linear function *ReLU* is monotone; however it is not odd, thus the model reduction method presented in the previous section is not straightforwardly applicable on *ReLU*-NN. Concretely, applying the approach presented in the previous section on a *ReLU*-NN does not guarantee the over-approximation, i.e., the output of



(a) A network N: s_2 and s_3 are to be merged.



(b) The obtained abstract model $\overline{N}$.

Figure 4.6: A counterexample of applying Algorithm 2 on a *ReLU*-NN.

the original network may not be within the range of the output of the reduced network.

For illustration, a counterexample is presented in Figure 4.6, where the original network is on the left (Figure 4.6b) and the reduced network on the right (Figure 4.6a) is obtained using our method presented in Section 4.3.1. Let consider that the activation function of the network is the *ReLU* functoin, and let assume that $v(s_1) = 2$, then the value of the neuron s_4 of the original network (Figure 4.6a) is $v(s_4) = 8$, and for the same input value ($v(s_1) = 2$), the value of $\hat{s}_4$ of the reduce network (Figure 4.6b) is $v(\hat{s}_4) = [12, 18]$. Hence $v(s_4) \notin v(\hat{s}_4)$ and the over-approximation is not preserved.

The idea of this relaxation for *ReLU*-NNs is to check whether the incoming weight to every node $s \in \hat{S}$ has the same sign as its corresponding outgoing weight, and if so, the lower bounds are calculated using Equation (4.13) instead of Equation (4.6)[1].

$$\begin{cases} \hat{w}_k^l = \min\limits_{s_i \in \hat{S}} \{w(s_{i-1,k}, s_i)\} \\ \hat{w}_k^u = \max\limits_{s_i \in \hat{S}, s_{i+1,j} \in S_{i+1}} \{sign(w(s_i, s_{i+1,j})) \times w(s_{i-1,k}, s_i)\} \end{cases} \tag{4.13}$$

For two nodes $s_{i-1,k} \in S_{i-1}$ and $s \in \hat{S} \subseteq S_i$, we define $w^*_{i+1,j} \in \{w(s, s_{i+1,j}) : s_{i+1,j} \in S_{i+1}\}$ as the weight minimizing $\{sign(w(s, s_{i+1,j})) \times w(s_{i-1,k}, s)\}$. For a subset $\hat{S} \subseteq P_i$ of nodes to be abstracted, if all the concrete nodes $s \in \hat{S}$ satisfy Equation (4.14) below, then the incoming weights of the abstract node $\hat{s}$ are

[1]The formula for calculating the upper bounds of the incoming weights is the same as NN with odd and monotone, i.e., Equation (4.7)

calculated using Equation (4.13).

$$\bigwedge_{s \in \hat{s}} \Big(sign(w^*_{i+1,j}) = sign(w(s_{i-1,k}, s)) \Big) \tag{4.14}$$

The second step consists of replacing every scalar-weight w of layer l_i by an interval weight $\hat{w} = [min(w, 0), w]$. This is done at the end of the abstraction process of layer l_i and before switching to layer l_{i+1}. The main steps of abstracting a set of nodes of a hidden layer l_i of a *ReLU*-NN are summarized in Algorithm 3. The algorithm can be applied on multiple layers as presented in Algorithm 4.

Algorithm 3 Pseudo-algorithm of the abstraction method applied on one *ReLU* hidden layer l_i.

```
1: procedure Abstract1ReLULayer(N, i, P_i )  ▷ N: the network, i : the layer's number
2:     for every subset Ŝ in P_i do
3:         create a node ŝ
4:         for s_{i-1,k} ∈ S_{i-1} do
5:             calculate ŵ(s_{i-1,k}, ŝ) using Equation (4.6) and determine w*_{i+1,j}
6:             if Equation (4.14) holds then
7:                 update ŵ(s_{i-1,k}, ŝ) using Equation (4.13)
8:             end if
9:         end for
10:        for s_{i+1,j} ∈ S_{i+1} do
11:            calculate ŵ(ŝ, s_{i+1,j}) using Equation (4.7)
12:        end for
13:        for s_i ∈ Ŝ do
14:            remove s_i
15:        end for
16:        add ŝ to S_i
17:    end for
18:    replace all the scalar-weights w of layer l_i by an interval [min(w, 0), w]
19: end procedure
```

In the following, we state the main results pertaining to the abstraction of *ReLU*-NNs, then we provide the complete proof of Proposition 4.3. The proof of Proposition 4.4 can be derived from that of Proposition 4.3 by straightforwardly propagating the abstraction from layer l_i to l_{i+1}, up to l_{n-1}.

Proposition 4.3 *Let N be a NN. Assume that ReLU is the activation function of N. Let $\hat{S}_i = (S_i \backslash \hat{S}) \cup \{\hat{s}\}$ be the set of nodes in the i^{th} layer upon abstracting the nodes in*

Algorithm 4 Pseudo-algorithm of INNAbstract for *ReLU*-NNs.

```
1: procedure Abstract(N, selectStrategy)
2:     N̄ ← duplicate(N)                          ▷ Initialize the abstract network
3:     for i ← 1 to n − 1 do                       ▷ Explore all layers
4:         Ŝ ← SelectNodes(N̄, i, selectStrategy)
5:         Abstract1ReLULayer(N̄, i, Ŝ)
6:     end for
7:     return N̄
8: end procedure
```

$\hat{S} \subseteq S_i$ *into* $\hat{s}$. *Then we have :*

$$\forall v(S_{i-1}),\, v(S_{i+1}) \in\ v(\hat{S}_{i+1})$$

Proposition 4.4 *Let N be a NN. Assume that ReLU is the activation function of N. Let $\bar{N}$ be the abstract network of N obtained using Algorithm 4. Then the following holds:*

$$\forall x \in \mathbb{R}^{|S_0|}, N(x) \subseteq \bar{N}(x)$$

Proof of Proposition 4.3

Proposition 4.3 considers merging certain nodes in a hidden layer l_i; yet for the sake of clarity and readability, we provide the proof of merging two nodes. The generalization of this proof by considering multiple nodes can be done following the same steps.

Assume that we want to merge two hidden nodes s_1 and s_2 of some given hidden layer l_i, such that l_{i-1}, l_i and l_{i+1} are all *Relu*-layers, and denote by $\hat{s}$ the obtained abstract node be.

We denote by a_k the incoming weight of s_1 from node $s_{i-1,k} \in S_{i-1}$, similarly, b_k denotes the incoming weight of s_2 from node $s_{i-1,k}$. i.e.,

$$\begin{cases} a_k = w(s_{i-1,k}, s_1) \\ b_k = w(s_{i-1,k}, s_2) \end{cases}$$

Let us denote by c_j (resp. d_j) the outgoing weights of s_1 (resp. s_2) toward node $s_{i+1,j} \in S_{i+1}$, i.e.,

$$\begin{cases} c_j = w(s_1, s_{i+1,j}) \\ d_j = w(s_2, s_{i+1,j}) \end{cases}$$

The weights of the remaining edges connecting other nodes than s_1 and s_2

$(s \in S_i \setminus \{s_1, s_2\})$ to any node $s_j \in S_{i+1}$ are defined as follows:

$$w(s, s_j) = w_{s,s_j}$$

We denote by $\hat{s}$ the obtained abstract node after merging s_1 and s_2. The set of nodes of the new abstract layer is $\hat{S}_i = \hat{s} \cup (S_i \setminus \{s_1, s_2\})$.

We know that the activation function α is $Relu(x) = max(x, 0), \forall x \in \mathbb{R}$, and for clarity, we use v_k instead of $v_{i-1,k}$. For each $j \in \{1, 2, ..., n\}$, such that $n = |S_{i+1}|$, we have:

$$v_{i+1,j} = \alpha(z_{i+1,j}) = \alpha\left(c_j \times \alpha(\sum_{k=1}^{m} a_k v_k) + d_j \times \alpha(\sum_{k=1}^{m} b_k v_k) + \sum_{s \in S_i \setminus \{s1,s2\}} w_{s,s_j} v(s)\right)$$

and

$$\begin{aligned}\hat{v}_{i+1,j} &= \alpha(\hat{z}_{i+1,j}) \\ &= \alpha\left((|c_j| + |d_j|) \times \alpha(\sum_{k=1}^{m} \hat{w}_k v_k) + \sum_{s \in S_i \setminus \{s1,s2\}} \hat{w}_{s,s_j} v(s)\right)\end{aligned}$$

where $z_{i+1,j}$ (resp. $\hat{z}_{i+1,j}$) is the value of $s_{i+1,j}$ (resp. $\hat{s}_{i+1,j}$) before applying the activation function α, and $m = |S_{i-1}|$ is the number of nodes of layer l_{i-1}, $\hat{w}_{s,s_j} = [\min(w_{s,s_j}, 0), w_{s,s_j}]$, and $\hat{w}_k = [\hat{w}_k^l, \hat{w}_k^u]$ is the abstract weight connecting node $s_k \in S_{i-1}$ to the abstract node $\hat{s} \in \hat{S}_i$ as defined in Algorithm 3. The main result is to prove that $v_{i+1,j} \in \hat{v}_{i+1,j}$, which is equivalent to prove that:

1. $v_{i+1,j} \leq \hat{v}_{i+1,j}^u$, and
2. $\hat{v}_{i+1,j}^l \leq v_{i+1,j}$

1- Proving that $v_{i+1,j} \leq \hat{v}_{i+1,j}^u$

By definition of $\hat{w}_k$, we have:

$$\begin{cases} sign(c_j) a_k \leq \hat{w}_k^u \\ sign(d_j) b_k \leq \hat{w}_k^u \end{cases} \quad \forall k = \{1, 2, ..., m\}, \forall j \in \{1, 2, ..., n\}$$

In addition, by definition of *Relu* we know that: $v_k \geq 0, \forall k \in \{1, 2, ..., m\}$. Thus:

$$sign(c_j) a_k v_k \leq \hat{w}_k^u v_k, \forall k = \{1, 2, ..., m\}, \forall j \in \{1, 2, ..., n\}$$

This implies:

$$\sum_{k=1}^{m} sign(c_j)a_k v_k \leq \sum_{k=1}^{m} \hat{w}_k^u v_k$$

Because *Relu* is monotone increasing, applying *Relu* on the previous inequality gives:

$$\alpha(\sum_{k=1}^{m} sign(c_j)a_k v_k) \leq \alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k)$$

Using the property of *Relu*: $x \times Relu(y) \leq Relu(x \times y)$, we obtain:

$$sign(c_j)\alpha(\sum_{k=1}^{m} a_k v_k) \leq \alpha(\sum_{k=1}^{m} sign(c_j)a_k v_k) \leq \alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k)$$

Which implies:

$$c_j\alpha(\sum_{k=1}^{m} a_k v_k) \leq |c_j|\alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) \tag{4.15}$$

Following the same steps for d_j and b_k, we obtain:

$$d_j\alpha(\sum_{k=1}^{m} b_k v_k) \leq |d_j|\alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k) \tag{4.16}$$

From Equations (4.15) and (4.16):

$$c_j\alpha(\sum_{k=1}^{m} a_k v_k)+d_j\alpha(\sum_{k=1}^{m} b_k v_k)+\sum_{s\in S_i\setminus\{s1,s2\}} w_{s,s_j}v(s) \leq (|c_j|+|d_j|)\alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k)+\sum_{s\in S_i\setminus\{s1,s2\}} w_{s,s_j}v(s)$$

Finally, applying the activation function *Relu*:

$$\alpha(c_j\alpha(\sum_{k=1}^{m} a_k v_k)+d_j\alpha(\sum_{k=1}^{m} b_k v_k)+\sum_{s\in S_i\setminus\{s1,s2\}} w_{s,s_j}v(s)) \leq \alpha((|c_j|+|d_j|)\alpha(\sum_{k=1}^{m} \hat{w}_k^u v_k)+\sum_{s\in S_i\setminus\{s1,s2\}} w_{s,s_j}v(s))$$

which is equivalent to: $v_{i+1,j} \leq \hat{v}_{i+1,j}^u$.

2- Proving that $v_{i+1,j} \geq \hat{v}_{i+1,j}^l$

By definition of $\hat{w}_{s,s_j}$, we have $\hat{w}_{s,s_j}^l = \min(w_{s,s_j}, 0) \leq w_{s,s_j}$ for all j and all $s \in S_i \setminus \{s1, s2\}$. Then, since $v(s) \geq 0$ for all $s \in S_i \setminus \{s1, s2\}$, we have

$$\sum_{s\in S_i\setminus\{s1,s2\}} \hat{w}_{s,s_j}^l v(s) \leq \sum_{s\in S_i\setminus\{s1,s2\}} w_{s,s_j}v(s). \tag{4.17}$$

The definition of $\hat{w}_k$ in Algorithm 3 can be decomposed into three cases for each $k \in \{1,\dots,m\}$:

- Case 1: $sign(a_k) \neq sign(c_j^*)$ or $sign(b_k) \neq sign(d_j^*)$.
- Case 2: $sign(a_k) = sign(c_j^*)$ and $sign(b_k) = sign(d_j^*)$, with either $a_k \leq 0$ or $b_k \leq 0$.
- Case 3: $sign(a_k) = sign(c_j^*)$ and $sign(b_k) = sign(d_j^*)$, with both $a_k \geq 0$ and $b_k \geq 0$.

where c_j^* and d_j^* are the corresponding weights $w_{i+1,j}^*$ for s_1 and s_2, respectively (please see Section IV-B).

We also know that by definition of $\hat{w}_k$, in Cases 1 and 2 we have $\hat{w}_k^l \leq 0$, and in Case 3 we have $\hat{w}_k^l \geq 0$.

If we are in Cases 1 or 2 for all $k \in \{1,\dots,m\}$, then we have $\hat{w}_k^l \leq 0$ for all k, which implies that $\sum_{k=1}^m \hat{w}_k^l v_k \leq 0$ (since $v_k \geq 0$ for all k), and then $(|c_j| + |d_j|)\alpha(\sum_{k=1}^m \hat{w}_k^l v_k) = 0$ since $\alpha(x) = \max(x,0)$. From the definition of $\hat{w}_{s,s_j}^l = \min(w_{s,s_j},0) \leq 0$, this implies that $\hat{z}_{i+1,j}^l \leq 0$ and thus $\hat{v}_{i+1,j}^l = \alpha(\hat{z}_{i+1,j}^l) = 0$. Since $\alpha(x) \geq 0$ for all x, we have $v_{i+1,j} = \alpha(z_{i+1,j}) \geq 0$ which allows us to conclude that $\hat{v}_{i+1,j}^l \leq v_{i+1,j}$.

Otherwise, there is necessarily at least one k in Case 3, which means that for this particular k we have $a_k \geq 0$, $b_k \geq 0$, $c_j^* \geq 0$ and $d_j^* \geq 0$. By definition of c_j^* and d_j^*, we can conclude that for all $j \in \{1,\dots,n\}$ we have $c_j \geq 0$ and $d_j \geq 0$. This implies that the definition of $\hat{w}_k^l$ is reduced to $\hat{w}_k^l = \min\{a_k, b_k\}$ for all $k \in \{1,\dots,m\}$. From the knowledge that $\hat{w}_k^l \leq a_k$, $\hat{w}_k^l \leq b_k$ and $v_k \geq 0$ for all $k \in \{1,\dots,m\}$, and that the activation function α is monotonically increasing, we obtain:

$$\alpha\Big(\sum_{k=1}^{m} \hat{w}_k^l v_k\Big) \leq \alpha\Big(\sum_{k=1}^{m} a_k v_k\Big) \tag{4.18}$$

and

$$\alpha\Big(\sum_{k=1}^{m} \hat{w}_k^l v_k\Big) \leq \alpha\Big(\sum_{k=1}^{m} b_k v_k\Big). \tag{4.19}$$

Then, multiplying (4.18) by $|c_j| = c_j$, (4.19) by $|d_j| = d_j$, and taking the sum of the resulting two inequalities and (4.17) gives $\hat{z}_{i+1,j}^l \leq z_{i+1,j}$. From the monotonicity of α, we can finally conclude that $\hat{v}_{i+1,j}^l \leq v_{i+1,j}$. □

4.3.3 A Heuristic strategy for Nodes Selection

It is worth noticing that the efficiency and the precision of the reduced model strongly depends on the characteristics of the merged nodes. One of the most important features is the sign of the outgoing weights, since we use this feature to calculate the weights of the abstract node. Therefore, we propose a heuristic method to select the set of nodes to be merged in each layer based on the sign of their outgoing weights. Recall that the sign of outgoing weights of the initial network affects the sign of both incoming and outgoing weights of the abstract network after merging (multiplying incoming weights by this sign, and taking the absolute value for outgoing weights). Therefore, intuitively, it seems more likely that the abstract network will be closer to the original one when the outgoing weights are positive.

In our approach, we propose a heuristic for nodes selection giving priority to nodes having positive outgoing weights, i.e., the nodes that have more outgoing positive weights are to be merged first. This is done by calculating the *sign* of the outgoing weight matrix of the hidden layer l_i, sum the obtained matrix by rows, and then sort the obtained list. Finally, the selection of the nodes to be merged is done by picking a number of nodes from the top of this ordered list.

Algorithm 5 summarizes the main steps of the heuristics based nodes selection. In the algorithm, the parameter *nbAbst* determines the number of nodes to be merged. Notice also that $NodeSelectionHeuris$ is used within the procedure *SelectNodes* in Algorithms 1 and 4 by setting the parameter $selectStrategy$ to $heursitics$, i.e., $selectStrategy = heursitics$.

Algorithm 5 Nodes selection strategy based on heuristics.

1: **procedure** NodeSelectionHeuris(N, i, nbAbst)
2: $W \leftarrow$ the outgoing weight matrix of layer l_i
3: $W_{sign} \leftarrow sign(W)$ ▷ calculate the sign matrix of W
4: $sum_{rows} \leftarrow sum(W_{sign}, axis = 1)$ ▷ Sum by row
5: $sort(sum_{rows})$
6: save the first *nbAbst* nodes in a set $\hat{S}$
7: return $\hat{S}$
8: **end procedure**

4.4 Model Reduction approach for Non-negative Activation Functions

The approach presented in the previous section supports NNs with odd and monotone activation functions, and also *ReLU*-NNs with the proposed relaxation. In this section, we present a second approach that supports NNs with any monotone non-negative activation function. A non-negative function is a function that does not return negative values (see Definition 4.6). For instance, the *ReLU* and *Sigmoid* functions, presented in Equations (4.3) and (4.4), respectively, are examples of non-negative activation functions.

Definition 4.6 *A function* $\alpha : \mathbb{R} \rightarrow \mathbb{R}$ *is non-negative if:* $\forall x \in \mathbb{R} : \alpha(x) \geq 0$.

The aim of this method is to construct a reduced network, denoted as $\overline{N}$, from the original network N by merging a set of neurons within the same hidden layer, while ensuring that the output of $\overline{N}$ is always greater than or equal to the output of the original network N, formally written:

$$\forall x \in \mathbb{R}^{|S_0|} : \overline{N}(x) \geq N(x) \tag{4.20}$$

The approach is mainly used to check properties of the form $\forall x, Pre(x) \implies N(x) < c$. Notice that other properties of interest can be reformulated (or reduced) to a one-single output property by adding additional neurons to the network (see Elboher et al. (2020) for more details).

In the following and for the sake of simplicity, we consider that the network N has a single output y (i.e., S_n has one neuron, $|S_n| = 1$) and we want to verify that $N(x) < c$, for a given constant $c \in \mathbb{R}$ and some constraints Pre on the input x, formally written:

$$\forall x, Pre(x) \implies N(x) < c \tag{4.21}$$

Thus, whenever we are able to demonstrate that $\overline{N}(x) < c$, we can directly infer the correctness of the property for the original network N:

$$\forall x, [Pre(x) \implies \overline{N}(x) < c) \implies (Pre(x) \implies N(x) < c]$$

To construct an abstract network $\overline{N}$ from the original network N, we start by eliminating the negative connections of the relevant neurons. Concretely, to abstract a set of nodes $\hat{S} \subseteq S_i$ of the hidden layer l_i, we first remove the negative

weights starting from the hidden layer l_i towards the succeeding layers until reaching the last hidden layer of the network. Then, we replace a set of neurons $\hat{S}$ with a single abstract neuron $\hat{s}$. This abstract neuron is then connected to the previous and next layers through new weighted edges. The calculation of the incoming and outgoing weights for this abstract neuron $\hat{s}$ is computed in a way to ensure that the over-approximation relation presented in Equation (4.20) is always satisfied, namely:

- ***Incoming weight:*** the maximum value among the incoming weights of the individual neurons to merge.
- ***Outgoing weight:*** the sum of the outgoing weights of the corresponding individual neurons to merge.

Formally, let us consider a set of neurons $\hat{S} \subseteq S_i$, which belong to a hidden layer l_i. Each neuron s in the set $\hat{S}$ has incoming and outgoing weights denoted as $w(s_{i-1,k}, s)$ and $w(s, s_{i+1,j})$, respectively. Here, $s_{i-1,k} \in S_{i-1}$ represents a neuron of the preceding layer l_{i-1}, and $s_{i+1,j} \in S_{i-1}$ represents a neuron of the succeeding layer l_{i+1}. The first step consists in eliminating negative weights to ensure that: $\forall i \leq t \leq n-1, \forall s_1 \in S_t \wedge s_2 \in S_{t+1} : w(s_1, s_2) \geq 0$. The second step is to replace the set of neurons $\hat{S}$ by the abstract neuron $\hat{s}$ such that the weights of $\hat{s}$ are calculated as follows:

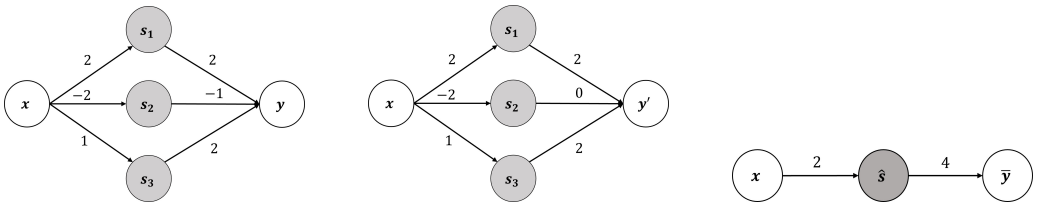
1. Incoming weights $w(s_{i-1,k}, \hat{s})$:

$$\forall s_{i-1,k} \in S_{i-1} : w(s_{i-1,k}, \hat{s}) = \max_{s \in \hat{S}} \{w(s_{i-1,k}, s)\} \tag{4.22}$$

2. Outgoing weights $w(\hat{s}, s_{i+1,j})$:

$$\forall s_{i+1,j} \in S_{i+1} : w(\hat{s}, s_{i+1,j}) = \sum_{s \in \hat{S}} w(s, s_{i+1,j}) \tag{4.23}$$

Figure 4.7 depicts an example of the model reduction method applied on a small network employing the *ReLU* activation function, where the objective in this example is to merge the set of neurons $\hat{S} = \{s_1, s_2, s_3\}$ of the original network N (Figure 4.7a) and replace it with the abstract neuron $\hat{s}$. The first step involves eliminating the negative outgoing weights of neurons in $\hat{S}$. The resulting network of this step is presented in Figure 4.7b. Subsequently, the model reduction method is applied on this network to merge the set of neurons $\hat{S}$,



(a) An example network N before abstraction (the original network).

