Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

VERIFICATION OF FNN WITH PIECEWISE LINEAR ACTIVATION FUNCTIONS USING CEGAR

Franz Leonard Link

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Thomas Noll

Additional Advisor:
László Antal

Aachen, 27.03.2025

Abstract

Feedforward neural networks (FNNs) are an important and widely used method to solve problems in many areas. For the verification of the correct behavior of FNNs, multiple approaches exist including verification based on exact or over-approximate reachability analysis using *star sets* as state set representation. In this work, we present a complete verification method for FNNs based on *counterexample-guided abstraction refinement* (CEGAR) that combines both types of this star-based reachability analysis and can be faster than only applying the exact analysis. This method conducts reachability analysis with over-approximation and replaces over-approximations with exact computation if unsafety and thus counterexamples are introduced by an over-approximate computation. With each replacement, the abstract reachability analysis is refined. After the presentation of an implementation of this verification method including a variety of heuristics for five common activation functions (ReLU, LeakyReLU, UnitStep, HardTanh, HardSigmoid), we test this method on different benchmarks and neural networks and compare it to the purely exact and over-approximate verification methods.

Contents

Chapter 1

Introduction

Artificial neural networks are widely used to solve problems in a variety of different areas. An important subcategory of artificial neural networks is *feedforward neural network* (FNN). This type of neural network, corresponds to sequences of affine mappings and the application of non-linear functions called activation functions. FNNs are thus complex algorithms often meant to approximate highly non-linear functions otherwise hard to compute or specify.

The verification that an FNN behaves according to its intended function is hard to perform by hand due to this complexity. In this work, we focus on verification based on reachability analysis using *star sets* (stars) [DV16, TMLM$^+$19] as a state set representation. This means, that for an input set in this representation the corresponding output is computed and verified according to a given specification. For the reachability analysis of FNNs using a specific activation function, two algorithms are presented in [TMLM$^+$19]. In [AMÁ23], the activation functions to which reachability analysis can be applied are extended from only the ReLU activation function to also include the LeakyReLU, HardTanh, HardSigmoid, and UnitStep activation functions.

These reachability algorithms compute either a representation of the exact output for the given input or an over-approximation of it. The exact method is thus complete and sound while the over-approximated reachability analysis can only verify safety but not unsafety (i.e., it is only sound). The advantage of completeness of the exact method comes with a reduction in performance compared to the over-approximated method. Thus, the existing methods are either complete but comparatively slow, or fast, but only lead to conclusive results in some cases.

We propose a new verification method based on *counterexample-guided abstraction refinement* (CEGAR) and both types of reachability analysis. We implement this method in the open-source C++ tool HyPro[1] [SÁMK17] for the aforementioned activation functions. Our method aims to be a complete alternative to verification with the exact reachability analysis with better performance. To this end, the verification method combines both the exact and over-approximated approaches.

Because over-approximation has better performance, as many computations as possible are performed with this method for an initial reachability analysis. This analysis corresponds to the abstraction in CEGAR which is then refined by exchanging over-approximated steps with exact computation. An over-approximated computation is replaced if it introduces a *spurious* counterexample. Such a counterexample does

[1] `https://github.com/hypro/hypro`

not correspond to any element of the input and does thus not imply incorrect behavior of the FNN. Both the identification of counterexamples and the identification of an operation introducing a counterexample require feasability checks. These checks are performed using the solver *Z3*[1] [dMB08].

1.1 Related Work

Stars were originally introduced as state set representation for the simulation of linear systems in [DV16, BD17]. This star-based simulation allows for efficient verification of such systems. Based on this initial definition, stars were adapted for exact and over-approximated reachability analysis of FNNs using the ReLU activation function in [TMLM+19]. This representation enables efficient computation of affine mappings and halfspace intersections. These properties allow efficient reachability analysis and thus verification in many practical cases.

In [Mas23, AMÁ23] the usable activation functions with both methods were extended to include LeakyReLU, HardTanh, HardSigmoid, and UnitStep. We use the reachability analysis with these activation functions and ReLU as the basis for our verification method. In [AÁM25], this method is further expanded to allow star-based reachability analysis of FNNs using any piecewise linear activation functions. Our method is only adapted for the specified activation functions and not all piecewise linear activation functions.

An extension to the reachability analysis of FNNs using ReLU proposed the application of *zonotope* pre-filters [TPL+21]. These zonotopes allow for fast estimation of the bounds of stars, which are essential of both exact and over-approximated reachability analysis. These pre-filters thus allow for an increased verification speed compared to the reachability analysis without them or other methods like Reluplex [KBD+17]. Note, that this performance already occurs comparing Reluplex to the method without the pre-filter. Reluplex verifies the full FNNs based on an adaption of the simplex method to handle the ReLU activation functions.

As mentioned before, the reachability analysis is refined based on counterexamples during verification with our method. Another approach also uses refinement to improve the performance of verification based on reachability analysis [Bak21]. Instead of using counterexamples to identify a relevant over-approximation, the first over-approximated computation is replaced.

HyPro [SÁMK17] is an open-source C++ library containing many common state set representations including stars. Originally, this tool was intended for reachability analysis of hybrid systems. This functionality has been extended to include stars-based reachability analysis of FNNs using the previously mentioned activation function. This library thus provides the foundations for the implement of our verification method.

1.2 Outline

In Chapter 2, the theoretical foundations for verification of FNN with CEGAR are introduced. This starts with FNNs and the activation functions that are supported by our verification method. Afterwards, the state set representation, stars, is introduced and used for the following section in which the exact and over-approximated reachability

[1] `https://github.com/Z3Prover/z3`

analysis are explained. This section additionally contains an introduction to safety checking which enables the verification with the reachability methods. The chapter is finished with a brief introduction to *Z3* and the types of formulae used in the following chapter.

In Chapter 3, our verification method using CEGAR including the implementation is introduced. This begins with the data structure of *reachability trees* used to represent an abstraction of reachability analysis. Afterwards, the algorithms for the identification of counterexamples and the operations introducing them are presented. Finally, these algorithms are combined into our verification in the last section of this chapter. Note, that for all these algorithms, multiple methods and heuristics are introduced. For some of these, the advantage gained by a heuristic relies on the algorithms introduced in a later section.

