REACHABILITY ANALYSIS AND REPAIR OF DEEP NEURAL NETWORKS IN AUTONOMOUS SYSTEMS

By

Xiaodong Yang

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

June 30, 2022

Nashville, Tennessee

Approved:

Taylor T. Johnson, Ph.D.

Alan Peters, Ph.D.

Bardh Hoxha, Ph.D.

Gautam Biswas, Ph.D.

Ipek Oguz, Ph.D.

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Taylor Johnson for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation. Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Alan Peters, Prof. Gautam Biswas, Prof. Ipek Oguz, and Dr. Bardh Hoxha, for their interest in my research, taking time to serve on my dissertation committee, and providing insightful comments and suggestions, which incented me to widen my research from various perspectives. My sincere thanks also go to Dr. Danil Prokhorov, Tom Yamaguchi, and Dr. Bardh Hoxha, who provided me an opportunity to join their team as an intern, and their treasured advice in my research on verification of neural networks and learning-enabled cyber-physical systems. I also would like to thank my colleagues and labmates for the cherished time spent together over the last five years. In particular, I am grateful to Dr. Weiming Xiang and Dr. Hoang-Dung Tran for their inspiring discussion and advices when I struggled with my research directions. Lastly but most importantly, I would like to take this opportunity to express my profound gratitude to my parents and two older sisters for their encouragement, unconditional and endless love.

The material presented in this paper is based upon work supported the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089, the Air Force Office of Scientific Research (AFOSR) award FA9550-22-1-0019, and the National Science Foundation (NSF) through grant numbers 1918450, 1910017, and 2028001. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of DARPA, AFOSR or NSF.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

During the past decade, there has been a surge of research on the artificial intelligence (AI). Because of its extraordinary capability of handling complex and practical problems, it has been increasingly prevalent in the economies and societies, such as autonomous driving, advertisement and healthcare. Its rising popularity also highlights some critical problems. One is the lack-of-explainability issue which raises one major concern about the trustworthiness of AI, where AI should consistently generate reliable outputs and perform task properly rather than conduct unpredictable devastating behaviors.

To protect safety-critical systems with learning-enabled components, there is an urgent demand to establish approaches that can verify and guarantee the safety of AI. The conventional methods are normally based on falsification tests, a type of statistical tests that can automatically identify system behaviors that fails to satisfy the system specification. However, these methods are significantly computational expensive due to large amount of simulations. Additionally, they are incomplete, which means they are capable of only finite conclusions and cannot check all the possible scenarios, especially in a continuous environment with infinite cases. Therefore, it is inevitably necessary to develop formal methods, mathematical approaches to explore all the scenarios in reasoning which would be left unverified in the falsification test, and thus can rigorously prove whether the system specification holds or not. For instance, in the autonomous driving, a collision-avoidance system controller with AI components takes information from a set of sensors such as speed, acceleration and distance of the ego vehicle relative to other objects, and makes reasonable plans to avoid the collision. Formal methods can provide safety certification by taking all uncertain measurement noise and adversarial attacks into account and verifying the reliability of such AI components in consistently outputting correct actions.

Among the fundamental AI techniques, deep neural networks (DNN) have been widely applied in a number of industrial systems including safety-critical systems because of their versatility and ease-of-use. DNNs are based on simple mathematical operations combining neurons containing non-linear activation functions in layers, whose functionality, however, is difficult to be interpreted by humans. This lack of transparency leads to one major barrier to realizing trustful autonomy. Moreover, it has been demonstrated that DNNs are extremely vulnerable to adversarial attacks Carlini and Wagner (2017); Szegedy et al. (2014). Slight perturbation on the input would generate an unpredictable output variation, resulting in object misclassifi-

cation and unsafe actions with devastating effects. Therefore, in this dissertation, I propose a framework based on computational geometry to conduct reachability analysis of DNNs including feedforward neural networks (FFNN) and convolutional neural networks (CNN), such that (1) the safety of FFNN, particularly neural network controllers in autonomous systems, can be formally verified efficiently, (2) unsafe behaviors can be fully identified to contribute to the safety improvement of DNNs, and (3) the adversarial robustness of image-classification CNNs can be fairly evaluated. Furthermore, based on these reachability analysis methods, I also propose a framework to repair unsafe DNNs to produce provably safe models on multiple safety properties with negligible performance degradation.

1.2 Contributions

1.2.1 Reachability Analysis of Deep ReLU Neural Networks using Facet-Vertex Incidence

The versatility and ease-of-use of DNNs has made them popular in a number of industrial systems, including safety-critical systems such as autonomous systems. However, a major barrier in realizing trustful autonomy is the lack of scalable methods for safety verification. One of the major approaches for safety verification is through reachability analysis which can provide a formal and insightful evaluation of the DNN behaviors.

In this work, we propose a set-based method for exact reachability analysis of DNNs. We utilize a Facet-Vertex Incidence Matrix (FVIM) for the set representation. It represents the combinatorial structure of convex sets by encoding the containment relation between their facets and vertices. The FVIM has very useful properties for set manipulation. In the reachability analysis of ReLU DNNs, the major challenge origins from the analysis of ReLU neurons. Most of the state-of-the-art works treat this analysis as a linear programming (LP) problem Xiang et al. (2017); Tran et al. (2020b); Bak et al. (2020). However, solving of numerous high dimensional LP problems can result in an undesired efficiency. In contrast, the reachability analysis based on the FVIM can handle this problem more efficiently by avoiding the LP problem. This is because the vertices of a set encoded in FVIM can be directly applied to determine whether the input set spans both negative and positive input ranges of the ReLU function in neurons. When the input set spans these two input ranges, it can also be quickly split into two subsets. This speeds up the computation of the reachable set significantly in comparison with the related methods. An additional feature of the proposed method is that it enables output-to-input backtracking. This enables the computation of the complete input subspace that causes safety violations, which is especially useful for debugging and retraining the network. The performance of our method is evaluated and compared to other state-of-the-art methods by using the ACAS Xu DNNs and other benchmarks.

1.2.2 Reachability Analysis of Convolutional Neural Networks with Face Lattice

Applications of Convolutional Neural Networks (CNNs) to safety-critical systems with learning-enabled components, such as autonomous vehicles and medical devices have increased substantially. However, neural networks are vulnerable to adversarial attacks Carlini and Wagner (2017); Szegedy et al. (2014). It has been shown that CNN performance is often susceptible to pixel-level variations in images. Formal methods are urgently needed to evaluate the safety and adversarial robustness of CNNs.

To address this challenge, first, we propose an approach to compute the exact output reachable domain of a CNN given a pixel-wise input domain, where the reachable domain is a union of many output reachable sets. To improve the computational efficiency, we utilize a novel set representation, face lattice, to encode reachable sets. The face lattice is a structure that encodes the complete combinatorial structure of a convex polytope. Besides the computation of reachable sets, our approach is also capable of backtracking to the input domain given an output reachable set, such that *problematic* subspace in the input domain that is responsible for misclassification can be identified. Second, an approach for fast analysis is also introduced, which conducts fast computation of reachable sets by considering selected sensitive neurons in each layer. It allows the practitioner to select the trade-off between computation speed and completeness. Third, our reachability analysis method can be used to evaluate the robustness of networks trained with defense methods by approximating the maximal output variation w.r.t. an input perturbation. Our exact pixel-level reachability analysis method is evaluated on a CIFAR10 CNN and compared to related works. The fast analysis method is evaluated over a CIFAR10 CNN and a VGG16 for the ImageNet dataset.

1.2.3 Neural Network Repair with Reachability Analysis

Recently, many works for analyzing behaviors of DNNs present post-training verification methods that generate safety certificate over input-output safety properties. An important challenge that remains is the repair problem, where given a DNN with erroneous behaviors, an automatic process repairs the network w.r.t. a set of safety properties.

In this work, we propose a repair method for general ReLU DNNs based on exact reachability analysis of DNNs. Compared to over-approximation analysis, exact reachability analysis enables us to precisely compute the entire unsafe reachable state domain of DNNs and its distance to the safe domain. In the repair process, this distance is constructed with a novel loss function for minimization. Additionally, by combining this loss function with another objective function that minimizes the deviation of the DNN parameters, a minimal-repaired DNN can be generated such that the original behaviors of the DNN can be preserved to the utmost extent. Overall, contributions can be summarized as follows. First, we develop a repair method for ReLU DNNs based on exact reachability analysis, which can successfully repair unsafe DNNs on multiple

safety properties with negligible impact on performance. Second, we also extend our repair method to deep reinforcement learning by integrating it with the deep deterministic policy gradient (DDPG) algorithm, a well-known algorithm that operates over continuous action spaces. Third, we present a novel depth-first-search reachability analysis algorithm that includes both exact and over-approximation methods. This results in a five-fold computational speedup and two-fold memory reduction when compared to other state-of-the-art approaches. Forth, the repair method is evaluated on two benchmark problems against a state-of-the-art method, where the detailed effects of repair methods on DNNs are thoroughly evaluated.

1.2.4 Veritex: A Tool for Reachability Analysis and Repair of Deep Neural Networks

This work presents a toolbox named Veritex for reachability analysis and repair of deep neural networks (DNNs). Veritex includes methods for the exact reachability analysis and over-approximation analysis of DNNs using different novel set representations, such as facet-vertex incidence matrix, face lattice, and $\mathcal{V}$-zono. In addition to sound and complete safety verification of DNNs, these methods can also efficiently compute the exact output reachable domain as well as the exact unsafe input space that causes safety violations of DNNs. More importantly, based on the exact unsafe input-output reachable domain, Veritex can repair unsafe DNNs on multiple safety properties with negligible performance degradation. The repair is conducted by updating the DNN parameter through retraining. It also works in the absence of the safe model reference and the original dataset for learning. Veritex primarily addresses the issue of constructing provably safe DNNs which is commonly missing in most of the current formal methods for trustworthy AI. The utility of Veritex is evaluated from two aspects, safety verification and DNN repair. The benchmark for verification is the ACAS Xu networks, and the benchmarks for the repair include the unsafe ACAS Xu networks and unsafe agents trained in deep reinforcement learning.

CHAPTER 2

State-of-Art Works for Safety Verification of Deep Neural Networks

2.1 Neural Network Verification

Recently, there has been significant effort to develop methods to establish robustness and formal guarantees of Learning-Enabled Components (LECs) Pulina and Tacchella (2010); Huang et al. (2017); Gehr et al. (2018); Dutta et al. (2018); Katz et al. (2017); Henriksen and Lomuscio (2019); Xiang et al. (2018); Wang et al. (2018b); Zhang et al. (2018); Singh et al. (2018a, 2019b); Wang et al. (2018a); Tran et al. (2019b,c); Liu et al. (2019). These verification algorithms are *sound*, denoting that when they return holds, the safety property actually holds. Some of them are also *complete*, denoting that whenever the safety property actually holds, they will return holds. In terms of their methodologies, these algorithms can be classed into four main categories. They are reachability analysis based, satisfiability (SAT) and satisfiability modulo theories (SMT) solvers based, mixed-integer linear programming (MILP) based, and symbolic interval based.

2.2 Reachability based Methods

Reachability analysis is that given an input domain to a DNN, either the exact output reachable domain or its over approximation will be computed. Through the computation of the output domain, the safety properties which define desired output constraints for specific input spaces can be directly checked. The over approximation provides *sound* analysis, while the exact computation can guarantee both the *soundness* and *completeness*. The primary challenge for the reachability analysis of ReLU DNNs origins from the piece-wise linear *max* function in ReLU neurons and the Max-pooling layer. The *max* function exhibits different linearity over different input ranges. For instance, the ReLU activation function $y = max(0, x)$ has two different input ranges, the negative domain $x < 0$ and the positive domain $x \geq 0$, over which the function exhibits different linearities. The exact analysis separately considers these linearities whenever the input set to each neuron spans these two input ranges. There are related works such as Xiang et al. (2017); Tran et al. (2019a, 2020a)

The common strategy of the over-approximation analysis is to apply a conservative convex domain to linearize the ReLU function. Thus, when an input set spans both the positive and negative ranges, those different linearities does not need to be separately considered. Such over-approximation methods generate a single output reachable domain and are capable of efficiently analyzing large scale neural networks. There are related works such as AI2 Gehr et al. (2018), Abstract domain Singh et al. (2019b). But the approximation error is accumulated w.r.t. each neuron in an exponential manner, and the conservativeness of the final over

approximation can be extremely high and their effectiveness in verification will be undermined. One such example is presented in Figure 2.1. We can notice that these over-approximated domains show significantly high conservativeness.

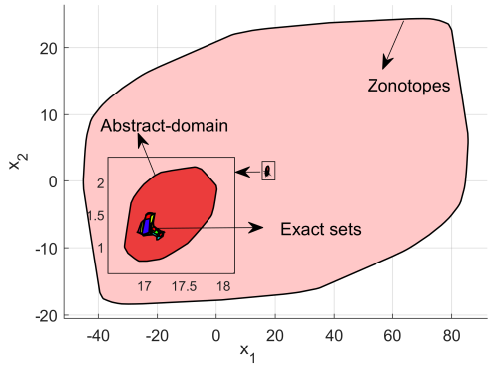


Figure 2.1: Conservativeness of output reachable domains of an ACAS Xu neural network computed by an abstract-domain-based method (over-approximation analysis) Singh et al. (2019b), a zonotope-based method (over-approximation analysis) Gehr et al. (2018), and our method (exact analysis).

2.3 SAT and SMT based Methods

In the work Katz et al. (2017), a method named Reluplex based on SMT solver is proposed to verify FFNNs. It extends the Simplex method to handle the piece-wise linear ReLU functions. To encode the ReLU node, the node is represented with two variables, where one is the input to the node named *backward-facing* variable and the other one is the output of the node named *forward-facing* variable. These two variables build the connection between ReLU layers with their preceding layer and following layer, respectively. Their compliance with the ReLU semantics is asserted in the solver. The pivot and update operations in the Simplex algorithm are included to handle the out-of-bound violations in variables. Thus, any broken ReLU connections can be fixed by updating the *backward-facing* or *forward-facing* variable. This method is sound and complete. The Marabou Katz et al. (2019); Elboher et al. (2020) method is an extension of Reluplex. It supports arbitrary piecewise-linear activation functions and parallel computation. In the work Ehlers (2017), a method named Planet based on SAT solver is proposed. This method combines the satisfiability solving and linear programming. An over-approximation approach is also included to filter out unnecessary search space to improve computational efficiency.

2.4 MILP based Methods

In the work Lomuscio and Maganti (2017), a method named NSVerify is proposed to verify FFNNs with ReLU neurons. NSVerify takes specifications with linear constraints and encodes the ReLU activation function by a set of mixed-integer linear constraints. It conducts *sound* and *complete* verification. Work Botoeva

et al. (2020) improves the computational efficiency of NSVerify by utilizing the dependency relation between ReLU nodes to prune search trees. Work Dutta et al. (2018) proposes a method named Sherlock to estimate the upper bound and lower bound of the output. The bounds are computed through local search and global search. The local search algorithm starts from a sample input points and utilizes the steepest projected gradient ascent or descent algorithm to search its adjacent points with their output towards the upper bound or lower bound. By encoding ReLU function with MILP, which is similar to NSVerify, the global search helps escape the local optima identified in the local search. For FFNNs with multiple outputs, this method is incomplete.

2.5 Symbolic Interval based Methods

ReluVal Wang et al. (2018b) is based on symbolic interval arithmetic to over approximate the output bounds of a DNN. The symbolic interval is utilized to preserve the dependency between ReLU nodes to the utmost extent when it propagates through layers, such that a less conservative approximation can be obtained. Neurify Wang et al. (2018a) which is an improved version of ReluVal introduces symbolic linear relaxation to achieve a tighter approximation, and also a *directed constraint refinement* algorithm to search space more efficiently. These methods iteratively refine the over-approximation of the output reachable sets by bisecting its input range, and they don't terminate until a counterexample is found, or the network is verified safe. Therefore, they are *sound* and *complete*.

CHAPTER 3

State-of-Art Works for Repair of Deep Neural Networks

3.1 Weight Modification with Singular Adversarial Example

Existing works that improve the safety and robustness of DNNs can be classified into two main categories. The first category relies on singular adversarial inputs to make specialized modifications on neural weights that likely cause misbehavior. In Sohn et al. (2019), the paper presents a technique named *Arachne*. There, given a set of finite adversarial inputs, with the guidance of a fitness function, *Arachne* searches and subsequently modifies neural weights that are likely related to unsafe behaviors. In Goldberger et al. (2020), the paper proposes a DNN verification-based method that modifies unsafe behaviors of DNNs by manipulating neural weights of the output layer. The correctness of the repaired DNN is then proved with a verification technique. In Usman et al. (2021), the repair approach first localizes the potential faulty neural weights at an intermediate layer or the last layer, and then conducts a small modification using constraint solving. In Majd et al. (2021), the method poses the repair problem as a mixed-integer quadratic program to adjust the parameter of a single layer, such that unsafe behaviors can be reduced and meanwhile the change in DNNs is minimized. However, these methods only enhance the robustness of DNNs for specific inputs and can hardly eliminate their whole unsafe domain, meaning that provably safe DNNs cannot be generated. In addition, the modification of weights based on individual adversarial examples may not capture the impact on the whole performance of the network. Slight changes in DNN parameters can result in introducing other unexpected unsafe behaviors.

3.2 Adversarial Training with Reachability Analysis

The second category of methods is based on adversarial training to update the DNN weights. Adversarial training methods such as Goodfellow et al. (2014); Madry et al. (2017) have shown that incorporating adversarial examples into the training process of a DNN can improve its adversarial robustness. However, these methods do not provide guarantees regarding the safety of a DNN. To solve this issue, some researchers incorporate reachability analysis in this process, such that they can train a model that is provably safe on a norm-bounded domain Wong and Kolter (2018); Mirman et al. (2018); Lin et al. (2020). Given a norm-bounded input range, these approaches over approximate the output reachable domain of a DNNs with one convex region. Then they minimize the worst-case loss over this region, which aims to migrate all unsafe outputs to the safe domain. The primary issue of these approaches is that the approximation error accumulates with each neuron during computation. For large input domains or complex DNNs, their approximated

reachable domain can be so conservative that a low-fidelity worst-case loss may result in significant accuracy degradation. We confirm this issue through experimental analysis in comparison with ART Lin et al. (2020). The method proposed in Leino et al. (2021) approaches the problem from a different perspective. Instead of adjusting the weights of the network, the method appends a new layer to the DNN which checks and repairs violating outputs. However, this method is limited to classification networks and the added layer may increase the run time overhead of the network.

CHAPTER 4

Reachability Analysis of Deep ReLU Neural Networks using Facet-Vertex Incidence

4.1 Preliminaries

4.1.1 Neural Network Verification

A DNN consists of one input layer, multiple hidden layers, and one output layer. Each layer contains multiple neurons which are interconnected with neurons in the next layer by weights and bias in a feed-forward way. The output of each neuron is associated with three components: its input weight ω, input bias b and the activation function f, namely:

$$y_i = f(\sum_{j=1}^{n} \omega_{i,j} x_j + b_i)$$

where ω_{ij} and b_j are respectively the weight and bias from the jth neuron of the preceding layer to the ith neuron of the current layer, and x_j is an input to this neuron and also the output of jth neuron in the preceding layer, and y_i is the output of this neuron. In this paper, we consider networks that contain ReLU activation functions which are defined as $\mathrm{E}(x) = max(0, x)$. Let $W_{(l,l\text{-}1)}$, b_l denote the weight matrix, the bias vector between the $(l\text{-}1)$th layer and lth layer, and x_l be its input, then the output of the lth layer will be Equation 4.1. For the first hidden layer, its input x_1 is equal to the input to the network x_0. The output of one layer is also an input of the next layer. Therefore, given an input x_0, the output y_l of the lth layer will be Equation 4.2.

$$\mathrm{L}(x_l) = \mathrm{E}(W_{(l,l\text{-}1)} \cdot x_l + b_l) \tag{4.1}$$

$$y_l = (\mathrm{L}_l \circ \mathrm{L}_{l-1} \circ \cdots \circ \mathrm{L}_1)(x_0) \tag{4.2}$$

Definition 4.1.1 (Reachable Sets and Reachable Domain of Neural Networks) *Given a neural network N and an input set $\mathcal{I} \subset \mathbb{R}^{|x|}$, an output reachable set $\mathcal{P} \subset \mathbb{R}^{|y|}$ refers to a convex polytope where $\forall y \in \mathcal{P}, \exists x \in \mathcal{I}$ and $y = N(x)$. The computation of reachable sets is $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\} = Reach(N, \mathcal{I})$. The output reachable domain $\mathcal{O}$ is the union of all the computed reachable sets $\bigcup_{i=1}^{k} \mathcal{P}_i$ where $\forall x \in \mathcal{I}, y = N(x)$ and $y \in \mathcal{O}$ besides the condition for reachable sets.*

An input set to one layer is denoted as $\mathcal{S}$. We denote the linear transformation by W and b on $\mathcal{S}$ as a function $\mathbb{T}(\mathcal{S})$ and the following operation from each ReLU neurons in the layer as $\mathbb{E}(\cdot)$. Overall, the whole process of the layer w.r.t. the input set $\mathcal{S}$ can be denoted as

$$\mathbb{L}(\mathcal{S}) = (\mathbb{E}_k \circ \mathbb{E}_{k-1} \circ \cdots \circ \mathbb{E}_1 \circ \mathbb{T})(\mathcal{S}) \tag{4.3}$$

The function $\mathbb{T}(\cdot)$ outputs one affine transformed set $\mathcal{S}'$ w.r.t. the input set $\mathcal{S}$. While each $\mathbb{E}(\cdot)$ yields at most 2 reachable sets for one input set because of the different linearities of the ReLU function w.r.t. its input range. Finally, the output reachable sets of the current layer are passed on as an input set to the next layer.

The reachable sets are also associated with the *linear region*s of neural networks Montufar et al. (2014); Serra et al. (2017); Hanin and Rolnick (2019). A *linear region* of a piecewise linear function $f : \mathbb{R}^{|x|} \rightarrow \mathbb{R}^{|y|}$ refers to a maximum convex subset of an input set in $\mathbb{R}^{|x|}$, on which the function f is linear. As described above, a set is either affine transformed by weights and bias, or divided into subsets to match the different linearities of ReLU functions in each neuron. Thus, the DNN exhibits unique linearity for each output reachable set over a subspace of the input set. Those subspaces are *linear region*s. Our method takes advantage of this property in the backtracking process, where it tracks back to the input space given a particular output reachable set. The details will be introduced in Section 4.4.

The verification of a neural network using reachability analysis is defined as follows.

Definition 4.1.2 (Verification of Neural Networks) *Given a neural network N, an input set $\mathcal{I} \subset \mathbb{R}^{|x|}$ and an unsafe domain $\mathcal{U} \subset \mathbb{R}^{|y|}$. The verification problem is to determine whether $(\bigcup Reach(N, \mathcal{I})) \cap \mathcal{U} = \emptyset$.*

4.1.2 Facet-Vertex Incidence Matrix

A convex polytope is a convex set $\mathcal{S} \in \mathbb{R}^d$ and can be represented by the vertex representation (V-rep) or the half-space representation (H-rep). The V-rep represents $\mathcal{S}$ as a finite number of extreme points (vertices), whereas the H-rep represents $\mathcal{S}$ as the intersection of a set of closed halfspaces (in terms of linear inequalities). In Xiang et al. (2017), the reachability analysis of neural networks utilizes both the V-rep and the H-rep to compute reachable sets. The V-rep of $\mathcal{S}$ is utilized to conduct affine transformations between layers, and the H-rep is utilized for the computation of new subsets while considering the linear constraints from the input range in ReLU neurons. The iterative process, which requires the back-and-forth conversion between representations, is called the enumeration problem in computational geometry and has a high computational complexity. In Tran et al. (2019c); Bak et al. (2020), the authors use the Star set based on the H-rep and can avoid the conversion issue. However, the computation for each neuron involves a large number of LP problems, which has a significant impact on its efficiency. Other works Singh et al. (2018a); Gehr et al. (2018) utilize zonotope representations, which are centrally symmetric polytope representations. The issue with these approaches is that with a large input set, the over-approximation in every layer is overly conservative, and this increases exponentially with every layer.

In this work, we utilize a facet-vertex incidence matrix to encode the combinatorial structure of a polytope. The FVIM enables quick operations over convex polytopes in the reachable-set computation. This enables fast affine transformations and can be used to derive vertex adjacency for fast division by hyperplanes. In the

rest of the section, we provide a formal definition of the FVIM. For details, see Henk et al. (2004); Grünbaum (2013).

In order to define a Facet-Vertex Incidence Matrix, we first need to define the notion of a supporting hyperplane and face of a polytope.

Definition 4.1.3 (Supporting Hyperplane) *A hyperplane $\mathcal{H}$ denoted by $a^\top x = b$ is a supporting hyperplane of polytope $\mathcal{S}$ if one of its closed halfspaces, $a^\top x \leq b$ or $a^\top x \geq b$ contains $\mathcal{S}$, and meanwhile, the intersection of $\mathcal{S}$ with the hyperplane is not empty.*

Definition 4.1.4 (Face & Facet) *The face of a d-dimensional polytope $\mathcal{S}$ is an intersection of $\mathcal{S}$ with a supporting hyperplane. When the dimension of $\mathrm{aff}(\mathcal{S} \cap \mathcal{H})$ is k, the face is denoted as k-$face$. The function $\mathrm{aff}(\mathcal{S})$ indicates the affine hull of $\mathcal{S}$, which is the smallest affine set that contains $\mathcal{S}$. $\mathcal{S}$ contains $\{0$-$face$, 1-$face$,..., $(d-1)$-$face\}$ where the 0-$face$ is named vertex and the $(d-1)$-$face$ is named facet. The cardinality of the set of all k-$faces$ is denoted as $f_k(\mathcal{S})$.*

Definition 4.1.5 (Facet-Vertex Incidence Matrix) *Given a full-dimensional polytope $\mathcal{S} \in \mathbb{R}^d$, the facet-vertex Incidence matrix is a matrix $M \in \{0,1\}^{f_{d-1}(\mathcal{S}) \times f_0(\mathcal{S})}$ where an entry $M(f,v) = 1$ indicates the facet f contains the vertex v, and an entry $M(f,v) = 0$ indicates that it does not.*

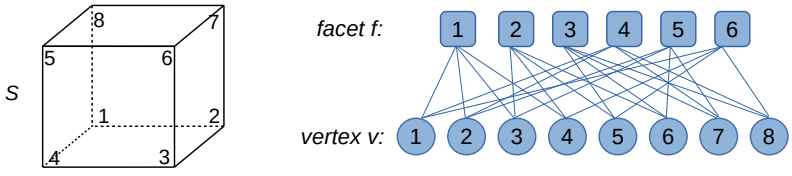


Figure 4.1: A 3-dimensional cube $\mathcal{S}$ and the containment relation between facets and vertices.

Consider a 3-dimensional cube as illustrated in Figure 4.1. The graph with blue blocks describes the containment relation between the facets and vertices and is equivalent to the FVIM shown in Equation 4.5. In FVIM, f denotes the facet/plane and v denotes the vertex of the cube. For instance, f_1 is the 2-dimensional face: $plane_{1\text{-}2\text{-}3\text{-}4}$. We define a polytope using its FVIM $\mathcal{F}$ and vertices V as follows:

$$\mathcal{S} = \langle \mathcal{F}, V \rangle \tag{4.4}$$

$$\text{FVIM} = \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{array} \begin{array}{c} \begin{array}{cccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \end{array} \\ \left(\begin{array}{cccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right) \end{array} \tag{4.5}$$

4.2 Operations on Polytopes

Our method relies on two basic operations for processing polytopes through a neural network. This includes *affine transformation* by weights and *division* by a hyperplane $\mathcal{H}$. In the following, we present each operation in detail.

4.2.1 Affine Transformation

An affine transformation can map a polytope to a higher-dimensional or lower-dimensional space. A polytope $\mathcal{S}$ is called full-dimensional if it is a d-dimensional object in $\mathbb{R}^d$. Otherwise, if $\mathcal{S}$ is in $\mathbb{R}^{n>d}$, then it is not full-dimensional.

Remark 4.2.1 *Polytopes that differ only by an affine transformation are combinatorially equivalent. The main reason is that an affine transformation only changes vertices but preserves polytopes' combinatorial structure Henk et al. (2004); Grünbaum (2013).*

Therefore, the FVIM remains unchanged under affine transformation, regardless if it is on a higher or lower dimensional space. Let W, b denote the weights and bias of an affine transformation. Then the computation of FVIM $\mathcal{F}'$ and vertices V' of the output polytope $\mathcal{S}'$, under an affine transformation, can be formulated as in Equation 4.6. This process computes the exact affine transformed polytope as represented by the function $\mathbb{T}$ in Equation 4.3.

$$\mathcal{F} = \mathcal{F}',\ V' = WV + b \tag{4.6}$$

4.2.2 Division by Hyperlanes

Another common operation when processing polytopes is the *divison* by hyperplanes. A hyperplane $\mathcal{H}$ divides the input range of a ReLU function into two sub-ranges over which the function exhibits different linearities. *Division* is only considered when the set $\mathcal{S}$ intersects $\mathcal{H}$. As the polytope representation has vertices V of $\mathcal{S}$, the determination whether an intersection occurs can be made by checking whether the distribution of vertices V is on one side of a hyperplane $\mathcal{H}$. For example, let $\mathcal{H}$ be $a^\top x + b = 0$. To compute

the distribution of vertices, we can substitute x with V. Formally, we define the set of *positive* vertices as $V^+ = \{v \in V | a^\top v + b > 0\}$ and *negative* vertices as $V^- = \{v \in V | a^\top v + b < 0\}$. Here, we make an assumption that no vertices are located on the hyperplane $\mathcal{H}$. In practice, this is an extremely unlikely occurrence due to floating point computation. When computing a division by hyperplane $\mathcal{H}$, one of the following three cases may occur:

(1) Both V^+ and V^- are non-empty. In this case, the hyperplane $\mathcal{H}$ intersects with $\mathcal{S}$. The polytope $\mathcal{S}$ will be divided by $\mathcal{H}$ into two non-empty polytopes $\mathcal{S}_{sub}^{+\mathcal{H}}$ and $\mathcal{S}_{sub}^{-\mathcal{H}}$. The *positive polytope* w.r.t. $\mathcal{H}$, $\mathcal{S}_{sub}^{+\mathcal{H}}$ is defined as $\forall x \in \mathcal{S}_{sub}^{+\mathcal{H}},\ a^\top x + b \geq 0$ On the other hand the *negative polytope* w.r.t. $\mathcal{H}$ is defined as $\forall x \in \mathcal{S}_{sub}^{-\mathcal{H}},\ a^\top x + b \leq 0$.