(b) The obtained network after the pre-processing phase applied on the NN presented in Figure 4.7a (removing negative weights).

(c) The abstract network $\overline{N}$ using the proposed method.

Figure 4.7: The application of the proposed model reduction method on a toy example of NN.

and calculate the incoming and outgoing weights of the obtained abstract neuron $\hat{s}$ using Equations (4.22) and (4.23), respectively. The final abstract network is shown in Figure 4.7c. Notice that $\overline{N}(x) \geq N(x)$, for all possible value of x.

Remark 4.2 *Theoretically, the proposed method allows for merging all neurons, leading to a network with only one neuron in each hidden layer. However, in practical applications, this may not be beneficial as there is a trade-off to be found between reducing the size of the original model and maintaining the precision of the abstract model.*

Algorithm 6 provides a summary of the main steps involved in the model reduction procedure applied to some given hidden layer l_i. The procedure assumes that all neurons in set s have positive outgoing weights. To apply the model reduction on different hidden layers, a general procedure is presented in Algorithm 7. The algorithm begins by eliminating negative weights and then, for each set of neurons to be merged in a hidden layer l_i, it invokes the *AbstractOneLayer* procedure (presented in Algorithm 6). The algorithm finally returns the abstract network $\overline{N}$. In this algorithm, *SelectNodes* is the procedure responsible for selecting the sets of nodes to be merged. In this work, we applied random nodes selection. However, this procedure can be re-implemented to support some heuristics.

In the following, we state the main results for model reduction of NN with non-negative activation function.

Lemma 4.1 *Let $v_1, v_2, ..., v_n$ be positive numbers, and $w_1, w_2, ..., w_n$ be real numbers. Let $z = \sum_{i=1}^{n} w_i v_i$ and $\overline{z} = \sum_{i=1, w_i \geq 0}^{n} w_i v_i$. Then we have: $\overline{z} \geq z$.*

Algorithm 6 Pseudo-algorithm of the proposed model reduction method for one hidden layer.

procedure AbstractOnLayer(N, i, $\hat{S}$) ▷ N is the original network, $\hat{S}$is the set of nodes of layer l_i
  for $s_{i-1,k} \in S_{i-1}$ **do** ▷ Incoming weights
    $w(s_{i-1,k}, \hat{s}) \leftarrow \max_{s \in \hat{S}} \{w(s_{i-1,k}, s)\}$
  end for
  for $s_{i+1,j} \in S_{i+1}$ **do** ▷ Outgoing weights
    $w(\hat{s}, s_{i+1,j}) \leftarrow \sum_{s \in \hat{S}} w(s, s_{i+1,k})$
  end for
  replace $\hat{S}$ with $\hat{s}$
end procedure

Algorithm 7 Pseudo-algorithm of the proposed model reduction method.

procedure Abstract(N) ▷ N: the original network
  $\overline{N} \leftarrow duplicate(N)$ ▷ Create a copy of N
  for $i \leftarrow 1$ to $n-1$ in $\overline{N}$ **do** ▷ Pre-processing: remove negative weights
    for $s_1 \in S_i$ **do**
      for $s_2 \in S_{i+1}$ **do**
        if $w(s_1, s_2) < 0$ **then**
          $w(s_1, s_2) \leftarrow 0$
        end if
      end for
    end for
  end for
  for $i \leftarrow 1$ to $n-1$ in $\overline{N}$ **do**
    $\hat{S} \leftarrow SelectNodes(\overline{N}, i)$
    $AbstractOneLayer(\overline{N}, i, \hat{S})$
  end for
  return $\overline{N}$
end procedure

Proposition 4.5 *For a NN* $N : \mathbb{R}^{|S_0|} \to \mathbb{R}$ *that has a non-negative activation function. Let* $\overline{N} : \mathbb{R}^{|S_0|} \to \mathbb{R}$ *be the abstract network of N obtained using Algorithm 7. Then, the following holds:*

$$\forall x \in \mathbb{R}^{|S_0|}, \overline{N}(x) \geq N(x)$$

The proof of Proposition 4.5 relies on Lemma 4.1, which states that removing negative elements from a sum yields a value greater than the original sum including the negative values.

4.5 Discussion w.r.t. Related Works

As discussed in Chapter 3, existing formal verification methods for NNs do not scalar to large networks. This is mainly due to the fact that generated NN models can often be complex and non-linear. One promising solution to remedy this problem is model reduction. Model reduction methods aim in reducing the state space of the verification problem by reducing the number of neurons in the network. In this context, several approaches have been proposed in the literature (Elboher et al. 2020; Prabhakar 2022; Prabhakar and Rahimi Afzal 2019; Sotoudeh et al. 2020). Please refer to chapter 3 and to our review (Boudardara, Boussif, Meyer, et al. 2023a) for more details.

Compared to the aforementioned model reduction works, the approaches we proposed in the present chapter are more general. Indeed, while most of the discussed works in chapter 3 are limited to the *ReLU* activation function, our methods support NNs with different types of activation functions, including Tanh, Sigmoid, *ReLU*, etc. Moreover, compared to (Ashok et al. 2020), where the abstraction depends on the set of input images on which one wants to check the robustness of the network, the presented approaches generate abstract networks independently of any specific input domains.

We should mention that the two techniques presented in (Elboher et al. 2020) and (Liu, Xing, et al. 2022), focus on categorizing the weights of the network by analyzing their signs and impact on the output (increasing or decreasing). Both techniques require a preprocessing phase to classify all nodes of the networks. Additionally, these approaches can only merge nodes that belong to the same category. In contrast, INNAbstract does not require any preprocessing phase, and if applied until saturation, a layer can (theoretically) be reduced to a single abstract node. Finally, we should highlight that the approach proposed in Prabhakar and Rahimi Afzal (2019) is the closest to INNAbstract, since both approaches are based on INNs. However, there is a significant advantage within our approach. Indeed, while Prabhakar and Rahimi Afzal (2019) define the abstract weights as the interval-hull of the corresponding weights in the original network, in our method we define the abstract outgoing weights as the sum of absolute values over the original weights, and transfer their signs to the corresponding incoming weights. This subtle technique helps building more precise abstract networks[2], as can be observed through the experimental results discussed in the next chapter.

[2]In the sense that its output is closer to the output of the original network.

Among the previously presented works (see Chapter 3), the method of Elboher et al. (2020) is the most similar one to our second proposed approach (Boudardara, Boussif, and Ghazel 2023). Both approaches share the same objective of constructing abstract networks whose output are greater than or equal to that of the original network. However, in their approach, Elboher et al. (2020) categorize neurons into four distinct groups, and when this is not applicable on the original network, a neuron splitting phase is performed. Beside the computation cost of this phase, neuron splitting may lead to generating a network up to four times larger than the original network. In addition, the minimum size of the generated abstract network using the method presented in Elboher et al. (2020) is four neurons per layer. In contrast to their approach (Elboher et al. 2020), our technique theoretically allows for merging all neurons within the same layer, and replace the whole layer with a single abstract neuron. Additionally, compared to the approach proposed by Prabhakar and Rahimi Afzal (2019), our method does not rely on INN. Consequently, the reduced networks generated by this approach can be analyzed using a variety of NN verifiers without requiring any adjustments or modifications.

4.6 Conclusion

In this chapter, we presented two methods for NN abstraction with the aim of reducing the size of NNs and ensuring that the obtained network is an over-approximation of the original one. This contributes to enhance the sclability of NN analysis operations such as verification.

The first approach, called ***INNAbstract***, consists of merging nodes of the same layer and provides a formula to calculate the new weights ensuring that the output range of the original model is always included within that of the abstract one. The approach leverages INNs, and supports NNs with *ReLU*, odd and monotone activation functions, including *Tanh* and *Sigmoid*. With the aim of enhancing the performance of the approach, we proposed a heuristic for nodes selection based on some features of the network's nodes.

Our second approach is a model reduction method that supports NNs with non-negative activation functions, such as *ReLU* and *Sigmoid*, and provides formal guarantees that the output of the abstract network is always greater than the output of the original one.

In the following chapter, we will evaluate these two approaches on a series of experiments, including randomly generated NNs and well-known benchmarks

from the literature. Additionally, an experimental comparison study with other relevant works from the literature will be presented and discussed.

Chapter 5

Experimental Evaluation of NN Model-Reduction Approaches

Outline of the current chapter

5.1 Introduction

In this chapter, we present the experimental settings used to evaluate our model reduction approaches introduced in the previous chapter (Chapter 4), and we discuss the obtained results. This experimental study includes evaluating the proposed approaches on randomly generated NNs, and on some known benchmarks, namely MNIST (LeCun 1998) and ACAS Xu (Kochenderfer 2015). Additionally, it includes a comparison study between our approaches and other reference approaches from the literature.

The *Tanh*-NNs, including randomly generated NNs and the MNIST benchmark, are only used in the evaluation of our INNAbstract approach. Indeed, our technique is the only method that supports the *Tanh* activation function. In contrast, the *ReLU*-NNs, i.e., including randomly generated NNs and the ACAS Xu benchmark, are used to evaluated our two approaches, while performing comparisons with some approaches from the literature. It is worth noticing that for a fair comparison with other existing model reduction methods, a random selection strategy is employed for all methods. On the other hand, the proposed heuristics for INNAbstract are employed specifically to check its impact on the INNAbstract method's performance. Figure 5.1 illustrates through a diagram the organization of our evaluation experiments.

The remaining of this chapter is organized as follows: Section 5.2.1 gives details about the implemented tools and the used machine-configuration to conduct this series of experiments. In Section 5.2.2, we present the various used NNs and benchmarks. Next, Section 5.3 exposes the obtained results. A detailed discussion of these results is provided within the same section. Finally, some concluding remarks are given in Section 5.4.

5.2 Experimental Setup and Configuration

5.2.1 Implementation and Experimental Environment

We implemented our two model reduction approaches as Python software tools. For both methods, the implementation includes the following three main modules:

1. **`Network_Reader`:** reads the NN model and the inputs' constraints. The tool supports both NNET and ONNX formats.

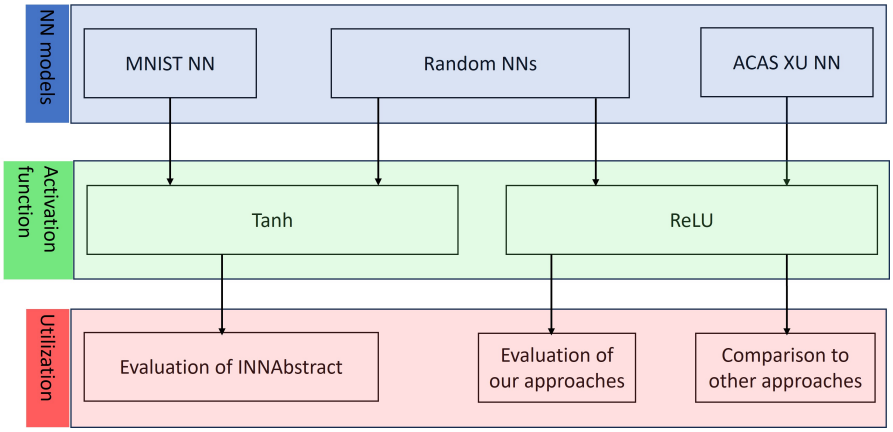


Figure 5.1: The organization of the experimental study.

2. **`Abstractor`:** proof-of-concept implementation of the approach. This module is used to generate abstract networks from the initial one brought by the first module (Network_Reader).

3. **`Analyzer`:** we re-implemented the Interval Bounds Propagation (IBP) algorithm to calculate the output range of neural networks by propagating the inputs' constraints through the layers of the network. For the INNAbstract method, since initially the IBP algorithm does not support interval neural networks, we have adjusted the algorithm to support this specific format of NN whose weights have the form of intervals, instead of scalars.

This modular architecture enables users to easily apply our methods (via the developed tools) to their NN models while customizing the corresponding parameters. Such parameters include the number of the (desired) remaining nodes on each hidden layer after the abstraction and the strategy for nodes selection.

We should mention that all the experiments in this chapter have been carried out on a machine equipped with an Intel(R) Core(TM) i7-1065G7 CPU, ranging from 1.30GHz to 1.50GHz in frequency, and featuring a RAM memory capacity of 8GB.

5.2.2 Used NN Models and Benchmarks

To effectively evaluate our approaches, we used randomly generated NNs with *Tanh* and *ReLU* activation functions. Moreover, in order to check the performance of the approaches, we also conducted a series of experiments on two well-used benchmarks, namely ACAS Xu and MNIST. More details about the generation

of random *Tanh*-NNs, *ReLU*-NNs, the ACAS Xu NN benchmark and the MNIST NN benchmark are provided in the following sections, successively.

5.2.2.1 Random NNs

The generated network has five inputs, a single output and twenty hidden layers ($L = 20$) with thirty neurons per layer ($n_l = 30$). The abstraction is then applied to reduce the size of the initial network. Different abstract networks are generated by varying the number of desired remaining nodes on hidden layers: $\hat{n}_l \in \{25, 20, 15, 10\}$. For instance, $\hat{n}_l = 15$ indicates that each layer of the abstract network has 15 neurons, i.e., the layer's size is reduced by 50%. More details about this step are outlined in Table 5.1. Next, for a randomly generated input $x^* \in [-1, 1]^5$ we apply a perturbation $\epsilon \in [0.01, 0.1]$ to generate input bounds $[x^* - \epsilon, x^* + \epsilon]$. Notice that this procedure is performed for generating both *Tanh*-NNs and *ReLU*-NNs. The only difference lies in the considered activation function.

5.2.2.2 MNIST Benchmark

To evaluate INNAbstract on a more practicle *Tanh*-NNs, we use the ONNX networks trained on the dataset of handwriting digits, namely MNIST (LeCun 1998). An illustrative example of the MNIST benchmark is presented in Figure 5.2. In this dissertation, we utilized the benchmark networks available in the VNN 2020 Competition's GitHub repository[1]. An MNIST network has 784 inputs and 10 outputs. The inputs represent the vectorial representation of images of 28×28 pixel, and the outputs correspond to the ten digits $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$. While these networks have the same input and output size, there may be variations in terms of their depth (number of layers) and width (number of neurons per hidden layer). In this study, our experiments are conducted on the network *tansig_200_100_50_onnx.onnx* which consists of three hidden layers with 200, 100, 50 nodes, successively. We select an image from the MNIST dataset and then we apply a perturbation (ϵ) on each pixel using the L_∞ norm (Huang, Kroening, et al. 2020). This perturbation is then propagated through the networks (original and abstract networks) using the IBP algorithm.

[1]Available in: https://github.com/verivital/vnn-comp/tree/master/2020/NLN/benchmark/mnist/tanh

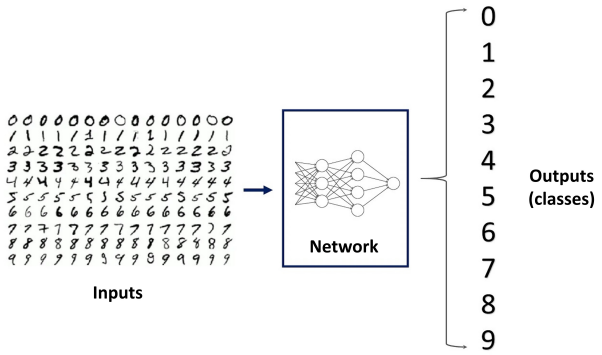


Figure 5.2: An illustration of the MNIST benchmark

5.2.2.3 ACAS Xu Benchmark

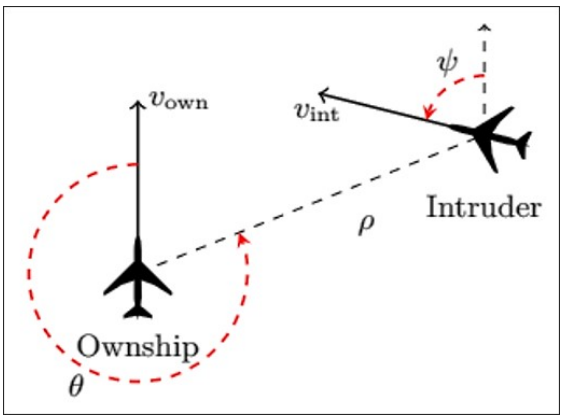


Figure 5.3: A representation of the sensors measurements inputs of the ACAS Xu networks.

To evaluate our methods on realistic *ReLU*-NN, we consider the Airborne Collision Avoidance System (ACAS) Xu benchmark. ACAS Xu is derived from ACAS X (Kochenderfer 2015) and it is created for unmanned airborne. Originally, it was designed using a large lookup table mapping the inputs (sensors measurements) to their corresponding outputs (advisories). Nonetheless, this approach requires a huge amount of memory (over 2 GB), which raises some resources concerns regarding the certification of such an avionics system. To address this challenge, an alternative solution involving the replacement of the lookup table with a neural network has been proposed. The resulting system requires less than 3 MB while maintaining an acceptable performance level, and in some cases, even outperforms the initial lookup table (Julian et al. 2016).

The ACAS Xu benchmark is a collection of 45 networks trained to offer advisories for an unmanned aircraft (ownship) to avoid a collision with another

aircraft (intruder) (Julian et al. 2016). Each network has seven inputs and five outputs. The network's inputs represent sensors measurements, and the outputs correspond to the given advisories. The objective is to establish a mapping between the inputs and the advised actions that ensure collision avoidance. The seven inputs are:

1. ρ: the distance between the ownship and the intruder (in feet).
2. θ: the angle to the intruder, w.r.t. the ownship's direction (in radians).
3. ψ: the heading angle (direction) of the intruder, w.r.t. the direction of the ownship (in radians).
4. v_{own}: the speed of the ownship (in feet/second).
5. v_{int}: the speed of the intruder (in feet/second).
6. τ: the required time until loss of vertical separation between the two aircraft (in seconds).
7. a_{prev}: the previous advisory (the previous output of the network).

The network processes these inputs and returns five outputs that are scores corresponding to potential advisories for the ownship. The output with the lowest score is considered as the best action. These outputs include the following advisories: *(i)* clear-of-conflict (COC), *(ii)* strong turn to the left (SL), *(iii)* weak turn to the left (WL), *(iv)* strong turn to the right (SR), and *(v)* weak turn to the right (WR). The terms strong turn and weak turn respectively correspond to heading rates of 3.0°/*second* and 1.5°/*second*.

5.3 Results & Discussion

5.3.1 Results on Tanh Networks

In this section, we present and discuss the obtained results on *Tanh*-NNs. The experiments are conducted on randomly generated NNs and the MNIST benchmark. Notice that in these experiments we only used INNAbstract to generate the abstract networks. This is due to the fact that the other model reduction methods does not support the *Tanh* activation function.

For a given network (random *Tanh*-NN or MNIST), we calculate the average values of the output's upper bound and the IBP computation time over 50 runs.

For abstract networks obtained using INNAbstract, we additionally calculate the abstraction time and the total computation time, which represents the sum of the abstraction time and the IBP computation time.

The obtained results on random *Tanh*-NNs are presented in Tables 5.1 and 5.2. To provide a visual representation, we also graphically depict these results in Figures 5.4 and 5.5.

For the MNIST benchmark, Figures 5.6 and 5.7 summarize the obtained results. As the hidden layers do not have the same number of nodes, the X-axis in Figures 5.6 and 5.7 denotes the percentage of remaining nodes after abstraction. For instance, the value 0.9 indicates a reduction 10% in the number of nodes of each hidden layer of the abstract network (180, 90, and 45 nodes successively).

	Output range (UB)		Output range's width	
Nb. Nodes	Original	INNAbstract	Original	INNAbstract
25	6.90	7.19	13.78	14.39
20	6.90	7.39	13.78	14.78
15	6.90	7.52	13.78	15.05
10	6.90	7.61	13.78	15.21

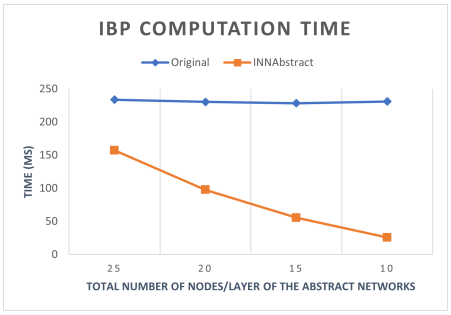
Table 5.1: Output results on randomly generated *Tanh*-NN, $L = 20$ layers.

	Abstraction time (ms)	Total computation time (ms)	
Nb. Nodes	INNAbstract	Original	INNAbstract
25	48.77	233.66	205.86
20	70.91	230.21	168.47
15	85.59	228.23	141.19
10	88.07	230.97	113.64

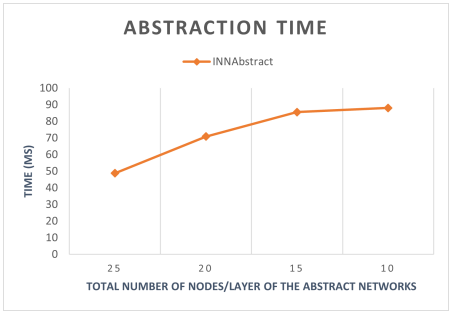
Table 5.2: Abstraction time and total computation time on randomly generated *Tanh*-NN, $L = 20$ layers.

As depicted in Figures 5.4 and 5.6, reducing the size of the network (i.e., decreasing the number of remaining nodes) by performing more abstraction leads to a linear increase in the abstraction time and a significant decrease in the IBP computation time. Consequently, as shown in Figure 5.4c and 5.6c, the total computation time (the sum of the abstraction time and the IBP computation time) is also decreased. Additionally, the IBP algorithm is consistently faster on the abstract networks generated using INNAbstract than on the original ones.

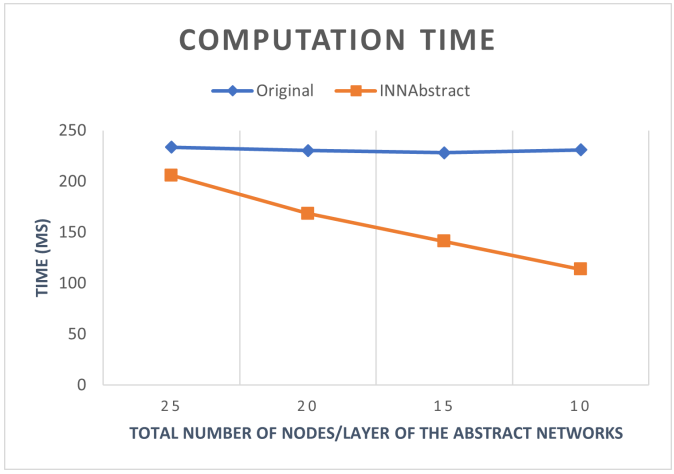
Figures. 5.5 and 5.7 illustrate the output range results on the original network and on the obtained abstract ones using INNAbstract. Figure. 5.5 shows that



(a) IBP computation time.



(b) Abstraction time.



(c) Total computation time.

Figure 5.4: Total computation time, IBP and abstraction time obtained on randomly generated *Tanh*-NN.