After the introduction of our verification method in the form of multiple heuristics, the fastest combination for the verification with CEGAR is determined in Chapter 4. First this combination is determined based on one benchmark. Then the performance of our new verification method is compared to the existing exact and over-approximated verification methods.

Finally, the results of this work are summaries and discussed in Chapter 5. In addition, further improvements to our method are listed in this chapter.

Chapter 2

Preliminaries

In this chapter, the foundations for the verification of FNN using CEGAR will be introduced. After a brief introduction to the notation, a formal definition of FNNs with a variety of activation functions will be provided in Section 2.1. Subsequently, the state set representation for the reachability analysis of FNNs will be introduced in Section 2.2. We use this state set representation to store sets of values (reachable sets) corresponding to the full or partial computation of an FNN.

This computation corresponds to the reachability analysis of the FNN for which two methods are introduced in Section 2.3. One of these methods is complete and sound, but slow. The other approach is fast and still sound, but no longer complete. We conclude this section by presenting a verification method for FNNs based on reachability analysis.

For this verification, the feasibility of formulas needs to be checked. We do this with the solver *Z3* which is introduced in Section 2.4 alongside a brief overview of the arithmetic formulas we use. These formulas are also the basis for the verification method using CEGAR introduced in the next chapter.

For this thesis, the real numbers $\mathbb{R}$, rational numbers $\mathbb{Q}$, integers $\mathbb{Z}$, and natural numbers $\mathbb{N}$ are relevant. $\mathbb{N}$ is defined to include 0. To represent relevant subsets of the natural numbers, $\mathbb{N}_{\circ x}$ for $\circ \in \{\geq, >\}$ and $x \in \mathbb{N}$ is used to describe all values $n \in \mathbb{N}$ with $n \circ x$. Additionally, $\mathbb{N}_+ := \mathbb{N}_{\geq 1}$ is used to describe all positive natural numbers.

Vectors $\boldsymbol{v} \in \mathbb{R}^n$ will be written as lower-case, bold letters and refer to column vectors. Thus, $\mathbb{R}^n = \mathbb{R}^{n \times 1}$ is implied. If we require a row vector we apply transposition $\boldsymbol{v}^T$ to a column vector $\boldsymbol{v}$. The entry corresponding to dimension i of $\boldsymbol{v}$ is v_i. Matrices $\boldsymbol{M} \in \mathbb{R}^{n \times m}$ will be written as upper-case, bold letters. The i-th row of a matrix $\boldsymbol{M}$ is the row vector $\boldsymbol{M}_i \in \mathbb{R}^{1 \times m}$. Similarly, column j of $\boldsymbol{M}$ corresponds to the column vector $\boldsymbol{M}_{\_,j} \in \mathbb{R}^n$. The value at position (i, j) for $1 \leq i \leq n$ and $1 \leq j \leq m$ in the matrix $\boldsymbol{M}$ will be referred to as $m_{i,j} \in \mathbb{R}$. Note, that we typically write the k-th matrix or vector in a sequence as $\boldsymbol{M}^{(k)}$ or $\boldsymbol{v}^{(k)}$ with entries $m_{i,j}^{(k)}$ and $v_i^{(k)}$.

In general, standard matrix multiplication and scalar multiplication can be assumed for all vectors and matrices. This also holds, if the multiplication symbol is omitted. Thus, $\boldsymbol{v}\boldsymbol{M} = \boldsymbol{v} \cdot \boldsymbol{M}$. The identity matrix $\boldsymbol{I}^{(n)} \in \mathbb{R}^{n \times n}$ for any $n \in \mathbb{N}_+$ is the matrix with $i_{j,k}^{(n)} = 1$ with $j = k$ and entry $i_{j,k}^{(n)} = 0$ otherwise for $j, k \in \{1, \ldots, n\}$. The standard unit vector $\boldsymbol{e}_i^{(n)}$ refers to the i-th column of $\boldsymbol{I}^{(n)}$. If the dimension n of this vector is implied by the context, we sometimes write $\boldsymbol{e}_i$. Finally, the null matrix $\boldsymbol{0}^{(n \times m)} \in \mathbb{R}^{n \times m}$ refers to a matrix where every entry is 0. We abbreviate the notation

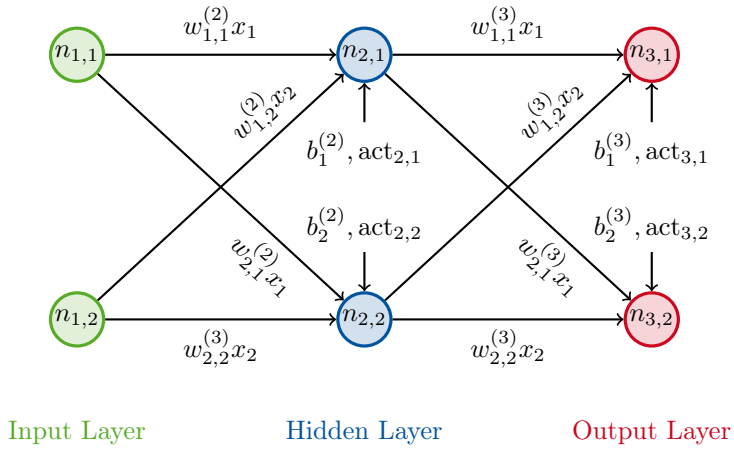


Figure 2.1: FNN with 3 layers of size 2 each

for a null vector $\mathbf{0}^{(n\times 1)}$ to $\mathbf{0}^{(n)}$.

For any two sets A and B we define the union as $\mathsf{A}\cup\mathsf{B} := \{x \mid x \in \mathsf{A} \vee x \in \mathsf{B}\}$, the intersection as $\mathsf{A}\cap\mathsf{B} := \{x \mid x \in \mathsf{A} \wedge x \in \mathsf{B}\}$, the difference as $\mathsf{A}\setminus\mathsf{B} := \{x \in \mathsf{A} \mid x \notin \mathsf{B}\}$, and the Cartesian product as $\mathsf{A}\times\mathsf{B} := \{(a,b) \mid a \in \mathsf{A} \wedge b \in \mathsf{B}\}$. The definitions for union, intersection, and the Cartesian product can be naturally extended to sequences of sets for which corresponding larger symbols are used similar to sum and product notation for numbers.