(2) Only set V^+ in non-empty. In this case, polytope $\mathcal{S}$ is on the positive closed halfspace of $\mathcal{H}$ and we have $\mathcal{S}^{+\mathcal{H}}$.

(3) Only set V^- in non-empty. In this case, polytope $\mathcal{S}$ is on the negative closed halfspace of $\mathcal{H}$ and we have $\mathcal{S}^{-\mathcal{H}}$.

4.2.2.1 Identification of New Facets

When a hyperplane $\mathcal{H}$ intersects with $\mathcal{S}$ and we have both $\mathcal{S}_{sub}^{+\mathcal{H}}$ and $\mathcal{S}_{sub}^{-\mathcal{H}}$, the intersection creates one new common facet with multiple new vertices for both subsets. The derivation of the facet is as follows. In Henk et al. (2004); Grünbaum (2013), the authors show that an intersection of a polytope with an affine subspace (hyperplane) is a polytope. Therefore, the intersection generates one polytope $\mathcal{S}_I$ where $\mathcal{S}_I \subset \mathcal{S}_{sub}^{+\mathcal{H}}$ and $\mathcal{S}_I \subset \mathcal{S}_{sub}^{-\mathcal{H}}$. In the following, we show that $\mathcal{S}_I$ is a new facet.

Theorem 1 *Given a d-dimensional polytope $\mathcal{S}$ located in $\mathbb{R}^{n \geq d}$ space, and a hyperplane $\mathcal{H} \subset \mathbb{R}^{n-1}$ where $\mathcal{S} \nsubseteq \mathcal{H}$, then the intersection of $\mathcal{S}$ with $\mathcal{H}$ is a one $(d$-$1)$-dimensional polytope $\mathcal{S}_I$. $\mathcal{S}_I$ is also a common facet of subsets $\mathcal{S}_{sub}^{+\mathcal{H}}$ and $\mathcal{S}_{sub}^{-\mathcal{H}}$.*

Proof. In dimension theory Hurewicz and Wallman (2015), given a vector space V, and U,W, two subspaces of V, we have that

$$\dim(U + W) = \dim(U) + \dim(W) - \dim(U \cap W)$$

where $dim(\cdot)$ returns the dimension of the object. Since $\mathcal{S} \nsubseteq \mathcal{H}$, we have $\dim(\mathcal{S} + \mathcal{H}) = n$. By substituting U,W with $\mathcal{S}$,$\mathcal{H}$, we have that

$$\dim(\mathcal{S} \cap \mathcal{H}) = \dim(\mathcal{S}) + \dim(\mathcal{H}) - \dim(\mathcal{S} + \mathcal{H}) = d - 1$$

Therefore $\mathcal{S}_I$ is $(d$-1$)$-dimensional polytope. Here, we use a result from polytope theory which states that a face of polytope is equivalent to a polytope Henk et al. (2004). We can conclude that a polytope $\mathcal{S}_I$ generated from the intersection is a common facet of both subsets.

4.2.2.2 Identification of New Vertices

When a hyperplane $\mathcal{H}$ intersects with $\mathcal{S}$, new vertices are also generated. To compute these new vertices, the edges of $\mathcal{S}$ first need to be identified from the FVIM. For a regular d-dimensional polytope, if $d < 4$, then two vertices are adjacent and form an edge if and only if they are contained by at least $(d-1)$ common facets. But this criterion does not apply to higher-dimensional polytopes where vertices may be nonadjacent under the same condition.

However, with the assumption above that no vertices of $\mathcal{S}$ lie on the hyperplane, the criterion becomes suitable for the higher-dimensional polytopes in our reachability analysis. First, we can show that all the polytopes in our analysis are *simple polytope*s.

Definition 4.2.1 (Simple polytope) *A d-dimensional polytope is a* ***simple polytope*** *if each vertex is contained in exactly* d *facets.*

Theorem 2 *Given a d-dimensional simple polytope* $\mathcal{S}$ *and a hyperplane* $\mathcal{H}$ *which intersects with* $\mathcal{S}$ *without any vertices of* $\mathcal{S}$ *locating on* $\mathcal{H}$*, then the two subsets from the division are simple polytopes*

Proof. Suppose there are m edges $e \subset \mathcal{S}$ intersecting with $\mathcal{H}$. Then, m new vertices v are generated respectively with $v \in e$ and $v \in \mathcal{H}$. Since $\mathcal{S}$ is a *simple polytope*, each edge is contained in exactly $(d$-1$)$ facets. Then, each new-generated vertex v is also contained in $(d$-1$)$ facets of $\mathcal{S}$. As Theorem 1 states, the intersection creates a common facet for both subsets. Then each v is also contained by this facet. Therefore, all vertices in two subsets are adjacent to exactly d facets,and the subsets are *simple polytope*s.

The infinity-norm bounded input set is a hyperrectangle. The hyperrectangle is a *simple polytope* which is closed under affine-transformation Henk et al. (2004). According to Theorem 2, the subsets from the division of a *simple polytope* are also *simple polytope*s. In terms of the polytope theory, two vertices of a d-dimensional *simple polytope* are adjacent and form an edge if they are contained in exactly $(d$-1$)$ common facets. Suppose a pair of vertices v_1 and v_2 form an edge e of $\mathcal{S}$, the new vertex v generated from $e \cap \mathcal{H}$ can be computed by Equation 4.7 where λ is a derivable scalar.

$$\begin{cases} v = v_1 + \lambda(v_2 - v_1) \\ a^\top v + b = 0 \end{cases} \tag{4.7}$$

4.2.2.3 Derivation of Two Subsets

After identifying the new facet and vertices generated from the intersection, the next step is dividing the original polytope $\mathcal{S}$ into two subsets according to the containment relation between facets and vertices. An example of a division of the 3-dimensional cube is shown in Figure 4.2. The intersection of $\mathcal{S}$ with $\mathcal{H}$ is the new facet $plane_{9\text{-}10\text{-}11\text{-}12}$ which is denoted as facet $\{7\}$ in the graph. *Negative vertices* $\{1, 2, 3, 4\}$ and *positive vertices* $\{5, 6, 7, 8\}$ w.r.t. $\mathcal{H}$ are first identified. As $\mathcal{S}$ is a simple polytope, vertices that are contained in exactly 2 common facets are adjacent and form an edge of $\mathcal{S}$. Thus, the edges $e_{1\text{-}8}$, $e_{2\text{-}7}$, $e_{3\text{-}6}$ and $e_{4\text{-}5}$ are identified intersecting with $\mathcal{H}$ and generate new vertices v_9, v_{10}, v_{11} and v_{12}. Their values can be computed according to Equation 4.7. Their containment relation with facets is inherited from the relation between the edges with facets. For instance, v_9 is from the intersection of $e_{1\text{-}8}$ with $\mathcal{H}$, and $e_{1\text{-}8}$ are contained in facets $plane_{1\text{-}4\text{-}5\text{-}8}$ and $plane_{1\text{-}2\text{-}7\text{-}8}$. Therefore, v_9 is also contained in these two facets. This inheritance relation is described by the red-line connection between Facet $\{4, 6\}$ and Vertex $\{9\}$ in the graph. Finally, the graph is split in terms of $positive$ and $negative$ vertices as well as the new vertices, generating two compact and exact FVIMs for the subsets and completing the division.

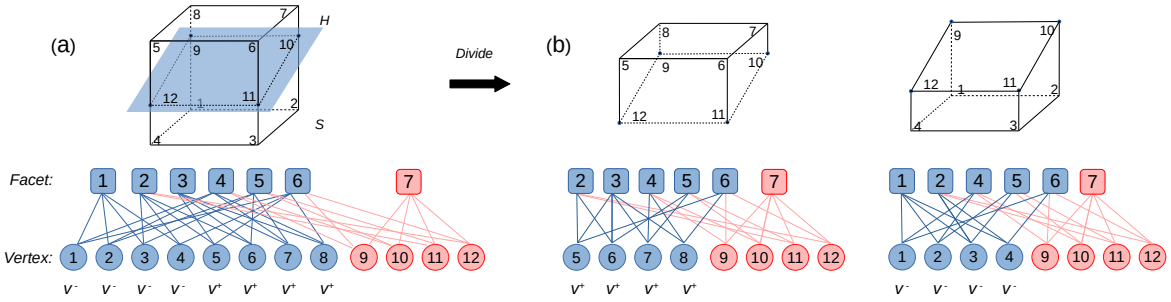


Figure 4.2: Division of a 3-dimensional cube $\mathcal{S}$ by a hyperplane $\mathcal{H}$. The graph which describes the containment relation between facets and vertices is equivalent with the FVIM. The blue blocks denote the faces of $\mathcal{S}$ and the red blocks denote the new faces generated from the intersection.

4.3 Reachability Analysis with FVIM

In the previous section, we described the *affine transformation* and *division* operations, two basic operations for reachability analysis with FVIM. In this section, we present the application of these operations in one layer of the neural network and then extend them to the entire network. The process is illustrated in Figure 4.3. In each layer, first we apply an affine transformation to the input set $\mathcal{S}$ by the weights and bias (❶). Then, we apply a *divison* operation on the resulting polytope to process it through the ReLU activation function (❷).

Suppose a neuron layer contains k neurons, and let $x \in \mathbb{R}^k$ be an input. In the i^{th} neuron, the input range of its ReLU function is divided by $x_i = 0$ into two domains over which the function exhibits different linearities. The $x_i = 0$ can also be formulated as a hyperplane $\mathcal{H}_i : a^\top x = 0$ where $a_i = 1$ and the rest are

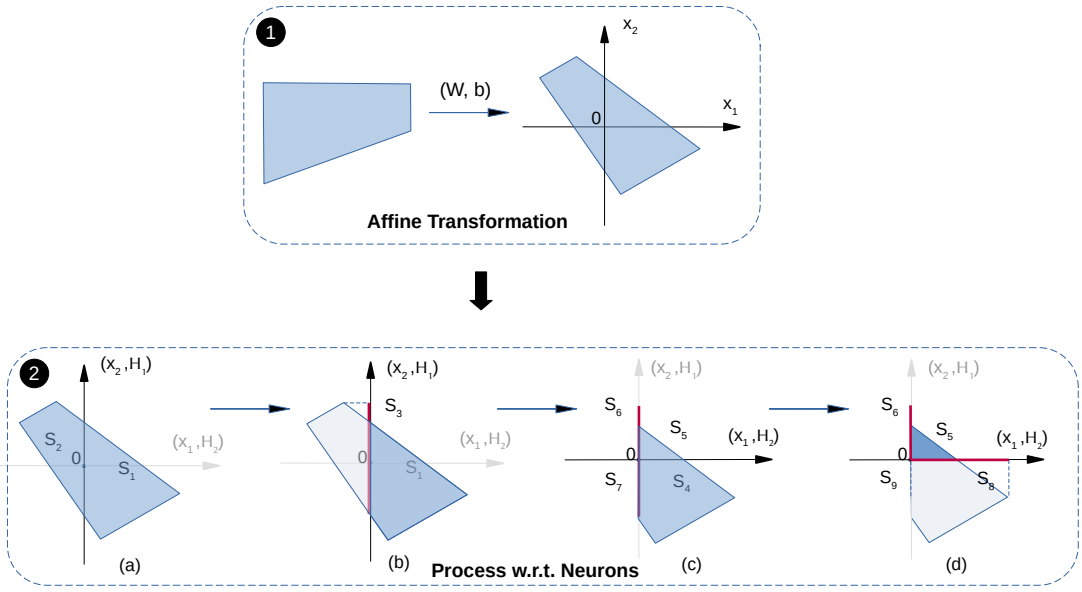


Figure 4.3: A simple layer that contains 2 neurons to demonstrate the processing of an input polytope

zeros. When the polytope $\mathcal{S}$ spans two input domains of this ReLU neuron which indicates that it intersects with $\mathcal{H}_i$, this polytope will be divided by $\mathcal{H}_i$ according to the *division* operation. Then two subsets $\mathcal{S}_{sub}^{-\mathcal{H}_i}$ and $\mathcal{S}_{sub}^{+\mathcal{H}_i}$ are generated. For the output *negative* subset that is located in the closed halfspace $a^\top x < 0$, the x_i dimension of all points belonging to it is set to zero in terms of the property of the ReLU function. This process is equivalent to a linear transformation and can be implemented with Equation 4.6. Then by sequentially processing the affine transformed set w.r.t. each neuron, we can consider all combinations of linearities, as formulated in Equation 4.3.

An example is demonstrated in Figure 4.2 (❷). The layer contains 2 ReLU neurons. First, $\mathbb{E}_1(\cdot)$ is applied to the input polytope as shown in (a). The polytope is divided by the hyperplane $\mathcal{H}_1$ into two polytopes $\mathcal{S}_1$ and $\mathcal{S}_2$ which are $\mathcal{S}_{sub}^{+\mathcal{H}_1}$ and $\mathcal{S}_{sub}^{-\mathcal{H}_1}$. Then the x_1 dimension of all element points in $\mathcal{S}_{sub}^{-\mathcal{H}_1}$ is set to zero, generating a new polytope $\mathcal{S}_3$ (red solid line) in (b). Then the output $\mathcal{S}_1$ and $\mathcal{S}_3$ will be the inputs to the function $\mathbb{E}_2(\cdot)$. As shown in (c). $\mathcal{S}_1$ and $\mathcal{S}_3$ are divided by $\mathcal{H}_2$ into $\{\mathcal{S}_4, \mathcal{S}_5\}$ and $\{\mathcal{S}_6, \mathcal{S}_7\}$, respectively. $\mathcal{S}_4$ and $\mathcal{S}_7$ are in $\mathcal{S}_{sub}^{-\mathcal{H}_2}$ and the x_2 dimension of points is set to zero as shown in (d), creating $\mathcal{S}_8$ and $\mathcal{S}_9$ (origin), respectively. Finally the output polytopes of this layer w.r.t. $\mathcal{S}$ are $\{\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8, \mathcal{S}_9\}$, and the whole process can be formulated as

$$\{\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8, \mathcal{S}_9\} = (\mathbb{E}_2 \circ \mathbb{E}_1 \circ \mathbb{T})(\mathcal{S})$$

The output polytopes of the current layer will be the inputs to the next layer. Given an input set $\mathcal{I}$, and

number of layers l, the computation of reachable sets can be defined as:

$$\mathcal{O} = (\mathbb{L}_l \circ \cdots \circ \mathbb{L}_2 \circ \mathbb{L}_1)(\mathcal{I}) \tag{4.8}$$

The algorithm for the computation of reachable sets is shown in Algorithm 1. Its implementation is highly parallelizable because (1) the input sets to each layer are independent in the *for* loop in Line 5, (2) the processing of each set w.r.t. one neuron is also independent in the *for* loop in Line 14. The details of functions are as following:

(1) The **Reach**() function corresponds to Equation 4.8. Given an input set to the neural network, it computes all the reachable sets.

(2) The **singleLayer**() function computes reachable sets of the target layer given an input set.

(3) The **affineTransform**() function corresponds to Equation 4.6. It computes the affine transformation on the input set with the weight matrix and bias vector between layers.

(4) The **Divide**() function corresponds to $\mathbb{E}(\cdot)$, which is illustrated in Figure 4.2. Here, we only present one case where $\mathcal{S}$ intersects with the hyperplane.

(5) The **Project**() function is used to set the target dimension of the elements of a polytope to zeros, which is equivalent to a linear transformation.

Given an input set to the neural network that has n ReLU neurons, the maximum number of reachable sets is bounded by 2^n. This number is essentially associated with the volume of the input set, where a larger volume increase the likelihood of its intersecting with hyperplanes imposed by neurons. For instance, in the experimental evaluation, the number of reachable sets for the ACAS Xu N_{27} varies depending on the input sets. For property 1 with a large input set, it computes over six hundred thousand reachable sets, while for property 3 which has a smaller input set, it outputs around five hundred reachable sets.

4.4 Backtracking

Given an input set, the algorithm introduced in the previous section can compute the exact reachable sets for neural networks. However, our method also supports *backtracking*, which is the process of computing the corresponding input subspace given a subset of the output domain. This is particularly useful in verification, since we can use this process to find input areas which are associated with possible violations. We track the connection between the polytopes processed in layers and their corresponding *linear regions* in the input space. Here, we revise the representation of polytopes in Equation 4.4 by introducing a matrix M and a

Algorithm 1 Reachable set computation of a neural network

Input: $\mathcal{I}$ # an input set to the neural network
Output: $\mathcal{O}$ # exact output reachable domain of the network

procedure $\mathcal{O}$ = REACH($\mathcal{I}$)
  $\mathcal{I}^{[k]} = \mathcal{I}$ # $\mathcal{I}^{[k]}$: input sets to the kth layer
  for $k = 1 : layers$ **do** # *layers: the number of layers*
    $\mathcal{O}^{[k]} = empty$ # $\mathcal{O}^{[k]}$: output reachable sets of the kth layer
    for $\mathcal{S}$ in $\mathcal{I}_k$ **do**
      $\mathcal{S}^{[k]}$ = singleLayer(k, $\mathcal{S}$) # $\mathcal{S}^{[k]}$: output reachable sets for one input set $\mathcal{S}$
      Add $\mathcal{S}^{[k]}$ to $\mathcal{O}^{[k]}$
    $\mathcal{I}^{[k]} = \mathcal{O}^{[k]}$
  return $\mathcal{O} = \mathcal{O}^{[k]}$
procedure $\mathcal{S}^{[k]}$ = SINGLELAYER(k, $\mathcal{S}$)
  $\mathcal{S}^{[k]}$ = affineTransform(k, $\mathcal{S}$)
  for i = $1 : m$ **do** # m: the number of neurons
    $list = empty$
    for $\mathcal{S}$ in $\mathcal{S}^{[k]}$ **do**
      $\mathcal{S}^+, \mathcal{S}^-$ = Divide($\mathcal{H}_i$, $\mathcal{S}$)
      $\mathcal{S}^-$ = Project(i, $\mathcal{S}^-$)
      Add $\mathcal{S}^+, \mathcal{S}^-$ to $list$
    $\mathcal{S}^{[k]} = list$
  return $\mathcal{S}^{[k]}$

vector d that describe an affine transformation:

$$\mathcal{S} = \langle \mathcal{F}_i, V_i, M, d \rangle \tag{4.9}$$

$\mathcal{F}_i$ and V_i represents the FVIM and vertices of a convex subset of the input space, respectively. The vertices of $\mathcal{S}$ can be computed with V_i by $V = MV_i + d$. This linear relation between V_i and V indicates that the input convex subset is the *linear region* for $\mathcal{S}$. Matrix M and d actually describe a linear relation between $\mathcal{S}$ and its linear region.

With the representation of polytopes as a tuple $\langle \mathcal{F}_i, V_i, M, d \rangle$, the *affine transformation* in Equation 4.6 will be slightly different. As shown in Equation 4.10, the new operation on polytopes will only update M and d instead and preserve V_i and $\mathcal{F}_i$. Hence, the *linear region* for $\mathcal{S}$ can be tracked. The updated polytope $\langle \mathcal{F}_i', V_i', M', d' \rangle$ is calculated as follows:

$$\mathcal{F}_i' = \mathcal{F}_i, \quad V_i' = V_i, \quad M' = WM, \quad d' = Wd + b \tag{4.10}$$

which is equivalent to updating the linear relation between $\mathcal{S}$ and its linear region.

With the new set representation, we also have a new *division* operation $\mathbb{E}(\cdot)$. The checking of vertices

distribution of $\mathcal{S}$ w.r.t. a hyperplane $\mathcal{H}$ is by computing the vertices of $\mathcal{S}$ with $V = MV_i + d$. When conducting a *division* operation on $\mathcal{S}$ with a hyperplane $\mathcal{H} : a^\top x + c = 0$, $\mathcal{H}$ will be first mapped back into the input space through M and d, generating a new hyperplane $\mathcal{H}' : a^\top M + ad + c = 0$. Then, we utilize this hyperplane to conduct the *division* operation on the input subset represented by $\mathcal{F}_i$ and V_i.

4.4.1 Identification of Unsafe Input Subspace

Let the output unsafe domain be $\mathcal{U} : \{y \,|Ay + v \leq 0\}$. Suppose we have an output reachable set of a network $\mathcal{P}$=$\langle \mathcal{F}_i, V_i, M, d \rangle$. Then, we conduct *division* operations on $\mathcal{P}$ sequentially with each hyperplane from $\mathcal{U}$. If there exists a subset of $\mathcal{P}$ locating in $\mathcal{U}$, it indicates the neural network is unsafe. Otherwise, it is safe. In each *division*, the hyperplane is obtained from one of the linear inequalities of $\mathcal{U}$. One linear inequality $a^\top y + v \leq 0$ is essentially one closed halfspace of a hyperplane $\mathcal{H} : a^\top y + v = 0$. The $\mathcal{P}_{sub}^{-\mathcal{H}}$ generated from the *division* with $\mathcal{H}$ contains all element points that satisfy this linear inequality. Then $\mathcal{P}_{sub}^{-\mathcal{H}}$ is the set to the *division* with the next hyperplane. We can obtain the final unsafe set after sequentially considering all linear inequalities of $\mathcal{U}$.

4.5 Evaluation

4.5.1 Safety Verification of ACAS Xu Networks

In this section, we evaluate our method against the Marabou Katz et al. (2019), NNV-exact Tran et al. (2020b), nnenum Bak et al. (2020), Venus Botoeva et al. (2020) and Neurify Wang et al. (2018a) methods, which conduct a sound and complete verification. We also include ERAN Singh et al. (2018a), which is an over-approximation method. We include it to demonstrate that performance-wise, over-approximation may not provide a significant improvement in efficiency and effectiveness as opposed to our exact method.

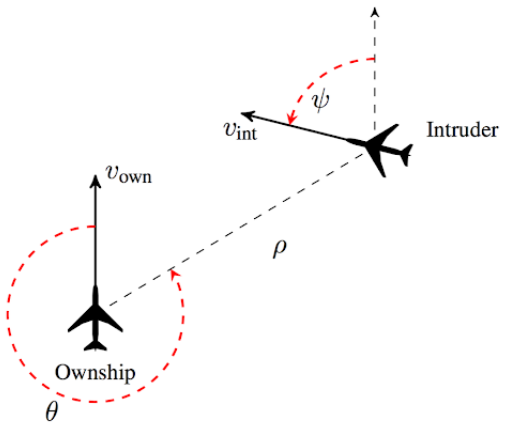


Figure 4.4: Airborne Collision Avoidance System

We verify safety of the DNN benchmarks proposed in Julian et al. (2016) for the Airborne Collision System X Unmanned (ACAS Xu). These benchmarks have been widely tested by the verification community Katz et al. (2019); Tran et al. (2020b, 2019c); Bak et al. (2020); Wang et al. (2018a). The ACAS Xu networks are used to approximate a large lookup table that converts sensor measurements into maneuver advisories in an Airborne Collision Avoidance System as shown in Figure 4.4 Katz et al. (2017), such that the massive memory occupation by the table and the lookup time can be reduced. The set of networks contain 45 fully-connected DNNs for different combinations of discretized parameters. Each one has five inputs, five outputs, and six hidden layers. Each layer consists of 50 ReLU neurons. There are 300 hidden neurons total in each network. The five inputs consist of the sensor measurements: (1) ρ: Distance from ownship to intruder; (2) θ: Angle to intruder relative to ownship heading direction; (3) ψ: Heading angle of intruder relative to ownship heading direction; (4) v_{own}: Speed of ownship; (5) v_{int}: Speed of intruder. The outputs are advisory scores for five different actions, where the lowest score corresponds to the best action. The five actions are respectively, clear of conflict (COC), weak right, strong right, weak left, and strong left. There are ten safety properties 1-10 which specify input bounds $\mathcal{I}$ and unsafe domains $\mathcal{U}$ in the output. The details can be found in the Appendix of Katz et al. (2017).

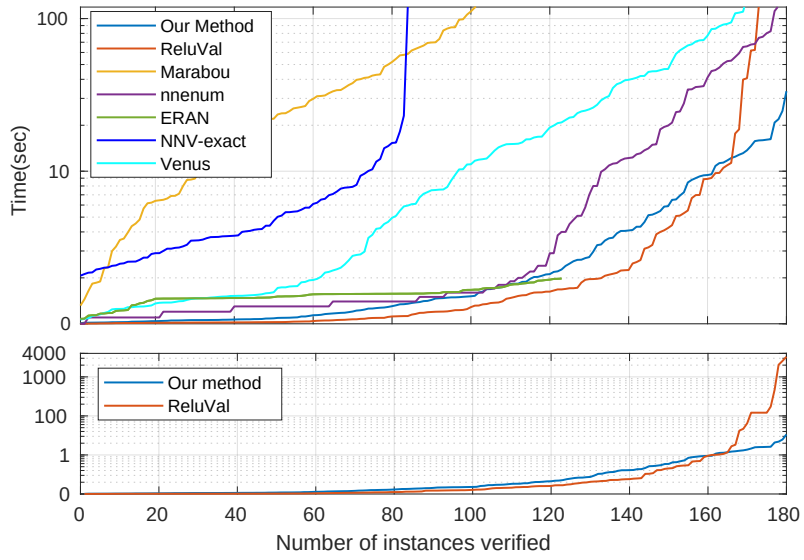


Figure 4.5: Test on ACAS Xu Property 1-4 with timeout. The timeout for each instance is set to 120 seconds. ERAN fails to complete all instances. The figure below is without timeout.

We tested all 45 networks for properties 1, 2, 3 and 4 with 180 instances in total. The hardware configuration is Intel Core i9-10900K CPU @3.7GHz×, 10-core and 20-thread Processor, 128GB Memory, 64-bit Ubuntu 18.04. The results as a cactus plot are shown in Figure 4.5. Our method is the only method that can verify all the 180 instances without timing out (120s). Although Neurify Wang et al. (2018a) is faster than

Property	Network	Result	Our Method	nnenum	NNV-Exact	Neurify	ERAN	Venus
5	N_{11}	Safe	1.7	4.3	84.1	4.8	-	174.6
6.1	N_{11}	Safe	4.7	20.9	Timeout	1.2	6.0	325.7
6.2	N_{11}	Safe	5.6	24.1	Timeout	0.5	-	402.4
7	N_{19}	Unsafe	1757.0	3730.5	Timeout	344.5	-	5298.0
8	N_{29}	Unsafe	5.2	0.1	Timeout	24.4	-	0.7
9	N_{33}	Safe	6.9	26.0	Timeout	146.2	-	926.8
10	N_{45}	Safe	1.1	4.0	Timeout	0.4	10.3	78.4

Table 4.1: Running time (*sec*) and results of verification on properties 5-10 with 1hr timeout.

our methods on some instances by a few seconds, there are instances where it performs significantly worse. In some instances, Neurify takes up to one hour. For properties 5-10, we select an individual network that is commonly used by other publications. The result is shown in Table 4.1. Marabou is not included since their published code does not support disjunctive conditions for the input domain. In addition, our approach can also identify the subspace of the input space that leads to the safety violation.

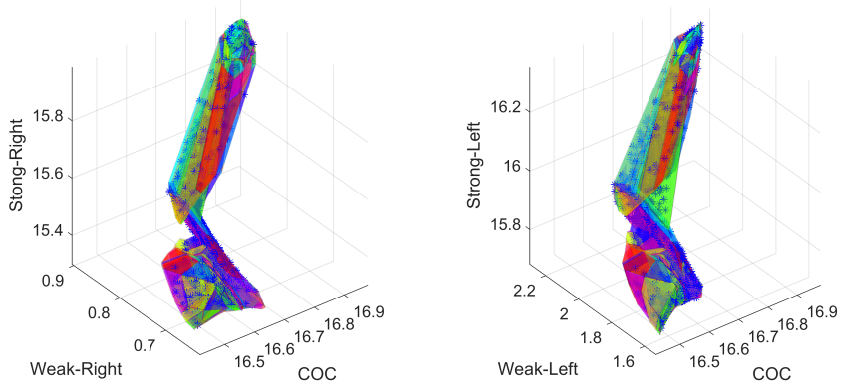


Figure 4.6: Exact reachable set of the network N_{47} on Property 4. The blue star points are the output w.r.t. 2000 random input samples. They all locate on or inside the output sets.

An example of the exact reachable set of the network N_{47} is demonstrated in Figure 4.6, whose lower bound and upper bound of the input are $[1500, \text{-}0.06, 3.1, 1000, 700]$ and $[1800, 0.06, 3.14, 1200, 800]$. 5000 random inputs are sampled to demonstrate the method's correctness. The reachable sets are visualized separately in two figures. With such output sets, we can verify different safety properties. Besides, our method can also compute the exact input sets that lead to property violations. For instance, the inputs that make the network N_{12} violate Property 2 are computed as shown in Figure 4.7. It is a union of 104 polytopes. The element belonging to these polytopes will yield property violations. For the element that doesn't belong to them but are contained in the input range, they won't cause any violation.