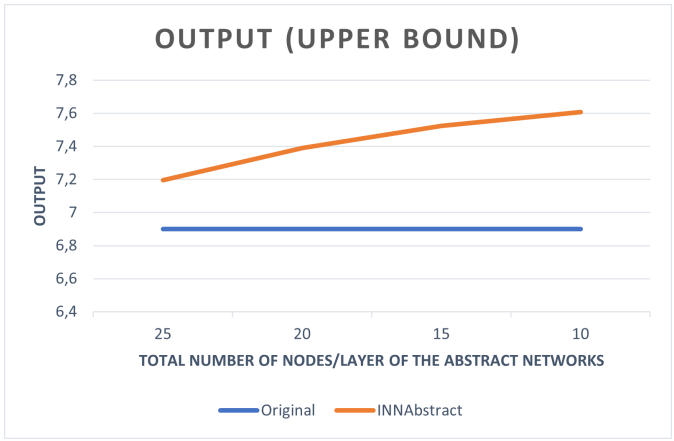
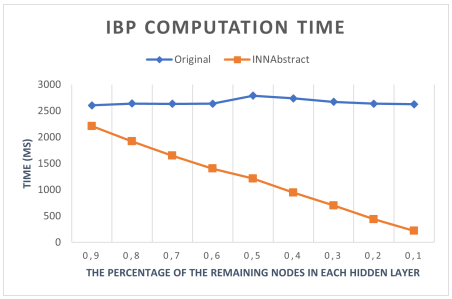
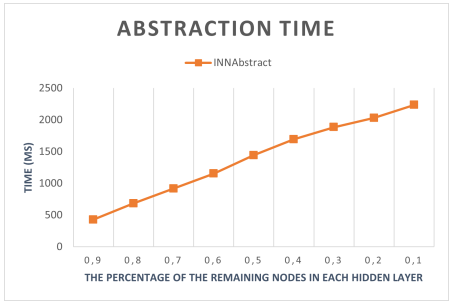


Figure 5.5: Output range results on randomly generated *Tanh*-NNs.

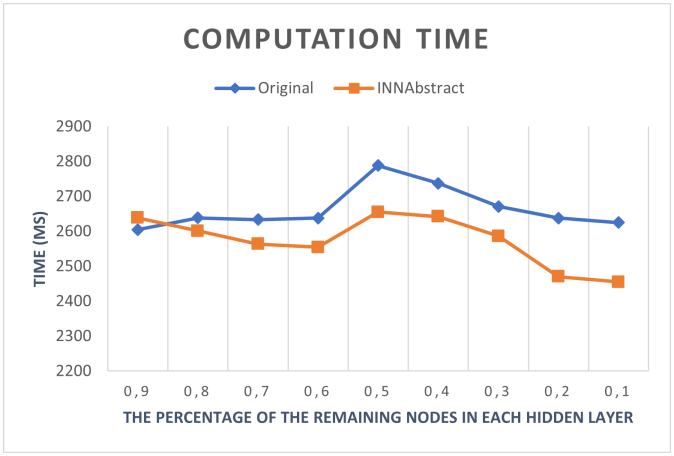
decreasing the size of the network leads to wider output range, hence negatively impacting the precision of the network. From Figure 5.7, it can be observed that the upper bound of the abstract networks is greater than that of the original one. However, they both remain fairly close. Interestingly, we can observe that the output's upper bound of different abstract networks does not change



(a) IBP computation time.



(b) Abstraction time.



(c) Total computation time.

Figure 5.6: Total computation time, IBP and abstraction time results on MNIST *Tanh*-NN.

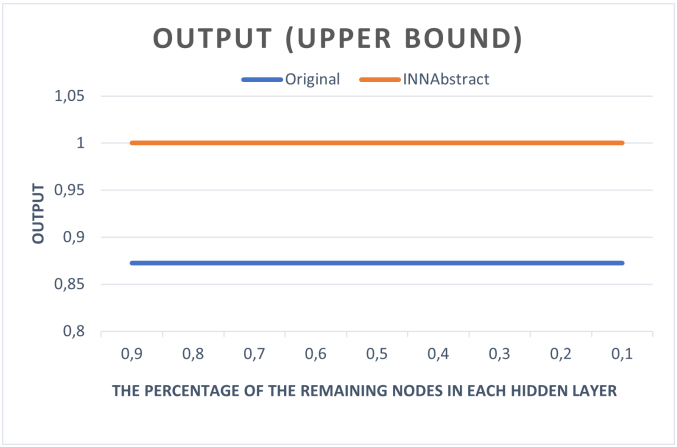


Figure 5.7: Output ranges on MNIST *Tanh*-NN.

while varying the number of remaining nodes. This is most likely related to the behavior of the *Tanh* activation function with large (resp. small) values, where the slope is almost null and the output of *Tanh* is basically 1 (resp. −1).

Subsequently, we conducted a series of experimental evaluations on ReLU networks to further assess the performance of INNAbstract and compare

it against other methods, including our second approach (for non-negative activation functions).

5.3.2 Results on ReLU Networks

The section is dedicated to the experimental study on ReLU NNs. Our two approaches are evaluated on both randomly generated ReLU-NNs (Section 5.2.2.1) and on the ACAS Xu benchmark network (Section 5.2.2.3). Furthermore, we conducted a comparison study between our approaches and two other model reduction methods proposed by Prabhakar and Rahimi Afzal (2019) and Elboher et al. (2020) using the same networks.

To evaluate and compare the performance of our methods on randomly generated *ReLU*-NNs, we used the configuration steps presented in Section 5.2.2.1. The same procedure is used to derive abstract networks using the two selected model reduction methods from the literature, namely, (Elboher et al. 2020) and INN (Prabhakar and Rahimi Afzal 2019). Over 50 runs, we calculate the average of the output's upper bound, the IBP computation time, the abstraction time, and the total computation time. Figures 5.8 and 5.9 illustrate the obtained results on the randomly generated *ReLU*-NNs.

Considering the evaluation on the ACAS Xu benchmark, for a given network and inputs' constraints of a specific property[2], we generated eight abstract networks by varying the number of remaining neurons in each hidden layer. The eight configurations have 45, 40, 35, 30, 25, 20, 15, and 10 neurons, respectively. Then, by means of the IBP algorithm, we compute the corresponding output range on the original network as well as on the generated abstract networks using the four methods, namely, INNAbstract, our non-negative model reduction method (denoted as NoNeg on the graphs), INN (Prabhakar and Rahimi Afzal 2019), and the technique presented in Elboher et al. (2020) (denoted as Elboher on the graphs). Over 50 runs, and for each network, we calculate the output upper bound average, the average of the IBP computation time, the abstraction time and the total computation time. The results are shown in Figures 5.11 and 5.12.

Through the performed experiments on both ACAS Xu and randomly generated networks, we can observe that the computation time and the abstraction time values (Figures 5.9 and 5.12) obtained using the Elboher

[2]We used the property ϕ_5 presented in Katz, Barrett, et al. (2017), and the network ACASXU_experimental_v2a_1_1.nnet

et al.'s method were excessively high, making it challenging to compare the original network with the abstract networks generated using the model reduction methods. Therefore, we separately present the computation time values for INN and our two approaches (INNAbstract, NoNeg), and compare them to the computation time on the original networks. Figures 5.10 and 5.13 summarize the obtained results on the *ReLU*-NNs and ACAS Xu benchmark, respectively.

As shown by Figures 5.9, 5.10, 5.12, and 5.13, the four model reduction methods exhibit similar evolution of the abstraction time, the IBP computation time and the total computation. When more abstraction is performed, i.e., reducing the number of remaining nodes, we observe that the abstraction time increases while the IBP computation time experiences a noticeable decrease. For the latter, it is consistently lower on the abstract networks than on the original ones. This reduction in the IBP computation time contributes to an overall decrease in the total computation time, whose values remain always below those obtained on the original networks (see Figures 5.10a and 5.13a). However, there is an exception for the method proposed by (Elboher et al. 2020). In fact, the computation time for the abstract networks generated using this method is higher than the computation time on the original networks. Additionally, from Figures 5.9 and 5.12, we can clearly notice that this method is computationally expensive compared to the other methods. This could be attributed to the pre-processing phase. Recall that this phase involves classifying neurons into different categories, and potentially splitting a number of neurons prior to the abstraction procedure per-se. Hence, the network to be abstracted might end up being larger than the original one, thereby significantly increasing the abstraction time.

Furthermore, by comparing the abstraction time among the selected methods, it can be observed that our methods are faster. Specifically, when considering the same number of abstract neurons, both our methods generate abstract networks in less time compared to the other methods from the literature.

Moreover, the abstraction time using INNAbstract and NoNeg increases at a slower rate compared to INN approach (see Figures 5.10b and 5.13b). As a result, the total computation time, defined as the sum of the abstraction time and the IBP computation time, is also lower when using our methods. This illustrates the efficiency of the proposed method, which is capable of handling larger networks with a significant decrease in the overall computation time.

Additionally, when comparing our approaches exclusively, it is clear that the NoNeg method outperforms INNAbstract across all metrics. This includes

the abstraction time (Figures 5.10b and 5.13b), the IBP computation time (Figures 5.9a and 5.12b) and the total computation time (Figures 5.10a and 5.13a). While the abstraction time for INNAbstract increases linearly and steadily compared to the two alternative methods, the abstraction time using the NoNeg method exhibits only minor fluctuations and tends to remain stable when varying the number of neurons of the abstract networks. Moreover, the NoNeg method is computationally more efficient, consistently showcasing lower abstraction and IBP computation times compared to all other selected methods.

Figures 5.8 and 5.11 present the output range results for both the original networks and on the abstract networks generated using the four abstraction methods. A prominent observation from these figures is that as the number of remaining neurons decreases, indicating a reduction in the network size, the upper bound of the output tends to increase. Furthermore, when comparing the output range results of the four abstraction methods, we can see that the output range's upper bounds of the abstract networks obtained using our methods are generally smaller than the output ranges of the abstract networks generated using Elboher et al. (2020) and INN (Prabhakar and Rahimi Afzal 2019). Specifically, our methods significantly outperforms that of Elboher et al. considering different sizes of the abstract networks. In fact, although the INN abstraction method initially demonstrates a slightly better performance than INNAbstract when only few neurons are merged, its performance drops drastically as the abstraction becomes more intensive (more neurons are merged). This is reflected in the rapid increase of the output range of the obtained abstract networks using INN method. In contrast, we can notice that the output range on abstract networks obtained using our methods exhibits a slow and gradual increase as the number of remaining nodes decreases, hence outperforming the INN method when substantial abstraction is applied.

While INNAbstract ourperforms the two model reduction methods, our NoNeg method outperforms all the aforementioned methods, including INNAbstract. Indeed, the generated abstract networks using NoNeg provide tighter bounds compared to the other three methods. The upper bounds obtained on the abstract networks of the NoNeg method are always lower than the upper bounds of abstract networks generated using the three other methods.

The main conclusion that can be drawn from this series of experiments is that, reducing the network's size by applying the established model reduction methods allows for significantly reducing the total computation time (including both the abstraction and output range computation). On the other hand, this size

reduction leads to a linear and gradual increase in the output range. Furthermore, according to the conducted experiments and the obtained results, our methods outperform the approach from (Elboher et al. 2020) and INN (Prabhakar and Rahimi Afzal 2019) approaches in the sense that they produce tighter output bounds in less time. This highlights the effectiveness of our methods in reducing the computation time, thus allowing for faster analysis and operations on NN. Particularly, for the NoNeg method, which exhibits a high performance compared to all other methods, the computation time increases slowly and it remains always lower than the computation time of all considered methods. In addition, its generated abstract networks' upper bounds are tighter. Finally, we can see that the precision of the abstract model is strongly influenced by the number of merged nodes. Specifically, as more nodes are merged, resulting in a higher level of abstraction and more abstract nodes, the upper bound of the abstract networks tends to increase.

5.3.3 Heuristic's Improvement for INNAbstract

The objective of this section is to assess the performance of INNAbstract when combined with the proposed heuristic for nodes selection (Algorithm 5). To achieve this, we conduct a series of experiments on a set of randomly generated *ReLU*-NNs using two nodes selection strategies: random selection and heuristic-based selection. Namely, we generate a set of networks with different sizes by varying the number of layers L and the number of nodes n_l in each layer to examine their impact on the the heuristic's performance. More details about the generation of these networks can be found in Section 5.2.2.1. Next, we calculate the output range of the obtained abstract networks with both nodes selection strategies and we determine the improvement rate (I_{rate}) in terms of precision using the proposed heuristic. We define this rate as the percentage of the difference between the output range (using heuristics) with respect to the output range obtained when INNAbstract is combined with the random nodes selection. Formally speaking, I_{rate} is calculated as follows:

$$I_{rate} = \frac{U_{rand} - U_{heuris}}{U_{rand}} \times 100$$

such that U_{rand} and U_{heuris} are the upper bounds of the abstract network using INNAbstract with the random and heuristic based strategies for nodes selection, respectively.

Figure 5.14 depicts the obtained results. Specifically, Figure 5.14a represents the results on networks with different numbers of layers ($L = 20$, $L = 40$) and the same number of nodes in hidden layers ($n_l = 30$), while Figure 5.14b represents the results on networks whose numbers of layers are set to 20 ($L = 20$) and have different hidden layer sizes ($n_l = 30$, $n_l = 40$).

As shown by the results presented in Figure 5.14, the combination of the proposed heuristic with INNAbstract has improved the precision of the generated abstract networks. We can also notice that the number of layers and hidden nodes in the initial network has an impact on the performance of the heuristic. Particularly, the proposed heuristic performs better with large networks, as the population from which the nodes are selected is larger. This can be explained by the fact that the algorithm has a greater pool of options, increasing the likelihood of finding suitable candidates.

To illustrate the improvement in terms of precision through the proposed heuristics, we chose the configuration $L = 20$ and $n_l = 40$ to create a random network of 20 layers, with 40 nodes in each hidden layer. Subsequently, we compute the output range and the abstraction time of the resultant abstract networks, employing both random and heuristic selection strategies.

As demonstrated in Figure 5.15a, the output range obtained on abstract networks generated using the combination of INNAbstract and the proposed heuristic is tighter compared to the output range of those obtained using INNAbstract with a random selection of nodes. Furthermore, these improvements become more significant when more nodes are merged. However, it is important to note that achieving these improvements comes at the cost of increased abstraction time, as illustrated in Figure 5.15b.

Finally, it is worth noticing that we have conducted the same experiments on *Tanh*-NNs; however, the improvement rate remains negligible. This is mainly due to the nature of the *Tanh* function (illustrated by its hyperbolic behavior) which tends to push the output value to either ends of the curve (1 or -1) due to its S-like shape, i.e., in the region close to zero, if we slightly change the input value, the respective changes in the output are very large and vice versa.

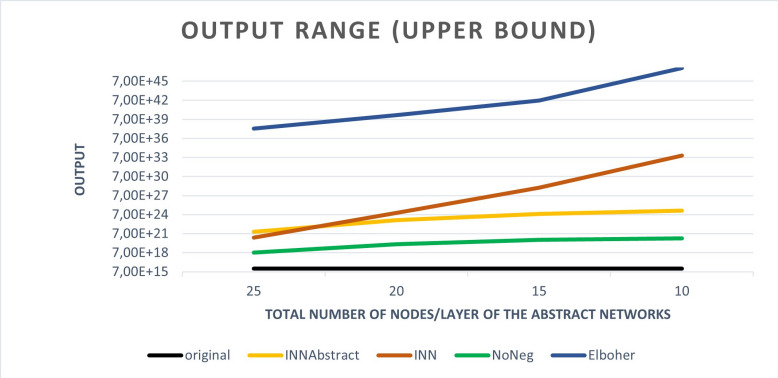
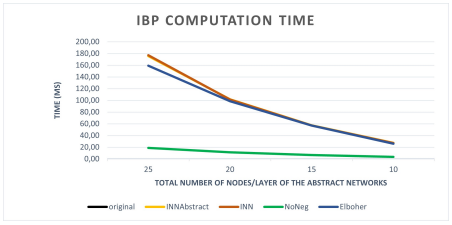
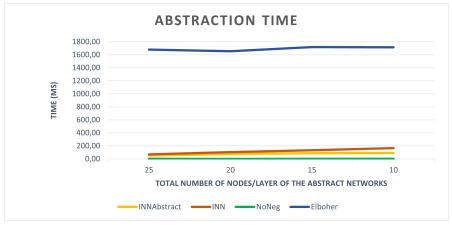


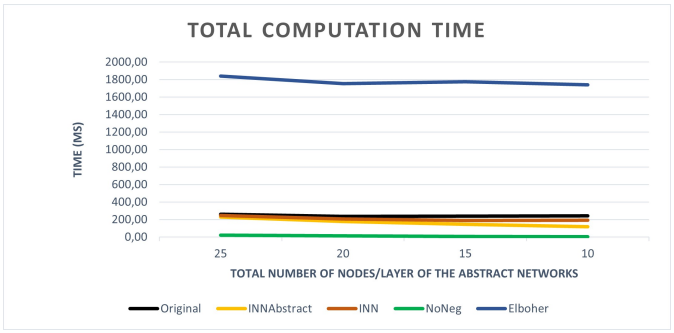
Figure 5.8: Output range results on *ReLU*-NN.



(a) IBP computation time.

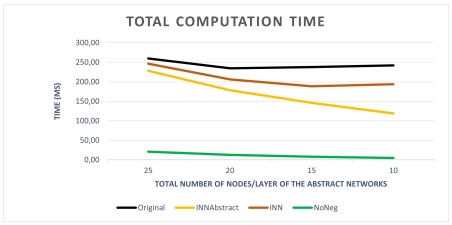


(b) Abstraction time.

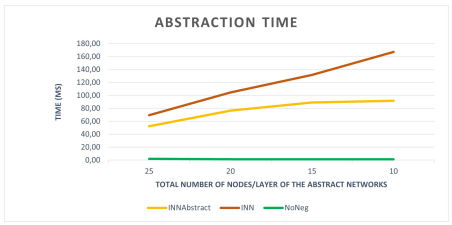


(c) Total computation time.

Figure 5.9: Total computation time, IBP and abstraction time results on *ReLU*-NN.



(a) Total computation time.



(b) Abstraction time.

Figure 5.10: The total computation time and the abstraction time for INNAbstract, NoNeg, and INN on *ReLU*-NN.

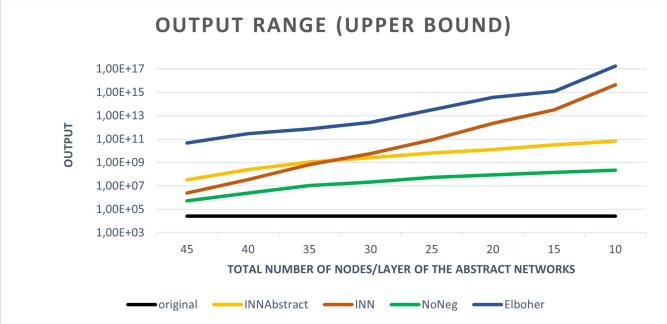
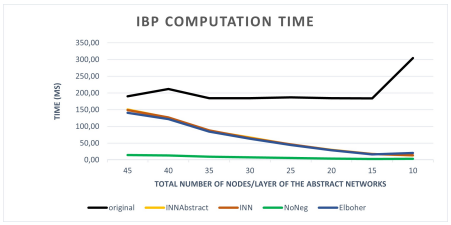
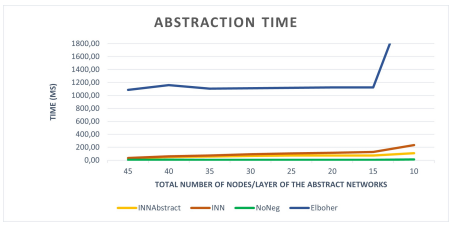


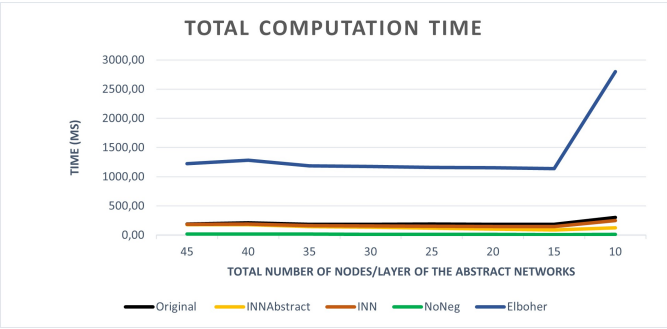
Figure 5.11: Output range results on ACAS Xu *ReLU*-NN.



(a) IBP computation time.

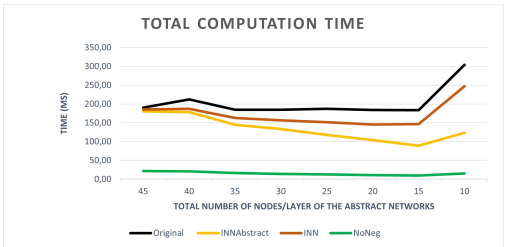


(b) Abstraction time.

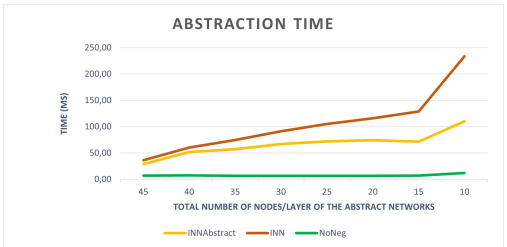


(c) Total computation time.

Figure 5.12: Total computation time, IBP and abstraction time results on ACAS Xu *ReLU*-NN.

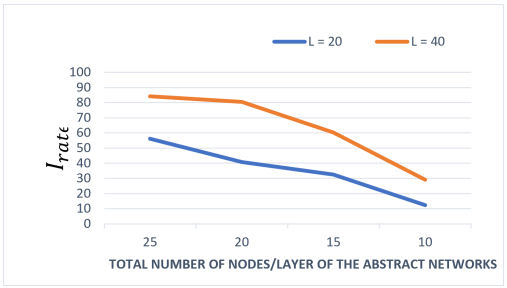


(a) Total computation time.

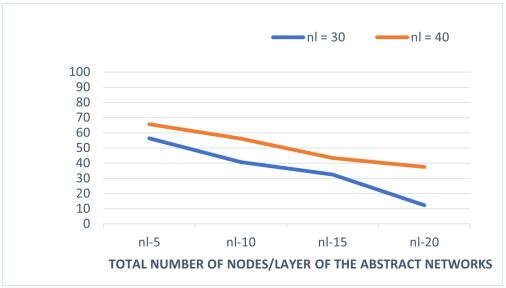


(b) Abstraction time.

Figure 5.13: The total computation time and abstraction time results of INNAbstract, NoNeg, and INN on ACAS Xu *ReLU*-NN.

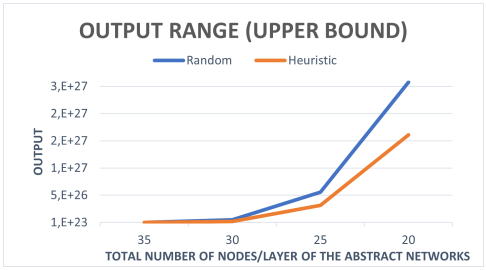


(a) Results on networks with different number of layers, $L = 20$ and $L = 40$.

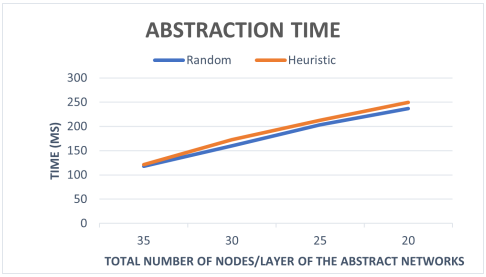


(b) Results on networks with different number of nodes per layer, $n_l = 30$ and $n_l = 40$.