2.1 Feedforward Neural Networks

A *feedforward neural network* (FNN) [TMLM+19, AMÁ23] computes an output vector based on an input vector of potentially different dimension by propagating it *forward* through the FNN. This computation consists of the application of multiple affine mappings and so-called activation functions and is structured into layers. Each layer consists of an affine mapping followed by the application of an activation function. An exception to this structure is the first layer which is called input layer due to its correspondence to the input vector. This input layer is followed by an arbitrary number of hidden layers and a single output layer. This last layer corresponds to the output vector. Figure 2.1 is a graphic representation of this structure as well as the additional details explained in the following.

There are two ways to consider FNNs formally. Firstly, an FNN can be understood as a directed graph (V,E) [AMÁ23] composed of $L \in \mathbb{N}_{\geq 2}$ layers where the affine mapping and activation function is applied between each consecutive layer. These layers $\mathsf{l}_1,\ldots,\mathsf{l}_L$ are disjoint, ordered sets of so-called neurons forming the set of vertices $\mathsf{V} = \bigcup_{i=1}^{L}\mathsf{l}_i$. In general, the notation $\mathsf{l}_i = \{n_{i,1},\ldots,n_{i,\langle i\rangle}\}$ is used to describe a layer and its neurons. For this, $\langle i\rangle$ is the number of neurons in the layer which is also called the size of layer i. Each neuron $n_{i,j}$ for $j \in \{1,\ldots,\langle i\rangle\}$ and layer l_i with $i \in \{1,\ldots,L-1\}$ is connected to all neurons of the next layer l_{i+1}. Thus, the set of

edges is defined as follows:

$$\mathsf{E} = \bigcup_{i=1}^{L-1} \mathsf{l}_i \times \mathsf{l}_{i+1}.$$

For the affine mapping, a weight which acts as a linear factor is added to each edge. These weights are grouped into weight matrices $\boldsymbol{W}^{(i)} \in \mathbb{R}^{\langle i \rangle \times \langle i-1 \rangle}$ for each non-input layer l_i with $i \in \{2, \ldots, L\}$. A weight $w_{r,s}^{(i)}$ with $r \in \{1, \ldots, \langle i \rangle\}$ and $s \in \{1, \ldots, \langle i-1 \rangle\}$ of a weight matrix $\boldsymbol{W}^{(i)}$ is added to the edge from neuron $n_{i-1,s}$ to neuron $n_{i,r}$. Additionally, a bias $b_j^{(i)} \in \mathbb{R}$ for $j \in \{1, \ldots, \langle i \rangle\}$ is added to each neuron $n_{i,j}$. These values are grouped into a bias vector $\boldsymbol{b}^{(i)} \in \mathbb{R}^{\langle i \rangle}$ for each non-input layer l_i with $i \in \{2, \ldots, L\}$. Thus, the value $y_j^{(i)}$ of neuron $n_{i,j}$ before activation function application is the bias value added to the linear combination of the values $x_k^{(i-1)}$ of the neurons of the previous layer using the weights as coefficients:

$$y_j^{(i)} = b_j^{(i)} + \sum_{k=1}^{\langle i-1 \rangle} w_{j,k}^{(i)} x_k^{(i-1)}.$$

This calculation on individual neurons corresponds to the affine mapping applied between layer l_{i-1} and l_i. This computes the value $\boldsymbol{y}^{(i)}$ of the whole layer before activation function application using the vector $\boldsymbol{x}^{(i-1)}$ of values in layer l_{i-1} as follows:

$$\boldsymbol{y}^{(i)} = \boldsymbol{b}^{(i)} + \boldsymbol{W}^{(i)} \boldsymbol{x}^{(i-1)}.$$

The activation function is applied to each neuron independently of the values of the other neurons. Thus, the final value $x_j^{(i)}$ of neuron $n_{i,j}$ in a layer l_i for $i \in \{2, \ldots, L\}$ and $j \in \{1, \ldots, \langle i \rangle\}$ is the application of the activation function:

$$x_j^{(i)} = \mathrm{act}_{i,j}(y_j^{(i)}).$$

For the application of the activation function on the whole layer $\boldsymbol{x}^{(i)} = \mathbf{act}_i(\boldsymbol{y}^{(i)})$, the function is split into iterative applications of activation functions $\mathbf{act}_{i,j}$ that only effect neuron $n_{i,j}$ for $j \in \{1, \ldots, \langle i \rangle\}$:

$$\mathbf{act}_i : \mathbb{R}^{\langle i \rangle} \to \mathbb{R}^{\langle i \rangle}, \boldsymbol{y} \mapsto \mathbf{act}_{i,\langle i \rangle}(\mathbf{act}_{i,\langle i \rangle - 1}(\cdots \mathbf{act}_{i,1}(\boldsymbol{y}) \cdots)),$$

$$\mathbf{act}_{i,j} : \mathbb{R}^{\langle i \rangle} \to \mathbb{R}^{\langle i \rangle}, \boldsymbol{y} \mapsto \begin{pmatrix} y_1 \\ \vdots \\ \mathrm{act}_{i,j}(y_j) \\ \vdots \\ y_{\langle i \rangle} \end{pmatrix}.$$

For an input $\boldsymbol{x}^{(1)} \in \mathbb{R}^{\langle 1 \rangle}$, the output of an FNN $\boldsymbol{x}^{(L)} \in \mathbb{R}^{\langle L \rangle}$ is therefore the forward propagation of the input through this graph. Thus, the second way to interpret an FNN is as a function [SGPV19]

$$F : \mathbb{R}^{\langle 1 \rangle} \to \mathbb{R}^{\langle L \rangle}, \boldsymbol{x}^{(1)} \mapsto f_L(f_{L-1}(\cdots f_2(\boldsymbol{x}^{(1)}) \cdots))$$

composed of multiple functions f_i corresponding to non-input layers l_i for $i \in \{2, \ldots, L\}$. Each of these functions f_i is thus the application of the affine mapping and the activation function during the transition from layer l_{i-1} to layer l_i:

$$f_i : \mathbb{R}^{\langle i-1 \rangle} \to \mathbb{R}^{\langle i \rangle}, \boldsymbol{x}^{(i-1)} \mapsto \mathbf{act}_i(\boldsymbol{W}^{(i)} \boldsymbol{x}^{(i-1)} + \boldsymbol{b}^{(i)}).$$