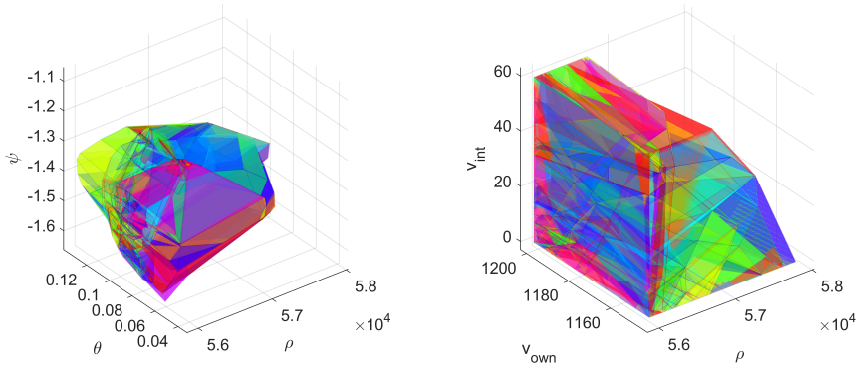


Figure 4.7: Complete input sets that lead to the property ϕ_2 violation on the network N_{12}.

4.5.2 Reachability Analysis of Microbenchmarks

To further evaluate our approach, we compare them on a set of microbenchmarks that are proposed in Dutta et al. (2018). These benchmarks consist of neural networks are created from the unwinding of a closed-loop controller and a plant model. Here we check these networks' safety on some synthetic unsafe domains. Neurify is not included as we fail to apply their published code to these new networks. The performance on each benchmark is shown in Table 4.2. The neural networks consist of thousands of neurons, as indicated in column m in Table 4.2. As shown in Table 4.2, our approach computes the reachable sets for large networks. For Marabou, the *HD Error* indicates a high-degradation error. The $Error$ means Marabou returns a false verification result for the instance. The *LP Error* for the nnenum indicates that the algorithm is terminated by a LP-solver error.

Network	x	k	m	**Our Method**	**Marabou**	**NNV**	**nnenum**
N_{11}	3	9	1427	65.5	567.7	Timeout	LP Error
N_{12}	3	14	2292	96.5	Timeout	Timeout	LP Error
N_{13}	3	19	3057	128.9	HD Error	Timeout	LP Error
N_{14}	3	24	3822	150.9	HD Error	Timeout	LP Error
N_{15}	3	127	6845	2.3	Error	383.5	12.7

Table 4.2: Performance results on the microbenchmarks. The label x, k and m respectively denote the number of input, the number of layers and total number of ReLU neurons. The running time is in sec. The timeout is set to 10 minutes.

4.6 Discussion

In this paper, we presented a set-based method to conduct exact reachability analysis of feed-forward neural networks with ReLU neurons. The exact reachable set can be reusable for different properties that have

the same input domain, thus the extra computation time can be saved. To further improve the efficiency and convenience, we can also partition a wide-range input into smaller subsets and build a library for their reachable sets so that further analysis of any input contained in such range can be reduced to merely looking up the library. Overall, there are multiple directions to improve our approach. One is a more concise and efficient polytope representation than FVIM because it becomes complicated for high dimensional polytopes. Another potential enhancement is to detect the polytopes in the input set that determine the output range so that the redundant polytopes can be ignored, and the computation burden can be reduced. Additionally, one direction is to apply the approach to reachability analysis and safety verification of learning-enabled control systems with neural networks involved.

CHAPTER 5

Reachability Analysis of Convolutional Neural Networks

5.1 Preliminaries

The architecture of CNNs usually consist of multiple layers including convolutional layers (L_c), Max-pooling layers (L_m), batch-normalization layers (L_b), ReLU layers (L_r) and feed-forward layers (L_f). Let N denote a CNN with l layers and L_i be its ith layer where $i \in [1, 2, \ldots, n]$. Then, a CNN can be formulated as:

$$N = L_l \circ L_{l-1} \circ \cdots \circ L_1 \tag{5.1}$$

where $L_i \in \{L_c, L_m, L_b, L_r, L_f\}$. Inputs to the network will be sequentially processed by each layer. The reachability analysis of neural networks refers to the computation of output reachable sets for the network given an input set. As introduced, a *reachable set* refers to a domain where all the element points are reachable. This set is usually represented using linear constraints or a geometric objects such as convex polytopes Xiang et al. (2018), Star set Tran et al. (2019a), Zonotope Tran et al. (2019a); Singh et al. (2018a) and Abstract domain Singh et al. (2019b). The reachable-set computation of the network N can be formulated by

$$\mathbb{N}(\mathcal{S}) = (\mathbb{L}_l \circ \mathbb{L}_{l-1} \circ \cdots \circ \mathbb{L}_1)(\mathcal{S}) \tag{5.2}$$

which is similar to Equation 4.8. The output reachable sets of one layer are inputs to the next layer.

Depending on the type of the layer, the function $\mathbb{L}(\cdot)$ has two different operations. First, an *affine transformation* operation is applied in layers L_c, L_b and L_f. This operation linearly transforms element points of an input set with weights and bias (convolutional layer) or the mean and standard deviation (batch normalization layer). Second, the input-domain division operation of the max function is applied in layers L_r and L_m. The max function exhibits an unique linearity over each input range. For instance, the ReLU activation function y=$max(0, x)$ has two different input ranges $x < 0$ and $x \geq 0$ over which the output of the max function has different linearities. This is a source of non-linearity in CNNs and makes the set computation challenging.

Recently, various approaches have been proposed to handle such challenges, and they can be classified in two main categories. One is an over-approximation based approach, such as Zonotope Gehr et al. (2018) and Abstract Domain Singh et al. (2018a), and the other one is an exact-analysis approach such as Polytope Xiang et al. (2017), Star set Tran et al. (2019a) and ImageStar Tran et al. (2020a). The approximation methods

normally apply a geometric object such as Zonotope to approximate the max function, which simplifies representation of the nonlinearity of the max function. This simplification results in very efficient algorithms, however, the over-approximation bound may be large, since the approximation error w.r.t the max function is accumulated in an exponential manner. The exact-analysis methods separately consider the divided input domains of the max function. These types of methods compute the exact reachable sets and can guarantee a complete and sound verification of networks. But the number of reachable sets increases exponentially w.r.t. the neurons, yielding an expensive computation process. Several set representations have been utilized by other works Xiang et al. (2018); Tran et al. (2019a, 2020a), aiming to simplify this process. However, these approaches do not scale very well for verification of deep neural networks. Here we propose a novel representation of sets using the face lattice structure which can exhibit a significantly higher efficiency compared to the aforementioned methods.

5.2 Reachable Set Representation

In this section, we introduce the notion of a face lattice Henk et al. (2004); Grünbaum (2013) and show how it can be used for constructing reachable sets, and how its properties make it particularly efficient for the computation of reachable sets for CNNs.

5.2.1 Face Lattice Structure

The face lattice of a set is a complete combinatorial structure that contains all its faces and partially orders them by face containment. A 3-dimensional tetrahedron and its face lattice is shown in Figure 5.1. Its faces are organized in terms of their dimensions. The highest-dimensional face *3-face* is the tetrahedron itself. There are a total of 15 faces described by blue blocks and their dimension ranges from 0 to 3. The downwards paths along the solid lines from one face to its adjacent faces represent their containment relation, which is transitive. For instance, the 2-dimensional face *2-face*:1 which is the $plane_{2\text{-}3\text{-}4}$ contains the 1-dimensional face *1-face*:2 which is the $edge_{2\text{-}4}$. The indices of the *0-face* correspond to the vertices. Given a set $\mathcal{S}$, let $\mathcal{F}$ denote its face lattice and V denote its vertices' values, then $\mathcal{S}$ is represented by

$$\mathcal{S} = \langle \mathcal{F}, V \rangle \tag{5.3}$$

More geometric details on the face lattice representation can be found in Henk et al. (2004).

5.2.2 Affine Transformation

Affine transformation operations are very common in computing reachable sets. One of the advantages of the face lattice structure is that affine transformation only changes the values of vertices while preserving the

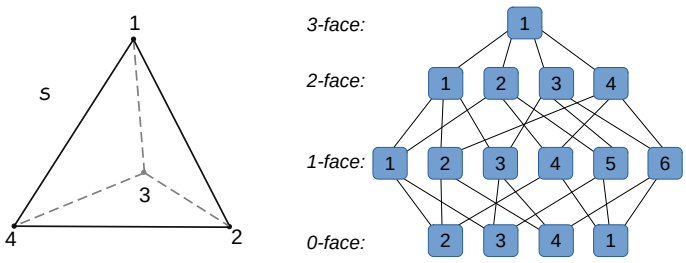


Figure 5.1: Face lattice structure of a tetrahedron.

face lattice structure Henk et al. (2004); Grünbaum (2013). An affine transformation on a set $\mathcal{S}$ is applied as follows. Given a set $\mathcal{S} = \langle \mathcal{F}, V \rangle$, and an *affine transformation* with a weight matrix W and a bias vector b, then the output set $\mathcal{S}' = \langle \mathcal{F}', V' \rangle$ is computed by

$$\mathcal{F}' = \mathcal{F}, \quad V' = WV + b \tag{5.4}$$

We note that an *affine transformation* may project a set into a higher-dimensional or lower-dimensional space. Projection onto higher dimensional space normally happens in the convolutional layer. On the other hand, projection onto lower dimensional space commonly happens in the linear layer where neurons are fully connected. In this case, the dimension of the set will be reduced to the dimensions of the target space accordingly. When the set is affine transformed to a lower dimension, its face lattice is preserved as described in Equation 5.4. The faces whose dimensions are higher than the target do not influence the computation of reachable sets, and the face lattice including all faces together will be processed for future operations, such that the geometric information from the preceding layer is maintained.

5.2.3 Split Operation

The split operation is applied when a set spans over multiple input domains of the max function. For instance, when dealing with the max function in a ReLU neuron, the input set is split in two subsets in order to capture the two different linearities. For a general max function $y = max\{x_1, x_2, \ldots, x_n\}$, we have that $y = x_i$ if and only if $\forall x_j, j \neq i$, $x_i - x_j \geq 0$. We note that the input domain over which x_i is the maximum is characterized by a set of linear constraints, and that input domains specifying different x_i as the maximum are adjacent and divided by hyperplanes. In exact reachability analysis, an input set will need to be split when it spans such domains. This case can be determined by inspecting whether the set intersects with those hyperplanes. In practice, making this determination is the most common and challenging issue in the exact reachability analysis of DNNs. It's usually handled by linear-optimization or similar approaches Katz

et al. (2019); Kouvaros and Lomuscio (2018); Tran et al. (2020a); Singh et al. (2018b, 2019b), which are computational expensive due to the high dimension of the sets and the large amount of neurons. The face lattice structure does not require such optimization and is therefore much more efficient. To formally illustrate the split process, we also utilize the concepts *positive* and *negative* vertices, and *positive* and *negative* subsets as defined in Section 4.2.2.

Given a $\mathcal{H} : \mathrm{a}^\top x + b = 0$ and a set $\mathcal{S}$, the split operation starts by determining whether there is an intersection between $\mathcal{H}$ and $\mathcal{S}$. As the face lattice contains vertices of $\mathcal{S}$, this determination can be simplified to finding the distribution of vertices on the sides of $\mathcal{H}$. This is achieved by substituting the x in $\mathrm{a}^\top x + b$ with vertices values. In terms of the vertices distribution, a vertex v is positive v^+ if $\mathrm{a}^\top x + b > 0$, negative v^- if $\mathrm{a}^\top x + b < 0$, and v^0 if $\mathrm{a}^\top x + b = 0$. Overall, there are three cases to consider:

1) The hyperplane $\mathcal{H}$ intersects with $\mathcal{S}$, where $\mathcal{S}$ has both v^+s and v^-s w.r.t. to $\mathcal{H}$. In this case, $\mathcal{S}$ is split into two non-empty subsets $\mathcal{S}_1^+$ and $\mathcal{S}_2^-$.

2) The hyperplane $\mathcal{H}$ doesn't intersect with $\mathcal{S}$, where $\mathcal{S}$ doesn't have any v^-s w.r.t. to $\mathcal{H}$. In this case, $\mathcal{S}$ itself is *positive* w.r.t. the $\mathcal{H}$.

3) The hyperplane $\mathcal{H}$ doesn't intersect with $\mathcal{S}$, where $\mathcal{S}$ doesn't have any v^+s w.r.t. to $\mathcal{H}$. In this case, $\mathcal{S}$ itself is *negative* w.r.t. the $\mathcal{H}$.

In the following, we focus on the first case, as the last two cases can be easily processed. The split process in the first case includes four steps, as illustrated in Figure 5.2. They are: 1) identification of faces in each dimension that intersects with $\mathcal{H}$, 2) derivation of the face lattice structure that consists of the new faces generated from the intersection of $\mathcal{H}$ with those faces obtained from Step 1, 3) splitting of the original face lattice into two sub-structures according to the types of its vertices, 4) merging of the structure generated from Step 2 respectively with the two sub-structures from Step 3, and forming the final face lattice for $\mathcal{S}_1^+$ and $\mathcal{S}_2^-$.

The identification of faces in Step 1 is based on Lemma 3, with which we can identify different dimensional faces that intersect with the hyperplane from the lower dimension to the higher along the containment relation. The identification starts with the *1-faces* (edges). A *1-face* is said to be intersecting with $\mathcal{H}$ if it contains a v^+ and a v^-. Based on these 1-$faces$, we search faces along the upward path to the next upper dimension, and repeat this process until we reach the highest-dimensional faces. This ensures all faces are searched. This process is demonstrated in Figure 5.2 (b).

The derivation of the new face structure in Step 2 is based on Lemma 4. The intersection of $\mathcal{H}$ with each face identified in Step 1 generates a new face. Additionally, their containment relation can also be inherited

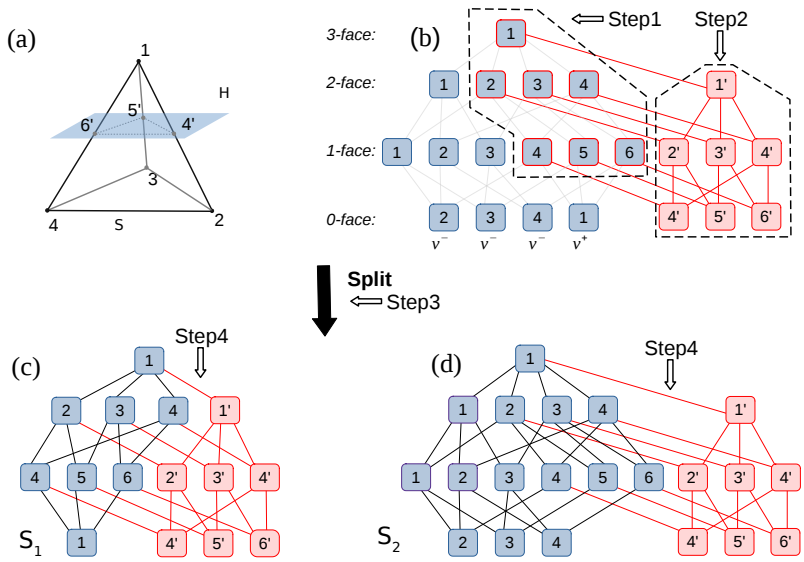


Figure 5.2: A demonstration of the split process of the face lattice of a tetrahedron $\mathcal{S}$ with a hyperplane $\mathcal{H}$. In (b), the blocks filled with blue and connected by solid lines is the face lattice. The connection by solid lines represents face containment. The blocks with red frames and blue color are the faces intersecting with $\mathcal{H}$ in $\mathcal{S}$, which are extracted from the listed $positive$ and $negative$ vertices in Step 2. The new face lattice generated in Step 2 is in red. In (c) and (d), the sub-face lattices consisting of the blue blocks are split from Step 3, and the whole structures with the red blocks represent the final face lattice. The one in (c) represents the *positive* set and the one in (d) represents the *negative* set.

from the original faces based on Lemma 4. This process is illustrated in Figure 5.2 (b), where the red solid lines represent the inherited containment relations. From another perspective, this derivation is essentially equivalent to a duplication of a sub-face lattice that only contains the faces that intersect with $\mathcal{H}$ in the original structure.

The split operation in Step 3 separates the face lattice into two sub-face lattices according to the type of their vertices. Faces of different dimensions belonging to one sub-face lattice are searched starting from the vertices along the containment relation up to the highest-dimensional faces. With v^+:$\{1\}$ and v^-:$\{2,3,4\}$, it generates two sub-face lattices which consists of blue blocks, as shown in (c) and (d). In terms of containment relations inherited in Step 2, the sub-face lattices from Step 3 and the new face lattice from Step 2 are merged, generating the final structures for two subsets $\mathcal{S}_1^+$ and $\mathcal{S}_2^-$, which is Step 4.

Lemma 3 *Given a set $\mathcal{S}$ in the face lattice structure and a hyperplane $\mathcal{H}$, if $\mathcal{H}$ intersects with a k-$face \subseteq \mathcal{S}$, then the $\mathcal{H}$ also intersects with all $(k$+1$)$-$faces$ where $(k$+1$)$-$faces \subseteq \mathcal{S}$ and k-$face \subseteq (k$+1$)$-$faces$.*

Proof. Let $\mathcal{H} \cap k$-$face = \psi$ and $\psi \neq \emptyset$, then $\psi \subseteq \mathcal{H}$ and $\psi \subseteq k$-$face$. As k-$face \subseteq (k$+1$)$-$faces$, then $\psi \subseteq (k$+1$)$-$faces$ and $\psi \subseteq (\mathcal{H} \cap (k$+1$)$-$faces)$. Thus $\mathcal{H}$ intersects with $(k$+1$)$-$faces$.

Lemma 4 *Assume a hyperplane $\mathcal{H}$ intersect with a $(k$+1$)$-$face$ and k-$face$ in a set and generate new faces*

k-$face'$ and (k-1)-$face'$ accordingly. If k-$face \subseteq$ (k+1)-$face$, then (k-1)-$face' \subseteq k$-$face'$.

Proof. As k-$face \subseteq$ (k+1)-$face$, it can be inferred that (k-$face \cap \mathcal{H}$) $\subseteq$ ((k+1)-$face \cap \mathcal{H}$), from which we can derive that (k-1)-$face' \subseteq k$-$face'$.

5.3 Computation of Reachable Sets of CNNs

As discussed in Section 2, there are two primary operations on a set when it passes through a CNN. The *affine transformation* is applied in the layers such as the convolutional layer and the linear layer, and the *split* operation is applied to the ReLU layer and the Max-pooling layer. As introduced in Section 5.2.1, the *affine transformation* only linearly updates the vertices of a set and its face lattice structure will be preserved. The algorithmic implementation of this operation is straightforward. Therefore, this section will mainly focus on the *split* operation on sets in ReLU and Max-pooling layers. Additionally, a tracking method is also proposed which enables backtracking to the input domain of the network given an output reachable set. This method can be utilized to identify space of adversarial examples.

5.3.1 ReLU Layer

Assume a ReLU layer contains n neurons and its input is denoted as $x \in \mathbb{R}^n$. Then, the output of the k^{th} neuron is $y_k = \max\{0, x_k\}$, and the input space of this max function is divided by the boundary $x_k = 0$ into two domains over which it has different linearities. The boundary can be generalized to the hyperplane $\mathcal{H}_k : a^\top x = 0$ where $a = [a_1, a_2, \ldots, a_n]$ and $a_k = 1$ and $a_{i(i\neq k)} = 0$. When the input is a set $\mathcal{S}$ where $x \in \mathcal{S}$, it will be sequentially processed with the hyperplane of each neuron. The sets generated w.r.t. one hyperplane will be further processed by the next one until all hyperplanes in this layer are considered.

The process w.r.t. one hyperplane $\mathcal{H}_k$ is as follows. In terms of the discussion in Section 5.2.3, $positive$ and $negative$ sets can be obtained in the split operation w.r.t. $\mathcal{H}_k$. The *positive* set will stay unchanged as an output w.r.t to $\mathcal{H}_k$. On the other hand, the *negative* set will be projected on $\mathcal{H}_k$ and transformed to a new set as an output, which is essentially done by setting the x_k dimension of all the elements in the set to zero. Let the function $\mathbb{E}^{[k]}_{relu}$ denote this process, then the function $\mathbb{L}$ in Equation (5.2) for the ReLU layer is equal to

$$\mathbb{L}(\mathcal{S}) = (\mathbb{E}^{[n]}_{relu} \circ \mathbb{E}^{[n-1]}_{relu} \circ \cdots \circ \mathbb{E}^{[1]}_{relu})(\mathcal{S}) \tag{5.5}$$

An example is used to illustrate this process as shown in Figure 4.3. The ReLU layer contains two neurons n_1 and n_2. The input set $\mathcal{S} \subset \mathbb{R}^2$ and element points $x = [x_1, x_2]^\top \in \mathcal{S}$. When the set $\mathcal{S}$ passes through the layer, it will be split sequentially by two hyperplanes that are imposed by two ReLU activation functions $max(0, x_1)$ and $max(0, x_2)$ from the n_1 and n_2. These two hyperplanes are respectively $\mathcal{H}_1 : \{[1, 0]^\top x = 0\}$

and $\mathcal{H}_2 : \{[0,1]^\top x = 0\}$. Together, there are four unique domains having different linearities over the input x to this layer, and they are

$$\{x \mid [1,0]^\top x \geq 0 \text{ and } [0,1]^\top x \geq 0\}$$
$$\{x \mid [1,0]^\top x \geq 0 \text{ and } [0,1]^\top x \leq 0\}$$
$$\{x \mid [1,0]^\top x \leq 0 \text{ and } [0,1]^\top x \geq 0\}$$
$$\{x \mid [1,0]^\top x \leq 0 \text{ and } [0,1]^\top x \leq 0\}$$

The input set $\mathcal{S}$ is first split by Hyperplane $\mathcal{H}_1$ into two subsets $\{\mathcal{S}_1, \mathcal{S}_2\}$. The $\mathcal{S}_2$ locates in the halfspace of $\mathcal{H}_1 : [1,0]^\top x \leq 0$ and thus is the *negative* set w.r.t. $\mathcal{H}_1$. Due to the property of the negative domain in the ReLU function, $\mathcal{S}_2$ will be projected on the $\mathcal{H}_1$, generating a new set $\mathcal{S}_3$. This process is essentially setting the x_1 of all the elements in the set to zeros. It's also equivalent to a linear transformation, and the matrix is an identity matrix with the first diagonal entry being zero. While the sets locating in the halfspace of $\mathcal{H}_1$ will remain. So far, sets $\{\mathcal{S}_1, \mathcal{S}_3\}$ are obtained from the *split* by $\mathcal{H}_1$. Then, these sets will be split by $\mathcal{H}_2$ into four subsets $\{\mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_7\}$ as shown in (c), where $\{\mathcal{S}_4, \mathcal{S}_7\}$ are *negative* sets w.r.t. $\mathcal{H}_2$. Similarly, sets $\mathcal{S}_4$ and $\mathcal{S}_7$ will be mapped on $\mathcal{H}_2$ by setting the x_2 of all their element points to zeros, generating new sets $\mathcal{S}_8$ and $\mathcal{S}_9$ which is the original point. Then the final outputs of this layer are $\{\mathcal{S}_5, \mathcal{S}_6, \mathcal{S}_8, \mathcal{S}_9\}$. Let the function $\mathbb{E}^{[i]}_{relu}$ denotes the *split* process by a hyperplane $\mathcal{H}_i$ as well as the subsequent linear projection for sets that locate in the negative halfspace. Then we have the following process:

$$\{\mathcal{S}_2, \mathcal{S}_3\} = \mathbb{E}^{[1]}_{relu}(\{\mathcal{S}_1\})$$
$$\{\mathcal{S}_5, \mathcal{S}_7, \mathcal{S}_8, \mathcal{S}_9\} = \mathbb{E}^{[2]}_{relu}(\{\mathcal{S}_2, \mathcal{S}_3\})$$

For one layer, given an input set, the maximum number of output reachable sets is 2^n, where n is the number of neurons. Our experimental results indicate that the actual number is associated with the input set's volume. An input set with a larger volume will generate a larger number of sets, as it has a higher likelihood of being split by hyperplanes. This process is described in Algorithm 2. It's based on a recursive function which aims to reduce the repeated computation on the hyperplanes that don't intersect with sets. Given an input set $\mathcal{S}$, it starts with **FnReLULayer**($\mathcal{S}$, *neurons*=empty, *flag*=False). In each recursion, only one valid neuron whose hyperplane truly intersects with $\mathcal{S}$ is considered. Function **FnValidNeurons** is first to compute all such valid neurons denoted as *news* as well as the neurons *negs* where $\mathcal{S}$ is *negative* w.r.t. the target hyperplanes. Subsequently, $\mathcal{S}$ will be projected on those hyperplanes using Function **FnMap**. Function **FnSplit** denotes the split process introduced in Section 5.2.3.

Algorithm 2 Computation in ReLU layers

Input: $\mathcal{S}$, *neurons* # input set and neurons to process
Output: $\mathcal{O}$ # output reachable sets

procedure $\mathcal{O}$ = FNRELULAYER($\mathcal{S}$, *neurons*, *flag*=True)
  if $neurons$ is empty and $flag$ **then**
    return $\mathcal{S}$
  news, *negs* = FnValidNeurons($\mathcal{S}$, *neurons*)
  $\mathcal{S}$ = FnMap($\mathcal{S}$, *negs*)
  outputs = FnSplit($\mathcal{S}$, *news*.pop(0))
  $\mathcal{O}$ = *empty*
  for $\mathcal{S}$ in *outputs* **do**
    $\mathcal{O}$.extend(FnReLULayer($\mathcal{S}$, *news*))
  return $\mathcal{O}$

5.3.2 Max-pooling Layer

The purpose of the Max-pooling layer is to reduce the dimensionality of an input by pooling with a max function. For parameter configurations, we consider the most common kernel size 2×2 and stride size 2×2. Each pool contains four unique element dimensions, and there are no overlaps between pools. Suppose a pool contains dimensions $\{x_1, x_2, x_3, x_4\}$, then the pooling process will be formulated by $y = max\{x_1, x_2, x_3, x_4\}$. Element x_i will be the output if $\forall x_{j(j \neq i)},\ x_i - x_j \geq 0$. We notice that the input domain over which x_i is the maximum is the common space of the three halfspaces $\mathcal{H}_{(x_i, x_{j(j \neq i)})}$: $x_i - x_j \geq 0$. When an input set to this pool spans this domain, its subset can be computed by sequentially applying the split operation with the three aforementioned hyperplanes. Considering the other three dimensions as well, there are totally six unique hyperplanes, and they are $\mathcal{H}_{(x_1,x_2)}, \mathcal{H}_{(x_1,x_3)}, \mathcal{H}_{(x_1,x_4)}, \mathcal{H}_{(x_2,x_3)}, \mathcal{H}_{(x_2,x_4)},$, and $\mathcal{H}_{(x_3,x_4)}$, respectively. With them, the input space of the max function is divided into four domains over which they exhibit unique linearity, and they are

$$\{x \mid x_1 - x_2 \geq 0 \,\&\, x_1 - x_3 \geq 0 \,\&\, x_1 - x_4 \geq 0\}$$

$$\{x \mid x_2 - x_1 \geq 0 \,\&\, x_2 - x_3 \geq 0 \,\&\, x_2 - x_4 \geq 0\}$$

$$\{x \mid x_3 - x_1 \geq 0 \,\&\, x_3 - x_2 \geq 0 \,\&\, x_3 - x_4 \geq 0\}$$

$$\{x \mid x_4 - x_1 \geq 0 \,\&\, x_4 - x_2 \geq 0 \,\&\, x_4 - x_3 \geq 0\}$$

The split operation is the same as introduced in Section 5.2.3. However, in this case, the split subset will be handled differently. In the ReLU layer, the *negative* sets will be mapped to their corresponding hyperplane. While in the Max-pooling layer, non-maximum dimensions will be eliminated. Here, the concepts of the *negative* set and the *positive* set are extended to be w.r.t a specific dimension as defined in Definition 5.3.1.

Definition 5.3.1 (Set Types) *In the Max-pooling layer, given a hyperplane $\mathcal{H}_{(x_i,x_j)}$, a set is named x_i-*

negative or x_j-positive if it belongs to the closed halfspace $x_i - x_j \leq 0$, and is named x_j-negative or x_i-positive if it belongs to the closed halfspace $x_j - x_i \leq 0$.