Figure 5.14: A graph representing the improvement of the output range using the proposed heuristic.



(a) Output range (upper bound).



(b) Abstraction time.

Figure 5.15: The Output range and the abstraction time on *ReLU*-NNs with random and heuristics based selection strategies.

5.4 Conclusion

This chapter summarizes the results of our comprehensive evaluation study, which aimed to assess the performance of our proposed methods in the context of NN model reduction. The conducted study includes experiments on randomly generated networks and some well-known benchmarks from the literature, namely ACAS Xu and MNIST. The former networks are generated by varying their sizes in order to analyse the performance of the abstraction methods across networks of different scales. On the other hand, the used benchmarks allowed us to evaluate our methods on real-world networks. In addition, to assess the performance of our methods against existing ones, a comparison study with some reference methods from the literature is carried out within this experimental study.

The results obtained through these experiments highlight the efficiency of our methods, namely *INNAbstract* and *NoNeg*, in reducing the size of networks while preserving acceptable precision (i.e., the output range). Furthermore, our

two methods outperformed all other selected model reduction methods in terms of computational time and precision. Particularly, the *NoNeg* method which has proven to be the top performer, allowing the generation of more precise abstract networks while requiring significantly less computation time compared to all other methods, including *INNAbstract*.

While NN model reduction methods are applied to enhance the scalability of NN verification, and thus brought into play before the deployment of the NN on the intended system, the following part will shift the focus to the evaluation of networks during runtime. Hereafter, we present our contribution in this domain regarding NN monitoring.

Part III

Neural Networks Monitoring

Chapter 6

NAPath: Runtime Monitoring of Neural Networks

Outline of the current chapter

6.1 Introduction

As discussed earlier in this dissertation (see Chapter 1), testing, formal verification, and monitoring are three key techniques to help ensure the safe and reliable deployment of neural network based systems. Testing involves evaluating the trained model through the process of comparing the predictions made by the model on various data sets to the actual labels in the data, with respect to given performance metrics (Goodfellow, Bengio, et al. 2016); and formal verification involves using mathematical techniques to verify whether the model satisfies a set of desired properties (Gehr et al. 2018; Katz, Barrett, et al. 2017). While testing and formal verification are performed during the system design phase, monitoring involves online tracking of the performance and the behavior of the model during the operation phase (Cheng, Nührenberg, and Yasuoka 2019). It is particularly essential in the scenarios where the consequences of incorrect predictions may be severe.

In this chapter, we present our contribution for monitoring NNs with ReLU activation function used for image classification. The proposed approach is used to supervise the network's outputs with respect to the input images, towards adding another level of confidence and reliable guarantees to the NN. The approach relies on the concept of Neuron Activation Path (NAPath) that we introduce as part of this contribution. NAPath is used to identify relevant paths of activations (i.e., activation patterns) for each class in the network. A path is defined as a sequence of neurons of the same status in terms of activation (active or inactive) linking the network's input to the output. For a NN with a *ReLU* activation function, NAPath extracts (learned) features of the network by processing the representative set of images that belong to the same class. We define a feature, or a pattern, as a set of paths on the network that are similarly activated or deactivated for the selected images. It is worth mentioning that our approach is inspired by the Neuron Activation Pattern (NAP) concept discussed by (Geng et al. 2022). The key difference between NAP and NAPath is that NAP defines a pattern as a set of active and inactive neurons, while NAPath defines a pattern as a set of paths linking the input of the network to its output.

Recall that a monitoring systems has mainly two phases: the construction of a monitor and runtime monitoring. In our approach, the former phase consists of identifying active and inactive paths for each class of inputs. A NAPath, which is a set of active and inactive paths, is then associated with this class of image. During the second phase which intervenes during runtime, the monitor

uses the set of NAPaths to supervise the network's output when it processes an input-image. Concretely, the monitor analyzes the NAPath generated when the network is fed with some given input, and compares it to the most similar NAPath from the pre-calculated set of NAPaths. The intuitive idea is that the inputs belonging to the same class should activate / deactivate the same paths. Accordingly, the monitor can raise misclassification alarms, or novelty alarms. While the former indicates that the network may have misclassified the input, and it suggests a new reclassification, the latter indicates that the input at hand does not have any similar NAPath with the set of pre-calculated NAPaths, thus the input is considered as a novel sample. More details about how the set of NAPaths is extracted, which data are used, how the similarity between NAPaths is calculated, and other information about the approach is provided in the remaining of this chapter.

The remaining of this chapter is organized as follows: in Section 6.2 we introduce the NAPath concept, and we provide necessary definitions and notations. Section 6.3 presents the proposed monitoring method, including a detailed discussion of each phase: the monitor construction, and runtime monitoring. The experiments' setups and the obtained results on the MNIST are given in Section 6.4. In Section 6.5, we present the experimental results of applying the approach to an AI-based perception system used within the autonomous train, the weather conditions' recognizer. In Section 6.6, we provide a review of related works. Finally, we conclude this chapter in Section 6.7.

6.2 Background

As mentioned in the Introduction section, NAPath draws inspiration from the concept of NAPs. Therefore, before delving into the definition of NAPath, we first recall the concept of Neuron Activation Pattern (NAP).

6.2.1 Neural Activation Patterns (NAP)

A NAP is a representation of the activation status of neurons in a neural network in response to a given input or a set of inputs, indicating which neurons (from the hidden layers) are active or inactive (Geng et al. 2022). A definition of this concept for networks with the ReLU activation function is given below.

Definition 6.1 *Let N be a ReLU-NN, and let S be the set of its hidden neurons. The set of all active (resp. inactive) neurons for an input x on the network N is denoted as*

A_x (resp. D_x) and defined such that:

$$\begin{cases} A_x = \{s \in S : v_s(x) > 0\} \\ D_x = \{s \in S : v_s(x) = 0\} \end{cases} \tag{6.1}$$

In definition 6.1, $v_s(x)$ denotes the output value of the hidden neuron $s \in S$ when we consider an input x of the network. Recall that S is the set containing all hidden neurons of the network N, i.e, $S = \{S_1, S_2, \ldots, S_{n-1}\}$, where S_i is the set of neurons of the hidden layer l_i, for $1 \leq i \leq n-1$. For a ReLU-NN, a NAP of an input x on N (denoted by $NAP(N,x)$) is defined as the set of its corresponding active and inactive sets of neurons, i.e., $NAP(N,x) = (A_x, D_x)$.

For a set of inputs X_c such that $\forall x \in X_c : N(x) = c$, we define its associated NAP, denoted as NAP_c, as the common NAP between all inputs in X_c:

$$\begin{cases} A_c = \{s \in S : \forall x \in X_c, v_s(x) > 0\} \\ D_c = \{s \in S : \forall x \in X_c, v_s(x) = 0\} \end{cases} \tag{6.2}$$

For a set X_c considered as a representative set of inputs of class c, (A_c, D_c) is calculated using Equation (6.2) represents the NAP corresponding to this class, which is denoted as $NAP_c = (A_c, D_c)$. In general, X_c represents the set of inputs of class c used to train the network.

Notice that the definition of NAP_c according to Equation (6.2) is too constrained, and in practice it may lead to an empty NAP, i.e., $A_c = \emptyset$ and $D_c = \emptyset$. Thus, a control parameter $\delta \in]0,1]$ is used to build a relaxed NAP, denoted as δ_NAP (Geng et al. 2022). A neuron is considered active (resp. inactive) if it is activated (resp. deactivated) at least $\delta \times |X_c|$ times. In other words, for each neuron $s \in S$, we firstly calculate its value for all $x \in X_c$; and then, we calculate the total number of activations and deactivations of s, let us denote them by a_s and d_s, respectively. Next, if $a_s \geq (\delta \times |X_c|)$ then s is added to A_c). Similarly, if $d_s \geq (\delta \times |X_c|)$ then s is added to D_c. Finally, the relaxed NAP is defined as: $\delta\_NAP_c = (A_c, D_c)$ and expressed by Equation (6.3):

$$\begin{cases} A_c = \{s \in S : a_s \geq \delta \times |X_c|\} \\ D_c = \{s \in S : \ d_s \geq \delta \times |X_c|\} \end{cases} \tag{6.3}$$

6.2.2 Neuron Activation Paths (NAPath)

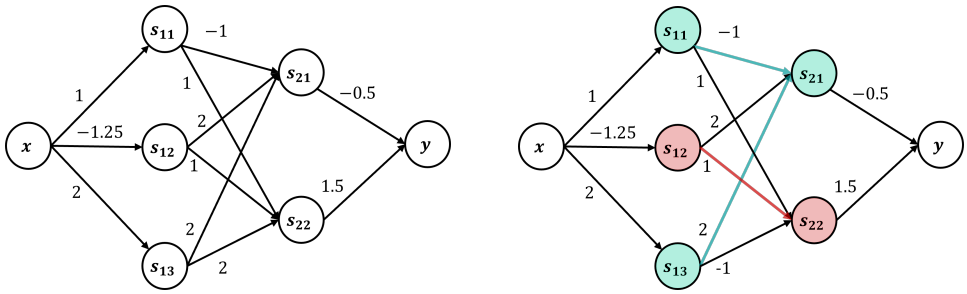
In this section, we introduce the concept of NAPath. Although NAPath is built upon the concept of NAP, NAPath aims to represent learned features through paths, instead of solely focusing on the activation status of individual neurons. Specifically, a path is a sequence of neurons that connects the input and output layers, passing through all hidden layers. This concept provides a more comprehensive view of the network's learned features, as it allows us to maintain the relation between neurons of different layers. By following the paths, we can better understand the sequence of (neural) activations that leads to a classification decision.

Formally, a path on a network N of $(n-1)$ hidden layers is a sequence of neurons $P = s_1, \ldots, s_{n-1}$ such that s_k is a neuron of the k^{th} layer, i.e., $s_k \in S_k$, for $1 \leq k \leq n-1$. Recall that l_0 and l_n are the input and the output layers, respectively. In our work, we define two types of paths:

1. ***Active path:*** For an input x and a network N of $(n-1)$ hidden layers, an active path is a sequence of neurons $P = s_1, \ldots, s_{n-1}$ such that for all k: $1 \leq k \leq n-1$, we have $s_k \in S_k$ and $v_{s_k}(x) > 0$.

2. ***Inactive path:*** For an input x and a network N of $(n-1)$ layers, an inactive path is a sequence of neurons $P = s_1, \ldots, s_{n-1}$ such that for all k: $1 \leq k \leq n-1$, we have $s_k \in S_k$ and $v_{s_k}(x) = 0$.

Figure 6.1 presents an example of network with marked active and inactive paths for the input $x = 1$. In Figure 6.1b, the active paths are highlighted in green, while the inactive paths are in red.



(a) An example of a neural network.

(b) The paths of the network shown in Figure 6.1a.

Figure 6.1: An example showing the NAPath of an input x on a neural network.

Definition 6.2 *For a given input x and a network N, the corresponding NAPath is the tuple $NAPath(N,x) = (\mathcal{A}_x, \mathcal{D}_x)$, such that:*

1. *$\mathcal{A}_x$ is the set of paths that are active for x.*
2. *$\mathcal{D}_x$ is the set of paths that are inactive for x.*

To maintain consistency with the definition of NAP, in Definition 6.2 we use the notation $\mathcal{A}_x$ and $\mathcal{D}_x$ to denote the set of active and inactive paths, respectively. When there is no ambiguity, we omit the argument N and write $NAPath(x) = NAPath_x = (\mathcal{A}_x, \mathcal{D}_x)$.

For a set of inputs X_c of class c, we define its associated NAPath as the intersection of all active and inactive paths for all $x \in X_c$, $NAPath_c = (\mathcal{A}_c, \mathcal{D}_c)$, such that:

1. $\mathcal{A}_c$: is the set of paths that are active for all x in the X_c.
2. $\mathcal{D}_c$: is the set of paths that are inactive for all x in the X_c.

Defining the NAPath as the intersection between all active and inactive paths for a large set of images can lead to an empty NAPath, i.e., no common active or inactive path. Therefore, similarly to the NAP feature, we introduce a relaxed NAPath using a control parameter $\delta \in]0,1]$. In this relaxed version, $NAPath_c$ is defined as a set of active and inactive paths, $\mathcal{A}_c$ and $\mathcal{D}_c$, respectively. For a path to be in $\mathcal{A}_c$ (resp. $\mathcal{D}_c$), it must be activated (resp. deactivated) for at least $\delta \times |X_c|$ inputs. For example, $\delta = 1$ means that a path has to be active for all inputs within X_c in order to belong to $\mathcal{A}_c$. Similarly, setting $\delta = 0.5$ means that an active path within $\mathcal{A}_c$ has to be active for at least half of inputs in X_c. This relaxed version of NAPath is explained in details in the remaining of this chapter.

Remark 6.1 *It is worth noting that the NAP of an individual input x, denoted as $NAP(N,x) = (A_x, D_x)$, is a partition of the set of all hidden neurons S of the network N, such that $A_x \cap D_x = \emptyset$ and $A_x \cup D_x = S$. However, the NAP of a class c generated using a set of inputs and denoted as $NAP_c = (A_c, D_c)$ only forms a subset of S, i.e., $A_c \cup D_c \subseteq S$. This is due to the fact that NAP_c is an intersection of the NAPs of the inputs belonging to the class c. On the other hand, a NAPath of the same input is a subset of paths of N, and consequently, the set of participating neurons (the neurons in $\mathcal{A}_x$ and $\mathcal{D}_x$) is a subset of S. This is due to the fact that some paths of the network may be neither active nor inactive. For example, a sequence of neurons that are all active except one neuron is not considered as an active path, neither an inactive one.*

6.3 Monitoring using NAPaths

In this section, we present our NAPath-based approach for runtime monitoring of NN image classifiers. The approach consists in firstly (*i*) building offline the monitor using the NAPath concept, and then (*ii*) using the monitor in runtime operation (in parallel to the NN model). The former phase, which we call *NAPathing*, allows for computing the set of NAPaths, and the latter phase, namely *runtime monitoring*, enables the use of the set of NAPaths to supervise the classification decision of the network during the runtime operation. The general structure of the monitoring process is presented in Figure 6.2, and a detailed explanation of the two phases is given below.

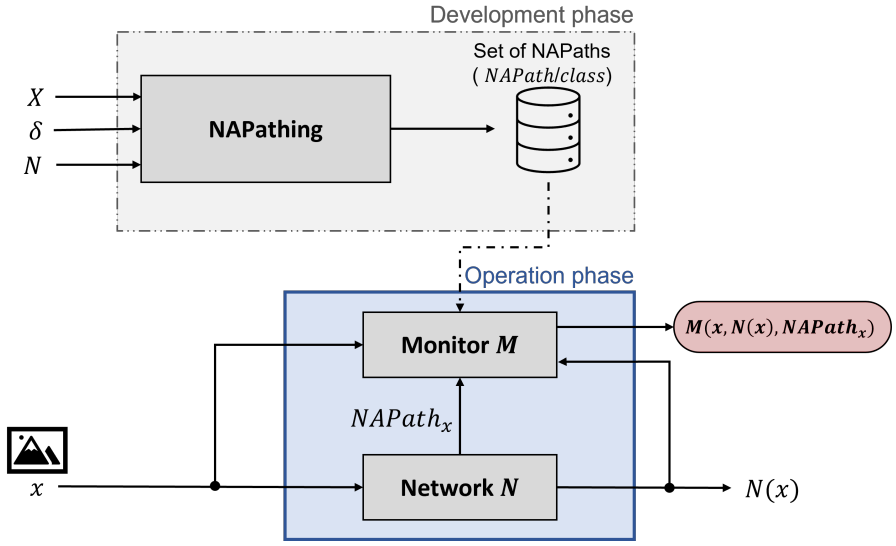


Figure 6.2: The general structure of the monitoring system.

6.3.1 NAPathing Phase

NAPathing is performed during the development phase of the monitor, and the objective is to compute a set of NAPaths, i.e., a NAPath for each class. For this purpose, NAPathing involves computing the set of active and inactive paths for each class based on the training data set. The specification of the following three parameters is required for this phase:

1. A network N.
2. The set of inputs X from which the NAPaths shall be computed, e.g., the training set used to establish N.

3. The parameter δ specifies the threshold of activations and inactivations required for a path to be considered as active or inactive, respectively.

Paths' computation is an important step. First, we partition the set X into subsets such that images within a subset must belong to the same class. Then, for a given class c, we start by filtering its associated subset of images to remove those that are misclassified by the network N. The obtained subset of images is denoted as X_c. Next, we define the control parameter $\delta \in]0,1]$, which defines the threshold for paths to be considered as activated or deactivated. Concretely, a path is considered as active if it is activated at least for $\delta \times |X_c|$ samples from X_c. Similarly, a path is considered inactive if it is inactive for $\delta \times |X_c|$ samples. The next step consists in assuring that the NAPath is valid; this is performed by checking that the sets of active and inactive paths are not empty, and they cover a sufficient number of inputs in X_c. If the NAPath is valid, it is saved as the NAPath of class c ($NAPath_c$); otherwise, the value of δ is updated to generate a new NAPath. The main steps of this process are presented in Figure 6.3. These steps are performed for each class of images to obtain at the end a set of NAPaths, where each NAPath is associated with a single class. Once all the NAPaths are computed and validated, they are saved and can be used during the monitoring phase.

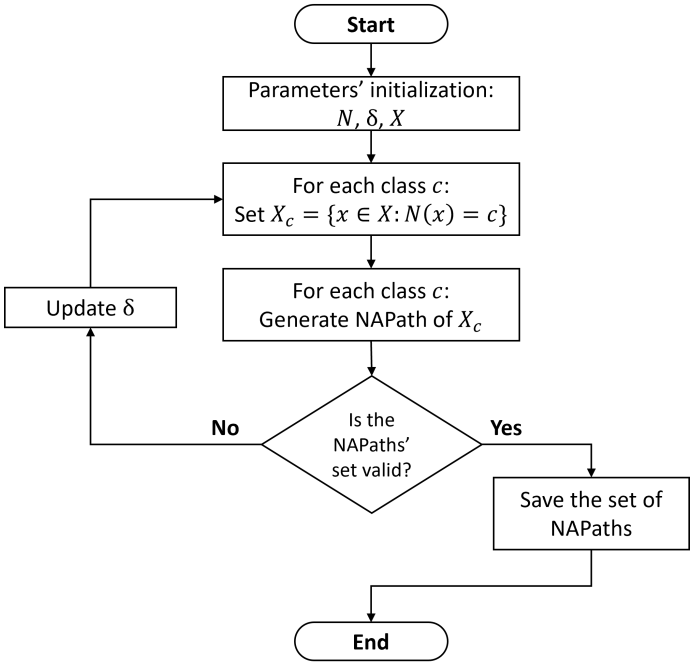


Figure 6.3: The flowchart of the NAPathing phase.

6.3.2 Monitoring Phase

During the operational phase of the system, the monitor is executed in parallel to the network N to supervise its decisions in real-time. The monitor takes as input:

- the set of pre-calculated NAPaths (determined during the NAPathing phase),
- the input-image x,
- the output y corresponding to input x, i.e., $N(x) = y$,
- the NAPath calculated in run-time using the network N, i.e., $NAPath(N, x) = NAPath_x$,

Next, the monitor calculates the similarity between $NAPath_x$ and the other NAPaths obtained from the NAPathing phase. This involves comparing the set of active and inactive paths of $NAPath_x$ to those of each pre-calculated NAPath. Accordingly, we can distinguish two different cases:

1. **Misclassification detection:** among the pre-computed NAPaths, a high similar (similarity will be formulated in the following) NAPath to $NAPath_x$ is found. Lets assume that this NAPath is associated with a class c, thus we denote it as $NAPath_c$. The next step consists in checking whether the classification decision by the network ($N(x) = y$) is consistent with the NAPath similarity to class c detected by the monitor.
    - ***Case*** $y = c$: this strengthens the classification decision and confirms that the input image shares similar features with the other images of the same class, represented by $NAPath_c$.
    - ***Case*** $y \neq c$: since $NAPath_x$ and $NAPath_c$ are similar and share a large portion of common paths, it is expected that x belongs to the class c. Therefore, the monitoring system shall raise an alarm indicating a need to check the network's classification decision. Furthermore, the monitor suggests a re-classification of the input based on its NAPath, recommending that this image should be of class c instead of class y.
2. **Novelty detection:** Novelty detection arises when the $NAPath_x$ exhibits no similarity with any other NAPath within the set. In other words, $NAPath_x$ does not share any path with any other NAPath in the set. Consequently,

the monitor shall raise a "novelty" alarm indicating that the image is new to the network and may represent an out of the operational domain input.

The main steps of the monitoring phase are illustrated in Figure 6.4.

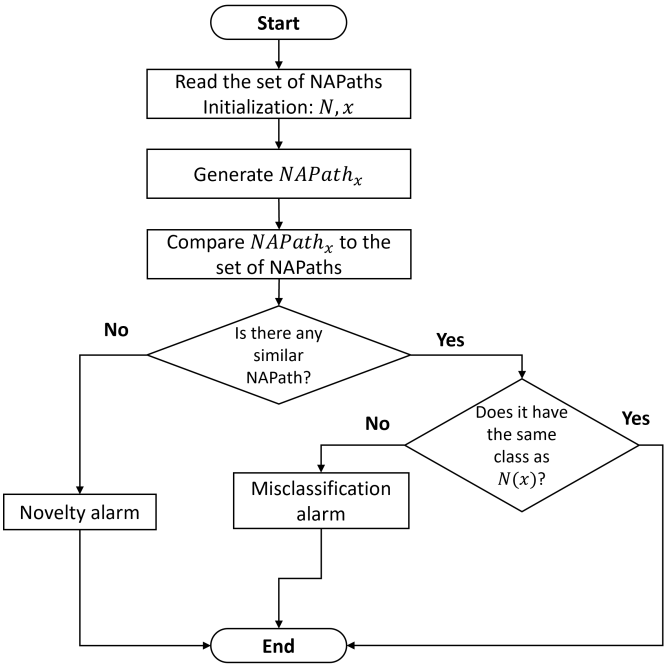


Figure 6.4: The flowchart of the the monitoring procedure using NAPaths.

An essential function during the monitoring phase is measuring the similarity between the NAPath of the current input ($NAPath_x$) and each of the pre-computed NAPaths. Such function is responsible for identifying the most similar pre-computed NAPath, if exists, to that of the considered input. For an input x and its corresponding NAPath: $NAPath_x = (\mathcal{A}_x, \mathcal{D}_x)$, and a NAPath of class c denoted as $NAPath_c = (\mathcal{A}_c, \mathcal{D}_c)$, we define the similarity degree between $NAPath_x$ and $NAPath_c$ as follows:

$$sim(NAPath_c, NAPath_x) = p \times \frac{|\mathcal{A}_c \cap \mathcal{A}_x|}{|\mathcal{A}_c|} + (1-p) \times \frac{|\mathcal{D}_c \cap \mathcal{D}_x|}{|\mathcal{D}_c|} \tag{6.4}$$

where $|\cdot|$ stands for the cardinality of the set and the parameter $p \in [0,1]$ is used to control the rate of active (or inactive) paths that contribute to calculating the similarity degree. In the context of the proposed approach, two NAPaths are considered similar if they have a high degree of overlap in terms of their paths. By adjusting the value of p, we can gain insights into which type of paths have more importance in the similarity degree's calculation. For instance, when p is close to 1, it indicates that similar NAPaths share more identical active paths

and less inactive paths. Notice that the parameter p can be tuned empirically from the evaluation of the training and testing data.