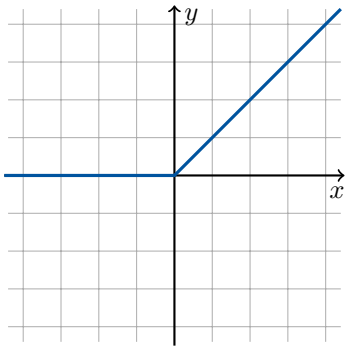


Figure 2.2: ReLU

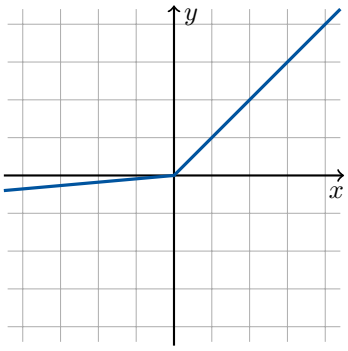


Figure 2.3: LeakyReLU

The size $\langle i \rangle$ of a layer and thus the number of neurons in a layer corresponds to the dimension of the intermediate values $\boldsymbol{x}^{(i)} \in \mathbb{R}^{\langle i \rangle}$ of the layer.

In the following, the activation functions [Mei72] relevant to this thesis will be introduced. All of them are piecewise linear functions of the form:

$$\text{act} : \mathbb{R} \to \mathbb{R}, x \mapsto \begin{cases} a_1 x + b_1 & \text{, if } x \in \mathsf{X}_1 \\ & \vdots \\ a_n x + b_n & \text{, if } x \in \mathsf{X}_n. \end{cases}$$

A piecewise linear function is a function that is constructed from a finite number $n \in \mathbb{N}_{\geq 2}$ of affine functions $(a_i x + b_i)$ which domains are intervals that form a partition of the real numbers. These domains $\mathsf{X}_1, \ldots, \mathsf{X}_n$ are called segments or subregions of the piecewise linear function. The following commonly used activation functions have been generalized for reachability analysis in [AMÁ23].

The *rectified linear unit* (ReLU) [Gus22] is one of the most common activation functions. It combines a low cost of computation with a solution to the problem of vanishing gradients [Gus22, YAT+20] which appears during training when the gradients approach 0 and reduces the learning efficiency.

Definition 2.1.1 (ReLU)**.** *The ReLU activation function represented in Figure 2.2 is defined as follows:*

$$ReLU : \mathbb{R} \to \mathbb{R}, x \mapsto \max(0, x) = \begin{cases} 0 & \text{, if } x < 0 \\ x & \text{, otherwise.} \end{cases}$$

Despite this advantage of ReLU, its first segment setting values to 0 causes the dead neuron problem. This problem also reduces learning efficiency and appears in gradient-based optimization, because weights are not adjusted during learning, if the value of a neuron is 0 [Maa13]. A solution to this problem is the *leaky rectified linear unit* (LeakyReLU) [Gus22]. This activation function adds a non-zero factor to the input in case it is negative instead of projecting to 0. This solves the dead neuron problem of ReLU, but once more introduces the vanishing gradient problem. Despite this, the LeakyReLU activation function performs satisfactory in many cases though not in all [XLD+20].

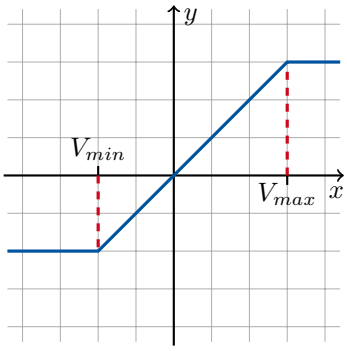


Figure 2.4: HardTanh

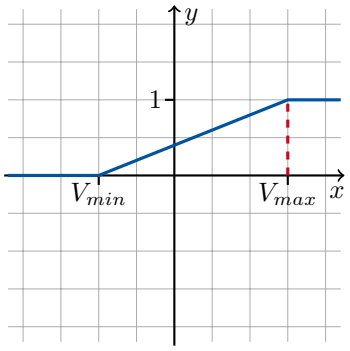


Figure 2.5: HardSigmoid

Definition 2.1.2 (LeakyReLU)**.** *The LeakyReLU activation function represented in Figure 2.3 is defined for a constant* $\gamma \in (0,1) \subseteq \mathbb{R}$ *as follows:*

$$\mathit{LeakyReLU} : \mathbb{R} \to \mathbb{R}, x \mapsto \max(\gamma x, x) = \begin{cases} \gamma x & , \text{ if } x < 0 \\ x & , \text{ otherwise.} \end{cases}$$

The *hard hyperbolic tangent* (HardTanh) activation function is a linearization of the hyperbolic tangent function that has been further generalized by exchanging the constant upper and lower bounds with corresponding variables.

Definition 2.1.3 (HardTanh)**.** *For* $V_{min}, V_{max} \in \mathbb{R}$ *with* $V_{min} \leq V_{max}$*, the HardTanh activation function represented in Figure 2.4 is defined as follows:*

$$\mathit{HardTanh} : \mathbb{R} \to \mathbb{R}, x \mapsto \begin{cases} V_{min} & , \text{ if } x < V_{min} \\ x & , \text{ if } V_{min} \leq x \leq V_{max} \\ V_{max} & , \text{ if } V_{max} < x. \end{cases}$$

The *hard sigmoid* (HardSigmoid) activation function is, similarly to HardTanh, a generalization of the linearization of the sigmoid activation function.