When a set is split by a hyperplane $\mathcal{H}_{(x_i,x_j)}$ into two non-empty subsets, we have a x_i-*negative* and a x_j-*negative* subset. The dimension for which the subset is *negative* will be eliminated. Then, these new sets will be processed w.r.t. the next hyperplane. Let $\mathbb{E}_{max}^{[x_i,x_j]}$ denote the process with the hyperplane $\mathcal{H}_{(x_i,x_j)}$, and $\mathbb{P}$ denote the sequential splits of an input set $\mathcal{S}$ by these hyperplanes in one pool. Then we can derive Equation (5.6).

$$\mathbb{P}(\mathcal{S}) = (\mathbb{E}_{max}^{[x_3,x_4]} \circ \mathbb{E}_{max}^{[x_2,x_4]} \circ \mathbb{E}_{max}^{[x_1,x_4]} \circ \mathbb{E}_{max}^{[x_2,x_3]} \circ \mathbb{E}_{max}^{[x_1,x_3]} \circ \mathbb{E}_{max}^{[x_1,x_2]})(\mathcal{S}) \tag{5.6}$$

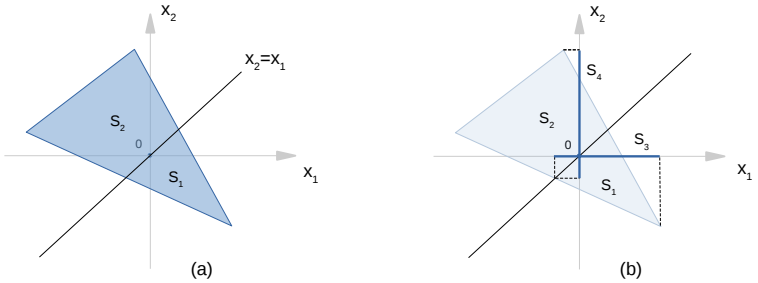


Figure 5.3: Demonstration of the split operation in the Max-pooling layer

The process $\mathbb{E}_{max}^{[x_i,x_j]}$ is illustrated in Figure 5.3. There are two dimensions x_1 and x_2 and correspondingly there is one hyperplane $\mathcal{H}_{(x_1,x_2)} : x_1 - x_2 = 0$. As shown in Figure (a), the input set $\mathcal{S}$ is first split by $\mathcal{H}_{(x_1,x_2)}$ into two subsets S_1 and S_2 which respectively are x_2-*negative* and x_1-*negative*. The elimination of dimensions is shown in Figure (b), where the 2-dimensional sets $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively generate new 1-dimensional sets $\mathcal{S}_3$ and $\mathcal{S}_4$. Finally, this pooling yields reachable sets $\{\mathcal{S}_3, \mathcal{S}_4\}$. We have

$$\{\mathcal{S}_3, \mathcal{S}_4\} = \mathbb{E}_{max}^{[x_1,x_2]}(\mathcal{S}) \tag{5.7}$$

Suppose the layer has n pooling operations and the k^{th} pooling is denoted as $\mathbb{P}_k$, the $\mathbb{L}$ in Equation (5.2) for the Max-pooling layer with an input set $\mathcal{S}$ can be formulated as Equation (5.8).

$$\mathbb{L}(\mathcal{S}) = (\mathbb{P}_n \circ \mathbb{P}_{n-1} \circ \cdots \circ \mathbb{P}_1)(\mathcal{S}) \tag{5.8}$$

Given an input set, the maximum number of the output reachable sets of this layer is 2^{3n}. Similar to the ReLU layer, the actual number is related to the input set's volume. In addition to Equation (5.8), The process

in the Max-pooling layer is also described in Algorithm 3. It's conducted in a similar recursive manner as Algorithm 2. Given an input set $\mathcal{S}$, this algorithm starts with **FnMaxpooling**(S, pools=empty, flag=False). Function **FnValidPools** helps identify a new set of pools *news* where intersections exist between their hyperplanes and $\mathcal{S}$. The set $\mathcal{S}$ is processed with one pool in each recursion, which is described by Equation (5.6) and Line 9-14 in the algorithm. Function **FnValidHplanes** helps identify hyperplanes. Function **FnSplit** splits a set with a hyperplane.

Algorithm 3 Computation in max-pooling layers

Input: $\mathcal{S}$, *pools* # input set and pools to process
Output: $\mathcal{O}$ # output reachable sets

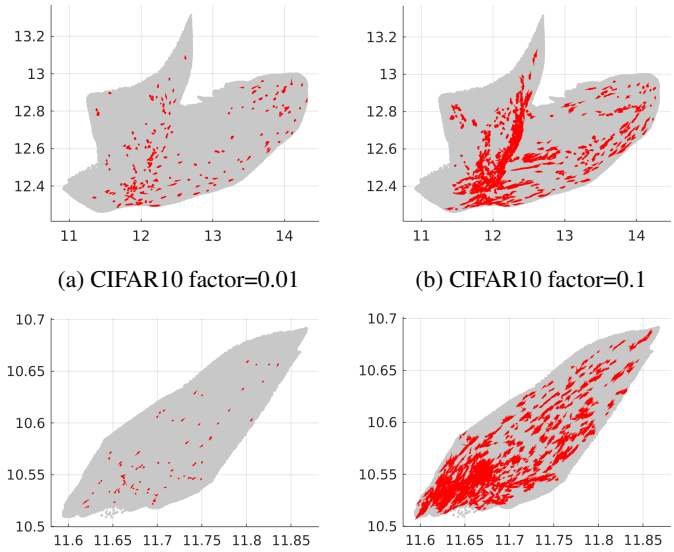
```
1:  procedure O = FNMAXPOOLING(S)
2:      if pools is empty and flag then
3:          return S
4:      news = FnValidPools(S, pools)
5:      if news is empty then
6:          return FnEliminateDims(S)
7:      hps = FnValidHplanes(news.pop(0))
8:      sets = S
9:      for H in hps do
10:         temp = empty
11:         for S in sets do
12:             temp.extend(FnSplit(S, H))
13:         sets = temp
14:     O = empty
15:     for S in sets do
16:         O.extend(FnMaxpooling(S, news))
```

5.3.3 Backtracking Method

Given output reachable sets, this method enables backtracking to find the corresponding subspace in the input domain. This enables us to identify the adversarial input space. As described in Equation (5.5) and (5.6), the domains over which the max function exhibits different linearities are always separately considered for processing reachable sets. Therefore, for each output reachable set, there always exists a subset in the input space over which the neural network is linear, and this subset is a *linear region* as introduced in Section 4.1.1. There have been some works on the quantification of linear regions to assess the expressibility of neural networks Montufar et al. (2014); Serra et al. (2018); Hanin and Rolnick (2019).

Our backtracking method tracks the *linear region*s of reachable sets when they are sequentially processed by neurons. Here, we represent the *linear region* using vertices (V-representation). As reachable sets are related to their *linear regions* with a linear map, the vertices of the *linear region* and the vertices of the set itself are in a one-to-one correspondence. This relation is maintained in the *affine transformation*. For the *split* process in the ReLU neuron $\mathbb{E}_{relu}$ and the Max-pooling $\mathbb{E}_{max}$, in addition to splitting the reachable set,

we also split its *linear region* accordingly, such that their one-to-one relation can be maintained and tracked. The *linear region* of the initial input set to the network is the input set itself. Overall, all the computed output reachable sets have their *linear region*s, and thus any violation in the output reachable sets can be backtracked to the input space.



(a) CIFAR10 factor=0.01 (b) CIFAR10 factor=0.1

(c) VGG16 factor=0.01 (d) VGG16 factor=0.1

Figure 5.4: The fast computation of reachable sets with different *relaxation* factors. The gray area is the exact reachable domain. The red area are the reachable domains computed by the fast method. The x axis is the logit of the correct class, while the y axis is the second highest logit.

5.4 Fast Reachability Analysis

With the face lattice structure, we can efficiently compute the exact output reachable sets for most of the deep neural networks and guarantee complete and sound verification. However, due to the exponential computational complexity in the ReLU layer and Max-pooling layer, the analysis of a deeper CNN with larger input sets is still challenging. Here we propose an efficient alternative approach which only considers the sensitive neurons. The sensitivity of each neuron is ranked using the gradient of the DNN output w.r.t. the input to this neuron. We use a *relaxation* factor to select a fraction of the highest ranked neurons in each layer and compute reachability analysis using only the selected neurons. The *relaxation* factor defines the trade-off between computation time and completeness of the computed reachable sets. Our experimental results show that even with a small *relaxation* factor, the exact reachable sets can still be well approximated, as shown in Figure 5.4 and 5.6.

The reachable set computation is slightly different using the fast analysis method. In the ReLU layer, for the selected neurons, computation remains unchanged. However, for each of the non-selected neurons, at

most two subsets can be generated by the split operation and only the subset, which has the highest number of vertices, is retained. We choose the subset with more vertices since it is more likely that it carries more geometric information. In the Max-pooling layer, for the selected dimensions or neurons, computation remains unchanged. Here, we apply a split operation for each of the non-selected dimensions and discard the subset which is *positive* w.r.t. a non-selected dimensions (see Definition 5.3.1). When two dimensions are non-selected, we keep the subset with more vertices to avoid an empty output.

Algorithm 4 Fast Computation of Reachable sets

Input: $\mathcal{S}$, δ # input set and *relaxation* factor
Output: $\mathcal{O}$ # output reachable sets

```
1:  procedure O = FastCompute(S)
2:      O = S
3:      for l in layers do
4:          if l is affine-transformation layer then
5:              O = FnAffineTrans(O)
6:          else if l is ReLu layer then
7:              neurons = FnImpactNeurons(δ)
8:              O = FnReLULayerFast(O, neurons)
9:          else if l is Maxpooling layer then
10:             neurons = FnImpactNeurons(δ)
11:             O = FnMaxPoolingFast(O, neurons)
12:     return O
```

The fast analysis is presented in Algorithm 4. Given a *relaxation* factor, Function **FnImpactNeurons** selects *neurons* on which the fast computation of reachable sets are conducted. Functions **FnReLULayerFast** and **FnMaxPoolingFast** are the revised functions respectively from Algorithm 2 and 3. The revised part of these functions is that when two subsets are generated from Function **FnSplit** and the current neuron is not in *neurons*, the subset with more vertices will be kept while the other one will be discarded.

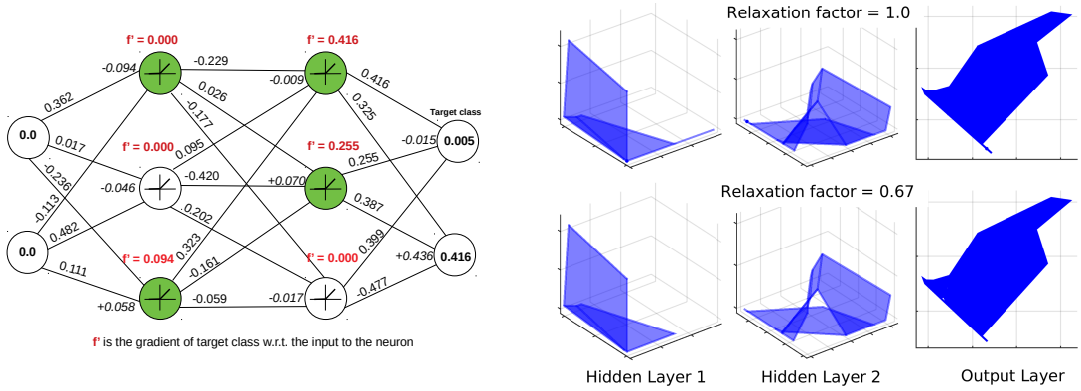


Figure 5.5: Reachable domains with different factors

Here is a running example to demonstrate our algorithm in Figure 5.5. It starts by ranking the neurons

in each layer in terms of sensitivity. The relaxation factor defines the percentage of top-ranked neurons considered in the fast computation of reachable sets. The lower the factor, the fewer the number of neurons considered and the faster the method will be. The input is $x = 0$ and the perturbed range is $x \pm 0.5$. When $factor = 1.0$, it considers all neurons and computes exact reachable domain. The result is shown in the first row where the exact reachable domains of each layer are included. When factor=0.67, it considers 2 neurons(in green) in each hidden layer which have the highest absolute gradients of the target output. The algorithm computes the subset of exact reachable domain as shown in the second row.

5.5 Experimental Results

This section presents experimental results demonstrating the utility of our methods. (1) We evaluate our reachable-set computation with various *relaxation* factors to demonstrate the trade-off between the computational complexity and the completeness of reachable sets. (2) We compare the performance of our reachability analysis method to other state-of-the-art tools for the safety verification of CNNs. (3) We apply our method to evaluate the adversarial robustness of CNNs trained with various defense algorithms to pixelwise perturbation. (4) We apply our fast analysis method to falsify CNNs for image classification. The hardware configuration for all experiments is CPU: Intel® Core™ i9-10900K, Memory: 128 GB, GPU: Nvidia® GeForce® RTX 2080 Ti.

Relaxation	0.01	0.2	0.4	0.6	0.8	1.0
ϵ = 0.02	5.3 / 124	8.6 / 1602	12.5 / 4109	23.7 / 14274	32.3 / 22449	48.7 / 42931
ϵ = 0.06	5.2 / 126	16.3 / 6409	37.0 / 22902	115.5 / 95929	289.5 / 261140	765.5 / 766849
ϵ = 0.10	5.2 / 143	24.2 / 11445	66.4 / 44986	268.3 / 223970	968.7 / 835045	2930.1 / 2722864

Table 5.1: Evaluation of the fast reachability analysis with different settings of epsilons and *relaxation* factors. The result format is **Running Time**(*sec*)**/ Number of Reachable Sets** for each computation.

The fast-analysis method is evaluated with a CNN for the CIFAR10 dataset and the VGG16 for the ImageNet dataset. Here, the input set is created with a perturbation on the independent channels of target pixels. Let $C = [c_1, c_2, \ldots, c_n]$ denote the channels of the target pixels and ϵ be the perturbation range. For each channel $c_{i,i\in\{1,2,\ldots,n\}}$, a range $[c_i - \epsilon, c_i + \epsilon]$ is created. Thus, we can obtain a n-hyperbox set with a face lattice structure. In this evaluation, we target one pixel with ϵ = 1.0. The input set is partitioned into several subsets for parallel computation. The reachable sets are shown in Figure 5.4. The density and the number of reachable sets increase with the *relaxation* factor, approaching the exact reachable domain as the relaxation factor increases. We can notice that with *relaxation* = 0.01, the sparse reachable sets can approximate well the outline of their exact reachable domains. Additional experiments are also included to reveal that the running time and the number of reachable sets computed increase along with the *relaxation*

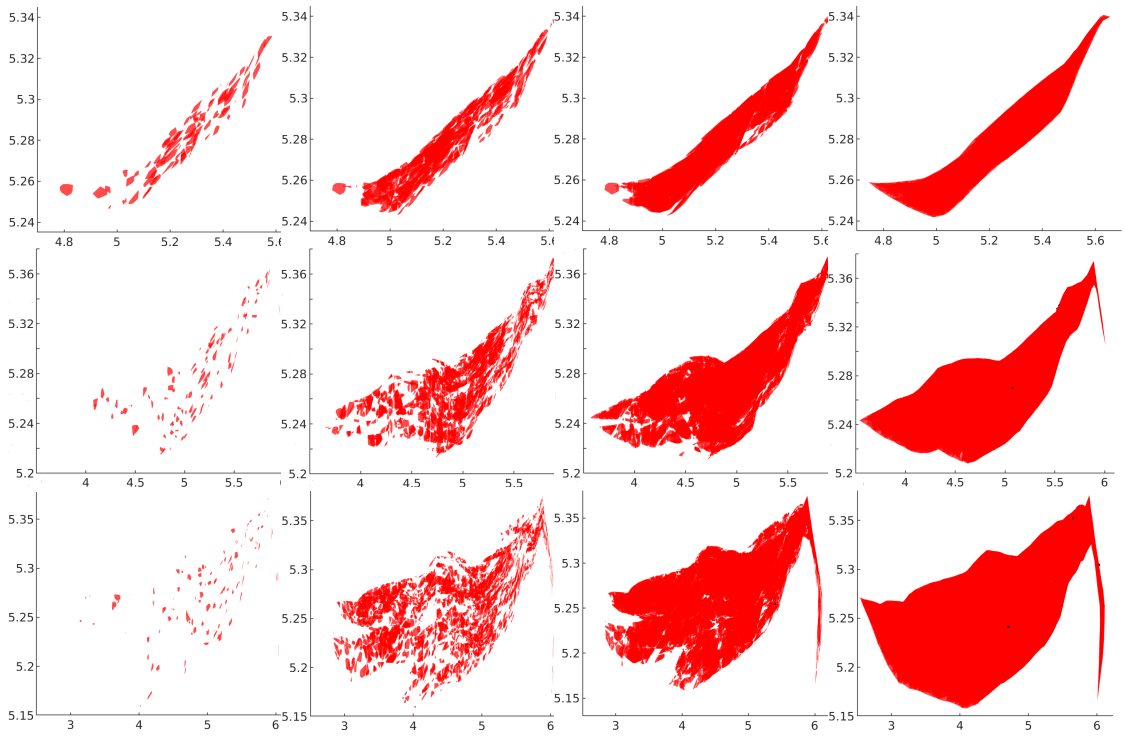


Figure 5.6: The fast computation of reachable sets with different *relaxation* factors on a CIFAR10 image. The computation in each row is with the different perturbation ϵs. From the top to the bottom, they are 0.02, 0.06 and 0.10, respectively. The computation in each column is with different *relaxation* factors. From the right to the left, they are 0.01, 0.2, 0.6 and 1.0, respectively. The x axis corresponds to the output of the correct class, and the y axis corresponds to the class which has the most likelihood to be misclassified to.

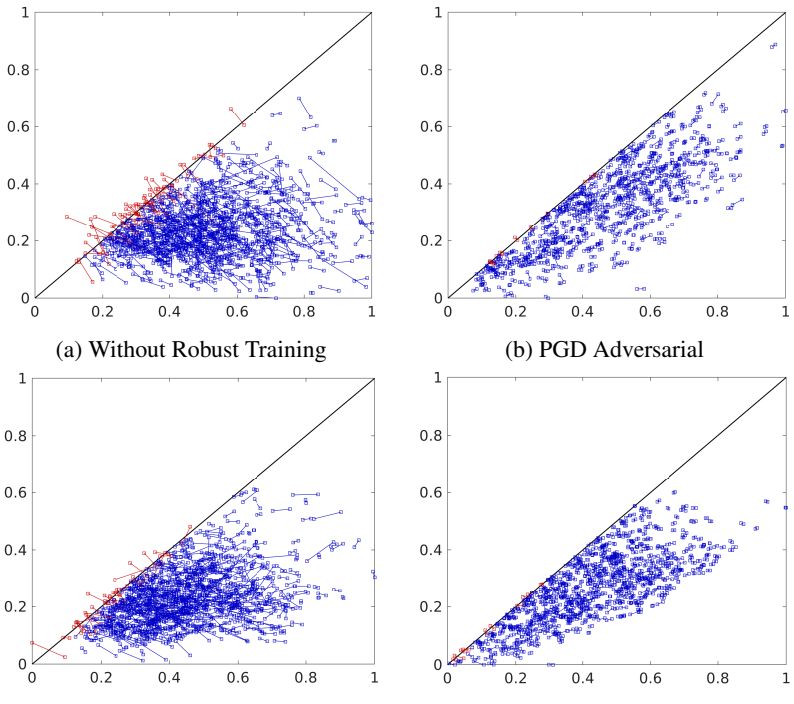
factor. The fast-analysis method is also evaluated by computing reachable sets w.r.t. one pixelwise variation on a CIFAR10 image. Different *relaxation* factors as well as ϵs are considered. The computed reachable sets are presented in Figure 5.6, and their number and the running time is shown in Table 5.1. We can notice that our algorithm can compute 1000 output sets per second, and that the running time and number of output sets increase exponentially with the *relaxation* factor.

We evaluate the exact reachability analysis of our approach by comparing it with related state-of-art-methods. Specifically: Deepzono Singh et al. (2018a), Refinezono Singh et al. (2018b), Deeppoly Singh et al. (2019b), Refinepoly Singh et al. (2019a) and the NNV tool Tran et al. (2020b). The proposed approach is evaluated on a CIFAR10 CNN which consists of 16 layers which include the convolutional layer, ReLU layer, and max pooling layer. The exact reachability analysis is tested on an image labeled with *cat*. 100 images are selected for the test. We apply perturbation with different ϵ values to one sensitive pixel and aim to verify the network. The result is shown in Table 5.2. Compared to NNV-Exact, which also computes the exact reachable domain, our method can quickly verify all images within the timeout limit, while the NNV-Exact can only verify the images with the small perturbation $\epsilon = 0.01$ and larger ϵ values causes out-

of-memory issues. For other over-approximation methods, some of them are faster in running time but mostly fail due to the large conservativeness of the approximated reachable domain.

ϵ	0.01					0.05					0.10					0.15				
Methods	**SF**	**US**	**UK**	**TT**	**TIME**	**SF**	**US**	**UK**	**TT**	**TIME**	**SF**	**US**	**UK**	**TT**	**TIME**	**SF**	**US**	**UK**	**TT**	**TIME**
NNV-Appr	66	0	34	0	2,303	0	0	98	2	29,738	0	0	67	33	193,907	0	0	<67	>33	<193,907
NNV-Exact	77	22	0	1	27,335	3	5	88	3	131,138	–	–	–	–	–	–	–	–	–	–
Deepzono	76	0	24	0	1,552	0	0	100	0	2,077	0	0	100	0	3,079	0	0	100	0	3,935
Refinezono	76	0	24	0	1,754	0	0	100	0	2,383	0	0	100	0	3,382	0	0	100	0	4,224
Deeppoly	75	0	25	0	7,6019	5	0	95	0	76,720	0	0	100	0	76,369	0	0	100	0	75,887
Refinepoly	77	0	23	0	146,763	10	0	90	0	186,558	0	0	100	0	198,058	0	0	100	0	219,365
Our Method	78	22	0	0	5,241	35	65	0	0	7,549	13	83	0	4	16,135	5	87	0	8	22,835

Table 5.2: Summary of the experimental results. The symbol **SF** stands for SAFE which indicates no misclassification in the input set. The **US** stands for UNSAFE indicating the existence of misclassification in the input set. The **UK** stands for UNKNOWN indicating a failed verification returned from the algorithm. The **TT** stands for the TIMEOUT which is set to one hour for each verification. The **TIME** stands for the total computation time for the verification of all images (in seconds). The incomplete result of NNV-Exact for ϵ=[0.05,0.10,0.15] is due to the out-of-memory (128 GB).



(a) Without Robust Training (b) PGD Adversarial

(c) Distillation (d) TRADES

Figure 5.7: Adversarial robustness evaluation with pixelwise perturbation. The x-axis is the output logits of the class *plane*. The y-axis is the second highest logit of the next likely class. The diagonal denotes the decision boundary. The segment length represents the approximated maximal output variation w.t.t. the input perturbation. The averaged segment lengths of each networks are respectively 0.040, 0.012, 0.034, 0.010. The standard deviations are 0.031, 0.008, 0.025, 0.006.

We also apply our method to analyze the adversarial robustness of CIFAR10 CNNs trained with different defense methods. These methods considered are PGD adversarial training Madry et al. (2018), Distilla-

tion Papernot et al. (2016) and TRADES Zhang et al. (2019). Test images include all the images correctly classified as *plane* in the test data. For each image, we select the most sensitive pixel and apply the fast reachability analysis to approximate the maximal output changes w.r.t. a perturbation of $\epsilon = 1$. A larger output variation indicates that a perturbation can have a larger effect on the output classification of the network. On the other hand, a smaller variation indicates that the network is more robust under the perturbation. This enables us to evaluate these networks' adversarial robustness. The experimental results over CNNs trained with different methods are shown in Figure 5.7. It shows the output change of each image before and after the perturbation. The end of the segment that is closer to the boundary denotes the worst output after perturbation, while the other end denotes the original output. The segment length reflects the maximal output variation w.r.t. the input perturbation. A red segment indicates that an adversarial example is found. For comparison, outputs are normalized into $[0, 1]$. We notice that networks trained without robust training or with the Distillation method exhibit much larger output variations, compared to the adversarial trained networks with PGD or TRADES. It indicates that these two networks of Figure 5.7 (a) and (c) are less resistant to the pixelwise perturbation and thus have lower robustness. This supports the results presented in Athalye et al. (2018); Tramer et al. (2020) where they show that methods such as Distillation which utilize gradient obfuscation give a false sense of robustness. This result indicates that our reachability analysis can be useful for a fair evaluation of robustness.

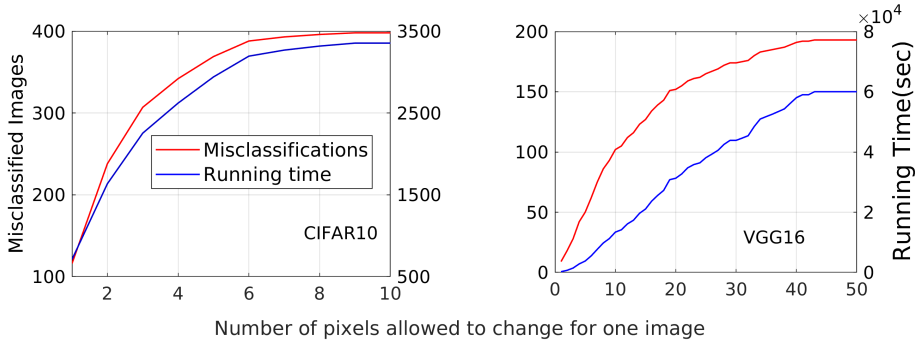


Figure 5.8: Falsification of CIFAR10 and VGG16 CNNs

Finally, to evaluate the falsification capability of the fast analysis method, we test it on a CIAFR10 CNN and the VGG16 network. Our falsification method changes the input image pixel by pixel until a misclassification is found. For each pixel, a fast-reachability computation is conducted with $\epsilon = 1$ and *relaxation* = 0.01. For each iteration, we first compute the output reachable domain for one pixel of the target image under the perturbation, from which we approximate the perturbed image which is the closest to the decision boundary. This perturbed image will be the input for the next iteration. The process is repeated until adversarial examples are found, or a timeout is reached. For the CIFAR10 and VGG16 networks, we attack 400

and 200 images that are correctly classified. The results are shown in Figure 5.8. With different number of pixels allowed to change in each image attack, we compute the number of misclassified images as well as the running time. We can notice that the selected CIFAR10 images can be successfully attacked when only a few pixels are affected. For VGG16, more pixels need to be changed to cause falsification. This is because a single pixel has a lower impact to the image classification due to the larger image size and a deeper network architecture. The averaged computation time of a single pixel for the CIFAR10 network is $\sim$2 seconds, while for a larger and deeper architecture VGG16 the computation time is $\sim$25 seconds.

CHAPTER 6

Neural Network Repair with Reachability Analysis

6.1 Deep Neural Network Repair

This section provides definitions and problem statements for DNN repair, and a brief overview of reachability analysis for DNNs. The problem is also extended to repair DNN agents in deep reinforcement learning.

6.1.1 Provably Safe DNNs

Let $f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X} \subseteq \mathbb{R}^{|x|}$ and $\mathcal{Y} \subseteq \mathbb{R}^{|y|}$ denote the input and output space of a DNN with the parameter set θ, where given an input $x \in \mathcal{X}$, it produces an output $y = f_\theta(x) \in \mathcal{Y}$. The safety problem of DNNs with reachability analysis on safety properties is formally defined as follows.

Definition 6.1.1 (Safety Property) *A safety property P of a DNN f_θ specifies a bounded input domain $\mathcal{I} \subseteq \mathcal{X}$ and a corresponding undesired output domain $\mathcal{U} \subseteq \mathcal{Y}$. The domain $\mathcal{I}$ refers to an interval with the lower bound $\underline{x}$ and the upper bound $\overline{x}$. The domain $\mathcal{U}$ refers to either a convex domain $Ay + b \leq 0$ or a non-convex domain consisting of multiple such convex domains.*

Definition 6.1.2 (DNN Reachable Domain) *Given an input domain $\mathcal{I} \subseteq \mathcal{X}$ to a DNN f_θ, its output reachable domain will be a subspace $\mathcal{O} \subseteq \mathcal{Y}$ where $\forall x \in \mathcal{I}$, its output $y = f_\theta(x)$ and $y \in \mathcal{O}$. The domain $\mathcal{O}$ is exact if it only contains the output of $\boldsymbol{x} \in \mathcal{I}$. Otherwise, it is over approximated. It is formulated as $\mathcal{O} = \mathbb{N}(\mathcal{I})$.*

Definition 6.1.3 (DNN Safety Verification) *A DNN f_θ is safe on a safety property P that specifies an input domain $\mathcal{I}$ and an output unsafe domain $\mathcal{U}$, or $f_\theta \models P$, if the exact output reachable domain $\mathcal{O} = \mathbb{N}(\mathcal{I})$ satisfies that $\mathcal{O} \cap \mathcal{U} = \emptyset$. Otherwise, it is unsafe, or $f_\theta \not\models P$.*

Given a set of safety properties $\{P\}_{i=1}^{n}$ and a candidate DNN f_θ, we define the DNN Repair problem as the problem of repairing the DNN to generate a new DNN f'_θ such that all the properties are satisfied, as defined in Problem 6.1.1. While repairing DNNs, it is also extremely important to preserve the parameter θ of the candidate DNN to the utmost extent because a subtle deviation on the parameter can lead to a high impact on the original performance and even introduce unexpected unsafe behaviors. These changes can be difficult to identify due to the black-box nature of DNNs. Therefore, our work also considers the minimal repair, which is formally defined in Problem 6.1.2.

Problem 6.1.1 (DNN Repair) *Given a DNN candidate f_θ and a set of safety properties $\{P\}_{i=1}^n$, at least one of which is violated, the repair problem is to train a new parameter set θ' based on θ, such that $f_{\theta'}$ satisfies all the safety properties, $f_{\theta'} \models \{P\}_{i=1}^n$.*

Problem 6.1.2 (Minimal DNN Repair) *Given a DNN candidate f_θ and a set of safety properties $\{P\}_{i=1}^n$, at least one of which is violated, the minimal repair problem is to train a new parameter set θ' based on θ, such that $f_{\theta'}$ satisfies all the safety properties, $f_{\theta'} \models \{P\}_{i=1}^n$ while minimizing the L-distance $\|\theta' - \theta\|_L$.*

For DNN classifier, we use classification accuracy on finite test data to analyze the performance of a repaired DNN $f_{\theta'}$ w.r.t. its original DNN f_θ. While for DNN regressor, we use the prediction error on test data. Additionally, we also analyze the impact of the DNN deviation on the reachability of a DNN. Here, reachability indicates the reachable domain of a DNN on its safety properties. As reachability characterizes the behaviors of a DNN, a desired repair method should preserve its reachability. In the repair of a DNN agent in Deep Reinforcement Learning (DRL), the DNN performance is set to the averaged rewards on a certain number of episode tests.