To evaluate the performance of the proposed approach, we present in the following section the experimental setup and the obtained results. Additionally, we compare our method to another NN monitoring approach from the literature.

6.4 Experimental Results on the MNIST Benchmark

In this section, we conduct a series of experiments using a neural network trained on the MNIST dataset. The NN model is obtained from the official website of VNNCOMP 2021[1], which is an international competition for researchers to test their NN verification tools on a set of benchmarks, such as MNIST. The conducted experiments aim to evaluate the effectiveness of our approach for monitoring image classification NN. The MNIST is a popular benchmark dataset in the field of machine learning and computer vision. It is a collection of handwritten digits ranging from 0 to 9, each digit represented as a 28x28 grayscale image. The dataset consists of 60,000 training images (around 6000 images per class) and 10,000 test images. The network used in this section consists of an input layer of size 784, which corresponds to the number of pixels of accepted input images. It is followed by four hidden layers, each containing 256 ReLU neurons. The output layer has a size of 10, which represents the 10 possible classes (digits 0 to 9).

6.4.1 NAPathing Phase

Firstly, we applied our NAPathing method on the training set to generate the set of NAPaths, where each class has its own NAPath. Next, to analyse the impact of the NAPathing precision's parameter δ, we conducted a series of experiments for various values of δ, and we calculated the number of samples following the NAPath of the corresponding class and the NAPath's size. In our experiments, we observed that the number of inactive paths was consistently large across all classes. Therefore, to represent the size of a NAPath, we focused solely on the number of its active paths.The results of our analysis are presented in Figures 6.5 and 6.6.

We observe in Figure 6.5 that the size of the NAPaths (active paths) decreases

[1]Available at https://github.com/stanleybak/vnncomp2021/tree/main/benchmarks/mnistfc

rapidely as the value of δ increases. On the other hand, Figure 6.6 shows that the number of inputs involved in the computation of NAPaths increases with the incrementation of parameter δ. This means that the obtained NAPaths cover a larger number of inputs. This is mainly due to the fact that increasing the value of δ leads to a decrease in the number of paths. Thus, the corresponding NAPaths are less constrained (the larger the NAPath is, the more constraints it contains). Consequently, as fewer paths are included, the number of inputs participating in building the NAPath increases. These results suggest that the value of the parameter δ can be adjusted to control the sensitivity of the NAPath approach, enabling a trade-off between the coverage and the size of the NAPath.

It is worth noticing that during our experiments, we have observed that the pre-trained network[2] misclassifies a significant portion of the training images for classes 3, 5, 6, and 7. As a result, we can see that the number of covered inputs belonging to these classes is significantly low, considering different values of δ (see Figure 6.6).

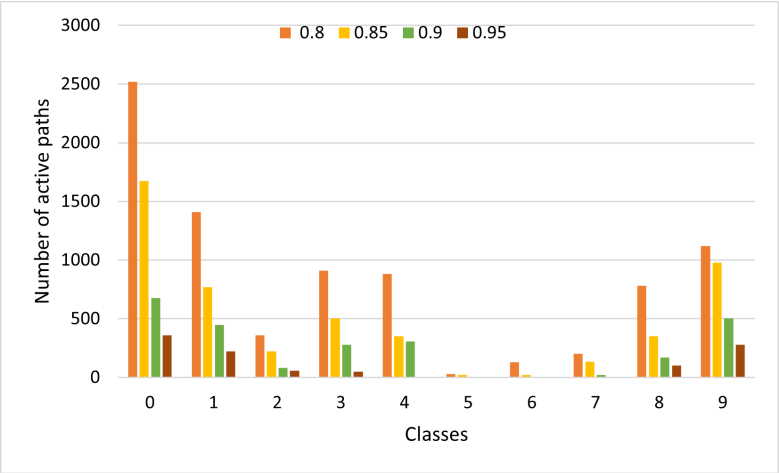


Figure 6.5: The number of active paths for different values of δ (0.8, 0.85, 0.90, 0.95).

6.4.2 Monitoring Phase

In the second part of our experiments, we investigate the impact of the control parameter p on the monitoring performances. To do so, we set $\delta = 0.9$, and we generate the NAPaths for the considered classes using their respective training data. This set of NAPaths is then used to assess the

[2]Available at: https://github.com/stanleybak/vnncomp2021/tree/main/benchmarks/mnistfc/mnist-net_256x4.onnx

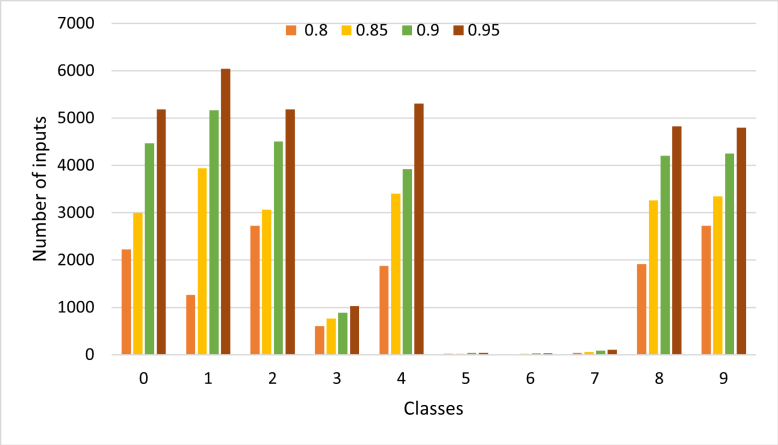


Figure 6.6: The number of inputs following the NAPath for different values of δ (0.8, 0.85, 0.90, 0.95).

monitoring system's performance across a range of p values, specifically, $p \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$.

The monitoring evaluation is conducted on the MNIST testing set, which includes a total of 6112 images equally distributed among the six selected classes: $\{0, 1, 2, 4, 8, 9\}$. For each value of p and the considered classes of images, we measure the monitoring precision in terms of the rate of correct no alarms (*true negative*), correct alarms (*true positive*), false alarms (*false positive*), missed alarms (*false negative*), and the rate of correctly re-classified samples proposed by the monitor. We define these four types of alarms as follows:

- ***Correct no alarms (true negative):*** the network correctly classifies the input-image, and the monitor does not raise an alarm

$$correct\ no\ alarms = \frac{number\ of\ passes\ for\ well\ classified\ images}{total\ number\ of\ well\ classified\ images} \tag{6.5}$$

- ***Correct alarms (true positive):*** the network misclassifies the input, and the monitor detects this misclassification and raises an alarm.

$$correct\ alarms = \frac{number\ of\ alarms\ for\ misclassified\ images}{total\ number\ of\ misclassified\ images} \tag{6.6}$$

- ***Missed alarms (false negative):*** the network misclassifies the input-image, and the monitor does not detect this misclassification (no alarm is triggered)

$$missed\ alarms = \frac{number\ of\ no\ alarms\ for\ misclassified\ images}{total\ number\ of\ misclassified\ images} \tag{6.7}$$

Value of p	All alarms (#)	Correct alarms (%)	Missed alarms (%)	Correct no alarms (%)	False alarms (%)	Re-classification (%)
0	185	76.06	23.94	99.46	0.55	61.34
0.1	178	76.06	23.94	99.68	0.32	67.40
0.2	177	74.65	25.35	99.71	0.30	69.32
0.3	175	74.18	25.82	99.73	0.27	69.54
0.4	175	73.18	25.82	99.75	0.25	69.94
0.5	175	73.71	26.29	99.75	0.25	69.77
0.6	176	73.71	26.29	99.75	0.25	70.35
0.7	176	73.71	26.29	99.75	0.25	70.93
0.8	176	73.71	26.29	99.75	0.25	71.51
0.9	176	73.71	26.29	99.76	0.24	71.93
1.0	177	75.12	24.88	99.24	0.77	43.42

Table 6.1: The impact of the parameter p on the precision of NAPath-based monitoring. The symbols # and % represent the number and the percentage, respectively.

- ***False alarms (false positive):*** the network correctly classifies the image, however the monitor considers it as a misclassification and raises an alarm

$$\textit{false alarms} = \frac{\textit{number of alarms for well classified images}}{\textit{total number of well classified images}} \tag{6.8}$$

For inputs misclassifed by the network, the monitor suggests to re-classify these inputs (based on NAPaths' similarities). The rate of correctly re-classified inputs is calculated as follows:

$$\textit{correct re-classification} = \frac{\textit{number of correctly re-classified images}}{\textit{total number of raised alarms (all alarms)}} \tag{6.9}$$

Figures 6.7 and 6.8 provide a visual representation of the monitoring performances based on the defined metrics for different values of the control parameter p. In general, the results demonstrate that the monitoring system reaches its best performance with respect to most of the metrics when the value of p is low, i.e., the rate of inactive paths participating in the calculation of the similarity degree is higher than the rate of active paths. As shown in Figure 6.7, the percentage of correct no alarms (correctly classified and no alarm is triggered) is almost stable and very close to 100%. Accordingly, the rate of false alarms (raised alarms for correctly classified images) is almost zero (see Figure 6.8).

For the other metrics, while the rate of correct raised alarms for misclassified

images slightly decreases with p, the rate of correct reclassification roughly increases by increasing the value of p. It means that for high values of p, the monitor raises less alarms for misclassified images, but the re-classification of these images is more accurate. The rate of missed alarms is generally between 23% and 27%. It slightly increases when the value of p increases, and then decreases when p is greater than 0.9. Moreover, the performance of the monitoring system is significantly reduced when only active paths or inactive paths are used in the similarity computation. This is demonstrated by the case when $p = 0$ or $p = 1$, where the rate of correct reclassification is notably low. Hence, it is crucial to consider both active and inactive paths for the similarity computation to achieve better results.

Through this series of experiments, we can conclude that tuning the parameter p directly affects the monitoring performance. Generally, the lower the value of p, the more alarms are raised by the monitor, resulting in fewer missed alarms and more correct alarms, but less precision (less correct re-classification). Therefore, considering a small value of p ($p \leq 0.3$), to include more inactive paths, tends to provide better performance of the monitoring system. For re-classification purposes, the system performs better when p is close to 0.9. However, the adequate value of p depends on the specific case study; including factors such as the network architecture and size, and the characteristics of the training and test sets. Therefore, these findings cannot be generalized, and additional research is required to determine the adequate values of p for different datasets and network configurations.

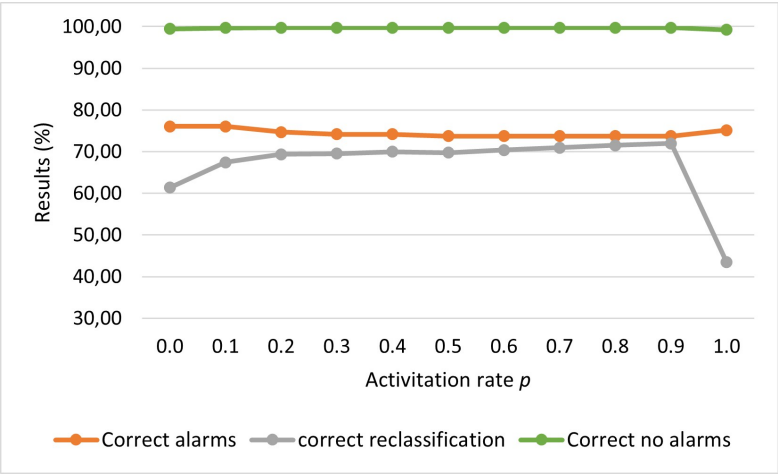


Figure 6.7: The rate of correct alarms and correct reclassification for different values of p.

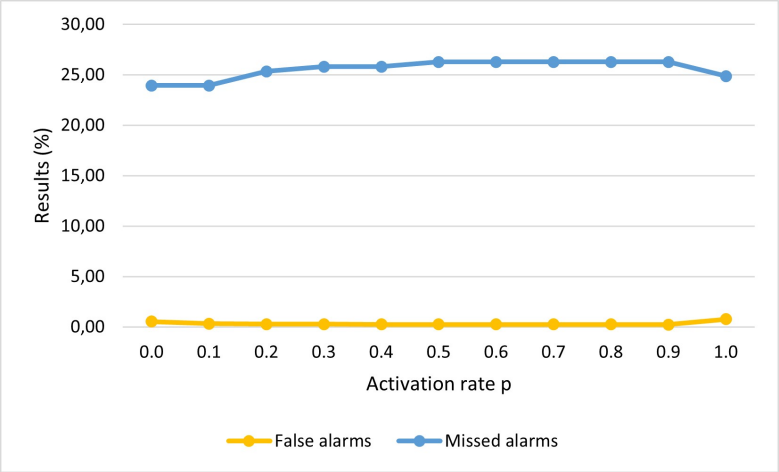


Figure 6.8: The rate of false and missed alarms for different values of p.

Metrics	**All alarms**	**Correct alarms** (%)	**False alarms** (%)	**Missed alarms** (%)
NAP	3995	84.98	64.66	15.02
NAPath	178	76.06	0.32	23.94

Table 6.2: Comparison results between NAP and NAPath.

In our last set of experiments, we conducted a comparison study between our approach based on NAPath and the NAP-based approach (Cheng, Nührenberg, and Yasuoka 2019). To this end, we generated the NAP set using the same training data and setting $\delta = 0.9$. During the monitoring phase, we used the Hamming distance[3] (Cheng, Nührenberg, and Yasuoka 2019), to compare the NAP of the input-image to the pre-computed NAPs, using the same configuration as for NAPath. Additionally, we fixed the control parameter p to 0.1. We evaluated the performance of both approaches based on the rate of correct alarms, false alarms, and missed alarms. Table 6.2 presents a summary of the obtained results.

When comparing the number of raised alarms, we observe that the NAP-based monitor generates more alarms than the NAPath-based monitor. However, the NAPath-based monitor is more precise, as it raises less alarms, and the greater part of them (76.06%) are correct, with very low rate of false alarms (0.32%). On the other hand, the NAP-based monitor triggers an important number of alarms (3995 in total), most of which are for correctly classified images, leading to a high rate of false alarms (64.66%). Although the rates of correct alarms and missed

[3]The Hamming distance between two vectors of the same size is the number of positions at which the corresponding values are different.

Metrics	**All alarms**	**Correct alarms** %	**False alarms** %	**Missed alarms** %
NAP	3995	4.53	95.47	1.15
NAPath	178	**89.50**	**10.50**	**0.86**

Table 6.3: Comparison between NAP and NAPath using updated formulas for computing performance metrics.

alarms using the NAP-based monitor are better, these results are less significant because most of alarms are unnecessary, and raising more alarms increases the chances of covering more misclassified images (less missed alarms). In order to obtain a more significant representation of the results and a more accurate comparison of the performance of the two monitoring systems, we utilize new formulas presented in Equations (6.10), (6.11), and (6.12) to calculate the rate of correct alarms, false alarms and missed alarms, respectively. The obtained results using these formulas are presented in Table 6.3.

$$correct\ alarms = \frac{number\ of\ alarms\ for\ misclassified\ images}{total\ number\ of\ raised\ alarms} \tag{6.10}$$

$$false\ alarms = \frac{number\ of\ alarms\ for\ well\ classified\ images}{total\ number\ of\ raised\ alarms} \tag{6.11}$$

$$missed\ alarms = \frac{number\ of\ passes\ for\ misclassified\ images}{total\ number\ of\ no\ alarms} \tag{6.12}$$

The results presented in Table 6.3 show that the NAPath-based monitor clearly outperforms the NAP-based monitor across all metrics. Specifically, the NAPath monitor is more precise with 89.50% of raised alarms being correct and low rates of missed alarms (0.86%) and false alarms (10.50%).

6.5 Experimental Results on Weather Conditions Detection Networks

In this section, we evaluate the efficiency and the scalability of the NAPAth monitoring approach using a real-world NN system developed within the Autonomous train project (TASV) coordinated by Railenium. The NN model is designed as part of the perception system of the autonomous train, and aims to detect and classify weather conditions based on real images captured by the AI-based perception system of the train. The experiments include evaluating the monitoring system on NN models with various sizes (number of layers and

number of neurons).

Before presenting the experiments' settings and results, let us first introduce the system and the architectures and configurations of its corresponding model.

6.5.1 System & Dataset Characteristics

Weather conditions recognition plays a crucial role in the operation of perception systems for autonomous trains. Such a function relies on accurate detection and classification of weather conditions to ensure safe and efficient train operation in diverse environmental settings. Therefore, there is a growing need for automated methods that can accurately identify weather conditions from images captured in railway environments. Addressing this challenge requires the development of robust and efficient algorithms that are capable of analyzing image data to classify various weather conditions accurately.

In the context of the TASV project, the AI team of IRT Railenium has conducted a study for weather detection and classification from images using NNs. The team has developed NN models using the Pytorch framework[4]. To train these models, a railway dataset of images comprising images representing four distinct weather conditions (represented by classes) was employed: *foggy*, *rainy*, *snowy*, and *sunny*. A total of 1963 images are used, with approximately 490 images per class. Figures 6.9, 6.10, 6.11, and 6.12 provide examples of images corresponding to these four classes.

The dataset is partitioned into three subsets: the training set, validation set, and test set. Approximately 70% of the dataset is allocated for training purposes, while 20% is set aside for validation, and the remaining 10% is designated as the test set. Recall that the validation set is used to fine-tune the model's hyperparameters and supervise its performance during training, while the test set serves as an independent dataset to evaluate the model's generalization ability and assess its performance on unseen data. Table 6.4 provides the number of inputs by class across training, validation, and test sets.

6.5.2 NN Models Configuration

Image classification is the process of segmenting images into different categories based on their features. A feature could be the edges in an image, the pixel intensity, the change in pixel values, and many others. In short, think of NN as a

[4]https://pytorch.org/

Class	Size of the training set	Size of the validation set	Size of the test set
Foggy	342	100	50
Rainy	349	100	50
Snowy	343	100	50
Sunny	329	100	50

Table 6.4: Size of training set, validation set, and test set, for each class.

(a) **(b)** **(c)**

Figure 6.9: Example images depicting foggy weather condition.

(a) **(b)** **(c)**

Figure 6.10: Example images depicting rainy weather condition.

(a) **(b)** **(c)**

Figure 6.11: Example images depicting snowy weather condition.

(a) **(b)** **(c)**

Figure 6.12: Example images depicting sunny weather condition.

machine learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other.

The trained networks work by extracting features from the images. The

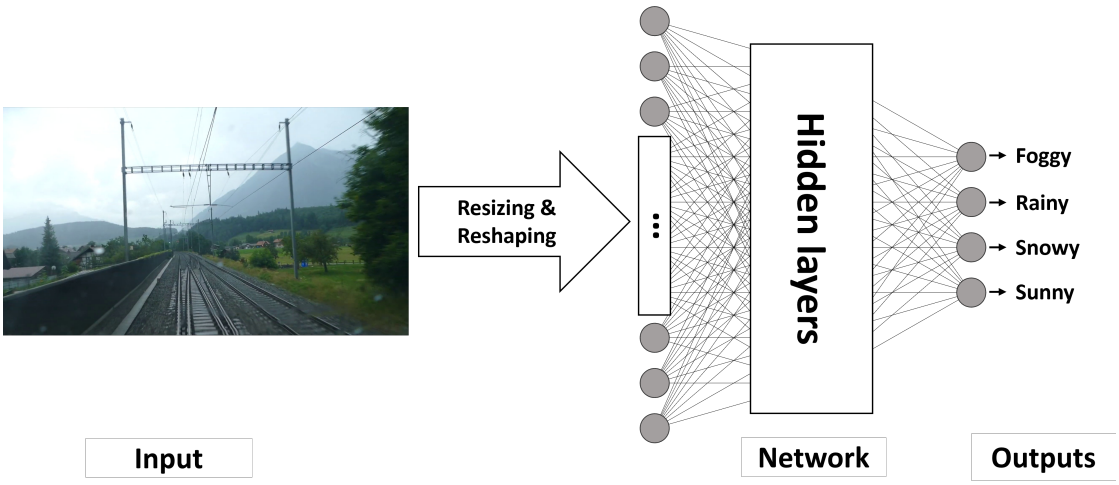


Figure 6.13: The general architecture of the used networks for classifying weather conditions.

implemented networks consist of the following characteristics:

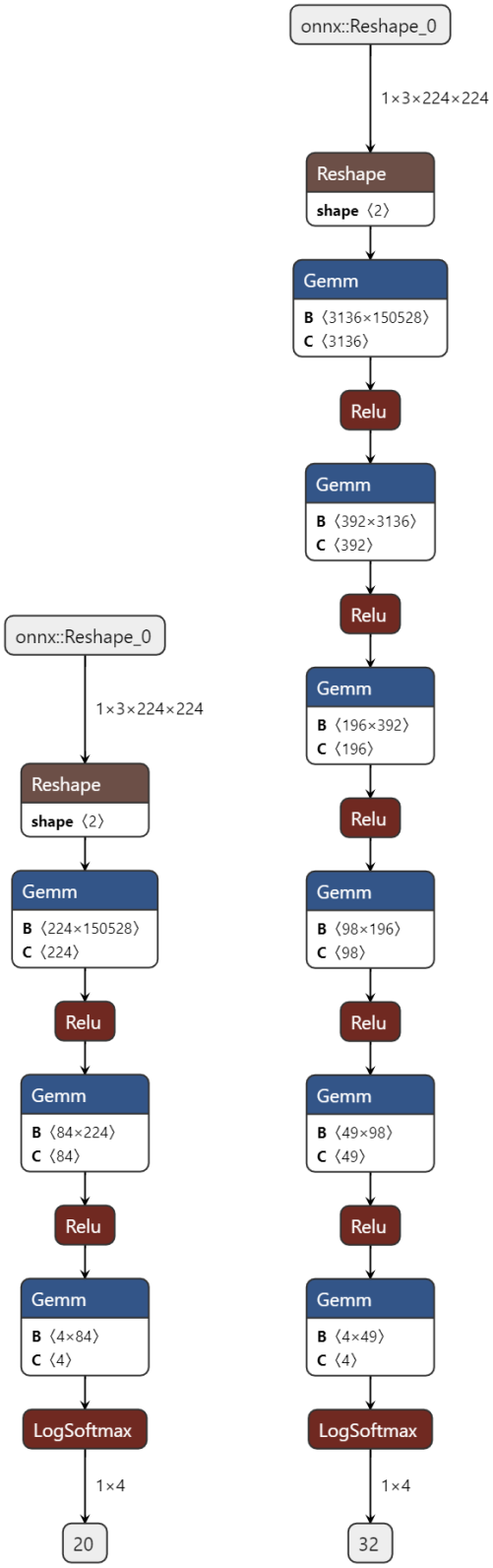
- *The input layer*: it accepts a colored input image of dimension $[3 * 224 * 224]$, which is equivalent to a vector of 150528 inputs, after reshaping it to one dimensional vector of pixels. Notice that a pre-processing phase, including resizing the inputs, is performed in order to obtain images of the above-mentioned dimension.

- *Hidden layers*: they are a sequence of fully connected layers of different widths. All hidden layers use *ReLU* as an activation function. By varying the configuration of the networks, namely the depth (number of hidden layers), a set of networks with different performances is generated. These networks are used to analyze the performance and the scalability of the monitoring system.