Definition 2.1.4 (HardSigmoid)**.** *For* $V_{min}, V_{max} \in \mathbb{R}$ *with* $V_{min} < V_{max}$*, the HardSigmoid activation function represented in Figure 2.5 is defined as follows:*

$$\mathit{HardSigmoid} : \mathbb{R} \to \mathbb{R}, x \mapsto \begin{cases} 0 & , \text{ if } x \leq V_{min} \\ \frac{x - V_{min}}{V_{max} - V_{min}} & , \text{ if } V_{min} < x < V_{max} \\ 1 & , \text{ if } V_{max} \leq x. \end{cases}$$

The *heaviside step* (UnitStep) activation function is an activation function that originally mapped an input to 0 or 1 depending on a comparison to a value [LMSR08].

Definition 2.1.5 (UnitStep)**.** *For* $R_{min}, R_{max}, v \in \mathbb{R}$*, the UnitStep activation function represented in Figure 2.6 is defined as follows:*

$$\mathit{UnitStep} : \mathbb{R} \to \mathbb{R}, x \mapsto \begin{cases} R_{min} & , \text{ if } x < v \\ R_{max} & , \text{ if } v \leq x. \end{cases}$$

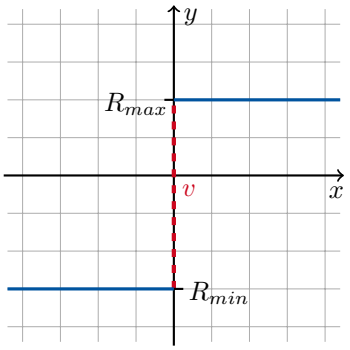


Figure 2.6: UnitStep

2.2 Star Sets

For a single input vector, the output of an FNN can be verified by propagating the input through the network and checking if the output fulfills our requirements. To facilitate the verification of FNNs for sets of input values, we require a state set representation that allows efficient propagation of all initial values through the network. This process is called *reachability analysis* and is the topic of the next section (Section 2.3). To perform this analysis, input, output, as well as intermediate sets which represent the set of values of a partial propagation though the network need to be representable. All of these sets are called reachable sets.

Based on the layered structure of FNNs, reachable sets need to allow efficient computation of the operations defining an FNN. These operations are affine mappings and activation function applications. A representation fulfilling these requirements is the star [BD17, DV16]. In the following, we will introduce this representation. Afterwards, we present the underlying properties that allow for the efficient performance of the aforementioned operations.

Definition 2.2.1 (Star Sets [BD17, AMÁ23])**.** *For any $n, m \in \mathbb{N}_+$, an (n, m)-dimensional* star set *(star) is a tuple $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathsf{P} \rangle$ with a center $\boldsymbol{c} \in \mathbb{R}^n$, a generator matrix $\boldsymbol{V} \in \mathbb{R}^{n \times m}$, which columns $\boldsymbol{V}_{\text{-},1}, \ldots, \boldsymbol{V}_{\text{-},m}$ are called generators or basis vectors, and a predicate $P \subseteq \mathbb{R}^m$.*
A star Θ represents the set $[\![\Theta]\!] := \{\boldsymbol{V}\boldsymbol{\alpha} + \boldsymbol{c} \mid \boldsymbol{\alpha} \in \mathsf{P}\}$.

Note, that with the unrestricted predicates $\mathsf{P} \subseteq \mathbb{R}^m$ of this definition, a star may represent a non-convex set.

Example 2.2.1. *Consider the $(2, 2)$-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathsf{P} \rangle$ where the center $\boldsymbol{c}$ is the origin, the generator matrix $\boldsymbol{V}$ is the identity matrix $\boldsymbol{I}^{(2)} \in \mathbb{R}^{2\times 2}$, and the predicate is $\mathsf{P} = \left\{\boldsymbol{e}_1^{(2)}, \boldsymbol{e}_2^{(2)}\right\} \subseteq \mathbb{R}^2$ where $\boldsymbol{e}_i^{(2)}$ is the standard unit vector. Since for example $\begin{pmatrix}0.5 & 0.5\end{pmatrix}^T \notin [\![\Theta]\!]$, $[\![\Theta]\!] = \left\{\boldsymbol{e}_1^{(2)}, \boldsymbol{e}_2^{(2)}\right\}$ is non-convex.*

For computing the bounds of stars during the reachability analysis (see Section 2.3), the definition of stars is further restricted to only allow the representation of convex sets. Additionally, the algorithms for reachability analysis presented in this thesis require boundedness. In general, this second constraint can be lifted as presented in [AMÁ23]. With this restriction, the predicate may only appear as a bounded convex

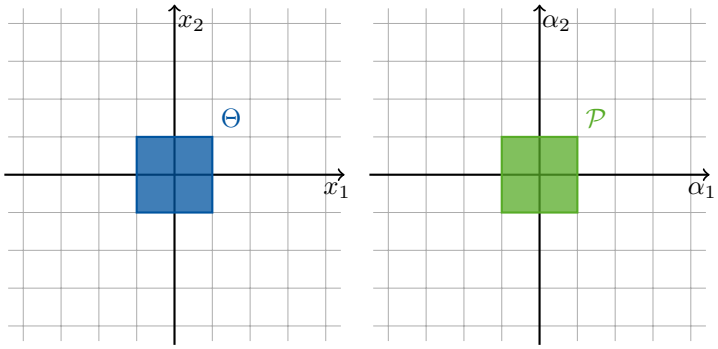


Figure 2.7: The representation of the polytope $\mathcal{P}$ in green (right) and the representation of the corresponding star in blue (left).

polytope $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d}\}$ for a matrix $\boldsymbol{A} \in \mathbb{R}^{p \times m}$ and a vector $\boldsymbol{d} \in \mathbb{R}^p$ with $p \in \mathbb{N}_+$. Equivalently, a polytope $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \psi_{\mathcal{P}}(\boldsymbol{\alpha})\}$ may be represented by the solution set of a conjunction of p linear constraints $\psi_{\mathcal{P}}(\boldsymbol{\alpha}) = \bigwedge_{i=1}^{p} \varphi_i(\boldsymbol{\alpha})$ where $\varphi_i(\boldsymbol{\alpha}) = \sum_{j=1}^{m} a_{i,j}\alpha_j \leq d_i$ is a linear constraint over the variables $\alpha_1, \ldots, \alpha_m$ for each $i \in \{1, \ldots, p\}$. A deeper and more general explanation of this formula is given in Section 2.4. Note, that these convex predicates are written as $\mathcal{P}$, while general predicates $\mathsf{P} \subseteq \mathbb{R}^m$ are written in a non-calligraphic style.