6.1.2 DNN Agent Repair for Deep Reinforcement Learning

In DRL, an agent is replaced with a DNN controller. The inputs to the DNN are states or observations, and their outputs correspond to agent actions. A *property* P for the DNN agent defines a scenario where it specifies an input state space $\mathcal{I}_{in}$ containing all possible inputs, and also a domain $\mathcal{U}$ of undesired output actions. Here, *safety* is associated with an input-output specification and reachability analysis refers to the process of determining whether a learned DNN agent violates any of its specifications and also the computation of unsafe state domain. This is formally defined as follows.

Definition 6.1.4 (Safe Agent) *Given multiple safety properties $\{P\}_{i=1}^n$ for a DNN agent N, the learned agent is safe if and only if for any P_i , the reachable domain $\mathcal{O}^{[i]}$ for its input state space $\mathcal{I}^{[i]}$ by $\mathcal{O}^{[i]} = \mathbb{N}(\mathcal{I}^{[i]})$ does not overlap with its unsafe action space $\mathcal{U}^{[i]}$, namely, $\mathcal{O}^{[i]} \cap \mathcal{U}^{[i]} = \emptyset$.*

An unsafe agent has the state domains $\mathcal{O}_u$ where their states will result in unsafe actions. Since the traditional adversarial training is usually for regular DNN training algorithms with existing training data, how to utilize such states or adversarial examples in DRL to repair unsafe behaviors remains a problem. By considering the fact that DRL learns optimal policies in interactive environments by maximizing the expectation of rewards, one promising way will be introducing penalty to the occurrence of unsafe actions during the learning, such that safety can also be naturally learned from the unsafe state domain. This strategy can also ensure the compatibility with the DRL algorithms, such that they can be seamlessly integrated. The repair problem is formally defined as follows:

Problem 6.1.3 (DNN Agent Repair) *Given a DNN agent candidate f_θ and a set of safety properties $\{P\}_{i=1}^n$, at least one of which is violated. The repair problem is to learn a DNN f'_θ through trial and error from the experience in the interactive environment, such that $f'_\theta \models \{P\}_{i=1}^n$ while maximizing the reward.*

6.1.3 Computation of Exact Unsafe Domain of ReLU DNNs

The unsafe reachable domain of a DNN refers to its unsafe input-output domain on the safety properties. In the repair, this domain is utilized not only for the safety verification, but also for its elimination by migrating this domain into safe domains in the retraining process. Traditional reachability analysis of neural networks focuses on computing the output reachable domain given an input domain. For neural network repair, it is just as important to backtrack the unsafe reachable domain to the corresponding unsafe input domain containing all adversarial examples. The input domain that generates unsafe behaviors is then used for the training/repair process of the DNN.

The computation of an unsafe input-output reachable domain is as follows. Given a DNN f_θ and a safety property P that specifies an input domain $\mathcal{I}$ and an unsafe output domain $\mathcal{U}$. The reachability analysis computes a set of output reachable sets $\mathcal{S}$, the union of which is the exact output reachable domain $\mathcal{O} = \bigcup_i^m \mathcal{S}_i$. Here, a set $\mathcal{S}$ refers to a convex set. The computation is denoted as $\mathcal{O} = \mathbb{N}(\mathcal{I})$. This process is illustrated in Figure 6.1. For each $\mathcal{S}$, we compute its overlap $\mathcal{S}_u$ with the specified unsafe output domain $\mathcal{U}$, and then apply a backtracking algorithm to compute its corresponding unsafe input subspace $\mathcal{E}_u$ which is also a convex set. The union of $\mathcal{S}_u$ is the exact unsafe output reachable domain $\mathcal{O}_u = \bigcup_i^m \mathcal{S}_{u,i}$ and the union of $\mathcal{E}_u$ is the exact unsafe input space $\mathcal{I}_u = \bigcup_i^m \mathcal{E}_{u,i}$. The backtracking process is denoted as $\mathcal{I}_u = \mathbb{B}(\mathcal{O}_u)$ as illustrated in Figure 6.1.

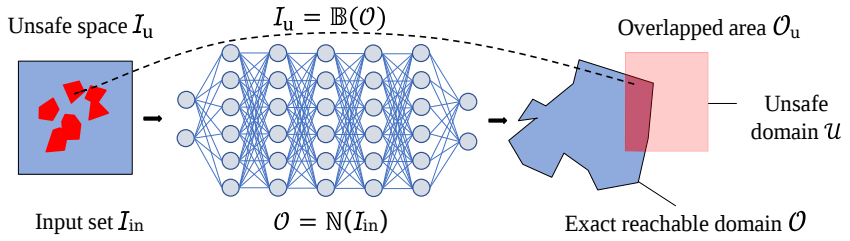


Figure 6.1: Computation of the unsafe input-output reachable domain $\mathcal{I}_u \times \mathcal{O}_u$.

In the reachability analysis above, the set representation for $\mathcal{E}_u$ and $\mathcal{S}_u$ includes their vertices. These vertices of all $\mathcal{E}_u$s and $\mathcal{S}_u$s distribute over the entire unsafe input domain $\mathcal{I}_u$ and unsafe output reachable domain $\mathcal{O}_u$, respectively. In addition, $\mathcal{E}_u$ is actually a subset of a *linear region* of the DNN. Recall that *linear region* is a maximal convex subset of the input domain to the DNN, over which the DNN is linear. Therefore, $\mathcal{S}_u$ and $\mathcal{E}_u$ have an affine mapping relation, and so do their vertices. Overall, these vertices can be utilized to

well approximate the distance between unsafe domains and safe domains in Equation 6.3 for the repair.

Despite this method can compute the unsafe input space for the DNN repair, it needs to first compute the $O(2^n)$ reachable sets. In practice, only a portion of these reachable sets violate the safety and a large amount of the computation is wasted on the safe reachable sets. Therefore, to improve the computational efficiency, it is necessary to develop a method to filter out such sets and avoid the additional computation.

Remark 6.1.1 *Our reachability analysis algorithm will require computation of $O(2^n)$ reachable sets with n ReLU neurons in $\mathbb{N}(\cdot)$ and $\mathbb{B}(\cdot)$. We aim to develop an over-approximation method to verify the safety of reachable sets of each layer, such that we can filter out the safe set that will not violate safety properties and avoid unnecessary subsequent computation.*

To solve this problem, we propose an algorithm that integrates an over-approximation method with the exact analysis method. Over-approximation methods can quickly check the safety of an input set to DNNs. The integration is done as follows. Before an input set $\mathcal{S}$ is processed in a layer $\mathbb{L}(\cdot)$, its safety will be first verified with the over-approximation method. If it is safe, it will be discarded, otherwise, it continues with the exact reachability method. Suppose there are m ReLU neurons involved in the computation in Equation 4.8, then it can generate $O(2^m)$ reachable sets whose computation can be avoided if $\mathcal{S}$ is verified safe. The integration of the over-approximation algorithm also improves the memory footprint of the algorithm, since many sets are discarded early in the process.

Another problem is the memory-efficiency issue. The computation may take up a tremendous amount of the computational memory, may even result in out-of-memory issues, due to the exponential explosion of sets. To solve this problem, the algorithm above is designed with the depth-first search, by which the memory usage can be largely reduced. The details of the over-approximation algorithm are presented in Section 6.3.3.

6.2 Framework for DNN Repair

6.2.1 Formulation of Loss Function for DNN Repair in Problem 1

The primary idea of this loss function is to construct the distance between its unsafe reachable domain and the safe domain with a loss function of the parameter set θ. By minimizing this distance, the unsafe reachable domain can be gradually eliminated. The loss function can be formulated as

$$\mathcal{L}_u(\theta) = \sum_{i=1}^{n} dist(\mathcal{O}_{\mathrm{u}}(\mathcal{I}^{[i]}, \mathcal{U}^{[i]}, \theta), \overline{\mathcal{U}^{[i]}}) \tag{6.1}$$

where $\mathcal{I}^{[i]}$ and $\mathcal{U}^{[i]}$ are the input domain and output unsafe domain specified by the property P_i, the function $dist$ computes the distance between the identified unsafe reachable domain $\mathcal{O}_u$ and the safe domain $\overline{\mathcal{U}}$.

This distance is designed to be the minimum distance of each $y \in \mathcal{O}_u$ to the safe domain $\overline{\mathcal{U}}$, such that the modification of unsafe behaviors can be minimal. This minimum l-norm distance of $y \in \mathcal{O}_u$ to the safe domain can be formulated as $\min_{\hat{y}\in\overline{\mathcal{U}}} \|y(x,\theta) - \hat{y}\|_l$.

The common strategy of related works which aim to repair or improve the safety of DNNs with reachability analysis Wong and Kolter (2018); Mirman et al. (2018); Lin et al. (2020) is by considering the largest distance which is formulated as

$$dist: \quad \max_{y\in\mathcal{O}_u} \min_{\hat{y}\in\overline{\mathcal{U}}} \|y(x,\theta) - \hat{y}\|_l. \tag{6.2}$$

However, the issues are twofold with this approach. First, it is unknown whether the unsafe domain $\mathcal{O}_u$ is convex or concave, which makes it significantly challenging to convert the computation of the exact largest distance to an LP problem. Secondly, Their solutions are based on the over-approximation of the output unsafe reachable domain by linearization of nonlinear activation functions. As a result, the approximation error is accumulated with neurons and it can be so conservative that a low-fidelity approximated distance may result in significant accuracy degradation. This remark is demonstrated in our experimental evaluation and comparison with the related work ART Lin et al. (2020).

Different from these strategies, our reachability analysis method can obtain the exact unsafe reachable domain $\mathcal{O}_u$ efficiently. As introduced in Sec. 6.1.3, this is achieved by computing the exact unsafe *linear region*s $\{\mathcal{E}_u\}_{i=1}^m$ and their unsafe output reachable sets $\{\mathcal{S}_u\}_{i=1}^m$, where $\mathcal{S}_u{=}\mathbb{N}(\mathcal{E}_u)$. The vertices of each pair of domains $\mathcal{E}_{(u,i)} \times \mathcal{S}_{(u,i)}$ can be denoted as $V_i : x \times y$. The $y \in \mathcal{O}_u$ having the largest distance in Equation 6.2 is also included in $\bigcup_{i=1}^m V_i$ which contains all the extreme points of $\mathcal{O}_u$. Here, instead of identifying the y, we choose to apply all the vertices to Equation 6.1. This enables us to avoid searching the y in $\bigcup_{i=1}^m V_i$, which significantly reduces computation time. More importantly, since these vertices distributes over the entire $\mathcal{O}_u$ and contains all its corner points, they encode more geometrical information of this domain and hence are more representative than a single point y which only captures its largest distance to the safe domain $\overline{\mathcal{U}}$. Therefore, we substitute the $dist$ function in Equation 6.1 with a more general formulation:

$$dist: \quad \sum_{i=1}^{m} \sum_{j=1}^{|V_i|} \min_{\hat{y}\in\overline{\mathcal{U}}} \|y_j(x_j,\theta) - \hat{y}\|_l \tag{6.3}$$

where $|V_i|$ denotes the vertices set's cardinality.

In the following, we will explain our approach of approximating the closest safe $\hat{y}$ to the unsafe y. Recall that the unsafe output domain $\mathcal{U}$ defined in a safety property is either a convex set formulated as $Ax + b \leq 0$ or a non-convex domain consisting of multiple such convex sets. Therefore, the problem of finding $\hat{y}$ can be encoded as an LP problem of finding the $\hat{y}$ on the boundaries of $\mathcal{U}$ whose distance to the interior y is minimal,

Algorithm 5 DNN Repair

Input: N, $\{P\}_{i=1}^{m}$, $(x,y)_{training}$ #an unsafe DNN, safety properties, training data
Output: N' #an safe DNN satisfying all its safety properties.

1: **procedure** N' = REPAIR(N)
2: $N' \leftarrow N$
3: **while** N' is not safe on $\{P\}_{i=1}^{m}$ **do**
4: $\mathcal{D}_{unsafe}$ = reachAnalysis(N, $\{P\}_{i=1}^{m}$) #compute unsafe data domains
5: $\mathcal{L}_u$ = Dist($\mathcal{D}_{unsafe}$) #approximate the distance using Equation 6.3.
6: $\mathcal{L}_c$ = Loss($(x,y)_{training}$) #compute loss on the training data in Equation 6.5
7: N' = Update(N', $\mathcal{L}_u$, $\mathcal{L}_c$) #learn through the loss function in Equation 6.6

where the optimal $\hat{y}$ is located on one of its boundaries along its normal vector from y. Let the vector from y to $\hat{y}$ along the normal vector be denoted as Δy. Then, the problem of finding $\hat{y}$ can be formulated as

$$\hat{y} = y + (1+\alpha)\Delta y, \quad \min_{\hat{y} \notin \mathcal{U}} \|y - \hat{y}\| \tag{6.4}$$

where α is a very small positive scalar to divert $\hat{y}$ from the boundary of $\mathcal{U}$ into the safe domain.

6.2.2 Formulation of Loss Function for the Minimal DNN Repair in Problem 2

The minimal repair problem is posed as a multi-objective optimization problem. In addition to the optimization for the repair problem explained in the last section, the minimal change of the DNN parameter θ is also considered in the problem formulation. For the minimization of the change, one simple and promising approach is to apply the training data in the retraining process of repair. Let the training input-output data be denoted as $\mathcal{X} \times \mathcal{T}$, then the function for measuring parameter change can be formulated as

$$\mathcal{L}_c(\theta) = \sum_{i=1}^{N} \|y_i(x_i, \theta) - t_i\|_l \tag{6.5}$$

where $(x,t) \in \mathcal{X} \times \mathcal{T}$. Here, we combine the function $\mathcal{L}_u$ for repair in Equation 6.3 and the function $\mathcal{L}_c$ in Equation 6.5 into one composite loss function using the weighted sum, and the minimal repair process can be formulated as

$$\underset{\theta}{\text{minimize}} \Big(\alpha \cdot \mathcal{L}_u(\theta) + \beta \cdot \mathcal{L}_c(\theta) \Big) \tag{6.6}$$

where $\alpha, \beta \in [0,1]$ and $\alpha + \beta = 1$. The configuration $\alpha = 1$, $\beta = 0$ indicates only the repair process, while $\alpha = 0$, $\beta = 1$ indicates the process does not include the repair but only the minimization of the parameter change.

The process of the DNN repair is described in Algorithm 5. Given an unsafe DNN candidate, in each iteration, its unsafe domain over safety properties are first computed. With the unsafe domain, the distance in

Equation 6.3 is then computed. Finally, together with the loss value on the training data, the total loss value in Equation 6.6 is computed and used to update the DNN parameters. The algorithm will be terminated when the DNN is verified safe or the iteration exceeds the maximum number.

6.2.3 Repair for Deep Reinforcement Learning

DRL is a machine learning technique where a DNN agent learns in an interactive environment from its own experience. Our method aims to repair an agent which violates its safety properties while its performance is maintained as defined in Problem 6.1.3. In each *time step* of DRL, the agent computes the action and the next state based on the current state. A reward is assigned to the state transition. This transition is denoted as a *tuple* $\langle s, a, r, s' \rangle$ where s is the current state, a is the action, s' is the next state, and r is the reward. Then, this tuple together with previous experience is used to update the agent. The sequence of *time step*s from the beginning with an initial state to the end of the task is called an *episode*. The DRL algorithm in this work considers one of the most popular algorithms, the deep deterministic policy gradients algorithm (DDPG) Lillicrap et al. (2015) and is utilized on the rocket-lander benchmark [1] inspired by the lunar lander Brockman et al. (2016).

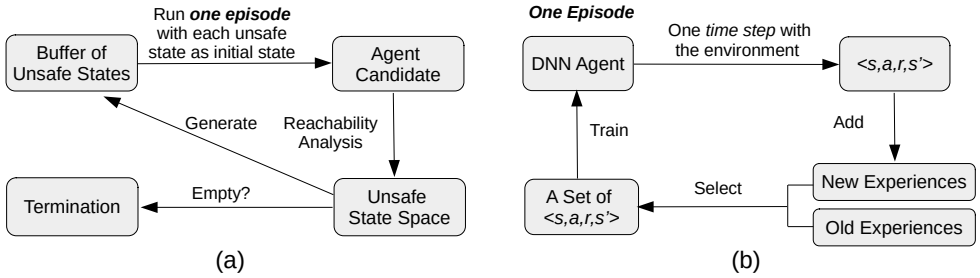


Figure 6.2: Repair framework for deep reinforcement learning. In the loop (a), given an agent, its unsafe state space is first computed with our reachability analysis method. Then, *eposide*s in (b) are run with unsafe states as initial states to update the agent, where the occurrence of unsafe states will be penalized.

Our repair method for DRL is demonstrated in Figure 6.2. Given an unsafe agent candidate in Figure 6.2(a), our reachability analysis method computes the unsafe state domain that leads to an unsafe action by the agent. The vertices of unsafe *linear region*s are selected as representative unsafe states for the unsafe domain. Instead of minimizing its distance to the closest safe state as proposed for the regular repair, we run one *episode* with the unsafe state as an initial state, as shown in Figure 6.2(b). In this process, a penalty r is applied to the unsafe action observed in the learning process, from which safety can be more naturally learned. The penalty r is normally set to the least reward in the old experience, where the *old experience* refers to the experience from learning the original unsafe agent. In the repair process, the $tuple$ in each *time step* will be stored into a global buffer for previous experience, which is named *new experiences*. For training,

[1] https://github.com/arex18/rocket-lander

a set of *tuple*s will be randomly selected from both experiences. The process in Figure 6.2(a) will be repeated until the agent becomes safe. The process is also described in Algorithm 6.

Algorithm 6 Repair for Deep Reinforcement Learning

Input: N, E, $\{P\}_{i=1}^{m}$ #an unsafe DNN agent, its old experience, safety properties
Output: N' #a safe agent satisfying all its safety properties

```
procedure N' = REPAIR(N)
    N' ← N
    while N' is not safe on {P}_{i=1}^{m} do
        D_unsafe = reachAnalysis(N, {P}_{i=1}^{m})   #compute unsafe state domains
        S_unsafe = Vertices(D_unsafe)   #representative unsafe states
        for s in S_unsafe do
            N' = Episode(N', s, E)   #one episode learning
```

6.3 Fast Reachability Analysis of DNNs

Fast reachability analysis is a core component in our DNN repair framework. However, different from traditional algorithms, for DNN repair the emphasis of the algorithm is on finding the unsafe input domain and its corresponding unsafe output domain. Our algorithm builds on the reachability analysis and backtracking method presented in Yang et al. (2021a). The method utilizes a FVIM set representation for efficient encoding of the combinatorial structure of polytopes. This set representation is suitable for set transformations that are induced by operations in a neural network. The reachability analysis method presented in Yang et al. (2021a) is able to compute the output reachable domain of a DNN, and subsequently identify the unsafe input regions. However, one disadvantage of the algorithm is that, in the worst case, the number of reachable sets is $O(2^n)$, where n is the number of ReLU neurons. Its efficiency could be impeded due to this computation of a huge number of sets.

To alleviate this problem, we utilize a novel over-approximation method to speed up computation in Equation 4.3 and 4.8 by filtering out safe regions in the early stages of the algorithm. Since our focus of retraining is on computing the unsafe input domain and its corresponding unsafe output domain, once we have guarantees of safety for a particular region, we do not need to compute its exact output reachable sets. Thereby, the computational efficiency and the memory footprint of the algorithm can be significantly improved.

Our over-approximation method is based on a new set representation for the linear relaxation of ReLU neurons. The new set representation named $\mathcal{V}$-zono is designed to efficiently encode the exponentially increasing vertices of reachable sets in each linear relaxation, and it is totally compatible with the FVIM. In the following section, the over approximation with $\mathcal{V}$-zono will be introduced. Additionally, to handle the memory-efficiency issue caused by the large amount of reachable sets computed in Equation 4.3 and 4.8, a

depth-first search algorithm is also presented.

6.3.1 Over Approximation with Linear Relaxation in Reachability Analysis

This section presents our over-approximation method based on the linear relaxation of ReLU. Linear relaxation is commonly used in other related works for fast safety verification of DNNs, such as Singh et al. (2019b); Gehr et al. (2018); Zhang et al. (2018). Instead of considering the two different linearities of ReLU neuron over its input in $\mathbb{E}(\cdot)$ of Equation 4.3, these works apply one convex domain to over approximate these linearities to simplify the reachability analysis, as shown in Figure 6.3. This over approximation is named *linear relaxation*. Recall the process of $\mathcal{S}'$ with the ith neuron in the layer in Section 6.1.3, when the lower bound and the upper bound of the x_i of $x \in \mathcal{S}'$ spans the two linearities bounded by $x_i = 0$, $\mathcal{S}'$ is supposed to be divided accordingly and their two subsets lying in the range $x_i < 0$ or $x_i \geq 0$ will be processed in terms of their linearity. The linear relaxation is applied only in such cases, as shown in Figure 6.3 (b) and (c). When $\mathcal{S}'$ only locates in $x_i < 0$ or $x_i \geq 0$, the computation will be the same as the computation in Section 6.1.3. We denote this process including the linear relaxation as $\mathbb{E}^{app}(\cdot)$, and the computation of one layer as $\mathbb{L}^{app}(\cdot)$. Thus, by simply substituting $\mathbb{E}(\cdot)$ with $\mathbb{E}^{app}(\cdot)$ in Equation 4.3, and substituting $\mathbb{L}(\cdot)$ with $\mathbb{L}^{app}(\cdot)$ in Equation 4.8, we can conduct the over-approximation method shown in Equation 6.7 and 6.8. Function $\mathbb{E}^{app}(\cdot)$ only generates one output set for each input set, instead of at most two sets. Therefore, given one input set to the DNN, Equation 6.8 computes one over-approximated output reachable set instead of $O(2^n)$ sets with n ReLU neurons.

$$\mathbb{L}^{app}(\mathcal{S}) = (\mathbb{E}_l^{app} \circ \cdots \circ \mathbb{E}_2^{app} \circ \mathbb{E}_1^{app} \circ \mathbb{T})(\mathcal{S}) \tag{6.7}$$

$$\mathbb{N}^{app}(\mathcal{S}) = (\mathbb{L}_k^{app} \circ \cdots \circ \mathbb{L}_2^{app} \circ \mathbb{L}_1^{app})(\mathcal{S}) \tag{6.8}$$

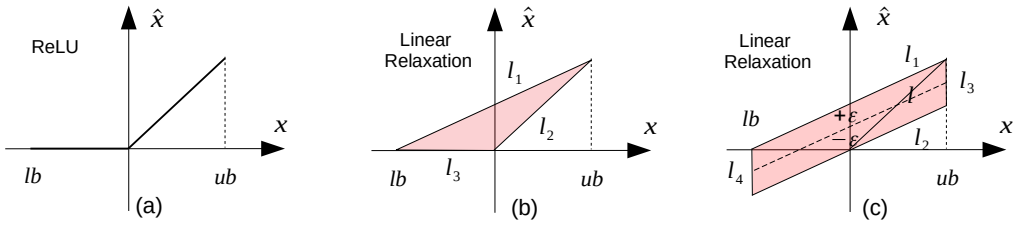


Figure 6.3: Linear relaxations of ReLU functions with a convex bound: (a) ReLU function, (b) one type of linear relaxation of ReLU function Wong and Kolter (2018), and (c) linear relaxation utilized in our over-approximation method based on a new set representation.

Here, we introduce two of the most common types of linear relaxations for ReLU functions, as shown in (b) and (c) of Figure 6.3. The linear relaxation in (b) uses the minimum convex bound. Compared to other

linear relaxations, it can over approximate the output reachable domain with the least conservativeness. A less conservative relaxation typically leads to more accurate reachability analysis algorithms. The primary challenge for this relaxation is the estimation of the lower bound lb and the upper bound ub for each ReLU activation function. This is normally formulated as an LP problem. However, since the number of variables equals to the number of activations, solving such problems with traditional methods for each verification may not be tractable.

One alternative to avoid the LP problem is to use vertices to represent a reachable set, This also enables us to easily integrate this representation with the FVIM set representation in the exact reachability analysis. One issue with this approach is that doubling of vertices in each ReLU relaxation may add a significant computation cost and memory occupation. The explanation is as follows. Suppose the relaxation is for the ith neuron. As shown in (b) and (c), the relaxation introduces an unknown variable $\hat{x}_i$ to the incoming set $\mathcal{S} \in \mathbb{R}^d$ and also the relation between x_i and $\hat{x}_i$ bounded in the convex domain.

Remark 6.3.1 *The introduction of $\hat{x}_i$ is equivalent to projecting $\mathcal{S}$ into $\hat{\mathcal{S}}$ in $(d+1)$-dimensional space. For each $x \in \mathcal{S}$, it will transform into $\hat{x} \in \hat{\mathcal{S}}$ with $\hat{x}=[x;\hat{x}_i] \in \mathbb{R}^{d+1}$. Accordingly, the faces of $\mathcal{S}$ will transform into new faces of $\hat{\mathcal{S}}$ with increasing their dimension by one. The new faces of $\hat{\mathcal{S}}$ are unbounded because of the unknown variable $\hat{x}_i$. The later intersection of $\hat{\mathcal{S}}$ with the domain of x_i and $\hat{x}_i$ in the relaxation will yield real values to $\hat{x}_i$.*

For instance, the vertex v of $\mathcal{S}$ which is a 0-dimensional face will turn into $\hat{v}=[v;\hat{x}_i]$ which is equivalent to an unbounded edge of $\hat{\mathcal{S}}$, a 1-dimensional face. With $\hat{\mathcal{S}}$ and the linear relaxation bounds of x_i and $\hat{x}_i$, $\mathbb{E}_i^{app}(\mathcal{S})$ can be interpreted as the intersection of $\hat{\mathcal{S}}$ with these bounds.

Remark 6.3.2 *The convex bounds of the ReLU relaxation consists of multiple linear constraint ls. Each $l : \alpha x + \beta \leq 0$ is one of two halfspaces divided by the hyperplane $\mathcal{H} : \alpha x + \beta$=0. The intersection of $\hat{\mathcal{S}}$ with each l is essentially identifying the subset of $\hat{\mathcal{S}}$ which is generated from division of $\hat{\mathcal{S}}$ by $\mathcal{H}$ and locates in the halfspace l.*

Take the (b) relaxation for instance which is bounded by three linear constraints l_1, l_2 and l_3. $\mathbb{E}_i^{app}(S)$ can be formulated as

$$\mathbb{E}_i^{app}(\mathcal{S}) = \hat{\mathcal{S}} \cap \{x \in \mathbb{R}^d \mid l_1 \cap l_2 \cap l_3\}. \tag{6.9}$$

As introduced above, the vertex $\hat{v}=[v;\hat{x}_i]$ of $\hat{\mathcal{S}}$ is symbolic with $\hat{x}_i$ and is equivalent to an unbounded edge. In a bounded set, an edge includes two vertices. After the intersection of $\hat{\mathcal{S}}$ with these linear constraints, $\mathbb{E}_i^{app}(\mathcal{S})$ generates a bounded subset of $\hat{\mathcal{S}}$ and meanwhile, each symbolic $\hat{v}$ will yield to two real vertices. Therefore, the vertices of $\hat{\mathcal{S}}$ are doubled from the vertices of $\mathcal{S}$.

$$\begin{cases} \mathcal{H}_1 : (ub - lb) \cdot \hat{x}_i - ub \cdot x_i + ub \cdot lb = 0 \\ \mathcal{H}_2 : (ub - lb) \cdot \hat{x}_i - ub \cdot x_i = 0 \\ \mathcal{H}_3 : x_i - ub = 0 \\ \mathcal{H}_4 : x_i - lb = 0 \end{cases} \tag{6.10}$$

To solve this problem, it is necessary to develop a new set representation that can efficiently encode the exponential explosion of vertices with ReLU relaxations. Here, we choose the relaxation in Figure 6.3(c) because the convex bound for the relaxation is a *zonotope* which can be simply represented by a set of finite vectors and be formulated as a *Minkowski sum*. An efficient representation of vertices can benefit from this simplification. The explanation is as follows. The zonotope in (c) is bounded by two pairs of parallel supporting hyperplanes, $(\mathcal{H}_1, \mathcal{H}_2)$ and $(\mathcal{H}_3, \mathcal{H}_4)$ as shown in Equation 6.10, and their linear constraints are denoted as l_1, l_2, l_3 and l_4. Since l_3 and l_4 are lower and upper bounds of the x_i in $\mathcal{S}$ and $\mathcal{S}$ itself locates in these constraints, then, the right part of the conjunction in Equation 6.9 can be replaced with $\{x \in \mathbb{R}^n \mid l_1 \cap l_2\}$. For each symbolic vertex $\hat{v}$=$[v; \hat{x}_i]$, two new vertices v_1' and v_2' can be computed from the intersection of the hyperplanes $\mathcal{H}_1$ and $\mathcal{H}_2$ with $\hat{x}_i$ involved. The vertices v_1' and v_2' are shown in Equation 6.11 where x_i equals to the v_i of v.

$$v_1' = \begin{bmatrix} v \\ \frac{ub \cdot x_i - ub \cdot lb}{ub - lb} \end{bmatrix}, \quad v_2' = \begin{bmatrix} v \\ \frac{ub \cdot x_i}{ub - lb} \end{bmatrix} \tag{6.11}$$

$$\{v_1', v_2'\} = \left\{ v_c' \pm v_v', \;\middle|\; v_c' = \begin{bmatrix} v \\ \frac{ub \cdot v_i}{ub - lb} - \frac{ub \cdot lb}{2(ub - lb)} \end{bmatrix}, \; v_v' = \begin{bmatrix} \mathbf{0} \\ \frac{ub \cdot lb}{2(ub - lb)} \end{bmatrix} \right\} \tag{6.12}$$

These two vertices can also be represented by Equation 6.12 where v_v' is a constant vector for the ReLU relaxation $\mathbb{E}_i^{app}(\mathcal{S})$. Let the vertices of $\mathcal{S}$ be V and $\mathcal{S}'$=$\mathbb{E}_i^{app}(\mathcal{S})$, then for each $v \in V$ it yields one v_c' for the vertices V' of $\mathcal{S}'$. Let the set of v_c' be denoted as V_c', then V' can be represented as Equation 6.13 where the doubled vertices are represented by plus-minus with the vector v_v'.

$$V' = V_c' \pm v_v' \tag{6.13}$$

The relaxation is illustrated by in Figure 6.4. The layer includes 2 ReLU neurons. The input set $\mathcal{S}$ is 2-dimensional with x=$[x_1, x_2]^\top \in \mathcal{S}$, and its vertices V consists of v_1, v_2 and v_3. Here, x_1, x_2 corresponds to the inputs of the first neuron and the second neuron, respectively. In this example, the process of $\mathcal{S}$ w.r.t.