- *The output layer*: it is a multi-class labels of dimension $[1 * 4]$ representing the four possible classes: *foggy*, *rainy*, *snowy*, and *sunny*. This layer returns the classification decision of the network. The output layer applies the logarithmic $Softmax$ function to smooth and sort out the final predictions.

The general architecture of the used networks is presented in Figure 6.13. Table 6.5 summarizes the different configurations of the learned models (with respect to the number of hidden layers), as well as the corresponding accuracy obtained by each configuration.

After having introduced the system of weather condition recognition, described the used dataset, and presented the configuration and performance



(a) N_1: two hidden layer.

(b) N_2: five hidden layer.

Figure 6.14: Details about the used networks (ONNX) format.

Network	Number of hidden layers	Size of hidden layers	Accuracy
N_1	2	224, 84	71%
N_2	5	3136, 392, 196, 98, 49	65%

Table 6.5: NN configurations and accuracies

metrics of the utilized NN models, the subsequent section presents the evaluation of our NN monitoring approach on these models. We begin by detailing the settings of the conducted experiments. Subsequently, we present the results obtained from these experiments, accompanied by a thorough discussion and analysis of our findings.

6.5.3 Experimental Settings & Results

As presented in Chapter 6, our NN monitoring approach is implemented as a Python tool. The tool is designed to accept several inputs, including inputs of the NN model in ONNX format, the training set, and the test set. Additionally, NAPath control parameters initialization is required. We divide our experimental study into two series: (i): NAPathing for extracting the set of NAPaths, and (ii) monitoring, for calculating the different alarms rates to assess the performance of the monitoring system.

6.5.3.1 NAPathing Phase

The first step of the NAPath technique consists in extracting the classes' NAPaths. With respect to the output of the NN models, a set of four NAPaths, where each NAPath is associated with a weather class, is extracted and stored to be used later during runtime monitoring. During this phase, the training set is used to compute the set of NAPaths. One of the important parameters of the NAPathing phase is the precision parameter δ. We experimented multiple values of δ, and we carried out a series of tests to determine an appropriate value of this parameter. Concretely, we considered the following values of δ: 0.80, 0.85, 0.90, 0.95, and we run the NAPath computation for each network. Figures 6.15 and 6.16 depict the obtained results on the network N_1, while the obtained results on the network N_2 are presented in Figures 6.17 and 6.18.

The obtained results on networks N_1 and N_2 are consistent with our findings on MNIST benchmark (Chapter 6), which show that increasing the value of δ leads to a decrease in the number of active paths. This implies that the NAPaths are less constrained, and therefore the number of covered inputs by

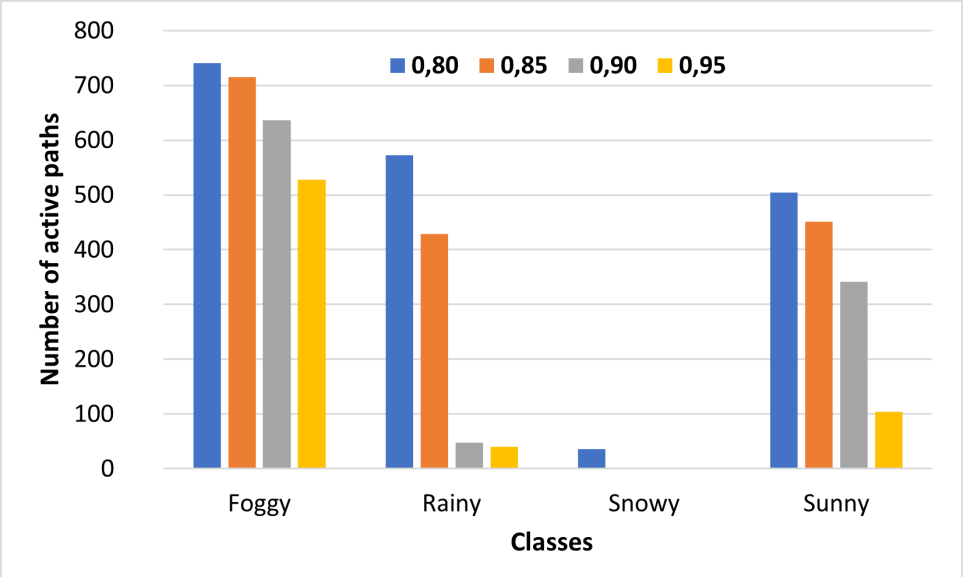


Figure 6.15: The number of active paths for N_1 across different values of δ

these NAPaths (or inputs that follow them) increases as δ decreases.

By focusing on the number of active paths and the number of covered inputs across different networks, we can observe that increasing the number of layers leads to a significant increase in the size of the NAPaths, and to a decrease in the number of covered inputs. For instance, when we compare the number of active paths obtained on N_1 and N_2, we can clearly see that the NAPaths on N_2 have more active paths than those on N_1. On the contrary, for the same value of δ, the number of covered inputs by NAPaths obtained using N_2 is often lower than the number of covered inputs by NAPaths obtained using N_1. This can be explained by the fact that adding more layers increases the total number of neurons within the network, which in turn increases the size of the NAPath. Larger NAPaths impose more constraints, resulting in fewer inputs that can satisfy these constraints. Therefore, as the NAPath size grows larger, the number of covered inputs decreases.

One interesting observation lies in the fact that the NAPath corresponding to class *snowy* has a very small size, and the number of active paths associated with it is almost zero for most of the values of δ. This phenomenon occurs for all the networks that we tested, regardless of their architecture or parameters. This suggests that the *snowy* class is either underrepresented or difficult to classify in the dataset, or the networks have not learned well to distinguish it from the other classes. A deep analysis of the learning process quality and the investigation of further performance metrics could help to understand and handle this issue (which is out of the scope of the present work).

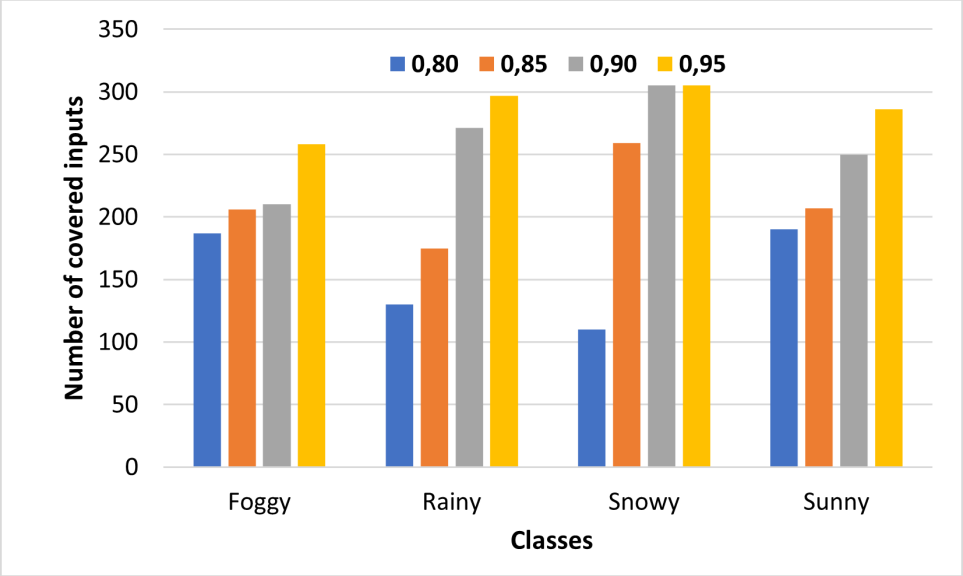


Figure 6.16: The number of covered inputs by the NAPaths of N_1 across different values of δ

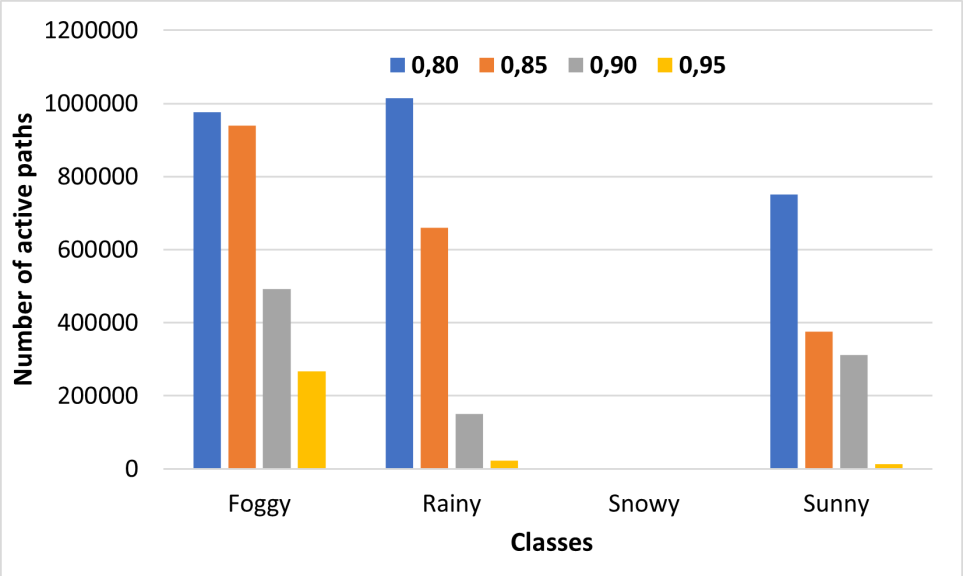


Figure 6.17: The number of active paths for N_2 across different values of δ

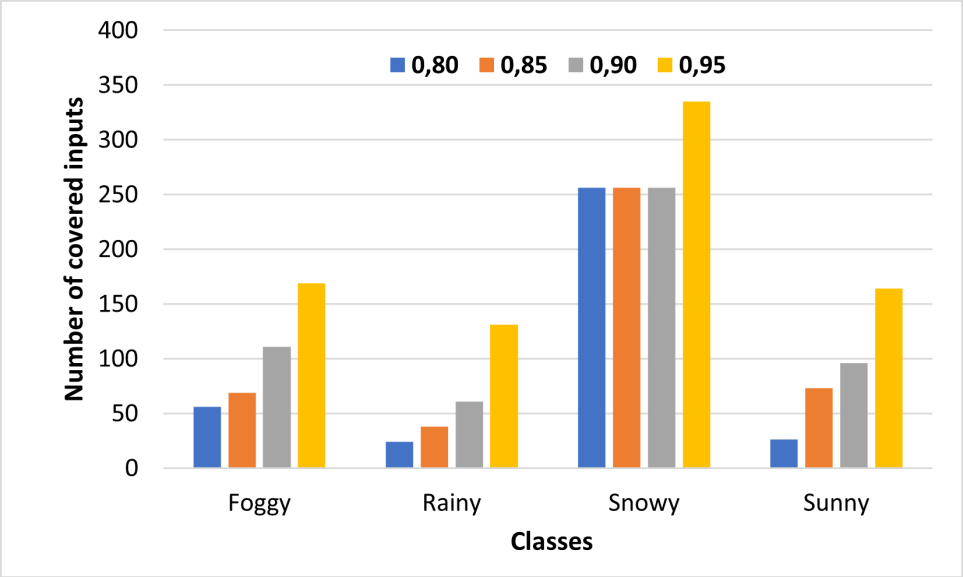


Figure 6.18: The number of covered inputs by the NAPaths of N_2 across different values of δ

6.5.3.2 Monitoring Phase

Once the NAPaths are generated, the next step consists in evaluating our monitor. As shown on the MNIST benchmark, small values of the control parameter p for similarity calculation between NAPaths ($p \leq 0.3$) provided satisfactory results. Therefore, in this second series of experiments, it is straightforwardly set at $p = 0.3$. Then, we employed the test set to evaluate the performance of the monitor using the set of NAPaths computed previously while considering various values of δ. It is worth mentioning that the NAPaths' set of the "snowy" class are excluded due to their low count (mostly equal to zero). The performance evaluation entails computing the various alarm rates, including false alarms, correct alarms, and missed alarms. Additionally, metrics such as correct no alarm (or correct passes) and correct re-classification rates are calculated. Tables 6.6 and 6.7 illustrate the obtained results for N_1, and N_2, respectively. For detailed insights into the computation of these metrics, please refer to Chapter 6.

As presented in Table 6.6, the monitoring performance of the network N_1 remains stable across all values of δ. The rate of false alarms is relatively high at 63.63%, while the rate of correct no alarms is slightly lower at 36.37%. Additionally, the rate of correctly reclassified images is around 6%. However, the rates of correct alarms and missed alarms are deemed acceptable, with a high correct alarm rate of 86.21% and a low missed alarm rate of 13.79%. It's worth noting that the latter metrics have a significant importance in safety critical systems, particularly the rate of missed alarms which must be as small

Value of δ	**All alarms (#)**	**Correct alarms (%)**	**Missed alarms (%)**	**Correct no alarms (%)**	**False alarms (%)**	**Re-classification (%)**
0.80	102	86.21	13.79	36.37	63.63	5.88
0.85	102	86.21	13.79	36.37	63.63	5.88
0.90	102	86.21	13.79	36.37	63.63	5.88
0.95	102	86.21	13.79	36.37	63.63	5.88

Table 6.6: Alarms rates for the network N_1

as possible.

For network N_2, as shown in Table 6.7, the performance of the monitoring system is impacted by the selected set of NAPaths (represented by the parameter δ). For $\delta = 0.80$, both the rates of missed alarms and correct alarms are equal to 50%. However, the rates of correct no alarms and false alarms are very acceptable, with high correct alarm rate and low false alarm rate. Using the set of NAPaths obtained with higher values of δ ($\delta \geq 0.85$) tends to increase the total number of alarms. Consequently, the rates of correct no alarms decreases, while the rate of false alarms increases. On the other hand, the monitor exhibits improved performance in terms of correct alarms (with a higher rate) and missed alarms (with a lower rate) when increasing the value of δ. Indeed, the best performance of the monitoring system considering these two metrics is achieved using the set of NAPaths for $\delta = 0.95$: the rates of correct alarms and missed alarms are equal to 90% and 10%, respectively.

Comparing the performances of our monitoring approach on both networks, we can notice that the approach performs better on the larger network (N_2). One possible explanation for this observation is that adding more layers increases the total number of paths within the network. Consequently, this allows for the extraction of more accurate NAPaths, which serve as reference patterns for

Value of δ	**All alarms (#)**	**Correct alarms (%)**	**Missed alarms (%)**	**Correct no alarms (%)**	**False alarms (%)**	**Re-classification (%)**
0.80	49	50	50	82.50	17.50	12.25
0.85	109	88.57	11.43	41.25	58.75	17.43
0.90	86	78.57	21.43	61.25	38.75	13.95
0.95	113	90	10	37.50	62.50	17.70

Table 6.7: Alarms rates for the network N_2

monitoring the network's behavior.

To summarize, our monitoring system shows an overall acceptable performance in weather condition detection, despite the inherent challenges of the task. Indeed, weather conditions do not manifest as distinct objects, but rather as states that affect the overall appearance of the image. Consequently, visually similar conditions pose a significant challenge for computer vision algorithms, as distinguishing between them accurately can be quite challenging. For example, the difference between an image taken in rainy weather and another in foggy conditions may be subtle. Moreover, it's worth noting that the used dataset is relatively small for this task. With less than 350 images per class in the training set and only 50 images per class in the testing set, the model may face limitations in learning the intricate details necessary for accurate weather condition detection. The small size of the dataset also impacts our approach, which relies on extracting NAPaths from the training set.

6.6 Related Works

Several works have been proposed in the NN literature for addressing the supervision and monitoring of NN-based systems. Our approach closely aligns with those that involve building a monitor by extracting some "learned" patterns. In this context, Neural Activation Pattern (NAP) (Krug et al. 2018) has been used as a basis of the monitoring of NN-based system. Cheng, Nührenberg, and Yasuoka (2019) used the training dataset to build a NAP for each class of images. The extracted patterns are saved using Binary Decision Diagrams (BDD). During execution, each class of images is supposed to have the same class by the NN as that of the closest NAP. Geng et al. (2022) introduced a new formula of the robustness property of NN, called *neural representation as specification*. New constraints using NAP are combined with the standard robustness property, stating that the inputs following the same NAP shall have the same class. While the previous works focus on FFNN, Olber et al. (2022) presented an extension of NAP to support CNN. Additionally, NAP have also been exploited for explaining the NN decisions (Bäuerle et al. 2022).

Henzinger et al. (2020) proposed a method for detecting novel inputs (or out-of-distribution inputs) using abstraction. The method constructs an over-approximation of a set of selected hidden layers using box abstraction. This over-approximation is used during runtime to check whether the hidden layers' values corresponding to an input are within the defined ranges, if not, the

input is marked as *novel*. Hashemi, Křetínskỳ, Mohr, et al. (2021) modelled the activation patterns as a Gaussian model, and then used this model to detect out-of-distribution inputs.

Examining the aforementioned approaches, the NAP-based approaches share some similarities with ours. They both focus on analyzing the network's behavior when fed with its training set in order to extract the activations of the hidden neurons. However, there are significant differences between them. Indeed, while the NAP method represents the activation patterns as a set of neurons, the NAPath uses the hidden activations and represents the features using the concept of paths. These paths connect the input to the output of the network. As shown through the experimental results, this concept allows for preserving the dependency between hidden neurons, which provide more accurate monitoring system compared to the NAP-based method presented in (Cheng, Nührenberg, and Yasuoka 2019).

6.7 Conclusion

In this chapter, we presented an approach called NAPath to extract and represent the learned features by NNs. The approach allows for analyzing the behavior of hidden neurons in response to a set of inputs belonging to the same class, and then computes paths (both active and inactive) that connect the input layer to the output layer of the network. These paths define a NAPath. In our approach, we construct a set of NAPaths, where each NAPath is associated with a single class. These NAPaths are subsequently used to monitor the classification decision of the network, whereby a novelty detection alarm is issued if the NAPath has no similar NAPath from the set of pre-computed NAPaths, or a misclassification alarm is issued if an input's predicted class is different from the class of the most similar NAPath to the one of this input. In the latter, the monitor suggests a new classification of this input based on the similarity degree between its NAPath and the set of pre-computed NAPaths. The similarity degree is calculated using both active and inactive paths. This is based on the assumption that inputs of the same class share similar paths.

To assess the effectiveness of the NAPath approach, we conducted an experimental study on the MNIST benchmark. The study included tuning various parameters and comparing NAPath to NAP. The experiments showed that NAPath can efficiently be used as a tool for monitoring NNs decisions used in image classification, and it can significantly enhance their reliability and

trustworthiness. Furthermore, to evaluate the applicability and the efficiency of the proposed approach within real-world application, we applied our monitoring approach on a set of networks with various sizes used for railway weather conditions' recognition. These NN models are developed within the TASV project. The results of these experiments show that NAPath can effectively scale up and be applied to monitor and supervise NN models of larger sizes.

Chapter 7

Conclusions & Perspectives

Outline of the current chapter

7.1 General Conclusion

This dissertation falls within the scope of verification and monitoring of NN systems in view of their application in safety-critical transportation systems. We firstly presented a comprehensive literature review of existing NN verification and monitoring techniques in Part I. Then, we introduced our main contributions related to NN abstraction for verification on one hand (Part II) and NN monitoring on the other hand (Part III).

In Chapter 2, we discussed the vulnerability and the sensitivity of NNs w.r.t. adversarial attacks and environment perturbations; hence motivating the need for rigorous NN evaluation techniques. In this chapter, We focused on NN verification and NN monitoring techniques. Regarding verification, we discussed two main categories of approaches, namely complete and incomplete methods. A NN verification method is called complete (or exact) if it always returns a complete, definitive answer: either the property holds or not. An NN incomplete verification method may return an "uncertain" answer when the verification process cannot decide whether the property holds or not.

In Chapter 3, we conducted a comprehensive review and critical analysis of the abstraction techniques that leverage neurons merging in order to reduce the NN size while preserving a formal relationship of over-approximation between the reduced network and the original one. Such a relationship is important to lift the verification output to the original network without explicitly applying the verification on it. Through our analysis, we compared several methods in terms of supported activation functions and provided formal guarantees. This study allowed us to identify some limitations of the existing techniques, which helped us to develop new approaches that are presented in Part II.

In the second part, we discussed our contributions for NN abstraction using model reduction. We split this part into two chapters: Chapter 4 and Chapter 5, where the former included the theoretical foundation of our NN model reduction approaches, and the latter presented the results of the experimental evaluation of these approaches. Namely, the first approach, called *INNAbstract*, supports NNs with odd-monotone activation functions, with an alternative to support the ReLU function. In INNAbstract, the incoming weights of an abstract neuron are intervals, and defined as min/max of the signs of the corresponding outgoing weights multiplied by the incoming weights of the original network. The outgoing weights are defined as the sum of absolute values of the outgoing weights of the merged neurons. As demonstrated by the provided proof, this

definition guarantees that the output of the original NN is always within the output interval of the reduced NN. The second approach is a Model reduction for non-negative activation functions. The approach can be applied to reduce the size of NNs with non-negative monotone activation functions. Using the presented approach, the outputs of the generated reduced NNs are always greater or equal to that of the original one. To fulfill this feature, the weights of the abstract neurons are computed in a way to ensure that the output of the network shall be increased. Concretely, we start by eliminating the negative outgoing weights. Then, the incoming weight of the new abstract neuron is defined as the max among the incoming weights of the set of neurons to merge, and the outgoing weight is the sum of the corresponding outgoing weights of the same set.

Chapter 5 was dedicated to a series of experiments to evaluate the two model reduction approaches. The evaluation included a set of randomly generated NNs, and two well-known benchmarks in NN verification, namely, ACAS Xu and MNIST. Additionally, the two approaches are compared to other relevant model reduction methods from the literature. Throughout the conducted experimental study, our methods consistently demonstrated high performances by efficiently and significantly reducing the overall computation time while preserving acceptable output ranges. Indeed, this is confirmed in the comparison study, where both approaches outperformed the others in most of the cases. Notably, the second approach outperformed all other methods, including INNAbstract, considering all metrics. In particular, the approach achieved the smallest computation times with the tightest output ranges. Although the obtained output range using our methods is tighter than those using other methods, there is always an important loss of precision due to the over-approximation.

In Part III, we presented our contribution regarding the monitoring of NN classifiers. In Chapter 6, we presented the fundamentals of our approach based on the NAPAth concept. NAPath is used to monitor image-classification decisions of NNs. The approach extracts internal activations for each class of image represented by its training set. Then, from these activations, we built paths linking the input to the output of the network. The shared set of paths by the inputs constructs a NAPath of the corresponding output class. This process is repeated for each class of images, which allowed us to have a set of NAPaths, where each one represents exclusively a class. Then, the set of NAPaths is used during runtime to check and supervise the network's decisions, and the monitor raises alarms if the decision does not fit the extracted NAPaths. We carried

out an experimental study on the MNIST benchmark in order to analyze the impact of some NAPath's control parameters. Next, we compared our approach to a reference monitoring approach in the literature. Through the conducted experiments, the alarms raised by our monitoring system were more accurate. This shows the effectiveness of the NAPath feature in monitoring NN decisions during runtime. Additionally, we applied our monitoring approach on a set of networks with various sizes used for weather conditions' recognition. These NN models are developed within the TASV project. The results of these experiments indicate that NAPath can effectively scale up and be applied to monitor and supervise NN models of larger sizes. The conducted study may have some critical points; for instance, the proposed approach remains empirical and does not offer any guarantees regarding the extracted features. We should also mention that the proposed extension for CNNs has not yet been tested.