The following properties of stars correspond to the operations used in the star-based reachability analysis and verification of FNNs. A proof for each of the following propositions and all other proposition presented in this thesis can be found in Appendix A.

Proposition 2.2.1 (Emptiness [TMLM+19])**.** *A star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ is empty if and only if $\mathcal{P}$ is empty.*

Thus, checking the emptiness of a star is equivalent to checking the emptiness of a convex bounded polytope or checking the unsatisfiability of a conjunction of linear constraints.

Proposition 2.2.2 (Representing Polyhedra with Star Sets [BD17])**.** *Any bounded, convex polytope $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{d}\}$ for $n \in \mathbb{N}_+$ can be represented by the (n, n)-dimensional star $\Theta = \langle \boldsymbol{0}^{(n)}, \boldsymbol{I}^{(n)}, \mathcal{P} \rangle$ with $\boldsymbol{0}^{(n)} \in \mathbb{R}^n$ the origin and $\boldsymbol{I}^{(n)} \in \mathbb{R}^{n \times n}$ the identity matrix.*

Example 2.2.2. *Consider the polytope $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^2 \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{d}\}$ representing a box around the origin in $\mathbb{R}^2$ represented on the right in green in Figure 2.7 with:*

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{d} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Since $\mathcal{P} \subseteq \mathbb{R}^2$, the $(2, 2)$-dimensional star

$$\Theta = \left\langle \boldsymbol{0}^{(2)}, \boldsymbol{I}^{(2)}, \mathcal{P} \right\rangle = \left\langle \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \mathcal{P} \right\rangle$$

represents the same set of values as $\mathcal{P}$. This is graphically represented on the left in blue in Figure 2.7.

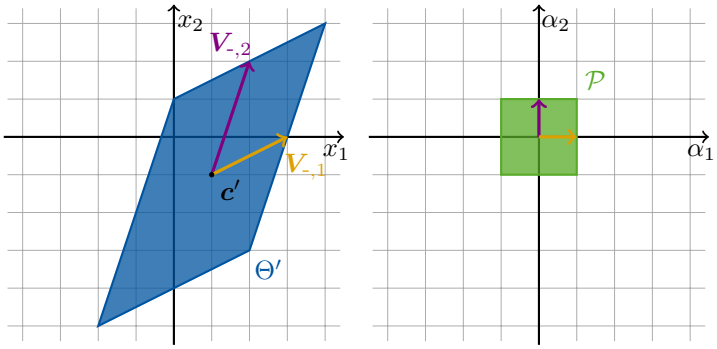


Figure 2.8: The star Θ' resulting from the application of the affine mapping to Θ is represented in blue (left). The representation of the predicate $\mathcal{P}$ of Θ' is given in green (right).

Proposition 2.2.3 (Affine Mapping [TMLM+19])**.** *Let $k, m, n \in \mathbb{N}_+$. For any matrix $\boldsymbol{W} \in \mathbb{R}^{k \times n}$ and any vector $\boldsymbol{b} \in \mathbb{R}^k$ the affine mapping $\{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \mid \boldsymbol{x} \in [\![\Theta]\!]\}$ of a (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, can be represented by a (k, m)-dimensional star $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ where $\boldsymbol{c}' = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b}$ and $\boldsymbol{V}' = \boldsymbol{W}\boldsymbol{V}$.*

Example 2.2.3. *Using the star $\Theta = \langle \boldsymbol{0}^{(2)}, \boldsymbol{I}^{(2)}, \mathcal{P} \rangle$ introduced in the prior Example 2.2.2, the $(2, 2)$-dimensional star Θ' representing the affine mapping $[\![\Theta']\!] = \{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \mid \boldsymbol{x} \in [\![\Theta]\!]\}$ for*

$$\boldsymbol{W} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

can be calculated based on the proposition. Thus, $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ for the center

$$\boldsymbol{c}' = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

and the generator matrix

$$\boldsymbol{V}' = \boldsymbol{W}\boldsymbol{V} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$$

represents the affine mapping. The values of Θ' are represented in Figure 2.8 on left. The predicate, which has remained unchanged, is represented on the right in this figure. Note, that the predicate does not change when applying an affine mapping to a star.

Proposition 2.2.4 (Intersection with a Halfspace [TMLM+19])**.** *Assume $n, m \in \mathbb{N}_+$ for a (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and a halfspace $\mathcal{H} = \left\{\boldsymbol{x} \in \mathbb{R}^n \;\middle|\; \boldsymbol{h}^T\boldsymbol{x} \leq g\right\}$ defined by $\boldsymbol{h} \in \mathbb{R}^n$ and $g \in \mathbb{R}$. Then, the intersection $\mathcal{H} \cap [\![\Theta]\!]$ can be represented by the (n, m)-dimensional star $\Theta' = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \cap \mathcal{P}' \rangle$ with*

$$\mathcal{P}' = \left\{\boldsymbol{\alpha} \in \mathbb{R}^m \;\middle|\; (\boldsymbol{h}^T\boldsymbol{V})\boldsymbol{\alpha} \leq g - \boldsymbol{h}^T\boldsymbol{c}\right\}$$

Example 2.2.4. *Consider the star $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ obtained in Example 2.2.3 and a halfspace $\mathcal{H} = \left\{\boldsymbol{x} \in \mathbb{R}^2 \;\middle|\; \boldsymbol{h}^T\boldsymbol{x} \leq g\right\}$ of all non-negative values in the first dimension*

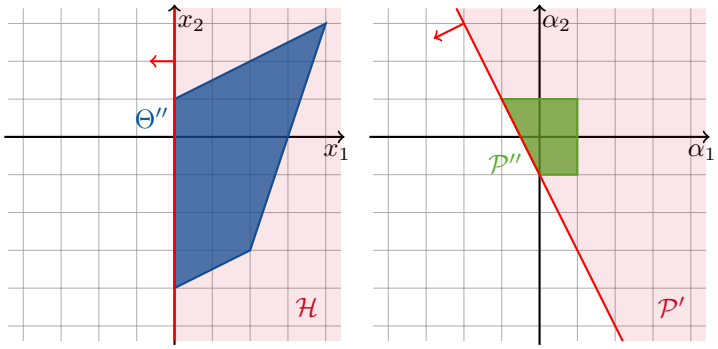


Figure 2.9: The star Θ'' resulting from intersection with halfspace $\mathcal{H}$ is represented in blue (left) as well as the halfspace in red. The representation of the predicate $\mathcal{P}''$ of Θ'' is given in green (right). Additionally, the new constraint $\mathcal{P}'$ resulting from the halfspace is represented on the right side in red.