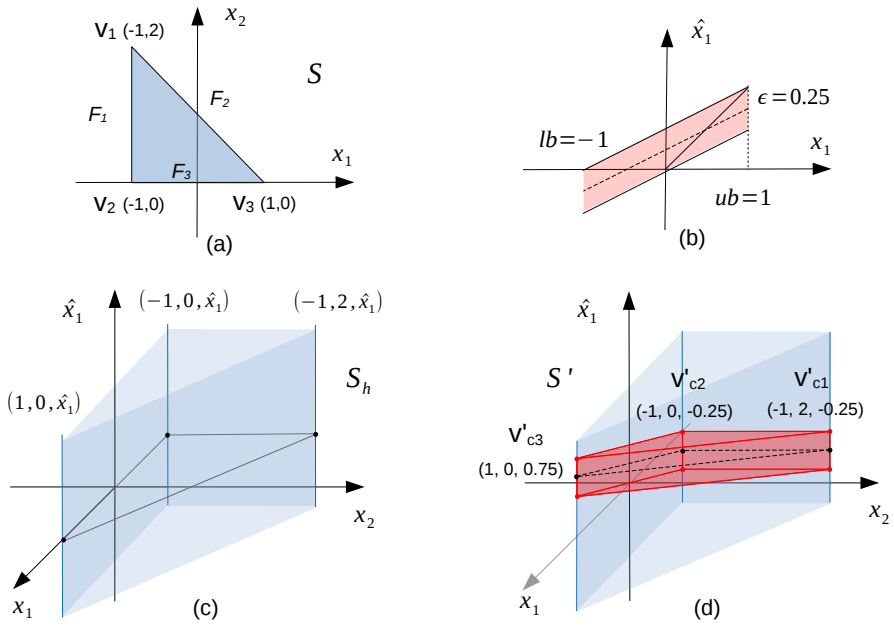


Figure 6.4: Example of the linear relaxation.

the first neuron is demonstrated. We can notice that the lower bound and the upper bound of x_1 in $\mathcal{S}$ are lb= $-$ 1 and ub=1, which indicates that $\mathcal{S}$ spans the input range of ReLU function over which the function exhibits two different linearities. Therefore, the linear relaxation is applied in terms of the subfigure (b). As introduced above, there are four linear constraints l_1, l_2, l_3 and l_4 bounding this relaxation of x_1 and $\hat{x}_1$, whose hyperplanes can be computed by Equation 6.10.

The introduction of new variable $\hat{x}_1$ projects 2-dimensional $\mathcal{S}$ into 3-dimensional $\hat{\mathcal{S}}$ with $x \in \mathcal{S}$ transforming into $\hat{x}$=$[x, \hat{x}_1]^\top \in \hat{\mathcal{S}}$. Accordingly, the vertices v_1, v_2 and v_3 of $\mathcal{S}$ transform into new symbolic vertices $[-1, 2, \hat{x}_1]^\top$, $[-1, 0, \hat{x}_1]^\top$ and $[1, 0, \hat{x}_1]^\top$ as shown in the subfigure (c), which are equivalent to three unbounded edges of $\hat{\mathcal{S}}$. After the intersection of $\hat{\mathcal{S}}$ with those linear constraints, we obtain the final over-approximated set $\mathcal{S}'$ represented by the red domain in the subfigure (d) with its 6 vertices represented by the red points. According to Equation 6.12 and 6.13, the vertices V' of $\mathcal{S}$ can be represented as

$$\begin{bmatrix} x_1 \\ x_2 \\ \hat{x}_1 \end{bmatrix} \in V', \quad V' = \left\{ v'_c \pm v'_v \;\middle|\; v'_c \in \left\{ \begin{bmatrix} -1 \\ 2 \\ -0.25 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ -0.25 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0.75 \end{bmatrix} \right\}, \; v'_v = \begin{bmatrix} 0 \\ 0 \\ 0.25 \end{bmatrix} \right\} \tag{6.14}$$

where v'_cs are denoted as $\{v'_{c1}, v'_{c2}, v'_{c2}\}$ and described by the dark points in (d).

6.3.2 Over approximation with $\mathcal{V}-$zono

In the previous section, a new set representation is preliminarily derived for the linear relaxation in Figure 6.3(c). This section mainly presents the formal definition of the set representation $\mathcal{V}$-zono, and its utilization in the over approximation of DNNs in Equation 6.7. The utilization includes the linear relaxation of ReLU neuron $\mathbb{E}^{app}(\cdot)$, the affine mapping $\mathbb{T}(\cdot)$, as well as the safety verification on the safety properties of DNNs.

The $\mathcal{V}$-zono is formally defined in Definition 6.3.1. It consists of *base vertices* $\mathcal{C}$ and *base vectors* $\mathcal{V}$. An example of the $\mathcal{V}$-zono representation is demonstrated in Figure 6.4(d). The convex set is the red domain. Its base vertices include three v_c' and the base vectors includes one v_v' as shown in Equation 6.14. Suppose $\mathcal{C}$ contains m base vertices and $\mathcal{V}$ contains n base vectors, then $\mathcal{C} \pm \mathcal{V}$ efficiently represents $m \times 2^n$ vertices in Equation 6.15.

Definition 6.3.1 ($\mathcal{V}$-zono) *In the set representation, the $\mathcal{C}$ that contains a set of finite real points $v_c \in \mathbb{R}^d$ is named as Base Vertices, and the $\mathcal{V}$ that contains a set of finite real vectors $v_v \in \mathbb{R}^d$ is named Base Vectors. Then $\langle \mathcal{C}, \mathcal{V} \rangle$ can represent a convex set $S \subset \mathbb{R}^d$ where $\mathcal{C} \pm \mathcal{V}$ encodes all its vertices.*

$$\mathcal{C} \pm \mathcal{V} = \{v_c + \sum_{i=1}^{n} (\pm v_{v,i}) \mid v_c \in \mathcal{C} \ \textit{and} \ \mathcal{V} = \{v_{v,1}, v_{v,2}, \ldots, v_{v,n}\}\} \tag{6.15}$$

6.3.2.1 Linear Relaxation with $\mathcal{V}$-zono

In the application of $\mathcal{V}$-zono in ReLU relaxation, the vertex computation in $\mathbb{E}_i^{app}(\mathcal{S})$ has been formulated as Equation 6.12, which deals with regular vertices not represented by $\mathcal{V}$-zono. Here, we will formally present the linear relaxation with $\mathcal{V}$-zono where given an input set $\mathcal{S}$ in $\mathcal{V}$-zono, the $\mathcal{V}$-zono of $\mathcal{S}' = \mathbb{E}_i^{app}(\mathcal{S})$ will be computed. Suppose the $\mathcal{V}$-zono of $\mathcal{S}$ has base vertices $\mathcal{C}$ and base vectors $\mathcal{V}$. In terms of Equation 6.12, a new base vertex v_c' can be computed for each v by

$$v_c' = \begin{bmatrix} v \\ \gamma \end{bmatrix}, \ \gamma = \frac{ub \cdot v_i}{ub - lb} - \frac{ub \cdot lb}{2(ub - lb)}, \ v \in V \text{ and } V = \mathcal{C} \pm \mathcal{V}.$$

Each v_c' is computed by incorporating one new dimension to each $v \in V$ with a real value γ, which is essentially adding one new dimension to each $v_c \in \mathcal{C}$ with γ, and adding one new dimension to each $v_v \in \mathcal{V}$ with zero. Accordingly, all the new base vertices v_c's can be computed as

$$\left\{ \begin{bmatrix} v_c \\ \gamma \end{bmatrix} + \sum_{i=1}^{n} \left(\pm \begin{bmatrix} v_{v,i} \\ 0 \end{bmatrix} \right) \ \middle| \ v_c \in \mathcal{C} \text{ and } v_{v,i} \in \mathcal{V} \right\}.$$

Based on the equation above, Equation 6.12 can be extended from the computation of new vertices v's for one v to all $v \in V$. The vertices V' of $\mathcal{S}'$ can be computed as below, from which we can derive the $\mathcal{C}'$ and $\mathcal{V}'$ as shown in Equation 6.16.

$$\left\{ \begin{bmatrix} v_c \\ \gamma \end{bmatrix} + \left(\sum_{i=1}^{n} \pm \begin{bmatrix} v_{v,i} \\ 0 \end{bmatrix} \right) \pm v'_v \;\middle|\; v_c \in \mathcal{C}, \text{ and } v_{v,i} \in \mathcal{V} \right\}$$

$$\mathcal{C}' = \left\{ \begin{bmatrix} v_c \\ \gamma \end{bmatrix} \;\middle|\; v_c \in \mathcal{C} \right\}, \quad \mathcal{V}' = \left\{ \begin{bmatrix} v_v \\ 0 \end{bmatrix}, \; v'_v \;\middle|\; v_v \in \mathcal{V} \right\} \tag{6.16}$$

From Equation 6.16, we notice that with the linear relaxation of each ReLU neuron $\mathbb{E}_i^{app}(\cdot)$, the dimension of vertices in $\mathcal{V}$-zono will be increased by one, because when introducing the new dimension or variable $\hat{x}_i$ the old dimension x_i still remains. It will result in the dimension inconsistency with the subsequent affine mapping between layers. This old dimension can be eliminated with projecting the set $\mathcal{S}'$ on it by replacing the old dimension x_i in the vertices with the new $\hat{x}_i$. The projection is reflected on Equation 6.17 with the updated $\mathcal{C}'$ and $\mathcal{V}'$.

$$\mathcal{C}' = \{v_c \mid \forall v_c \in \mathcal{C}, \; v_c[i] = \gamma\}, \quad \mathcal{V}' = \{v_v, \; v'_v \mid \forall v_v \in \mathcal{V}, \; v_v[i] = 0\} \tag{6.17}$$

After the projection, part of the points $\mathcal{C}' \pm \mathcal{V}'$ will become the actual vertices of the projected set and the rest will become its interior points. The projection of the polytope $\mathcal{S}'$ into a lower-dimensional space will generate another polytope $\mathcal{S}''$ whose every face is a projection of a face of $\mathcal{S}'$. It indicates that the vertices V'' of $\mathcal{S}''$ are from the projection of subset of the vertices V' of $\mathcal{S}'$. Since after the projection, Equation 6.17 preserves all projected vertices from Equation 6.16, there are redundant points in Equation 6.17 which are not vertices but only interior points of the projected polytope. This redundancy is allowed in the vertices representations of polytopes, as well as our set representation $\mathcal{V}$-zono. The detection and elimination of this redundancy requires an additional algorithm, which is out of the scope of this work and will be our future work. In addition, since $\mathcal{V}$-zono can efficiently encode vertices, the redundancy issue will not greatly affect the efficiency of the algorithm, which is also demonstrated in the experiments.

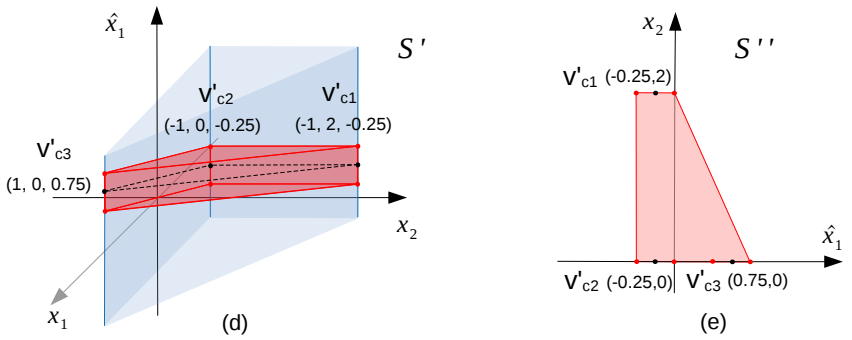


Figure 6.5: Example of the linear relaxation.

An example of the projection of the set S' in Figure 6.3(d) is shown in Figure 6.5. In the ReLU relaxation, x_1 is the old dimension and $\hat{x}_1$ is the new one. The x_1 will be eliminated by projecting the set on it, which maintains the exact bounded relation between $\hat{x}_1$ and x_2. The projection generates another polytope represented by the red domain in (e). Its $\mathcal{V}$-zono for the vertices is

$$\begin{bmatrix} \hat{x}_1 \\ x_2 \end{bmatrix} \in V'', \quad V'' = \left\{ v_c'' \pm v_v'' \;\middle|\; v_c'' \in \left\{ \begin{bmatrix} -0.25 \\ 2 \end{bmatrix}, \begin{bmatrix} -0.25 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.75 \\ 0 \end{bmatrix} \right\}, \; v_v'' = \left\{ \begin{bmatrix} 0.25 \\ 0 \end{bmatrix} \right\}. \right. \tag{6.18}$$

As shown in (d), its $\mathcal{V}$-zono contains 6 vertices denoted as red dots. After the projection as shown in (e), the new $\mathcal{V}$-zono contains 6 points which are the projections of these 6 vertices. 4 of them are actual the vertices of S'' and the rest 2 points are redundant.

6.3.2.2 Affine Mapping with $\mathcal{V}$-zono

As shown in Equation 6.7, the over approximation of reachable domain also includes affine mapping $\mathbb{T}_{(W,b)}(\cdot)$ between layers by weights W and bias b. Affine mapping on a set $\mathcal{S}$ only changes the value of its vertices. Suppose the vertices V of $\mathcal{S}$ is represented by $\mathcal{C} \pm \mathcal{V}$, then $\mathbb{T}_{(W,b)}(\mathcal{S})$ on V can be formulated as

$$\begin{aligned} W \cdot V + b &= W \cdot (\mathcal{C} \pm \mathcal{V}) + b \\ &= W \cdot \mathcal{C} + b \pm W \cdot \mathcal{V}. \end{aligned} \tag{6.19}$$

The new $\mathcal{C}'$, $\mathcal{V}'$ for new vertices V' after the affine mapping are

$$\begin{aligned} \mathcal{C}' &= W \cdot \mathcal{C} + b = \{W \cdot v_c + b \mid v_c \in \mathcal{C}\} \\ \mathcal{V}' &= W \cdot \mathcal{V} = \{W \cdot v_v \mid v_v \in \mathcal{C}\}. \end{aligned} \tag{6.20}$$

6.3.2.3 Safety Verification with $\mathcal{V}$-zono

The safety verification problem is to determine whether an output reachable domain overlaps with the unsafe domain bounded by linear constraints. Let one linear constraint be denoted as $l : \alpha^\top x + \beta \leq 0$. Suppose vertices V of the over approximated output domain are represented by $\mathcal{C} \pm \mathcal{V}$ and $\mathcal{V} = \{v_{v,1}, v_{v,2}, \ldots, v_{v,n}\}$. Then, the verification of V w.r.t. the linear constraint l can transform into checking if the minimum value of $\alpha^\top v + \beta$ over the vertices $v \in V$ is not positive, which is formulated as

$$\text{minimize}(\alpha^\top(\mathcal{C} \pm \mathcal{V}) + \beta). \tag{6.21}$$

The internal formula can be extended as

$$\alpha^\top(\mathcal{C} \pm \mathcal{V}) + \beta = \alpha^\top\mathcal{C} + \beta \pm \alpha^\top\mathcal{V} = \alpha^\top\mathcal{C} + \beta + \sum_{i=1}^{k} \pm(\alpha^\top v_v).$$

Then for each $v_c \in \mathcal{C}$, we can compute its minimum $\alpha^\top v_c + \beta + \sum_{i=1}^{k}$ -$|\alpha^\top v_v|$. By computing the minimums of all $v_c \in \mathcal{C}$, we can determine the global minimum value in Equation 6.21 and thus complete the safety verification.

6.3.3 Fast Computation of Unsafe Input Spaces of DNNs

In the previous sections, an over-approximation method for the reachability analysis of DNNs based on the linear relaxation of ReLU neurons is developed. The method is formulated as Equation 6.7 and 6.8. As introduced, it is utilized to integrate with the exact analysis method formulated in Equation 4.3 and 4.8 to filter out all the safe intermediate sets that are computed in each $\mathbb{L}(\cdot)$ and are not necessary for the computation of unsafe input spaces for DNNs. In this section, the algorithm for such integration will be presented. This algorithm is based on the depth-first search to handle the memory-efficiency issue due to large amount of reachable sets computed in each layer.

Algorithm 7 describes the integration of the computation of unsafe input spaces of DNNs with the proposed over-approximation method. Given an input set $\mathcal{S}$, it can compute all the unsafe input spaces w.r.t. the safety properties of the DNN. The details of each function are as follows.

1. Function **Reach**$(\cdot)$ is a recursive function which, in each recursion, computes only one of unsafe input sets generated from the last layer to reduce the burden on the computational memory. Its base case is Line 2-4 where the computation reaches to the last layer of the DNN and unsafe input spaces will be computed. The recursion depth is the number of the DNN layers

2. Function **outputOverApp**$(\cdot)$ over approximates the output domain of the DNN for each input set to

one layer. The input set is an output of the preceding layer computed with the exact analysis method in Equation 4.3 and 4.8. This function corresponds to Equation 6.7 and 6.8. The details are shown in Algorithm 8. In the beginning, the set $\mathcal{S}$ represented by FVIM is transformed into the $\mathcal{V}$-zono in Line 2. Subsequently, in each layer, it will be first processed by the affine mapping in Line 4 which corresponds to the $\mathbb{T}(\cdot)$ in Equation 6.7 and the computation of $\mathcal{C}'$ and $\mathcal{V}'$ in Equation 6.20. Then, in Line 5, it will be processed with ReLU neurons in the layer based on the linear relaxation. This process corresponds to the $\mathbb{E}^{app}$ in Equation 6.7 and also the computation of $\mathcal{C}'$ and $\mathcal{V}'$ in Equation 6.17.

3. Function **safetyCheck**$(\cdot)$ checks the safety of the over-approximated output domain computed in Line 6 w.r.t. safety properties. This function corresponds to Equation 6.21 where the vertices of the output domain are checked with each linear constraints of the unsafe domain defined in the properties. It returns *safe* if no vertices satisfy all the constraints. In this case, the computation in the following layers can be abandoned because $\mathcal{S}$ will not lead to safety violation.

4. Function **layerOutput**$(\cdot)$ computes the reachable sets for the current layer with input sets computed from the previous layer, which corresponds to Equation 4.3 and 4.8. All sets in this computation are represented by FVIMs. The details are shown in Algorithm 9.

The computational complexity of Algorithm 7 is that given an input set to a DNN with n ReLU neurons, $O(2^n)$ output reachable sets will be computed. In practice, the utilization of the over approximation method can significantly reduce the amount and improve the computational efficiency. The experimental results indicate that Algorithm 7 can be around five times faster than the algorithm without the over-approximation method.

Algorithm 7 Computation of unsafe input spaces of a neural network

Input: $\mathcal{S}$ # one reachable set
Output: $\mathcal{D}_u$ # unsafe input-output reachable domain of the DNN

1: **procedure** $\mathcal{D}_u$ = REACH(S, *layer*) # $\mathcal{S}$: an input set; $layer$: the layer index
2: **if** *layer* == *lastlayer* **then** # *lastlayer*: the ID of the last layer
3: $\mathcal{D}_u$ = Backtrack(S) # the unsafe domain in $\mathcal{S}$
4: **return** $\mathcal{D}_u$
5: $\mathcal{O}_o$ = outputOverApp(S) # over approximated output domain of the DNN
6: **if** safetyCheck($\mathcal{O}_o$) **then**
7: **return** None
8: $\mathcal{O}_l$ = layerOutput(S, *layer*) # $\mathcal{O}_l$: output reachable sets of the current layer for S
9: $\mathcal{D}_u = empty$
10: **for** S_l in $\mathcal{O}_l$ **do**
11: $\mathcal{D}_u$.extend(Reach(S_l, *layer*+1))
12: **return** $\mathcal{D}_u$

Algorithm 8 Reachable-domain Over approximation of a neural network

Input: $\mathcal{S}$ # one input set to the current layer
Output: $\mathcal{O}_o$ # one over approximated output reachable set of the current layer

procedure $\mathcal{O}_o$ = OUTPUTOVERAPP(S, *layer*)
  S = Vzono(S) # transform the FVIM representation of $\mathcal{S}$ to the $\mathcal{V}$-zono
  for *layer* = 1:*lastlayer* **do**
    S = affineMapping(S, *layer*) # update base vertices and base vectors
    S = reluLayerRelaxation(S) # relaxation of ReLU neurons
  return $\mathcal{O}_o \leftarrow S$

Algorithm 9 Reachable set computation of one layer

Input: $\mathcal{S}$ # one input set to the current layer
Output: $\mathcal{O}_l$ # output reachable sets of the current layer

procedure $\mathcal{O}_l$ = LAYEROUTPUT(S, *layer*)
  S = affineMapping(S, *layer*)
  $\mathcal{O}_l$=reluLayer(S)) # compute exact reachable sets with each ReLU neuron
  return $\mathcal{O}_l$

6.4 Experiments and Evaluation

In this section, we evaluate our repair methods with two benchmarks. One is the DNN controllers for the Airborne Collision System X Unmanned (ACAS Xu) Julian et al. (2016). Our repair method in Section 6.2.1 is evaluated against the work ART Lin et al. (2020). We also study the different performance between our non-minimal repair method in Equation 6.1 and our minimal repair in Equation 6.5. To measure the impact of repair algorithms on DNNs, besides the accuracy on finite test data, we also analyze the changes of the DNN's reachability. The other benchmark is a set of DNN agents for a rocket lander system based on the lunar lander Brockman et al. (2016). With this benchmark, we explore our repair method proposed in Section 6.2.3 to repair unsafe DNN agents for DRL. The hardware for all experiments is Intel Core i9-10900K CPU @3.7GHz×, 10-core Processor, 128GB Memory, 64-bit Ubuntu 18.04.

6.4.1 Repair of ACAS Xu Neural Network Controllers

The ACAS Xu DNN controllers consist of an array of 45 fully-connected ReLU DNNs. They are used to approximate a large lookup table that converts sensor measurements into maneuver advisories in an airborne collision avoidance system, such that they can significantly reduce the massive memory usage and also the lookup time. All DNNs have the same architecture which includes 5 inputs, 5 outputs and 6 hidden layers with each containing 50 ReLU neurons. The 5 inputs correspond to the sensor measurement of the relative dynamics between the ownship and one intruder. The 5 outputs are prediction scores for 5 advisory actions. There are 10 safety properties defined, and each neural network is supposed to satisfy a subset of them.

Among these 45 network controllers, there are 35 unsafe networks violating at least one of the safety

properties. Some works Lin et al. (2020); Katz et al. (2017) report there are 36 unsafe networks due to numerical rounding issues Wang et al. (2018b). Since the original dataset is not publicly available, we uniformly sample a set of 10k training data and 5k test data from the state space of DNNs, the same strategy as ART Lin et al. (2020). To repair these unsafe networks, our minimal repair and ART Lin et al. (2020) require those datasets, while our non-minimal repair approach does not.

The parameter configurations for the retraining process of a DNN in our repair are as follows. For non-minimal repair, the learning rate $lr = 0.001$. The learning rate for our non-minimal repair normally needs to be a small, because the retraining of the DNN is only guided by the modification of unsafe behaviors and a large value may also greatly affect other safe behaviors. For the minimal repair which is a multi-objective optimization problem, a set of configurations are applied to estimate the optimal performance. Here, the learning rate lr and the value (α, β) in Equation 6.6 are set as below.

	Learning Rate (lr)	(α, β)
Non-minimal Repair	0.001	-
Minimal Repair	{0.01, 0.001}	{(0.2,0.8),(0.5, 0.5),(0.8,0.2)}.

There are 6 different settings for the minimal repair of each unsafe network. The optimal result is selected for performance comparison. The loss functions in Equation 6.1 and 6.5 are computed with the Euclidean norm. As introduced, each iteration of our repair consists of the reachability analysis and epochs of retraining. Here, we empirically set the maximum iteration to 100 and the number of epochs to 200 for all our repair methods. For ART Lin et al. (2020), their default settings are applied for the comparison.

6.4.1.1 Success and Accuracy

The experimental results are shown in Table 6.1. Table 6.1 describes the repair successes and the accuracy of repaired networks. Recall that the test data are sampled from the original network, therefore, the accuracy of the original network on these data is 100%. As shown, in terms of success, our non-minimal repair and minimal repair methods both successfully repair all 35 unsafe networks. ART can repair 33 networks. While ART with refinement which computes tighter approximation than ART can repair all the networks. In terms of accuracy, our repaired networks exhibit a higher accuracy than ART and some of our repaired networks even have 100% accuracy, indicating less performance degradation. We hypothesize that the difference of performance in ART is primarily due to the over-approximation method for the unsafe domains of DNNs. It can be so conservative that the estimated distance in Equation 6.1 will be inaccurate, resulting in performance degradation. It can be also noticed that with the refinement in ART which computes a tighter approximation of domains, the number of repair successes and their accuracy increase. Overall, with exact reachability

analysis, our methods can outperform ART in terms of accuracy.

For our non-minimal repair and minimal repair methods, we can notice that the accuracy difference of their repaired networks in Table 6.1 is trivial. Recall that our minimal repair can have the Pareto optimality issue in its multi-objective optimization in Equation 6.6, while our non-minimal repair does not. It means that in the minimal repair, the minimization of the DNN deviation may impede the optimization for repair. By contrast, our non-minimal repair can consistently repair all networks with one parameter setting and meanwhile maintain the high accuracy without the usage of training data.

Methods	Repair Successes	Min Accu.	Mean Accu.	Max Accu.
Art	33/35	88.74%	94.87%	99.92%
Art-refinement	35/35	90.38%	96.23%	99.92%
Our Non-minimal Repair	**35**/35	**98.66**%	**99.74**%	**100.0**%
Our Minimal Repair	**35**/35	**99.38**%	**99.83**%	**100.0**%

Table 6.1: Repair of ACAS Xu neural network controllers.

6.4.1.2 DNN Deviation after Repair

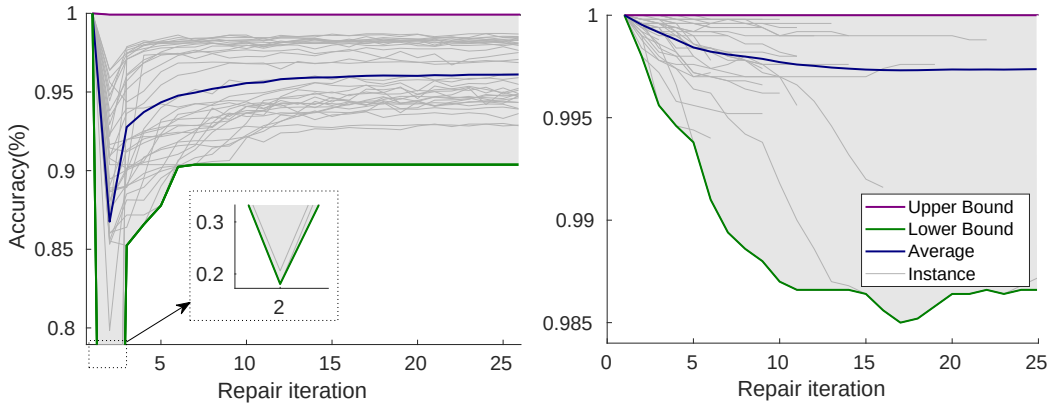
Additionally, the accuracy evolution of networks under repair is also demonstrated in Figure 6.6. It includes ART with refinement (a) and our non-minimal repair method (b). We can notice that at the beginning of ART, the accuracy of a DNN will first drop quickly and in some instances, it even drops below 20%. Then, the accuracy gradually converges to a higher value. We speculate that at the beginning of the repair, ART mainly generates a safe model with a large modification of the original network and then, train this safe network with the training data to improve the accuracy. This modification of the network parameter can result in a large parameter deviation (PD), and thus the minimal repair can hardly be enabled. This is shown by the measurement of PD in Table 6.2, where the PD of ART-repaired networks show a much larger deviation than our method. Here, PD is computed by $\|\theta' - \theta\|_2 / \|\theta\|_2$ in

$$PD = \frac{\sqrt{\sum_{l=1}^{L} \left(\sum_{i=1}^{R} \sum_{j=1}^{C} (W_{i,j}'^{[l]} - W_{i,j}^{[l]})^2 + \sum_{k=1}^{K} (b_k'^{[l]} - b_k^{[l]})^2 \right)}}{\sqrt{\sum_{l=1}^{L} \left(\sum_{i=1}^{R} \sum_{j=1}^{C} (W_{i,j}^{[l]})^2 + \sum_{k=1}^{K} (b_k^{[l]})^2 \right)}} \tag{6.22}$$

where L denotes the number of DNN layers, R and C denote the number of the row and the column of the weight matrix between each layer.