7.2 Perspectives & Future Works

Regarding our contributions and the critical points discussed above, we identified a number of future opportunities for enhancing the results of our research.

7.2.1 Neural Networks Abstraction

We plan to pursue our work on NN abstraction to further enhance the proposed methods and their applicability. In this regard, our future works and perspectives are delineated into short-term and long-term objectives.

Short term future works

- In this dissertation, we applied our model reduction methods on two benchmarks and some others which are randomly generated NNs. To broaden the scope of our findings, future work will involve extending the experimental study to include more benchmarks with larger networks to evaluate the performance and the scalability of the two reduction methods. This extension aims to provide a more comprehensive understanding of the methods' effectiveness across diverse NN benchmarks.

- To tackle the issue of precision loss caused by the abstraction, we intend to advance our heuristics for node selection. These heuristics should

produce tighter output ranges. The pivotal idea is to further explore the characteristics of neurons before merging, e.g., by examining the incoming and the outgoing weights' signs and the sign of the neurons' outputs. In this regard, some optimization algorithms can be investigated to extract optimal subgroups of neurons. By optimal, we mean that the constructed abstract network shall have the tightest output range with respect to the number of merged neurons. Additionally, applying a clustering algorithm is a way to group neurons behaving similarly. Then, each group can be merged and replaced by a single neuron that abstracts the group's behavior.

- Another line of works within this context may consist in suggesting a refinement phase to further enhance the precision of the abstraction approaches. Refinement is the inverse of abstraction, meaning that if the abstraction is too coarse, refinement can be used to split back some merged neurons, and thus providing more precise outputs. The naive way to do this is just to redo the previous iteration. This step can be repeated many times, till we get an acceptable precision-abstraction tradeoff. A smarter way is to identify an abstract neuron that causes the coarseness; thus, splitting this neuron should enhance the precision more than splitting any other abstracted neuron. This procedure can be implemented to iterate over a set of neurons till an appropriate precision is found. Some optimization algorithms and heuristics can be deployed to help identify the set of candidate neurons for splitting.

Long term future works

In the long term, we aim to explore the following research directions:

- We plan to combine our model reduction methods with NN verification techniques, and implement the obtained method as a full and complete NN verification tool. This combination, along with heuristics and a refinement strategy, aims to facilitate the formal verification of properties on NNs. The main idea is to extend existing verification tools to support INNs. Then, once this issue is solved, it will be interesting to apply our model reduction methods (depending on the network's characteristics, namely the activation function) to reduce the network's size. The next step consists in forwarding the obtained reduced network to the verification engine. During the verification process, several refinement steps may be invoked in order to get the final result.

- The second research subject consists in extending the applicability of our approach to support other NN architectures, such as Convolutional Neural Networks (CNNs). This extension is particularly relevant given our focus on verifying NN-based modules deployed in autonomous trains.
- Finally, our ultimate goal is to combine all these features towards verifying real-world NNs, particularly an application case on NN models implemented for the autonomous train.

7.2.2 Neural Networks Monitoring

Short term future works

- We believe that conducting an extensive experimental study is imperative to thoroughly assess our approach. In this regard, we intend to consider a wide and diverse range of networks and benchmarks, varying in size, in terms of the number of layers and the number of neurons per layer. Such extended experimental study shall enable us to gain deeper insights into the behavior of both the neural networks (NNs) and the monitor. Such a comprehensive analysis shall also enhance our understanding of how different network configurations (e.g., size, number of neurons per layer, etc.) and monitor settings influence the overall performance and efficacy of the monitoring system. Additionally, it shall allow us to identify potential challenges and limitations that may arise under various scenarios.
- Furthermore, we plan to meticulously tune the parameters of the monitor to provide comprehensive recommendations regarding their adequate values. By conducting rigorous experiments and parameter tuning, we aim to refine our approach and develop relevant guidelines for deploying and configuring the monitor in real-world scenarios.

Long term future works

- In this dissertation, the presented version of NAPath works specifically with feed-forward NNs. As future works, we plan to extend the NAPath concept to support other types of NNs, such as CNN, ResNet, RNN, etc. For instance, a CNN model contains mainly two parts: the convolutional part and the fully-connected one. While the former is mainly used to extract patterns from images, the latter is known to be in charge of the classification decision. Our first step towards adapting NAPath on CNNs

is to consider the fully-connected part as a separate FFNN; its inputs are the output of the convolutional part and its outputs are the output of the initial CNN model. Then, the main idea consists in applying the approach on the FFNN part. In the second step, we plan to extract paths through the whole network. To this aim, we need to identify active and inactive cells on the CNN kernels, and use these cells to build the NAPath.

- Another research direction involves analyzing the extracted paths to offer assurances regarding the triggered alarms by the network. While our approach has shown promising performance using test sets, the lack of certainty regarding the results remains a limitation. To mitigate this issue, we aim to delve deeper into the internal states and the outcomes of the monitor to provide statistical guarantees about the obtained results. This entails a comprehensive examination of the paths extracted during the NAPathing process. The first step consists of demonstrating that the extracted patterns represent (with some confidence level) the learned patterns by the network. Next, and through rigorous analysis and statistical techniques, we seek to quantify the level of confidence associated with the alarms triggered by the network.

- NAPath for explainable NNs: Finally, we plan to develop an approach based on NAPath to help explain NN decisions. The approach shall involve analyzing the paths associated with each class in the network to gain insights on the decision-making process. Recall that the key idea behind NAPath is to uncover the underlying patterns and features learned by the network during the training process. By examining the paths traversed by the network for the different classes, we aim to identify the distinctive patterns and characteristics that influence its decision-making. These paths represent the sequence of activated neurons or traversed by the network when processing some inputs that belong to a specific class.

Bibliography

Akhtar, Naveed, Ajmal Mian, Navid Kardan, and Mubarak Shah (2021). "Advances in adversarial attacks and defenses in computer vision: A survey". In: *IEEE Access* 9, pp. 155161–155196 (cit. on p. 34).

Akintunde, Michael E, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano (2019). "Verification of RNN-based neural agent-environment systems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01, pp. 6006–6013 (cit. on pp. 54, 62).

Amir, Guy, Haoze Wu, Clark Barrett, and Guy Katz (2021). "An SMT-based approach for verifying binarized neural networks". In: *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*. Springer, pp. 203–222 (cit. on p. 41).

Ashok, Pranav, Vahid Hashemi, Jan Křetínskỳ, and Stefanie Mohr (2020). "DeepAbstract: Neural network abstraction for accelerating verification". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, pp. 92–107 (cit. on pp. 7, 58, 61, 62, 64, 65, 95).

Bäuerle, Alex, Daniel Jönsson, and Timo Ropinski (2022). "Neural activation patterns (NAPs): Visual explainability of learned concepts". In: *arXiv preprint arXiv:2206.10611* (cit. on pp. 47, 145).

Bebis, George and Michael Georgiopoulos (1994). "Feed-forward neural networks". In: *IEEE Potentials* 13.4, pp. 27–31 (cit. on p. 21).

Berry, Michael W, Azlinah Mohamed, and Bee Wah Yap (2019). *Supervised and unsupervised learning for data science*. Springer (cit. on p. 19).

Besinovic, Nikola, Lorenzo De Donato, Francesco Flammini, Rob M. P. Goverde, Zhiyuan Lin, Ronghui Liu, Stefano Marrone, Roberto Nardone, Tianli Tang, and Valeria Vittorini (2022). "Artificial Intelligence in Railway Transport: Taxonomy, Regulations, and Applications". In: *IEEE Transactions on Intelligent Transportation Systems* 23.9, pp. 14011–14024. DOI: 10.1109/TITS.2021.3131637 (cit. on pp. 19, 28).

Biere, Armin, Marijn Heule, and Hans van Maaren (2009). *Handbook of satisfiability*. Vol. 185. IOS press (cit. on p. 30).

Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. (2016). "End to end learning for self-driving cars". In: *arXiv preprint arXiv:1604.07316* (cit. on pp. 60, 61).

Botoeva, Elena, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener (2020). "Efficient verification of relu-based neural networks via dependency analysis". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04, pp. 3291–3299 (cit. on p. 43).

Boudardara, Fateh, Abderraouf Boussif, and Mohamed Ghazel (2023). "A sound abstraction method towards efficient neural networks verification". In: *The 16th International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS), 18-20 October 2023, Marrakech, Morocco, Proceedings*, p. 14 (cit. on pp. 72, 96).

Boudardara, Fateh, Abderraouf Boussif, Pierre-Jean Meyer, and Mohamed Ghazel (2022). "Interval Weight-Based Abstraction for Neural Network Verification". In: *International Conference on Computer Safety, Reliability, and Security*. Springer, pp. 330–342 (cit. on pp. 7, 72).

— (2023a). "A Review of Abstraction Methods towards Verifying Neural Networks". In: *ACM Trans. Embed. Comput. Syst.* ISSN: 1539-9087. DOI: 10.1145/3617508. URL: https://doi.org/10.1145/3617508 (cit. on pp. 44, 50, 95).

— (2023b). "INNAbstract: an INN-based abstraction method for large-scale neural network verification". In: *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, p. xxx (cit. on p. 72).

Brenna, Morris, Federica Foiadelli, and Michela Longo (2016). "Application of genetic algorithms for driverless subway train energy optimization". In: *International Journal of Vehicular Technology* 2016 (cit. on p. 3).

Buciluǎ, Cristian, Rich Caruana, and Alexandru Niculescu-Mizil (2006). "Model compression". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 535–541 (cit. on p. 67).

Bunel, Rudy R, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda (2018). "A unified view of piecewise linear neural network verification". In: *Advances in Neural Information Processing Systems* 31 (cit. on pp. 40, 43).

Carlini, Nicholas and David Wagner (2017). "Towards evaluating the robustness of neural networks". In: *2017 ieee symposium on security and privacy (sp)*. Ieee, pp. 39–57 (cit. on p. 34).

CENELEC-EN50128 (2011). *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems* (cit. on p. 28).

Chakraborty, Anirban, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay (2021). "A survey on adversarial attacks and defences". In: *CAAI Transactions on Intelligence Technology* 6.1, pp. 25–45 (cit. on p. 34).

Cheng, Chih-Hong, Georg Nührenberg, and Harald Ruess (2017). "Maximum resilience of artificial neural networks". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, pp. 251–268 (cit. on pp. 6, 42, 51, 56).

Cheng, Chih-Hong, Georg Nührenberg, and Hirotoshi Yasuoka (2019). "Runtime monitoring neuron activation patterns". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 300–303 (cit. on pp. 5, 10, 33, 46, 120, 134, 145, 146).

Cheng, Yu, Duo Wang, Pan Zhou, and Tao Zhang (2017). "A survey of model compression and acceleration for deep neural networks". In: *arXiv preprint arXiv:1710.09282* (cit. on pp. 62, 66).

Clarke, Edmund M, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. (2018). *Handbook of model checking*. Vol. 10. Springer (cit. on pp. 29, 30).

Clarke, Edmund M and Jeannette M Wing (1996). "Formal methods: State of the art and future directions". In: *ACM Computing Surveys (CSUR)* 28.4, pp. 626–643 (cit. on p. 28).

Cousot, Patrick and Radhia Cousot (1977). "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252 (cit. on pp. 30, 31, 53).

Davis, Martin, George Logemann, and Donald Loveland (1962). "A machine program for theorem-proving". In: *Communications of the ACM* 5.7, pp. 394–397 (cit. on pp. 30, 41).

De Donato, Lorenzo, Francesco Flammini, stefano Marrone, Roberto Nardone, and Valeria Vittorini (2022). "Trustworthy AI for safe autonomy of smart railways: directions and lessons learnt from other sectors". In: *World Congress on Railway Research* (cit. on pp. 4, 5).

DIN-SPEC 92001 (2020). *Inofrmation technology - Life cycle processes and quality requirements - quality meta model*. Accessed: 2023-12-12 (cit. on p. 4).

Dorigo, Marco and Thomas Stützle (2019). *Ant colony optimization: overview and recent advances*. Springer (cit. on p. 2).

Duong, Hai, Linhan Li, ThanhVu Nguyen, and Matthew Dwyer (2023). "A DPLL (T) Framework for Verifying Deep Neural Networks". In: *arXiv preprint arXiv:2307.10266* (cit. on p. 41).

Dutta, Souradeep, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari (2018). "Output Range Analysis for Deep Feedforward Neural Networks". In: *Proc. 10th NASA Formal Methods*, pp. 121–138. ISBN: 978-3-319-77934-8 (cit. on pp. 6, 43, 51).

Ehlers, Ruediger (2017). "Formal verification of piece-wise linear feed-forward neural networks". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer, pp. 269–286 (cit. on pp. 6, 41, 52, 53, 55, 60, 61).

Elboher, Yizhak Yisrael, Justin Gottschlich, and Guy Katz (2020). "An abstraction-based framework for neural network verification". In: *International Conference on Computer Aided Verification*. Springer, pp. 43–65 (cit. on pp. 7, 59, 61, 62, 64, 65, 91, 95, 96, 108–111).

Eramo, Romina, Tiziana Fanni, Dario Guidotti, Laura Pandolfo, Luca Pulina, and Katiuscia Zedda (2022). "Verification of Neural Networks: Challenges and Perspectives in the AIDOaRt Project". In: *10th Italian Workshop on Planning*

and Scheduling, RiCeRcA Italian Workshop, and SPIRITWorkshop on Strategies, Prediction, Interaction, and Reasoning in Italy. (Cit. on p. 6).

Fantechi, Alessandro, Wan Fokkink, and Angelo Morzenti (2012). "Some trends in formal methods applications to railway signaling". In: *Formal methods for industrial critical systems: A survey of applications*, pp. 61–84 (cit. on p. 30).

Ferrari, Alessio and Maurice H Ter Beek (2022). "Formal methods in railways: a systematic mapping study". In: *ACM Computing Surveys* 55.4, pp. 1–37 (cit. on p. 28).

Fischetti, Matteo and Jason Jo (2018). "Deep neural networks and mixed integer linear optimization". In: *Constraints* 23.3, pp. 296–309 (cit. on p. 43).

Fowler, Martin (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional (cit. on p. 60).

Garavel, Hubert (2012). "Three decades of success stories in formal methods". In: *International Conference on Formal Methods for Industrial Critical Systems (FMICS)*, p. 2 (cit. on p. 30).

Gehr, Timon, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev (2018). "AI2: Safety and robustness certification of neural networks with abstract interpretation". In: *2018 IEEE Symposium on Security and Privacy*. IEEE, pp. 3–18 (cit. on pp. 5, 7, 44, 53, 55, 120).

Geng, Chuqin, Nham Le, Xiaojie Xu, Zhaoyue Wang, Arie Gurfinkel, and Xujie Si (2022). "Toward Reliable Neural Specifications". In: *arXiv preprint arXiv:2210.16114* (cit. on pp. 46, 120–122, 145).

Gholami, Amir, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer (2022). "A survey of quantization methods for efficient neural network inference". In: *Low-Power Computer Vision*. Chapman and Hall/CRC, pp. 291–326 (cit. on p. 67).

Ghorbal, Khalil, Eric Goubault, and Sylvie Putot (2009). "The zonotope abstract domain taylor1+". In: *International Conference on Computer Aided Verification*. Springer, pp. 627–633 (cit. on p. 53).

Gibert, Xavier, Vishal M Patel, and Rama Chellappa (2015). "Robust fastener detection for autonomous visual railway track inspection". In: *2015 IEEE winter conference on applications of computer vision*. IEEE, pp. 694–701 (cit. on p. 3).

— (2017). "Deep Multitask Learning for Railway Track Inspection". In: *IEEE Transactions on Intelligent Transportation Systems* 18.1, pp. 153–164. DOI: 10.1109/TITS.2016.2568758 (cit. on p. 3).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press (cit. on pp. 5, 120).

Goodfellow, Ian J, Jonathon Shlens, and Christian Szegedy (2014). "Explaining and harnessing adversarial examples". In: *arXiv preprint arXiv:1412.6572* (cit. on p. 34).

Grimm, Tomás, Djones Lettnin, and Michael Hübner (2018). "A survey on formal verification techniques for safety-critical systems-on-chip". In: *Electronics* 7.6, p. 81 (cit. on pp. 28, 37).

Hadded, Mohamed Amine, Ankur Mahtani, Sébastien Ambellouis, Jacques Boonaert, and Hazem Wannous (2022). "Application of Rail Segmentation in the Monitoring of Autonomous Train's Frontal Environment". In: *Pattern Recognition and Artificial Intelligence*. Ed. by Mounîm El Yacoubi, Eric Granger, Pong Chi Yuen, Umapada Pal, and Nicole Vincent. Cham: Springer International Publishing, pp. 185–197 (cit. on p. 3).

Han, Song, Huizi Mao, and William J Dally (2015). "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (cit. on pp. 62, 66).

Hashemi, Vahid, Panagiotis Kouvaros, and Alessio Lomuscio (2021). "OSIP: Tightened Bound Propagation for the Verification of ReLU Neural Networks". In: *Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings 19*. Springer, pp. 463–480 (cit. on p. 43).

Hashemi, Vahid, Jan Křetínskỳ, Stefanie Mohr, and Emmanouil Seferis (2021). "Gaussian-based runtime detection of out-of-distribution inputs for neural networks". In: *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings*. Springer, pp. 254–264 (cit. on pp. 47, 146).

Hashemi, Vahid, Jan Křetínskỳ, Sabine Rieder, and Jessica Schmidt (2023). "Runtime Monitoring for Out-of-Distribution Detection in Object Detection Neural Networks". In: *International Symposium on Formal Methods*. Springer, pp. 622–634 (cit. on p. 7).

Hawkins, Richard, Colin Paterson, Chiara Picardi, Yan Jia, Radu Calinescu, and Ibrahim Habli (2021). "Guidance on the assurance of machine learning in autonomous systems (AMLAS)". In: *arXiv preprint arXiv:2102.01564* (cit. on p. 5).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 (cit. on p. 41).

Henriksen, Patrick and Alessio Lomuscio (2021). "DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pp. 2549–2555. DOI: 10.24963/ijcai.2021/351. URL: https://doi.org/10.24963/ijcai.2021/351 (cit. on p. 43).

Henzinger, Thomas A, Anna Lukina, and Christian Schilling (2020). "Outside the Box: Abstraction-Based Monitoring of Neural Networks". In: *24th European Conference on Artificial Intelligence-ECAI 2020*, pp. 2433–2440 (cit. on pp. 46, 47, 145).

Hickish, Bob, David I Fletcher, and Robert F Harrison (2020). "Investigating Bayesian Optimization for rail network optimization". In: *International Journal of Rail Transportation* 8.4, pp. 307–323 (cit. on p. 3).

Hua, Gaofeng, Li Zhu, Jinsong Wu, Chunzi Shen, Linyan Zhou, and Qingqing Lin (2020). "Blockchain-Based Federated Learning for Intelligent Control in

Heavy Haul Railway". In: *IEEE Access* 8, pp. 176830–176839. DOI: 10.1109/ACCESS.2020.3021253 (cit. on p. 3).

Huang, Xiaowei, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi (2020). "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability". In: *Computer Science Review* 37, p. 100270. ISSN: 1574-0137 (cit. on pp. 28, 33, 37, 40, 44, 50, 102).

Huang, Xiaowei, Marta Kwiatkowska, Sen Wang, and Min Wu (2017). "Safety verification of deep neural networks". In: *International conference on computer aided verification*. Springer, pp. 3–29 (cit. on pp. 6, 40, 51).

ISO/IEC TR 24028 (2020). *Inofrmation technology - Artificial Intelligence - Overview of trustworthiness in artificial intelligence*. Accessed: 2023-12-12 (cit. on p. 4).

Jacoby, Yuval, Clark Barrett, and Guy Katz (2020). "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*. Springer, pp. 57–74 (cit. on pp. 54, 62).

Jia, Kai and Martin Rinard (2020). "Efficient exact verification of binarized neural networks". In: *Advances in neural information processing systems* 33, pp. 1782–1795 (cit. on p. 62).

Julian, Kyle D, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer (2016). "Policy compression for aircraft collision avoidance systems". In: *Proc. 35th Digital Avionics Systems Conference (DASC)*. IEEE, pp. 1–10 (cit. on pp. 11, 50, 56, 103, 104).

Karaboga, Dervis, Beyza Gorkemli, Celal Ozturk, and Nurhan Karaboga (2014). "A comprehensive survey: artificial bee colony (ABC) algorithm and applications". In: *Artificial intelligence review* 42, pp. 21–57 (cit. on p. 2).

Katz, Guy, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer (2017). "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *International conference on computer aided verification*. Springer, pp. 97–117 (cit. on pp. 5, 36, 40, 42, 51, 56, 61, 108, 120).

Katz, Guy, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. (2019). "The marabou framework for verification and analysis of deep neural networks". In: *International Conference on Computer Aided Verification*. Springer, pp. 443–452 (cit. on pp. 6, 40, 46, 51, 59, 61, 64).

Kochenderfer, Mykel J (2015). *Decision making under uncertainty: theory and application*. MIT press (cit. on pp. 9, 36, 100, 103).

Koopman, Philip, Uma Ferrell, Frank Fratrik, and Michael Wagner (2019). "A safety standard approach for fully autonomous vehicles". In: *Computer Safety, Reliability, and Security: SAFECOMP 2019 Workshops, ASSURE, DECSoS, SASSUR, STRIVE, and WAISE, Turku, Finland, September 10, 2019, Proceedings 38*. Springer, pp. 326–332 (cit. on p. 5).

Krug, Andreas, René Knaebel, and Sebastian Stober (2018). "Neuron activation profiles for interpreting convolutional speech recognition models". In:

NeurIPS Workshop on Interpretability and Robustness in Audio, Speech, and Language (IRASL) (cit. on pp. 47, 145).

Kurakin, Alexey, Ian J Goodfellow, and Samy Bengio (2018). "Adversarial examples in the physical world". In: *Artificial intelligence safety and security*, pp. 99–112 (cit. on pp. 27, 34).

Lagay, Rémy and Gemma Morral Adell (2018). "The Autonomous Train: a game changer for the railways industry". In: *2018 16th international conference on intelligent transportation systems telecommunications (ITST)*. IEEE, pp. 1–5 (cit. on p. 4).

Lambora, Annu, Kunal Gupta, and Kriti Chopra (2019). "Genetic algorithm-A literature review". In: *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE, pp. 380–384 (cit. on p. 2).

Larsen, Kim G and Arne Skou (1991). "Bisimulation through probabilistic testing". In: *Information and computation* 94.1, pp. 1–28 (cit. on p. 59).

Laurendin, Olivier, Sébastien Ambellouis, Anthony Fleury, Ankur Mahtani, Sanaa Chafik, and Clément Strauss (2021). "Hazardous Events Detection in Automatic Train Doors Vicinity Using Deep Neural Networks". In: *2021 17th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–7. DOI: 10.1109/AVSS52988.2021.9663863 (cit. on p. 3).

Lawford, Mark and Alan Wassyng (2012). "Formal verification of nuclear systems: Past, present, and future". In: *1st International Workshop on Critical Infrastructure Safety and Security (CrISS-DESSERT'11)*. Vol. 1, pp. 43–51 (cit. on p. 28).