$(x_1 \geq 0)$ with $\boldsymbol{h}^T = \begin{pmatrix}-1 & 0\end{pmatrix}$ and $g = 0$. This halfspace is represented in red in Figure 2.9 on the left. The $(2,2)$-dimensional star $\Theta'' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P}'' \rangle$ representing the intersection $[\![\Theta'']\!] = \mathcal{H} \cap [\![\Theta']\!]$, which is represented in the same coordinate system, can now be calculated based on the new predicate $\mathcal{P}'' = \mathcal{P} \cap \mathcal{P}'$. Based on the proposition, the addition to the predicate $\mathcal{P}'$ is defined as follows:

$$\begin{aligned}
\mathcal{P}' &= \left\{\boldsymbol{\alpha} \in \mathbb{R}^2 \mid (\boldsymbol{h}^T \boldsymbol{V}')\boldsymbol{\alpha} \leq g - \boldsymbol{h}^T \boldsymbol{c}'\right\} \\
&= \left\{\boldsymbol{\alpha} \in \mathbb{R}^2 \;\middle|\; \left(\begin{pmatrix}-1 & 0\end{pmatrix}\begin{pmatrix}2 & 1\\ 1 & 3\end{pmatrix}\right)\boldsymbol{\alpha} \leq 0 - \begin{pmatrix}-1 & 0\end{pmatrix}\begin{pmatrix}1\\ -1\end{pmatrix}\right\} \\
&= \left\{\boldsymbol{\alpha} \in \mathbb{R}^2 \mid \begin{pmatrix}-2 & -1\end{pmatrix}\boldsymbol{\alpha} \leq 1\right\}
\end{aligned}$$

$\mathcal{P}''$ is represented in green and $\mathcal{P}'$ in red in the right coordinate system of Figure 2.9. Due to the restriction of the predicate to bounded convex polyhedra, the intersection $\mathcal{P} \cap \mathcal{P}'$ can be represented as follows:

$$\mathcal{P}'' = \left\{\boldsymbol{\alpha} \in \mathbb{R}^2 \;\middle|\; \begin{pmatrix}1 & 0\\ -1 & 0\\ 0 & 1\\ 0 & -1\\ -2 & -1\end{pmatrix}\boldsymbol{\alpha} \leq \begin{pmatrix}1\\ 1\\ 1\\ 1\\ 1\end{pmatrix}\right\}$$

Proposition 2.2.5 (Bounds [AMÁ23])**.** *For $n, m \in \mathbb{N}_+$ and any (n,m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, the upper and lower bounds of the set in dimension $i \in \{1, \ldots, n\}$ can be calculated by:*

- *Lower bound:* $l_i := c_i + \min\{\boldsymbol{V}_i\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
- *Upper bound:* $u_i := c_i + \max\{\boldsymbol{V}_i\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$

2.3 Reachability Analysis

As mentioned in the previous section, the reachable set of an FNN is defined as the set of all values that can be obtained by propagating each value in an input set through

the network. For this, reachability is defined based on the reachable sets of each layer as follows.

Definition 2.3.1 (Reachability [TMLM+19]). *For an FNN F with $L \in \mathbb{N}_{\geq 2}$ layers, weight matrices $\boldsymbol{W}^{(i)}$, bias vectors $\boldsymbol{b}^{(i)}$ and, activation function $\boldsymbol{act}_i$ for all non-input layers $i \in \{2, \ldots, L\}$ and an input set $\mathcal{I} \subseteq \mathbb{R}^{\langle 1 \rangle}$, the reachable set of layer l_i of F given input set $\mathcal{I}$ is defined inductively as $\mathcal{R}_1 := \mathcal{I}$ and for $1 < i$ as follows:*

$$\mathcal{R}_i := \left\{ \boldsymbol{act}_i(\boldsymbol{W}^{(i)}\boldsymbol{x} + \boldsymbol{b}^{(i)}) \;\middle|\; \boldsymbol{x} \in \mathcal{R}_{i-1} \right\}$$

The reachable set of an FNN F given an input set $\mathcal{I}$ is then defined as the reachable set of the output layer:

$$F(\mathcal{I}) := \mathcal{R}_L$$

In general, the input set could be represented by a convex bounded polytope. Using Proposition 2.2.2, such an input set can be transformed into a star efficiently. For our reachability algorithms, the input set, and all other reachable sets are represented by stars, due to this transformation.

The computation of the reachable sets of an FNN is performed layer-by-layer. For each layer, we first compute an affine mapping. This computation can be performed efficiently on stars according to Proposition 2.2.3. Afterwards, an activation function is applied dimension-wise (i.e., for each neuron in the layer). For stars, this is not straight forward, because convexness is not necessarily preserved. Due to our restrictions to their predicates, stars always represent convex sets.