From (b), we can notice that compared to ART, our repaired networks gradually converge within 25 iterations. As shown in Table 6.2, the PDs of our repaired networks are negligible. There is also no obvious difference between PDs of repaired networks with our non-minimal repair and PDs with our minimal repair.

Besides PD, we also analyze its consequent impact on one of the critical properties of DNNs, their ex-



(a) ART with refinement

(b) Our method

Figure 6.6: Accuracy evolution of models w.r.t. the number of repair iterations. All models before the repair are unsafe on at least one safety property. All repairs successfully generate safe models in the end.

Methods	Min PD	Mean PD	Max PD
Art	3.96×10^{-2}	9.75×10^{-2}	1.77×10^{-1}
Art-refinement	2.31×10^{-2}	6.64×10^{-2}	1.17×10^{-1}
Our Method	6.75×10^{-6}	2.10×10^{-4}	9.31×10^{-4}
Our Minimal Repair	**3.56** $\times 10^{-6}$	**1.04** $\times 10^{-4}$	**4.21** $\times 10^{-4}$

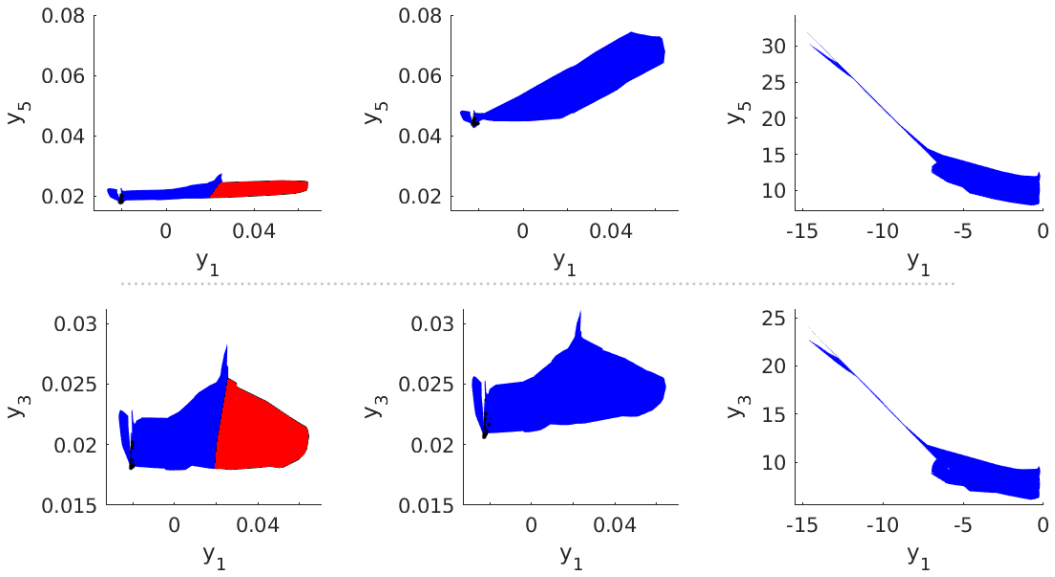
Table 6.2: Parameter Deviation (PD) after the repair.

pressibility, over the input-output continuous space. Recall that the expressibility of DNNs which refers to the capability of approximating nonlinearity in data is quantified by the number of linear regions of DNNs. A larger number of linear regions indicates a higher expressibility. Here, we identify the number of linear regions of each DNN over the input and output domains specified in all their safety properties. Then, we compute their change ratio of the linear regions of repaired DNNs to the linear regions of their original DNNs. The ratio is computed by $|N' - N|/N$ where N' represents the number of linear regions of repaired models over safety properties and N represents the number of linear regions of the original models over the same safety properties. A smaller ratio indicates a less change of the DNN expressibility after repair.

Methods	Min LRCR	Mean LRCR	Max LRCR
Art	8.21×10^{-2}	5.40×10^{-1}	1.26
Art-refinement	3.57×10^{-1}	6.32×10^{-1}	8.99×10^{-1}
Our Method	**5.00** $\times 10^{-6}$	**7.20** $\times 10^{-3}$	**3.82** $\times 10^{-2}$
Our Minimal Repair	**3.44** $\times 10^{-6}$	**1.92** $\times 10^{-2}$	**1.09** $\times 10^{-1}$

Table 6.3: Linear Regions Change Ratio (LRCR) after the repair of neural networks.

We also analyze the impact of repair on the reachability of DNNs. The reachability refers to the output reachable domain of DNNs on the input domains of their safety properties. Here, we consider the network



(a) The original (b) Our method (c) ART

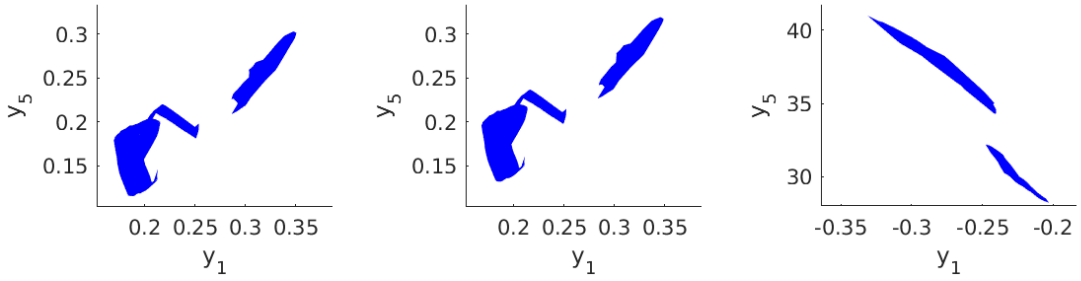
Figure 6.7: Reachability of repaired network N_{21} on Properties 1 & 2, projected on dimensions (y_1, y_5) and (y_1, y_3). It shows the output reachable domains of the original network and its repaired networks. The red area represents the unsafe reachable domain, while the blue area represents the safe domain.

N_{21} which includes safety properties 1,2,3,4. It violates Property 2 whose unsafe output domain is that y_1 is the maximum. The output reachable domain of N_{21} has 5 dimensions, and it is projected on two dimensions for visualization.

The output reachable domains of N_{21} on Properties 1 & 2, projected on (y_1, y_3) and (y_1, y_5), are shown in Figure 6.7. (a) represents the reachable domain of the original unsafe N_{21}. (b) and (c) represents the reachable domain of repaired N_{21} by our method and ART, respectively. The blue area represents the safe reachable domain, and the red area represents the unsafe reachable domain. We can notice that the unsafe domain is successfully eliminated by our method and ART. We can also notice that compared to ART, our method barely changes the reachable domain.

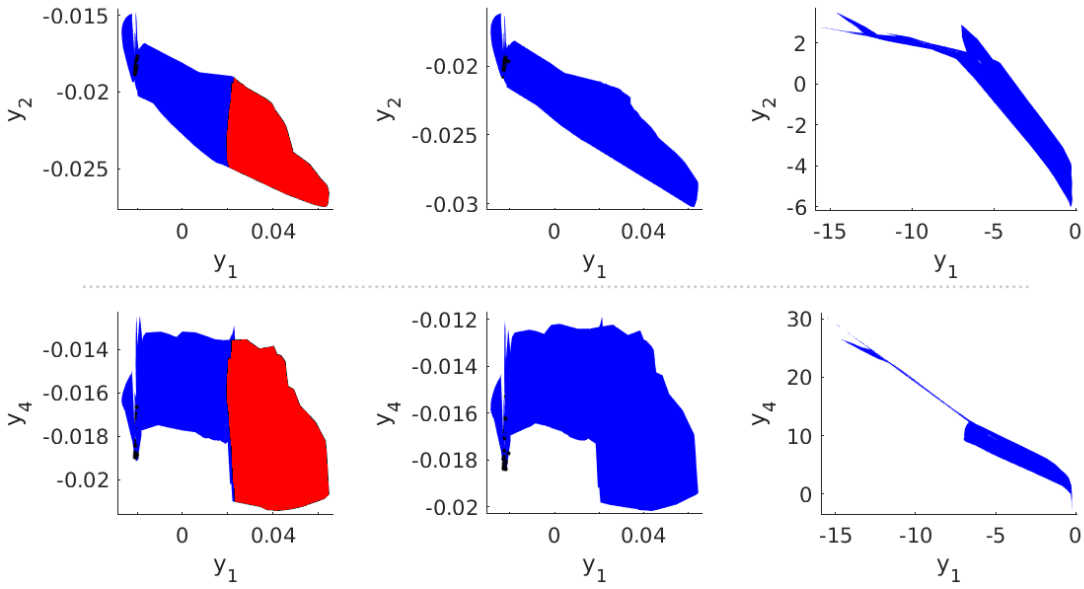
In addition, the reachability on Properties 3 & 4, projected on (y_1, y_5), is also shown in Figure 6.8. Similarly, we can notice that the impact of our repair method on the reachability of N_{21} is negligible, but ART changes the entire reachable domain. It can justify that a slight deviation on the DNN parameter may cause a tremendous change in its behaviors. It also shows that our DNN repair on one property hardly affects the DNN performance on other properties. The projection on other dimensions can be found in Figure 6.9 and 6.10

The running time of our repair method and ART is shown in Table 6.4. Here, we divide the repair of all 35 networks into regular cases and hard cases in terms of the volume of input domain of their safety



(a) The original (b) Our method (c) ART

Figure 6.8: Reachability of repaired network N_{21} on Properties 3 & 4, on which the original network is safe. The domain is projected on dimensions (y_1, y_5). We observe that due to the over approximation, ART repairs the network needlessly and changes the reachable set of the network.

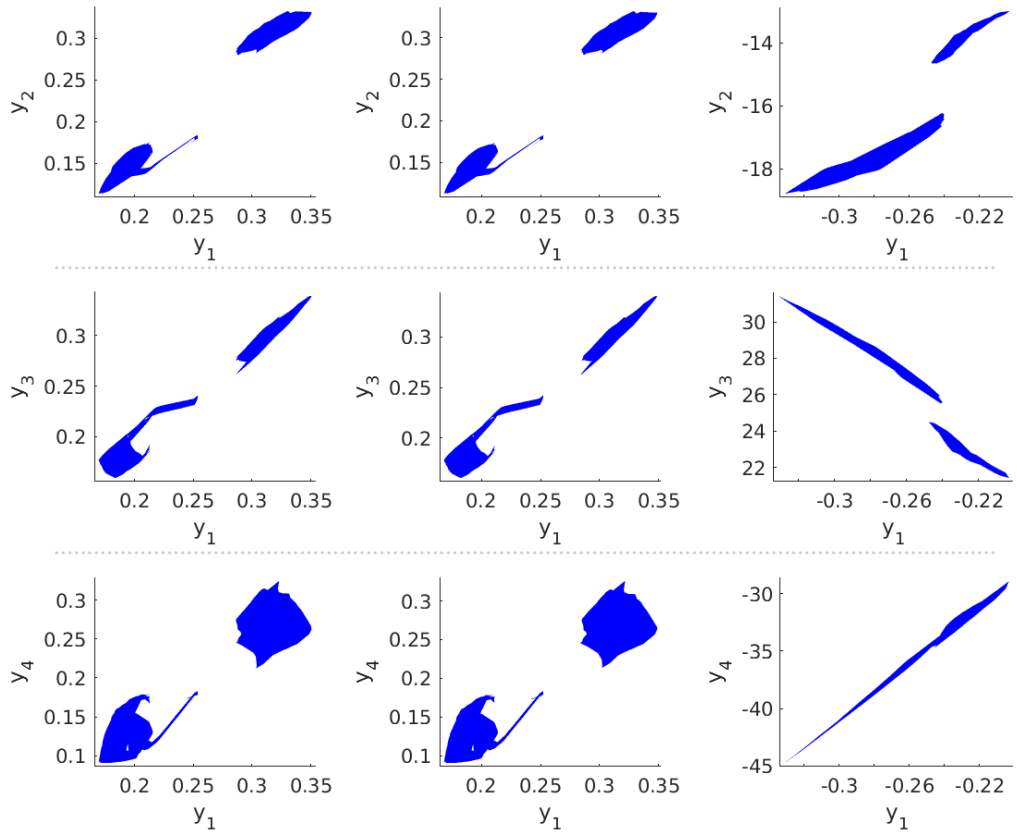


(a) The original (b) Our method (c) ART

Figure 6.9: Reachability of repaired network N_{21} on Property 1&2, projected on dimensions (y_1, y_2) and (y_1, y_4).

Methods	**Regular Cases** (33 Nets)			**Hard Cases** (2 Nets)	
	Min	Mean Time	Max	Time (N_{19})	Time (N_{29})
Art	66.5	71.4	100.3	**65.6**	**71.5**
Art-refinement	85.4	89.2	90.1	84.6	90.4
Our Method	7.4	**65.5**	230.1	7173.2	3634.9

Table 6.4: Running time (**sec**) of our repair method and ART



(a) The original (b) Our method (c) ART

Figure 6.10: Reachability of repaired network N_{21} on Property 3&4, projected on dimensions (y_1, y_2), (y_1, y_3) and (y_1, y_4).

properties. Normally, a larger input domain requires more computation for reachability analysis. The regular cases include all 33 networks whose safety properties specify small input domains. While the hard cases include the 2 networks N_{19} and N_{29} whose safety properties 7 and 8 specify large input domains. It can be noticed that our method is faster than ART in the regular cases, but slower in the hard cases. That is because that unlike the over-approximation method utilized by ART, the exact reachability analysis of networks is an NP-complete problem Katz et al. (2017). When handling hard cases, it becomes less efficient. Despite this undesired efficiency in the hard cases, the exact analysis enables our method to repair all networks with much less performance degradation than ART.

6.4.2 Rocket Lander Benchmark

The rocket lander benchmark is based on the lunar lander presented in Brockman et al. (2016). It is a vertical rocket landing model simulating SpaceX's Falcon 9 first stage rocket. Unlike the lunar lander whose action space is discrete, the action space is continuous, which commonly exists in the practical applications. Besides the rocket, a barge is also included on the sea which moves horizontally, and its dynamics are monitored. The benchmark is shown in Figure 6.11. The rocket includes one main engine thruster at the bottom with an actuated joint and two other side nitrogen thrusters attached to the sides of the top by un-actuated joints. The main engine has a power F_E ranging in $[0, 1]$ and its angle relative to the rocket body is φ. The power F_S of the side thrusters ranges in $[-1, 1]$, where -1 indicates that the right thruster has full throttle and the left thruster is turned off, while 1 indicates the opposite. The rocket landing starts in certain height. Its goal is to land on the center of the barge without falling or crashing by controlling its velocity and lateral angle θ through the thrusters.

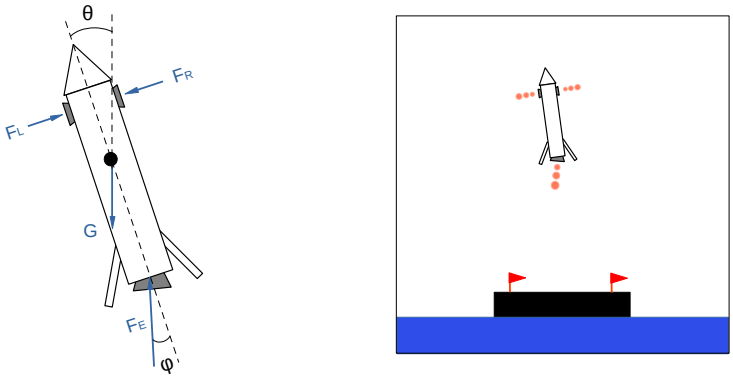


Figure 6.11: Rocket lander benchmark.

There are three actions, the main engine thruster F_E, its angle φ and the side nitrogen thrusters F_S. The original observation contains the position x and y of the rocket relative to the landing center on the barge, the velocity v_x and v_y of the rocket, its lateral angle θ, its angular velocity ω. To improve the performance

of agents, we also incorporate the last action advisory into the observation for reference. Then, the new observation can be denoted as $[x, y, v_x, v_y, \theta, \omega, F'_E, \varphi', F'_S]$. Their lower bound lb and upper bound ub are in Equation 6.23. The starting state and the reward are similar to the lunar lander The termination conditions of one *episode* includes (1) $|x| > 1$ which indicates the rocket moves out of the barge in x-space, (2) $y > 1.3$ or $y < 0$ which indicates the rocket moves out of the y-space or below the barge, (3) $\theta > 35^\circ$ which indicates the rocket tilts greater than the controllable limit.

$$\begin{aligned} lb &= [-\infty, 0, -\infty, -\infty, -\pi, -\infty, 0, -15^\circ, -1] \\ ub &= [+\infty, +\infty, +\infty, +\infty, \pi, +\infty, 1, 15^\circ, 1] \end{aligned} \tag{6.23}$$

Two safety properties are defined for the agent as below. Since reachability analysis processes bounded sets, the infinite lower bounds and upper bounds of states above will be replaced with the searched state space in the learning process of the original agent.

1. *property 1*: for the state constraints $-20^\circ \leq \theta \leq -6^\circ$, $\omega < 0$, $\varphi' \leq 0^\circ$ and $F'_S \leq 0$, the desired action should be $\varphi < 0$ or $F_S < 0$, namely, the unsafe action domain is $\varphi \geq 0 \cap F_S \geq 0$. It describes a scenario where the agent should always stop the rocket from tilting to the right.

2. *property 2*: for the state constraints $6^\circ \leq \theta \leq 20^\circ$, $\omega \geq 0$, $\varphi' \geq 0^\circ$ and $F'_S \geq 0$, the desired action should be $\varphi > 0$ or $F_S > 0$, namely, the unsafe action domain is $\varphi \leq 0 \cap F_S \leq 0$. It describes a scenario where the agent should always stop the rocket from tilting to the left.

The reinforcement learning algorithm Deep Deterministic Policy Gradients (DDPG) Lillicrap et al. (2015) is applied on this benchmark, which combines the Q-learning with Policy gradients. This algorithm is used for continuous action spaces. It consists of two models: Actor, a policy network that takes the state as input and outputs exact continuous actions rather than probability distribution over them, and Critic, a Q-value network that takes state and action as input and outputs Q-values. The Actor is our target agent controller with its safety properties. DDPG uses experience replay to update Actor and Critic, where in the training process, a set of *tuple*s are sampled from previous experiences.

Here, we first learn several agents with the DDPG algorithm. Then, we apply our framework to repair agents that violate the safety properties. The architecture of the Actor is designed with 9 inputs for states, 5 hidden layers with each containing 20 ReLU neurons, 3 outputs with subsequent $tanh$ function which maps input spaces into $[-1, 1]$. Let the three outputs before the $tanh$ be denoted as y_1, y_2 and y_3, the outputs of Actor are computed by $F_E = 0.5 \times \tanh(y_1)$+0.5, $\varphi = 15^\circ \times \tanh(y_2)$ and $F_S = \tanh(y_3)$. Our reachability analysis is applied to the architecture before the $tanh$ function, which contains only ReLU neurons. Although

the unsafe output domains defined in safety properties above are for outputs φ and F_S after the $tanh$ function, the domains $\varphi \geq 0 \cap F_S \geq 0$, $\varphi \leq 0 \cap F_S \leq 0$ are actually equivalent to $y_2 \geq 0 \cap y_3 \geq 0$, $y_2 \leq 0 \cap y_3 \leq 0$. The architecture of the Critic is designed with 12 inputs for state and action, 5 hidden layers with each containing 20 ReLU neurons, 1 output for the Q-value. The capacity of the global buffer to store previous experience is set to 4×10^5. The learning rate for Actor and Critic is set to 10^{-4} and 10^{-3}, respectively. 1000 *episode*s are performed for each learning. Overall, three unsafe agents are obtained.

For the repair, the parameters are as follows. the learning rates for Actor and Critic stay unchanged. A buffer stores all old experiences from the learning process. In addition, another global buffer is included to store new experiences with unsafe states as initial states. From these two buffers, old experiences as well as new experiences are randomly selected to form a set of training *tuple*s in Figure 6.2(b). As introduced, a new penalty reward is added for any wrong actions generated from input states. Its value is normally set to the lowest reward in the old experience. Here, the penalty is set to -30. To maintain the performance, the threshold of the change ratio which is defined in Equation 6.24 is set to -0.2.

$$\textit{Ratio} = \frac{\textit{Performance}_{(repaired\ agent)} - \textit{Performance}_{(original\ agent)}}{\textit{Performance}_{(original\ agent)}} \tag{6.24}$$

For each unsafe agent, we conduct the repair 5 times, with each repair aiming to obtain a safe agent. There are 15 instances used for evaluation. The experimental results are shown in Table 6.5 and 6.6. Table 6.5 describes the performance change ratio, the epochs of repair and the total time, where the performance of agents is evaluated by the averaged reward on 1000 episodes of running. We note that our framework can successfully repair the 3 agents in all 15 instances. In most cases, the performance of the repaired agents is slightly improved. The performance degradation in other instances is also trivial. The repair process takes 2-6 epochs for all instances with the running time ranging from 332.7 seconds to 2632.9 seconds. Also, during the repair process, we notice that repairing unsafe behaviors on one property occasionally leads to new unsafe behaviors on the other property. This is likely because the input regions defined in the properties are adjacent to each other, but their desired output regions are different, and the repaired behaviors over one input region can easily expand to other regions over which different behaviors are expected.

Next, we conduct a comparison of our new fast reachability analysis method with the method presented in Yang et al. (2021a). The results are shown in Table 6.6. In terms of computational efficiency and memory efficiency, our method outperforms this method in Yang et al. (2021a). The computational efficiency improvement of our method ranges between a maximum of 6.8 times faster and a minimum of 3.6 times faster, with an average of 4.7. The reduction in memory usage ranges between 61.7% and 70.2%, with an average of 64.5%.

ID	Agent 1			Agent 2			Agent 3		
	Ratio	**Epoch**	**Time**	**Ratio**	**Epoch**	**Time**	**Ratio**	**Epoch**	**Time**
1	+0.063	3	332.7	+0.048	3	635.7	+0.053	2	446.1
2	+0.088	3	302.0	+0.012	6	1308.4	+0.085	3	1451.6
3	+0.079	3	447.9	-0.084	4	812.9	-0.033	3	2417.1
4	+0.078	3	884.2	+0.025	3	620.3	+0.073	2	1395.3
5	+0.085	3	754.3	-0.001	4	813.5	-0.165	5	2632.9

Table 6.5: Repair of unsafe agents for the rocket lander. **ID** is the index of each repair. **Ratio** denotes the performance change ratio of the repaired agent compared to the original unsafe agent as formulated in Equation 6.24. **Epoch** denotes the number of epochs for repair. **Time** (*sec*) denotes the running time for one repair with our reachability analysis method.

ID	Agent 1				Agent 2				Agent 3			
	$\mathbf{T}_r$	$\mathbf{T}_{nr}$	$\mathbf{M}_r$	$\mathbf{M}_{nr}$	$\mathbf{T}_r$	$\mathbf{T}_{nr}$	$\mathbf{M}_r$	$\mathbf{M}_{nr}$	$\mathbf{T}_r$	$\mathbf{T}_{nr}$	$\mathbf{M}_r$	$\mathbf{M}_{nr}$
1	129.4	760.0	15.94	42.69	426.1	1583.6	13.38	36.39	700.5	2513.7	14.48	46.05
2	122.9	740.4	15.48	40.45	935.9	3352.8	13.58	37.23	618.6	2586.6	14.33	48.18
3	206.7	1361.9	16.74	45.41	572.6	2106.7	13.55	36.55	645.3	3054.0	15.13	44.91
4	329.0	2250.6	15.66	43.89	428.4	1569.7	13.67	35.97	714.4	3019.8	15.49	47.10
5	224.53	1454.2	15.32	40.77	579.5	2108.3	13.68	35.99	997.0	3277.3	15.99	48.86

Table 6.6: Comparison of our new reachability analysis method with the method Yang et al. (2021a) on computational efficiency and memory efficiency. $\mathbf{T}_r(sec)$ and $\mathbf{M}_r(GB)$ denote the computational time and the maximum memory usage of our method for all reachability analysis in one repair. $\mathbf{T}_{nr}(sec)$ and $\mathbf{M}_{nr}(GB)$ are for Yang et al. (2021a) on the same model candidate models.

In addition, we analyze the evolution of the reachable domain of candidate models in the repair. An example of Agent 1 on the first repair is shown in Figure 6.12. At the first iteration, which is before the repair, we can notice that the candidate agent has unsafe output reachable domain on Property 1 and is safe on Property 2. At the second iteration, the unsafe reachable domain becomes smaller, which indicates that the agent learns from the penalty and its action space moves towards the safe domain. After another repair, the agent becomes safe on both properties. We can also notice that during the repair, the output reachable domains do not change much, indicating that the performance of the agent is preserved.

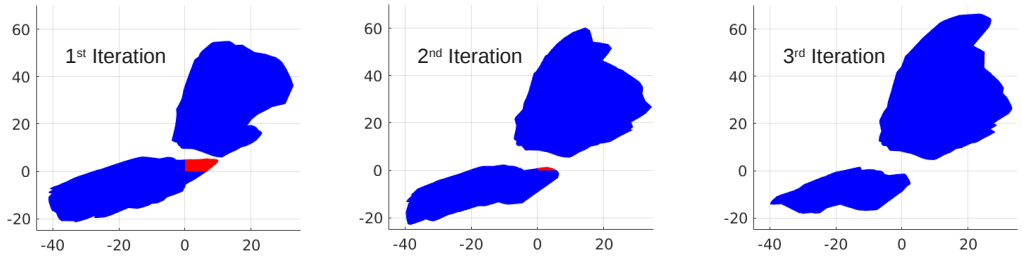


Figure 6.12: The evolution of the output reachable domain and the unsafe reachable domain in the repair of Agent 1 on the first repair. x axis represents y_2 and y axis represents y_3. The blue area represents the exact output reachable domain, while the red area represents the unsafe reachable domain which is a subset of the exact output reachable domain. The bottom left area is the reachable domain on Property 1 and the top right area is the reachable domain on Property 2.

6.5 Discussion

We have presented a reachability-based framework to repair unsafe DNN controllers for autonomous systems. The approach can be utilized to repair unsafe DNNs with only training data available. It can also be integrated into existing reinforcement algorithms to synthesize safe DNN controllers. Our experimental results on two practical benchmarks have shown that the proposed framework can successfully obtain a provably safe DNN while maintaining its accuracy and performance. We utilize a new set representation and integrate an over approximation method to improve the performance and memory footprint of our reachability analysis algorithm.

Nonetheless, safe training or repairing of DNNs with reachability analysis is still a challenging problem. There are several aspects we plan to study in the future. Firstly, computation of the unsafe set domain of larger-scale DNNs with higher dimensional inputs, such as DNNs for image classification, is still challenging. Therefore, new approaches are needed to repair such unsafe DNNs. Secondly, training of DNNs relies on appropriate meta parameters and cannot always guarantee the convergence to optimal performance, which can impose difficulties for the repair process to converge. Thus, analysis of the interaction between DNN training and its repair is necessary. Furthermore, from our observations, it becomes more difficult to repair unsafe DNNs when the input spaces of two safety properties are adjacent but with different desired output behaviors. This is due to the fact that it may be difficult for the DNN to learn an exact boundary to distinguish these adjacent input spaces and behave correctly. Therefore, a thorough study on understanding the convergence of the proposed framework is critical for enhancing its applicability to real-world applications.

CHAPTER 7

Veritex: A Tool for Reachability Analysis and Repair of Deep Neural Networks

7.1 Introduction

In this paper, we introduce a tool called Veritex that performs set-based reachability analysis of DNNs, safety certification, and repair of unsafe DNNs [1]. The reachability analysis includes the computation of both the exact and over-approximated output reachable domain for an input domain, and also the computation of the exact unsafe input space that causes the safety violation in the output. Here, the **reachable domain** contains all the possible reachable states of a system given an input bounded domain. It is a union of **reachable sets**, which refer to bounded convex polytopes. A variety of efficient set representations are utilized to construct the reachable set, such as facet-vertex incidence matrix (FVIM) Yang et al. (2021a), face lattice (FLattice) Yang et al. (2021c) and $\mathcal{V}$-zono Yang et al. (2021b). If the exact output reachable domain does not intersect with specified unsafe domains, the DNN is determined to be **safe** by Veritex. Otherwise, the DNN is **unsafe** and Veritex computes the entire unsafe input space. The repair in Veritex is a retraining process. Based on the unsafe input-output reachable domain computed in the reachability analysis, Veritex can repair an unsafe DNN on multiple safety properties with negligible impact on its original performance. Here, the **safety property** refers to specifications that describe a desired or unsafe output domain of a DNN for an input domain.

Veritex primarily supports feedforward neural networks (FFNNs) which are commonly used as controllers in learning-enabled control systems. It can perform reachability analysis, safety verification and unsafe network repair. It also supports the reachability analysis and safety verification of convolutional neural networks (CNNs). To speed up its computation, we also design a work-stealing parallel framework. It is a well known scheduling algorithm for dynamic multi-threaded computation. In the evaluation, two cases studies are pre-

[1] https://github.com/Shaddadi/veritex.git

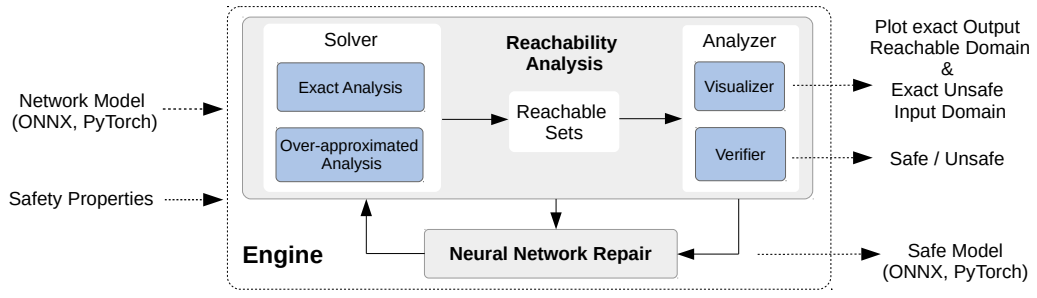


Figure 7.1: An overview of Veritex architecture.