Lazarus, Christopher and Mykel J Kochenderfer (2022). "A mixed integer programming approach for verifying properties of binarized neural networks". In: *arXiv preprint arXiv:2203.07078* (cit. on p. 62).

LeCun, Yann (1998). *The MNIST database of handwritten digits* (cit. on pp. 9–11, 61, 100, 102).

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *nature* 521.7553, pp. 436–444 (cit. on p. 34).

Leofante, Francesco, Nina Narodytska, Luca Pulina, and Armando Tacchella (2018). "Automated verification of neural networks: Advances, challenges and perspectives". In: *arXiv preprint arXiv:1805.09938* (cit. on pp. 35, 50).

Leucker, Martin (2020). "Formal Verification of Neural Networks?" In: *Formal Methods: Foundations and Applications*. Ed. by Gustavo Carvalho and Volker Stolz. Cham: Springer International Publishing, pp. 3–7. ISBN: 978-3-030-63882-5 (cit. on p. 8).

Leucker, Martin and Christian Schallhart (2009). "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), pp. 293–303. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2008.08.004. URL: https://www.sciencedirect.com/science/article/pii/S1567832608000775 (cit. on p. 45).

Li, Jianlin, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang (2019). "Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification". In: *International Static Analysis Symposium*. Springer, pp. 296–319 (cit. on p. 54).

Li, Zewen, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou (2022). "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12, pp. 6999–7019. DOI: 10.1109/TNNLS.2021.3084827 (cit. on p. 24).

Liang, Tailin, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang (2021). "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461, pp. 370–403 (cit. on pp. 62, 66).

Lipton, Zachary C, John Berkowitz, and Charles Elkan (2015). "A critical review of recurrent neural networks for sequence learning". In: *arXiv preprint arXiv:1506.00019* (cit. on pp. 25, 26).

Litjens, Geert, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. Van Der Laak, Bram Van Ginneken, and Clara I. Sánchez (2017). "A survey on deep learning in medical image analysis". In: *Medical Image Analysis* 42, pp. 60–88. ISSN: 1361-8415. DOI: https://doi.org/10.1016/j.media.2017.07.005 (cit. on p. 28).

Liu, Changliu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. (2021). "Algorithms for verifying deep neural networks". In: *Foundations and Trends® in Optimization* 4.3-4, pp. 244–404 (cit. on p. 50).

Liu, Jiaxiang, Yunhan Xing, Xiaomu Shi, Fu Song, Zhiwu Xu, and Zhong Ming (2022). "Abstraction and Refinement: Towards Scalable and Exact Verification of Neural Networks". In: *arXiv preprint arXiv:2207.00759* (cit. on p. 95).

Liu, Weibo, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi (2017). "A survey of deep neural network architectures and their applications". In: *Neurocomputing* 234, pp. 11–26 (cit. on pp. 2, 20, 24, 26).

Lomuscio, Alessio and Lalit Maganti (2017). "An approach to reachability analysis for feed-forward relu neural networks". In: *arXiv preprint* (cit. on pp. 6, 43, 51, 61).

Loquercio, Antonio, Ana I Maqueda, Carlos R Del-Blanco, and Davide Scaramuzza (2018). "Dronet: Learning to fly by driving". In: *IEEE Robotics and Automation Letters* 3.2, pp. 1088–1095 (cit. on pp. 60, 61).

Madry, Aleksander, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu (2017). "Towards deep learning models resistant to adversarial attacks". In: *arXiv preprint arXiv:1706.06083* (cit. on p. 34).

Mahesh, Batta (2020). "Machine learning algorithms-a review". In: *International Journal of Science and Research (IJSR).[Internet]* 9.1, pp. 381–386 (cit. on p. 19).

Mahtani, Ankur, Wael Ben-Messaoud, Abdelmalik Taleb-Ahmed, Smail Niar, and Clément Strauss (2020). "Pedestrian Detection and Classification for Autonomous Train". In: *2020 IEEE 4th International Conference on Image*

Processing, Applications and Systems (IPAS), pp. 52–57. DOI: 10.1109/IPAS50080.2020.9334938 (cit. on p. 3).

Mamalet, Franck, Eric Jenn, Gregory Flandin, Hervé Delseny, Christophe Gabreau, Adrien Gauffriau, Bernard Beaudouin, Ludovic Ponsolle, Lucian Alecu, Hugues Bonnin, et al. (2021). "White paper machine learning in certified systems". PhD thesis. IRT Saint Exupéry; ANITI (cit. on p. 5).

Masson, Émilie, Philippe Richard, Santiago Garcia-Guillen, and Gemma MORRAL Adell (2019). "TC-Rail: Railways remote driving". In: *Proceedings of the 12th World Congress on Railway Research, Tokyo, Japan*. Vol. 28 (cit. on p. 4).

McCarthy, John, Marvin Minsky, Nathan Rochester, and Claude Shannon (1955). *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. https://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html. Accessed: 2024-01-25 (cit. on p. 2).

Meng, Mark Huasong, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong (2022). "Adversarial Robustness of Deep Neural Networks: A Survey from a Formal Verification Perspective". In: *IEEE Transactions on Dependable and Secure Computing* (cit. on p. 36).

Müller, Mark Niklas, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev (2022). "PRIMA: general and precise neural network certification via scalable convex hull approximations". In: *Proceedings of the ACM on Programming Languages* 6, pp. 1–33 (cit. on p. 54).

Narodytska, Nina, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh (2018). "Verifying properties of binarized deep neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1 (cit. on pp. 41, 62).

Nativi, Stefano. and A. Gómez Losada (2019). "Artificial Intelligence at the JRC". In: Publications Office of the European Union. DOI: doi:10.2760/705074. URL: https://www.academia.edu/42766402/AI_at_JRC_final_29_Jul_2019_ (cit. on p. 2).

Neill, James O' (2020). "An overview of neural network compression". In: *arXiv preprint arXiv:2006.03669* (cit. on pp. 66, 67).

Nwankpa, Chigozie, Winifred L. Ijomah, Anthony Gachagan, and Stephen Marshall (2018). "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *ArXiv* abs/1811.03378 (cit. on pp. 22, 78).

Olber, Bartlomiej, Krystian Radlak, Adam Popowicz, Michal Szczepankiewicz, and Krystian Chachula (2022). "Detection of out-of-distribution samples using binary neuron activation patterns". In: *arXiv preprint arXiv:2212.14268* (cit. on pp. 47, 145).

Ostrovsky, Matan, Clark Barrett, and Guy Katz (2022). "An abstraction-refinement approach to verifying convolutional neural networks". In: *Automated Technology for Verification and Analysis: 20th International Symposium, ATVA 2022, Virtual Event, October 25–28, 2022, Proceedings*. Springer, pp. 391–396 (cit. on p. 62).

Ouimet, Martin and Kristina Lundqvist (2007). "Formal software verification: Model checking and theorem proving". In: *Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA* (cit. on p. 30).

Pimentel, Marco AF, David A Clifton, Lei Clifton, and Lionel Tarassenko (2014). "A review of novelty detection". In: *Signal processing* 99, pp. 215–249 (cit. on p. 47).

Prabhakar, Pavithra (2022). "Bisimulations for Neural Network Reduction". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, pp. 285–300 (cit. on pp. 59–61, 64, 65, 95).

Prabhakar, Pavithra and Zahra Rahimi Afzal (2019). "Abstraction based Output Range Analysis for Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc. (cit. on pp. 7, 56, 57, 61–63, 65, 75, 95, 96, 108, 110, 111).

Pu, Hao, Taoran Song, Paul Schonfeld, Wei Li, Hong Zhang, Jianping Hu, Xianbao Peng, and Jie Wang (2019). "Mountain railway alignment optimization using stepwise & hybrid particle swarm optimization incorporating genetic operators". In: *Applied Soft Computing* 78, pp. 41–57 (cit. on p. 3).

Pulina, Luca and Armando Tacchella (2010). "An abstraction-refinement approach to verification of artificial neural networks". In: *International Conference on Computer Aided Verification*. Springer, pp. 243–257 (cit. on pp. 39, 52, 53).

— (2012). "Challenging SMT solvers to verify neural networks". In: *Ai Communications* 25.2, pp. 117–135 (cit. on p. 39).

Rich, Elaine (1983). *Artificial intelligence* (cit. on p. 2).

Ristić-Durrant, Danijela, Marten Franke, and Kai Michels (2021). "A review of vision-based on-board obstacle detection and distance estimation in railways". In: *Sensors* 21.10, p. 3452 (cit. on p. 3).

Sheikh, Haroon, Corien Prins, and Erik Schrijvers (2023). "Artificial Intelligence: Definition and Background". In: *Mission AI: The New System Technology*. Cham: Springer International Publishing, pp. 15–41. ISBN: 978-3-031-21448-6. DOI: 10.1007/978-3-031-21448-6_2. URL: https://doi.org/10.1007/978-3-031-21448-6_2 (cit. on p. 2).

Sherstinsky, Alex (2020). "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network". In: *Physica D: Nonlinear Phenomena* 404, p. 132306 (cit. on p. 26).

Shriver, David, Dong Xu, Sebastian Elbaum, and Matthew B Dwyer (2019). "Refactoring neural networks for verification". In: *arXiv preprint* (cit. on pp. 60, 61, 65).

Singh, Gagandeep, Rupanshu Ganvir, Markus Püschel, and Martin Vechev (2019). "Beyond the single neuron convex barrier for neural network certification". In: *Advances in Neural Information Processing Systems* 32 (cit. on p. 54).

Singh, Gagandeep, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev (2018). "Fast and effective robustness certification". In: *Advances in neural information processing systems* 31 (cit. on pp. 44, 53, 55, 61).

Singh, Gagandeep, Timon Gehr, Markus Püschel, and Martin Vechev (2019a). "An abstract domain for certifying neural networks". In: *Proceedings of the ACM on Programming Languages* 3.POPL, pp. 1–30 (cit. on pp. 53–55).

— (2019b). "Boosting robustness certification of neural networks". In: *International Conference on Learning Representations* (cit. on p. 54).

Sotoudeh, Matthew and Aditya V Thakur (2020). "Abstract Neural Networks". In: *International Static Analysis Symposium*. Springer, pp. 65–88 (cit. on pp. 57, 58, 61–63, 65, 95).

Stano, Martin, Wanda Benesova, and Lukas Samuel Martak (2020). "Explaining predictions of deep neural classifier via activation analysis". In: *arXiv preprint arXiv:2012.02248* (cit. on p. 47).

Szegedy, Christian, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus (2014). "Intriguing properties of neural networks". In: *In 2nd International Conference on Learning Representations, ICLR 2014* (cit. on pp. 27, 28).

Tang, Ruifan, Lorenzo De Donato, Nikola Besinovic, Francesco Flammini, Rob M.P. Goverde, Zhiyuan Lin, Ronghui Liu, Tianli Tang, Valeria Vittorini, and Ziyulong Wang (2022a). "A literature review of Artificial Intelligence applications in railway systems". In: *Transportation Research Part C: Emerging Technologies* 140, p. 103679. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2022.103679. URL: https://www.sciencedirect.com/science/article/pii/S0968090X22001206 (cit. on pp. 3, 20).

Tang, Ruifan, Lorenzo De Donato, Nikola Besinović, Francesco Flammini, Rob MP Goverde, Zhiyuan Lin, Ronghui Liu, Tianli Tang, Valeria Vittorini, and Ziyulong Wang (2022b). "A literature review of Artificial Intelligence applications in railway systems". In: *Transportation Research Part C: Emerging Technologies* 140, p. 103679 (cit. on p. 2).

Taylor, Brian J (2006). *Methods and procedures for the verification and validation of artificial neural networks*. Springer Science & Business Media (cit. on p. 39).

Tjeng, Vincent, Kai Y. Xiao, and Russ Tedrake (2019). "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *International Conference on Learning Representations* (cit. on pp. 6, 42, 43, 51).

Tran, Hoang-Dung, Stanley Bak, Weiming Xiang, and Taylor T Johnson (2020). "Verification of deep convolutional neural networks using imagestars". In: *International conference on computer aided verification*. Springer, pp. 18–42 (cit. on p. 52).

Tran, Hoang-Dung, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T Johnson (2019). "Star-based reachability analysis of deep neural networks". In: *Formal Methods–The Next 30 Years: Third World Congress, FM 2019, Porto, Portugal, October 7–11, 2019, Proceedings 3*. Springer, pp. 670–686 (cit. on p. 52).

Tran, Hoang-Dung, Weiming Xiang, and Taylor T Johnson (2020). "Verification approaches for learning-enabled autonomous cyber-physical systems". In: *IEEE Design & Test* (cit. on p. 50).

Trentesaux, Damien, Rudy Dahyot, Abel Ouedraogo, Diego Arenas, Sébastien Lefebvre, Walter Schön, Benjamin Lussier, and Hugues Chéritel (2018a). "The Autonomous Train". In: *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, pp. 514–520. DOI: 10.1109/SYSOSE.2018.8428771 (cit. on p. 3).

— (2018b). "The autonomous train". In: *2018 13th Annual Conference on System of Systems Engineering (SoSE)*. IEEE, pp. 514–520 (cit. on p. 4).

Urban, Caterina and Antoine Miné (2021). "A review of formal methods applied to machine learning". In: *arXiv preprint* (cit. on pp. 6, 28, 44, 50).

VDE-AR-E 2842-61 (2021). *Development and trustworthiness of autonomous/cognitive systems*. Accessed: 2023-12-12 (cit. on p. 5).

Velastin, Sergio A and Diego A Gómez-Lira (2017). "People detection and pose classification inside a moving train using computer vision". In: *Advances in Visual Informatics: 5th International Visual Informatics Conference, IVIC 2017, Bangi, Malaysia, November 28–30, 2017, Proceedings 5*. Springer, pp. 319–330 (cit. on p. 3).

Wang, Shiqi, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana (2018a). "Efficient formal safety analysis of neural networks". In: *Advances in Neural Information Processing Systems* 31 (cit. on pp. 7, 43, 44, 52, 61).

— (2018b). "Formal security analysis of neural networks using symbolic intervals". In: *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1599–1614 (cit. on pp. 7, 42–44, 52).

Wäschle, Moritz, Florian Thaler, Axel Berres, Florian Pölzlbauer, and Albert Albers (2022). "A review on AI Safety in highly automated driving". In: *Frontiers in Artificial Intelligence* 5, p. 952773 (cit. on p. 2).

Watterson, Conal and Donal Heffernan (2007). "Runtime verification and monitoring of embedded systems". In: *IET software* 1.5, pp. 172–179 (cit. on p. 44).

Woodcock, Jim, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald (2009). "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4, pp. 1–36 (cit. on p. 28).

Wu, Changshun, Yliès Falcone, and Saddek Bensalem (2023). "Customizable Reference Runtime Monitoring of Neural Networks Using Resolution Boxes". In: *Runtime Verification*. Ed. by Panagiotis Katsaros and Laura Nenzi. Cham: Springer Nature Switzerland, pp. 23–41. ISBN: 978-3-031-44267-4 (cit. on p. 47).

Wu, Jianxin (2017). "Introduction to convolutional neural networks". In: *National Key Lab for Novel Software Technology. Nanjing University. China* 5.23, p. 495 (cit. on pp. 24, 25).

Xiang, Weiming, Hoang-Dung Tran, Xiaodong Yang, and Taylor T Johnson (2020). "Reachable set estimation for neural network control systems: A simulation-guided approach". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.5, pp. 1821–1830 (cit. on pp. 7, 42, 44).

Xu, Han, Yao Ma, Hao-Chen Liu, Debayan Deb, Hui Liu, Ji-Liang Tang, and Anil K Jain (2020). "Adversarial attacks and defenses in images, graphs and

text: A review". In: *International Journal of Automation and Computing* 17, pp. 151–178 (cit. on pp. 6, 27, 28).

Xu, Jin, Zishan Li, Miaomiao Zhang, and Bowen Du (2021). "Conv-Reluplex: a verification framework for convolution neural networks". In: *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering (SEKE)* (cit. on p. 62).

Yang, Hongfei, Yanzhang Wang, Jiyong Hu, Jiatang He, Zongwei Yao, and Qiushi Bi (2022). "Deep Learning and Machine Vision-Based Inspection of Rail Surface Defects". In: *IEEE Transactions on Instrumentation and Measurement* 71, pp. 1–14. DOI: 10.1109/TIM.2021.3138498 (cit. on pp. 3, 20).

Yang, Jingkang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu (2021). "Generalized out-of-distribution detection: A survey". In: *arXiv preprint arXiv:2110.11334* (cit. on p. 47).

Yang, Pengfei, Jianlin Li, Jiangchao Liu, Cheng-Chao Huang, Renjue Li, Liqian Chen, Xiaowei Huang, and Lijun Zhang (2021). "Enhancing robustness verification for deep neural networks via symbolic propagation". In: *Formal Aspects of Computing* 33.3, pp. 407–435 (cit. on p. 54).

Yuan, Chunyu and Sos S Agaian (2023). "A comprehensive review of binary neural network". In: *Artificial Intelligence Review*, pp. 1–65 (cit. on p. 41).

Zhang, Caiming and Yang Lu (2021). "Study on artificial intelligence: The state of the art and future prospects". In: *Journal of Industrial Information Integration* 23, p. 100224 (cit. on pp. 2, 19).

Zhang, Haoran, Meng Yuan, Yongtu Liang, Bohong Wang, Wan Zhang, and Jianqin Zheng (2018). "A risk assessment based optimization method for route selection of hazardous liquid railway network". In: *Safety science* 110, pp. 217–229 (cit. on p. 3).

Contributions to the verification and monitoring of neural network systems

Abstract

The evaluation and verification of neural networks (NNs), as a part of their safe design and deployment, becomes a hot research topic, particularly with the recent studies showing their sensitivity and vulnerability to operational conditions (adversarial attacks, environment conditions, etc.). Despite its importance in ensuring the accuracy and the reliability of NNs, test-based approaches for NNs evaluation suffer from several limitations that may impact their effectiveness. To overcome the limitation of NN testing, researchers are exploring formal verification as complementary activity to enhance the reliability and safety of NN-based systems. Indeed, while testing can illustrate the ability of a system to maintain its level of performance under varying conditions, proving this requires some form of formal analysis. NN verification aims to provide formal guarantees regarding the behavior and properties of NNs. It involves analyzing the model's inputs, outputs, and internal computations to ensure that the network behaves correctly and meets the desired specifications. While NN verification is applied before the deployment of the network, NN runtime verification (or monitoring) is used to continuously check and assess the correct behavior of the network during runtime. Broadly speaking, NN monitoring consists in building a monitor that runs in parallel to the network in order to supervise its behavior and decision. If the monitor detects a malfunctioning of the network, or some abnormal behavior, it raises alarms demanding an examination of the current decision. Although these techniques have been successfully applied in solving certain properties of NN, NN verification and monitoring remains challenging, particularly when it comes to verifying large networks with practical interests. This is mainly due to the complexity and the non-linearity of NN models, and the limitations of the traditional formal methods to scale up to large real-world models.

In this dissertation, we propose two main contributions to address these challenges, namely (*i*) NN abstraction (model reduction) for verification purposes and (*ii*) NN runtime monitoring. In order to enhance the scalability of NN verification, we propose two approaches involving the merging of neurons within the same hidden layer of a network to reduce its size. Both approaches ensure that the resulting reduced model over-approximates the original network. The over-approximation relation is crucial, as it guarantees that any verified property on the reduced model remains valid on the original network. Our proposed approaches rely on mathematical formulas to formally establish and ensure this over-approximation relationship. Additionally, we provide formal proofs of this relation for each approach, thus ensuring the rigor and reliability of our methods. The second contribution involves the development of a monitoring system specifically designed for NNs used in image classification tasks. The key idea behind this approach is to identify and extract relevant path activations that are referred to as *NAPath*. The computation of NAPaths is performed for each class of images using the training set. Each NAPath serves as a reference pattern that captures the essential characteristics of the associated class. During the runtime, the monitoring system compares the network's classification results to the most similar NAPath. This comparison analysis enables the monitor to evaluate the consistency of the network's classification decisions and detect any potential deviations or misclassification. To evaluate the proposed approaches, we have implemented them as Python tools and carried out a set of experiments on well-known NN benchmarks and/ or railway use cases.

Keywords: railway system safety; safe artificial intelligence (safe ai); formal verification; neural networks; neural networks verification; neural networks abstraction; neural networks monitoring

Résumé

L'évaluation et la vérification des réseaux neuronaux (NN), lors de leur conception et de leur déploiement sécurisé, suscitent un vif intérêt de recherche, en particulier à la lumière d'études récentes mettant en évidence leur sensibilité et leur vulnérabilité face à diverses conditions opérationnelles telles que les attaques adverses et les variations environnementales. Bien que leur importance soit cruciale dans l'assurance de la précision et de la fiabilité des NN, les approches fondées sur les tests pour l'évaluation des NN sont entravées par plusieurs limitations qui peuvent compromettre leur efficacité. Afin de remédier à ces limites, des travaux de recherche explorent la vérification formelle comme approche complémentaire pour vérifier la fiabilité et la sécurité des NN. Si les tests peuvent démontrer la capacité d'un système à maintenir ses performances dans des conditions variables, démontrer cela nécessite une analyse formelle approfondie. La vérification des NN vise ainsi à fournir des garanties formelles quant au comportement et aux propriétés des NN. Cela implique une analyse minutieuse des entrées, des sorties et des calculs internes du modèle pour s'assurer que le réseau se comporte correctement vis-à-vis des spécifications requises. Alors que la vérification des NN est, d'une manière générale, effectuée lors de la conception des systèmes, le monitoring des NN, quant à lui, est cruciale pour garantir l'exécution du bon comportement lors de la phase opérationnelle. Pour ce faire, un système de monitoring fonctionne en parallèle du NN, détectant toute anomalie ou comportement inattendu et déclenchant des alertes en cas de besoin. Bien que des succès aient été rencontrés dans certaines applications, la vérification et le monitoring des NN restent des défis, notamment lorsqu'il s'agit de NN complexes ou de taille importante. Cette difficulté découle en grande partie de la complexité et de la non-linéarité des modèles NN, ainsi que des limites des méthodes formelles existantes pour s'adapter à des modèles réels de grande taille.

Dans cette thèse, nous proposons deux contributions principales pour répondre aux deux défis susmentionnés. Concrètement, nous proposons des techniques d'abstraction des NN (réduction de modèle) à des fins de vérification, ainsi qu'une approche de monitoring des NN en temps réel spécifiquement conçu pour les tâches de classification d'images. Ces approches d'abstraction de NN visent à améliorer la scalabilité et l'efficacité de la vérification des NN. En se basant sur des formulations mathématiques, nous garantissons que les modèles abstraits résultants conservent les propriétés essentielles (e.g., la sur-approximation) des réseaux d'origine, assurant ainsi la validité des résultats de vérification. Pour la partie monitoring, nous développons un système de surveillance qui identifie et évalue en temps réel les décisions de classification des réseaux par rapport à des modèles de référence spécifiques (motives de surveillance). Enfin, nous validons nos approches à travers des expérimentations menées sur des benchmarks académiques, ainsi qu'un cas d'étude spécifique au domaine ferroviaire.

Mots clés : sécurité des systèmes ferroviaires ; sécurité de l'intelligence artificielle ; réseaux de neurones ; vérification formelle, vérification des réseaux de neurones ; abstraction des réseaux de neurones, monitoring des réseaux de neurones