Feature	Exact Analysis	Over-approximation Analysis
Set representations	FVIM, FLattice	$\mathcal{V}$-zono
Safety Verification	Sound and complete	Sound
Network Repair	Provably safe networks (FFNNs)	
Activation Function	ReLU	ReLU, Sigmoid, Tanh
Layer Types	FC, CONV, MaxPool, BN	
Parallel Computing	Work-stealing parallel	

Table 7.1: Overview of primary features in Veritex. FC stands for fully-connected layers. CONV stands for convolutional layer. MaxPool stands for max-pooling layer, BN stands for batch normalization.

sented. They include the safety verification and repair of ACAS Xu networks Katz et al. (2017), and an unsafe DNN agent for a rocket-lander system in DRL roc (2021). The experimental results show that Veritex has the highest efficiency in the safety verification of the ACAS Xu networks compared to all 13 related works, and that it can repair all unsafe DNNs with negligible performance degradation. Veritex currently supports DNNs with low-dimensional inputs, but maintain the possibility of supporting large-scale DNNs for image classification in the future.

7.2 Overview and Features

Veritex is an object-oriented software programmed in Python. It takes in two inputs as shown in Figure 7.1, the network model and safety properties. Veritex supports the standardized format ONNX and PyTorch for the network and the unified format VNN-LIB[2] for the safety property. In DNN verification, VNN-LIB is the emerging standard that can specify safety properties of a DNN by defining their input domains and their corresponding unsafe output domains. Roughly for specifications, it is an extension of SMT-LIB with additional assumptions. With the network model and its safety properties, Veritex can compute the exact or over-approximated output reachable domain and also the entire unsafe input space if exists. It supports the plotting of 2 or 3-dimensional polytopes. When the repair option is enabled, it will produce a provable safe network in ONNX or PyTorch format. Unlike tools era (2021); ver (2021); cro (2021); Bak (2021); Tran et al. (2020b); Katz et al. (2019), Veritex does not involve LP problems in the reachability analysis and verification of DNNs. Therefore, it does not require any commercial optimization solvers, which makes its installation straightforward. The main features of Veritex are summarized in Table 7.1.

[2]http://www.vnnlib.org/standard.html

7.2.1 Engine and Components

The engine of Veritex contains two main modules: reachability analysis of DNNs and DNN repair, as shown in Figure 7.1. The former contains functions to compute the reachable domains of a DNN. The latter contains functions to repair unsafe DNN on multiple safety properties.

7.2.1.1 Reachability Analysis Module

The module includes a solver for the computation of the reachable domain and an analyzer for the safety verification and reachable-domain visualization. The solver constructs the incoming network and its safety properties with a network object and a set of property objects. It can compute its exact or over-approximated output reachable domain. It can also compute the exact unsafe input space using the backtracking algorithm Yang et al. (2021a).

The exact analysis utilizes set representations FVIM and Flattice to compute output reachable sets whose union is the exact output reachable domain. These reachable sets can be sent to the verifier for a sound and complete safety verification, which returns either "safe" or "unsafe". The over-approximation utilizes the set representation $\mathcal{V}$-zono to over approximate the output reachable domain. This reachable domain can be sent to the verifier for a sound but incomplete safety verification, which returns either "safe" or "unknown". The visualizer plots a reachable domain by projecting it into a 2 or 3-dimensional space. This visualization is critical for the analysis of the impact of repair methods on DNN reachability.

7.2.1.2 DNN Repair Module

This module eliminates safety violations through optimization of a loss function in the retraining of a DNN. In each iteration of repair, it interacts with the reachability analysis module. Given a DNN and its violated safety properties, they are first fed into the reachability analysis module, where its exact unsafe input-output reachable domain over these properties are computed. Recall that the reachable domain consists of reachable sets, which are convex polytopes. Then, the vertices of these sets are selected as representative data pairs (x, y) to fully represent this reachable domain. They distribute over this domain, including all its extreme points. They are used to construct the distance between the unsafe reachable domain and the safe domain of the DNN. By minimizing this objective function, the repair can gradually eliminate the unsafe reachable domain, generating a provably safe DNN. When there is a safe model as a reference for the repair, adversarial x can be fed into this model to generate safe and correct $\hat{y}$ for the repair. Otherwise, $\hat{y}$ is set to the closet safe output to y for the minimal modification.

In addition to the objective function above, the repair also incorporates another objective function into the loss function, which aims to minimize the DNN parameter deviation. This is because slight changes in the

parameter can cause unexpected performance degradation. This function minimizes the difference between the predicted output of the repaired network for the training data and the true output in the training data. A weighted-sum method is applied for this multi-objective optimization problem. Two positive real-valued α and β represent the weights of each objective function and $\alpha + \beta = 1$. This repair is named the *minimal repair*. If the original dataset is not available, it can be sampled from the original network. The sampled data are purified by removing unsafe data before the training. Or users can set $\alpha = 1$ and $\beta = 0$ to transform the optimization into a single-objective optimization. Then, only the objective function for repair is considered, which is named the *non-minimal repair*.

In practice, the solving of the minimal repair is less efficient than the non-minimal repair due to the Pareto optimality issue in the multi-objective optimization, where one objective function cannot be optimized without worsening the optimization of other objective functions.

7.2.2 Work-stealing Parallel computation

In the exact analysis, different linearities that the ReLU activation function exhibits over its input ranges $x \geq 0$ and $x < 0$ are separately considered. Therefore, when an input reachable set to one ReLU neuron spans its two input ranges, this set will be divided into two subsets which are separately processed w.r.t. the linearity in that range. Afterward, these two subsets will be input sets to another neuron. Here, the state $\mathbf{S} = (\mathcal{S}, l, N)$ is defined for this computation, where $\mathcal{S}$ is a reachable set, l denotes the index of that layer, and N denotes a list of neurons in the layer that will process $\mathbf{S}$. After one neuron, the state $\mathbf{S}$ spawns at most two states $\mathbf{S}'$s with updated $\mathcal{S}'$s and N's. This state concept is also applied in the max-pooling layer. One pooling operation normally contains more linearities than the ReLU neuron and thus spawns more states. In the affine-mapping layer, such as fully-connected layer and convolutional layer, only $\mathcal{S}$ in the state will be transformed to one new reachable set $\mathcal{S}'$ accordingly.

In the work-stealing parallel computing, each processor computes their states and store additional states in a local queue for future processes. One processor becomes idle when its local queue is empty. Then this processor steals states from other processors with a globally-shared queue as the agent, such that it can enable the full use of the processors. The process of states will be terminated once they reach the end of the DNN, where different callback functions can be invoked. In this phase, the reachable set $\mathcal{S}$ in the state is an output reachable set of the DNN. The callback functions include the safety verification and the computation of unsafe input space with the backtracking algorithm.

7.3 Reachability Analysis and Set Representations

7.3.1 Reachability Analysis

The reachability analysis in Veritex includes the computation of exact or over-approximated output reachable domain and exact unsafe input subspace for a bounded input domain. This computation in FFNNs can be formulated in Equation 4.3 and 4.8 where $\mathbb{L}$ denotes the reachable-set computation in one layer, s denotes an input reachable set, $\mathbb{E}$ denotes the computation in one ReLU neuron and $\mathbb{T}$ denotes the preceding affine mapping. The reachable sets are computed layer by layer until the last layer. Similarly, in the computation of CNNs, $\mathbb{E}$ also refers to one pooling operation in the max-pooling layer, and $\mathbb{T}$ also refers to the convolutional computation or the batch normalization. The non-linearity of ReLU DNNs origins from piecewise linearity of the max function in the ReLU activation function and max-pooling operation. In the exact analysis, different linearities are separately considered for the reachable-set computation. Therefore, an output reachable set is actually the output of a linear region of the DNN. A **linear region** refers to a maximal convex subspace of the input domain, on which the DNN is linear.

7.3.2 Set Representations

The set representation encodes geometric information of a convex polytope, which directly affects the efficiency of reachability analysis. Veritex includes multiple set representations, FVIM, FLattice and $\mathcal{V}$-zono.

7.3.2.1 Facet-vertex Incidence Matrix (FVIM)

FVIM is an efficient representation to encode the combinatorial structure of a polytope. Since this set representation tracks its vertices (extreme points), any LP problems involved can be avoided. It is notable that the set representation including vertices will occupy more memory than methods involving LPs. The impact of the increased memory usage on the overall computational cost can be less than the runtime overhead from solving LPs. Thus, there is a space-time tradeoff in computational complexity.

There are two types of operations on FVIM in the reachable-set computation. The first one is the affine mapping from the weights in the fully-connected layer, the convolutional layer and the batch normalization. This operation will only modify the value of vertices but preserve the FVIM of a polytope. Therefore, its implementation in Veritex is straightforward. The other operation is the process by the max function in ReLU neurons and Max-pooling layers, whose details are discussed in Yang et al. (2021c). In brief, the vertex adjacency can be efficiently deduced from the FVIM, which facilitates the update of reachable sets in the max function.

With this representation, Veritex computes the exact output reachable domain. Furthermore, the computation also tracks the affine-mapping relation between an output reachable set and its linear region. Therefore,

Veritex can backtrack exact unsafe input subspace that causes safety violation in the output. This set representation can be only applied to simple polytopes Yang et al. (2021a). The common input interval domain to a DNN is a simple polytope. Affine mapping does not change this attribute. A reachable set computed from a simple polytope in the max function is still a simple polytope if none of its vertices lies in the boundary distinguishing the linearities of this max function. In practice, this situation extremely unlikely happens because of floating-point computation. Veritex can also detect this situation. In case, Veritex implements another set representation, Face Lattice.

7.3.2.2 Face Lattice (FLattice)

Compared to FVIM, the face lattice structure encodes the complete combinatorial structure of a polytope, describing all the containment relation between different-dimensional faces. Therefore, it is scalable to represent general polytopes. FLattice is also for the exact analysis of ReLU DNNs. The affine-mapping operation on it is the same as FVIM. Similarly, in the process of the max function, the vertex adjacency also needs to be achieved for the reachable-set update. Since FLattice has more face-containment relation to process, its efficiency is slightly lower than FVIM. This set representation also can backtrack exact unsafe input space given an unsafe output domain, the same strategy as FVIM. Overall, FLattice is compatible with the operations on FVIM in the reachable-set computation, and it is a convenient and effective alternative to address the issue in FVIM.

7.3.2.3 $\mathcal{V}-$zono

$\mathcal{V}$-zono is an enhanced vertex-representation of zonotope, which is used to construct the over-approximated reachable set in the linear relaxation of activation functions, such as ReLU, Sigmoid and Tanh. This zonotope-based reachability method computes the over-approximated output reachable domain of a DNN and can be used for sound but incomplete safety verification. Since it does not consider different linearities in the activation function, this method is faster than the exact analysis. However, the approximation error is accumulated w.r.t. each neuron, which can yield a conservative approximation. Normally, this method is used for safety verification with small input domains. Veritex also combines the exact analysis with this method in the safety verification and the computation of unsafe input-output reachable domain of DNNs. Because the over approximation method can quickly filter out spaces that does not contain unsafe elements in the beginning of its computation and significantly improve the computational efficiency.

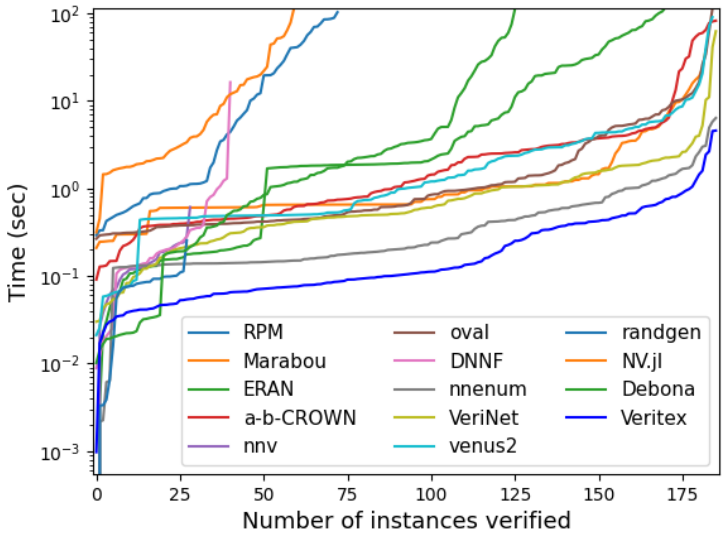


Figure 7.2: Cactus plot of the running time of the safety verification for ACAS Xu from VNN-COMP'21. The running time of failed instances which return 'unknown' or 'timeout' is not included. Timeout is 116 seconds. Compared to all the related works, Veritex exhibits the highest efficiency.

7.4 Evaluation

7.4.1 Safety Verification of ACAS Xu Networks

The performance of Veritex on the safety verification of 45 ACAS Xu networks is compared to the standardized competition results in VNN-COMP'21 Bak et al. (2021), where most of the state-of-art methods participated. All 13 methods and tools that participated are considered in the comparison. Our hardware is set to the standard configuration, AWS, CPU: r5.12$\times$large, 48vCPUs, 384 GB memory.

Veritex combines the exact analysis and the over-approximation analysis for a fast, sound and complete verification. The verification time of all 186 instances of each method is shown in the cactus-plot Figure 7.2. We can notice that compared to those 13 methods, Veritex can complete all the verification within the 116-second timeout and exhibits the highest efficiency. There are 10 methods that are over-approximation based fail to verify all the instances due to their conservativeness. There are 3 methods that can also verify all the instances within the timeout, and they are α-β-CROWN cro (2021), nnenum Bak (2021) and VeriNet Henriksen and Lomuscio (2020). In terms of the total running time, Veritex is $16.8\times$ faster, $1.8\times$ faster, and $5.0\times$ faster than these 3 methods, respectively. This is because the set representation in Veritex contains vertices of reachable sets, and thus can avoid LP problems that commonly exists in these related works. The other reason is that the incorporation of the over-approximation analysis can quickly filter out safe subspaces in the input domain and thus avoid further computation on them.

Methods	Repair Successes	Min Accu.	Mean Accu.	Max Accu.
Veritex	35/35	98.74%	99.70%	100.0%
Art	34/35	89.08%	94.57%	98.06%
Art-refinement	35/35	88.82%	95.85%	98.64%

Table 7.2: Repair of ACAS Xu neural network controllers. Veritex successfully repairs all 35 unsafe networks with little accuracy degradation.

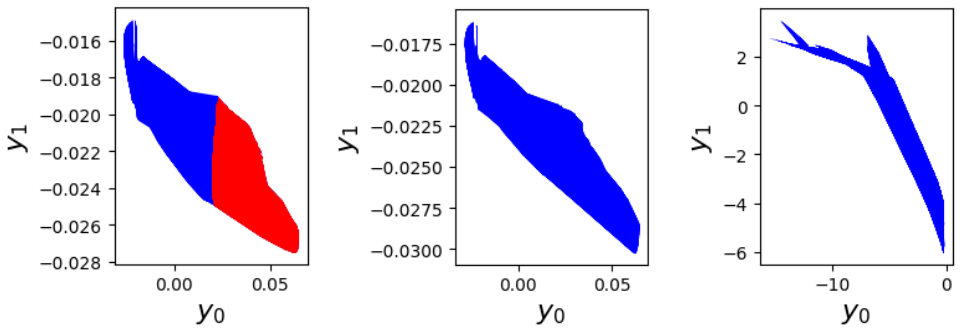
7.4.2 Repair of Unsafe ACAS Xu Networks and Unsafe DNN Agents

Among those 45 networks, there are 35 unsafe networks violating at least one of their safety properties. Their original dataset is not publicly available. Therefore, a set of 5k test data is sampled from each original network for the accuracy analysis of their repaired network, on which the accuracy of these original networks is 100%. In this case study, we apply the non-minimal repair and compare Veritex to ART Lin et al. (2020) which is a well-known repair method for DNNs.

The result of the ACAS Xu network repair is shown in Table 7.2. We can notice that Veritex can repair all the 35 unsafe networks. ART can repair 34 networks, and ART-refinement which is an improved version of ART can also repair all the networks. In terms of accuracy, our repaired networks exhibit a much higher accuracy than ART and ART-refinement. Some of our repaired networks even have 100% accuracy, showing much less performance degradation.

Besides the accuracy, we also analyze the reachability change of repaired networks, because the reachability of a network comprehensively reflects its behaviors. A desired repair should fix all safety violations of a network and meanwhile preserve its safe behaviors. Here, we apply Veritex to plot the output reachable domain of the original and repaired network N_{21} on their safety properties, and then analyze their difference. Network N_{21} has safety properties $1, 2, 3, 4$ and it violates the property 2. The output reachable domain of the original network, Veritex-repaired network and ART-refinement-repaired network on the property 1&2 is shown in (a), (b) and (c) in Figure 7.3. Their output reachable domains on the property 3&4 are shown in (d), (e) and (f). All domains are projected on (y_0, y_1) for visualization. The unsafe reachable domain is plotted in red. We can notice that the unsafe reachable domain on the property 2 is eliminated after the repair by Veritex and ART. We can also notice that compared to ART, Veritex modifies the reachability less. This is also shown by the reachability on the property 3&4 in (d), (e) and (f).

The running time of Veritex and ART is shown in Table 7.3. The repair of N_{19} and N_{29} by Veritex takes more time than ART. This is because that the exact reachability analysis of networks is an NP-complete problem Katz et al. (2017). The safety properties of these 2 networks specify very large input domains, therefore, their analysis is more computational expensive. For the repair of the other 33 networks, Veritex is



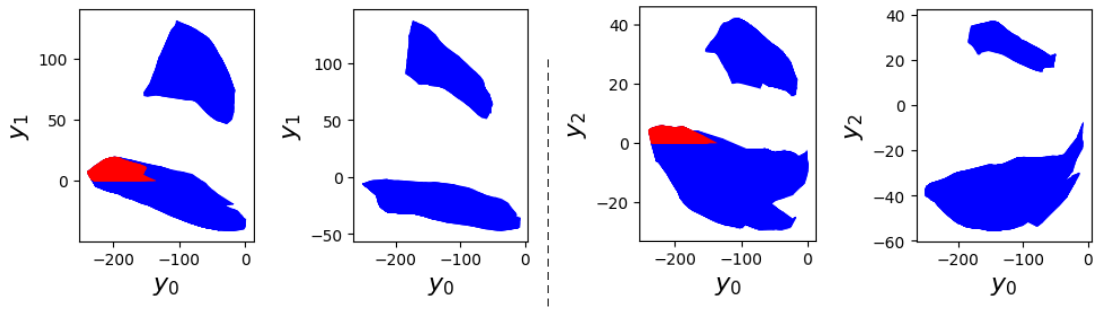
(a) Unsafe Network (b) Our Repair (c) ART Repair

(d) Unsafe Network (e) Our Repair (f) ART Repair

Figure 7.3: Reachability of the original network and the repaired networks on Properties 1&2 (a,b,c), 3&4 (d,e,f). The output reachable domains are projected on (y_0, y_1). Red area represents the unsafe reachable domain. When projected on the lower dimensional space, the unsafe reachable domain overlaps with the safe reachable domain, as shown in (a). The unsafe reachable domain is eliminated by Veritex, but the safe reachable domain is barely changed, as shown in (b).

Methods	Min	Mean Time	Max	Time (N_{19})	Time (N_{29})
Veritex	8.4	77.7	230.2	11250.7	2484.1
Art	63.6	64.6	91.7	67.5	72.6
Art-refinement	83.4	86.0	124.3	82.5	88.4

Table 7.3: Running time (**sec**) of Veritex and ART. Veritex shows a higher efficiency than ART-refinement in most of the instances.



(a) Unsafe Agent (b) Our Repair (c) Unsafe Agent (d) Our Repair

Figure 7.4: Reachability of the original agent and the repaired agent on Properties 1 & 2. The output reachable domains are projected on (y_0, y_1) and (y_0, y_2). Red area represents the unsafe reachable domain.

faster than ART-refinement.

The other case study is repairing an unsafe DNN agent for a rocket-lander system in DRL roc (2021). This agent has 9 state inputs, 5 hidden layers with each containing 20 ReLU neurons, and 3 outputs with a continuous action space. More details can be found in Yang et al. (2020). The repair by Veritex takes 304.9 seconds to produce a provable safe agent. The reachability of the original agent and our repaired agent is shown in Figure 7.4. Similar to the ACAS Xu network repair, Veritex repairs this unsafe agent without heavily affecting its original reachability. Overall, we can conclude that Veritex can efficiently repair unsafe DNNs on multiple safety properties with trivial impact on the original performance.

CHAPTER 8

Future Research and Challenges

8.1 Reachability Analysis and Verification of Large-scale DNNs

The primary challenge of the reachability analysis of large-scale DNNs origins from the more complex architecture with different types of layers such as convolutional layer and max-pooling layer and significantly more parameters involved, such as VGG16 with 144 million parameters. Since the exact analysis of DNNs is a NP-complete problem, it will require a substantial computational resource and thus become infeasible. Methods based on over approximation can be alternatives. These methods are more efficient because they can avoid exponential explore of reachable sets and generate a single set for the estimation of the reachable domain. However, the approximation error is accumulated w.r.t. each neuron, and the estimation can be conservative. Therefore, this project will aim to develop an efficient and less conservative over-approximation approach.

8.2 Reachability Analysis of Learning-enabled Control Systems

Our reachability-analysis methods will be extended to verify the safety of closed-loop dynamic systems with neural networks serving as controllers. Because of the disturbance and uncertainties are inevitable in the sensing and control channel, such control is still not reliable without formal verification. In the perspective of feedback channel, the main challenges include accumulated error when considering over-approximation methods, and exponential exploration of reachable sets when considering exact analysis. In the aspect of the plant system, the main challenge lies in the reachability analysis of non-linear dynamics.

References

(2021). alpha-beta-crown. https://github.com/huanzhang12/alpha-beta-CROWN.git.

(2021). Eran. https://github.com/eth-sri/eran.git.

(2021). Rocket-lander system. https://github.com/arex18/rocket-lander.git.

(2021). Verinet. https://github.com/vas-group-imperial/VeriNet.git.

Athalye, A., Carlini, N., and Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning*, pages 274–283. PMLR.

Bak, S. (2021). nnenum: Verification of relu neural networks with optimized abstraction refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer.

Bak, S., Liu, C., and Johnson, T. (2021). The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*.

Bak, S., Tran, H.-D., Hobbs, K., and Johnson, T. T. (2020). Improved geometric path enumeration for verifying relu neural networks. In *Proceedings of the 32nd International Conference on Computer Aided Verification*. Springer.

Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., and Misener, R. (2020). Efficient verification of relu-based neural networks via dependency analysis. In *AAAI*, pages 3291–3299.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Carlini, N. and Wagner, D. (2017). Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE.

Dutta, S., Jha, S., Sankaranarayanan, S., and Tiwari, A. (2018). Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer.

Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer.

Elboher, Y. Y., Gottschlich, J., and Katz, G. (2020). An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. (2018). Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE.

Goldberger, B., Katz, G., Adi, Y., and Keshet, J. (2020). Minimal modifications of deep neural networks using verification. In *LPAR*, volume 2020, page 23rd.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.

Grünbaum, B. (2013). *Convex polytopes*, volume 221. Springer Science & Business Media.

Hanin, B. and Rolnick, D. (2019). Complexity of linear regions in deep networks. *arXiv preprint arXiv:1901.09021*.

Henk, M., Richter-Gebert, J., and Ziegler, G. M. (2004). 16 basic properties of convex polytopes. *Handbook of discrete and computational geometry*, pages 255–382.

Henriksen, P. and Lomuscio, A. (2019). *Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search*. PhD thesis, Imperial College London.

Henriksen, P. and Lomuscio, A. (2020). Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI 2020*, pages 2513–2520. IOS Press.

Huang, X., Kwiatkowska, M., Wang, S., and Wu, M. (2017). Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer.

Hurewicz, W. and Wallman, H. (2015). *Dimension Theory (PMS-4)*, volume 4. Princeton university press.

Julian, K. D., Lopez, J., Brush, J. S., Owen, M. P., and Kochenderfer, M. J. (2016). Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE.

Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. (2019). The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer.

Kouvaros, P. and Lomuscio, A. (2018). Formal verification of cnn-based perception systems. *arXiv preprint arXiv:1811.11373*.

Leino, K., Fromherz, A., Mangal, R., Fredrikson, M., Parno, B., and Păsăreanu, C. (2021). Self-repairing neural networks: Provable safety for deep networks via dynamic repair. *arXiv preprint arXiv:2107.11445*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, X., Zhu, H., Samanta, R., and Jagannathan, S. (2020). Art: Abstraction refinement-guided training for provably correct neural networks. In *FMCAD*, pages 148–157.

Liu, C., Arnon, T., Lazarus, C., Barrett, C., and Kochenderfer, M. J. (2019). Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*.

Lomuscio, A. and Maganti, L. (2017). An approach to reachability analysis for feed-forward relu neural networks. *arXiv preprint arXiv:1706.07351*.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2017). Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*.

Majd, K., Zhou, S., Amor, H. B., Fainekos, G., and Sankaranarayanan, S. (2021). Local repair of neural networks using optimization. *arXiv preprint arXiv:2109.14041*.

Mirman, M., Gehr, T., and Vechev, M. (2018). Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586.

Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932.

Papernot, N., McDaniel, P., Wu, X., Jha, S., and Swami, A. (2016). Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE.

Pulina, L. and Tacchella, A. (2010). An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, pages 243–257. Springer.

Serra, T., Tjandraatmadja, C., and Ramalingam, S. (2017). Bounding and counting linear regions of deep neural networks. *arXiv preprint arXiv:1711.02114*.

Serra, T., Tjandraatmadja, C., and Ramalingam, S. (2018). Bounding and counting linear regions of deep neural networks. In *International Conference on Machine Learning*, pages 4558–4566. PMLR.

Singh, G., Ganvir, R., Püschel, M., and Vechev, M. (2019a). Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems*, pages 15098–15109.

Singh, G., Gehr, T., Mirman, M., Püschel, M., and Vechev, M. (2018a). Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, pages 10802–10813.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2018b). Boosting robustness certification of neural networks. In *International Conference on Learning Representations*.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019b). An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41.

Sohn, J., Kang, S., and Yoo, S. (2019). Search based repair of deep neural networks. *arXiv preprint arXiv:1912.12463*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks. In *International Conference on Learning Representations*.

Tramer, F., Carlini, N., Brendel, W., and Madry, A. (2020). On adaptive attacks to adversarial example defenses. In *Advances in Neural Information Processing Systems*, volume 33, pages 1633–1645. Curran Associates, Inc.

Tran, H.-D., Bak, S., Xiang, W., and Johnson, T. T. (2020a). Verification of deep convolutional neural networks using imagestars. In *32nd International Conference on Computer-Aided Verification (CAV)*. Springer.

Tran, H.-D., Lopez, D. M., Musau, P., Yang, X., Nguyen, L. V., Xiang, W., and Johnson, T. T. (2019a). Star-based reachability analysis of deep neural networks. In *International Symposium on Formal Methods*, pages 670–686. Springer.

Tran, H.-D., Musau, P., Lopez, D. M., Yang, X., Nguyen, L. V., Xiang, W., and Johnson, T. T. (2019b). Parallelizable reachability analysis algorithms for feed-forward neural networks. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering*, pages 31–40. IEEE Press.

Tran, H.-D., Musau, P., Lopez, D. M., Yang, X., Nguyen, L. V., Xiang, W., and Johnson, T. T. (2019c). Star-based reachability analsysis for deep neural networks. In *23rd International Symposisum on Formal Methods (FM'19)*. Springer International Publishing.

Tran, H.-D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L. V., Xiang, W., Bak, S., and Johnson, T. T. (2020b). Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 3–17. Springer.

Usman, M., Gopinath, D., Sun, Y., Noller, Y., and Pasareanu, C. (2021). Nnrepair: Constraint-based repair of neural network classifiers. *arXiv preprint arXiv:2103.12535*.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018a). Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6369–6379.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018b). Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1599–1614.

Wong, E. and Kolter, Z. (2018). Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5286–5295. PMLR.

Xiang, W., Tran, H.-D., and Johnson, T. T. (2017). Reachable set computation and safety verification for neural networks with relu activations. *arXiv preprint arXiv:1712.08163*.

Xiang, W., Tran, H.-D., and Johnson, T. T. (2018). Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783.

Yang, X., Johnson, T. T., Tran, H.-D., Yamaguchi, T., Hoxha, B., and Prokhorov, D. (2021a). Reachability analysis of deep relu neural networks using facet-vertex incidence. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, HSCC '21, New York, NY, USA. Association for Computing Machinery.

Yang, X., Tran, H.-D., Xiang, W., and Johnson, T. (2020). Reachability analysis for feed-forward neural networks using face lattices. *arXiv preprint arXiv:2003.01226*.

Yang, X., Yamaguchi, T., Tran, H.-D., Hoxha, B., Johnson, T. T., and Prokhorov, D. (2021b). Neural network repair with reachability analysis. *arXiv preprint arXiv:2108.04214*.

Yang, X., Yamaguchi, T., Tran, H.-D., Hoxha, B., Johnson, T. T., and Prokhorov, D. (2021c). Reachability analysis of convolutional neural networks. *arXiv preprint arXiv:2106.12074*.

Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. (2018). Efficient neural network robustness certification with general activation functions. In *Advances in Neural Information Processing Systems*, pages 4944–4953.

Zhang, H., Yu, Y., Jiao, J., Xing, E., El Ghaoui, L., and Jordan, M. (2019). Theoretically principled trade-off between robustness and accuracy. In *International Conference on Machine Learning*, pages 7472–7482. PMLR.