The application of a piecewise linear activation function to one dimension of a set preserves convexness, if the set of all values in the respective dimension is a subset of exactly one segment. In this case, an affine mapping $x \mapsto ax + b$ for $a, b \in \mathbb{R}$ corresponding to the segment of the activation functions is applied to the values in this dimension. For an application to dimension j, this corresponds to the affine mapping

$$\boldsymbol{x} \mapsto \boldsymbol{M}_{[a]}^{(n)}\boldsymbol{x} + \boldsymbol{v}_{[b]}^{(n)}$$

for $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{M}_{[a]}^{(n)} \in \mathbb{R}^{n \times n}$, $\boldsymbol{v}_{[b]}^{(n)} \in \mathbb{R}^n$ with the following definition:

$$\boldsymbol{M}_{[a]}^{(n)} = \begin{pmatrix} \boldsymbol{e}_1^{(n)} & \cdots & a\boldsymbol{e}_j^{(n)} & \cdots & \boldsymbol{e}_n^{(n)} \end{pmatrix} \quad \boldsymbol{v}_{[b]}^{(n)} = \begin{pmatrix} 0 & \cdots & b & \cdots & 0 \end{pmatrix}^T.$$

Based on Proposition 2.2.3, this affine mapping and thus the activation function can be applied to a (n, m)-dimensional star.

We present two methods for the application of the activation function, if convexness is not inherently preserved. The first method applies the activation function in an exact but computationally expensive manner via case-splitting. The other approach is a less expensive but over-approximates the application of the activation function. This over-approximation sacrifices the *completeness* of the safety verification introduced at the end of this section, but preserves its *soundness*.

The exact approach splits a given input star into multiple stars such that the activation function can be applied with the previous method. A star is thus split according to the segments of the activation function. This is done by intersecting the star with halfspaces based on Proposition 2.2.4. To each resulting star, the activation function can be applied directly. The union of these stars after application of the activation function is then equal to the direct application of the activation function

Algorithm 1:
Exact Reachability Analysis

input : Star Θ,
FNN F with L layers
output : List of stars $\mathcal{R}$

1 $\mathcal{R} \leftarrow [\Theta]$
2 **for** $i \leftarrow 2$ **to** L **do**
3 $\quad \mathcal{R}' \leftarrow []$
4 $\quad$ **for** $\langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle \in \mathcal{R}$ **do**
5 $\quad\quad \Theta' \leftarrow \left\langle \boldsymbol{W}^{(i)}\boldsymbol{c} + \boldsymbol{b}^{(i)}, \boldsymbol{W}^{(i)}\boldsymbol{V}, \mathcal{P} \right\rangle$
6 $\quad\quad T \leftarrow [\Theta']$
7 $\quad\quad$ **for** $j \leftarrow 1$ **to** $\langle i \rangle$ **do**
8 $\quad\quad\quad T' \leftarrow []$
9 $\quad\quad\quad$ **for** $\Theta'' \in T$ **do**
10 $\quad\quad\quad\quad T'$.append(exactAct(Θ'', j))
11 $\quad\quad\quad T \leftarrow T'$
12 $\quad\quad \mathcal{R}'$.append(T)
13 $\quad \mathcal{R} \leftarrow \mathcal{R}'$

Algorithm 2:
Approximate Reachability Analysis

input : Star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
FNN F with L layers
output : Star $\mathcal{R} = \Theta'$

1
2 **for** $i \leftarrow 2$ **to** L **do**
3
4
5 $\quad \Theta' \leftarrow \left\langle \boldsymbol{W}^{(i)}\boldsymbol{c} + \boldsymbol{b}^{(i)}, \boldsymbol{W}^{(i)}\boldsymbol{V}, \mathcal{P} \right\rangle$
6
7 $\quad$ **for** $j \leftarrow 1$ **to** $\langle i \rangle$ **do**
8 $\quad\quad$
9 $\quad\quad$
10 $\quad\quad \Theta' \leftarrow$approximateAct$(\Theta', j)$
11 $\quad\quad$
12 $\quad\quad$
13 $\quad$

to the original set. Note, that splitting does not necessarily result in a partition of the original star. It still ensures that the union of the resulting stars is equal to the original set.

Therefore, the exact application outputs a union or list of (generally) multiple stars with each application. Further calculations of the reachability algorithm are then performed on each of the stars in the list. This process is described algorithmically by Algorithm 1 where lists are represented with square brackets. The method `append` adds a new item to a list. `exactAct` refers to the exact application of a specific activation function. This process is described later in this section. Note, that the execution of the loop starting in line 4 can be parallelized as the computation relies only on the selected star. Besides this option, parallelization is not further explored in this thesis.

The over-approximating approach to this calculation instead over-approximates an activation function with a convex relaxation and thereby ensures the convexness of the result. The convex relaxation used for this purpose is as tight as possible, when only considering the output and input domains of a single neuron [TAH+20]. For the application of the relaxation to dimension j, all values x_j of an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ with $\boldsymbol{x} \in [\![\Theta]\!]$ are represented by a new variable α_{m+1} of the predicate. The relaxation is then applied to the star in the form of constraints on α_{m+1}. These constraints depend on the star and the specific activation function. They fully define the value x_j. A detailed explanation of the constraints for each activation function follows later in this section.

A value x_j corresponds to a vector $\boldsymbol{\alpha} \in \mathcal{P}$ with $x_j = c_j + \sum_{i=1}^{m} v_{i,j}\alpha_i$. To represent the values x_j with the new variable α_{m+1}, the values $v_{i,j}$ for all $j \in \{1, \ldots, m\}$ and c_j are set to 0. This is equivalent to the application of the affine mapping with matrix $\boldsymbol{M}_{[0]}^{(n)}$ and vector $\boldsymbol{0}^{(n)}$ to the star Θ. Afterwards, the unit vector $\boldsymbol{e}_j^{(n)}$ is added to the generator. The star

$$\Theta' = \left\langle \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}, \begin{pmatrix} \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V} & \boldsymbol{e}_j^{(n)} \end{pmatrix}, \mathcal{P}' \right\rangle$$

Algorithm 3:
Exact Activation Function Application

input : Star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Dimension j,
List of cases C
output : List of stars R

$l_j \leftarrow c_j + \min\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
$u_j \leftarrow c_j + \max\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
for $(\mathsf{P}, \mathit{splits}) \in C$ **do**
  if $\mathsf{P}(l_j, u_j) = \top$ **then**
    $R \leftarrow []$
    for $(\boldsymbol{A}, \boldsymbol{b}, \mathit{halfspaces}) \in \mathit{splits}$ **do**
      $\Theta' \leftarrow \langle \boldsymbol{A}\boldsymbol{c} + \boldsymbol{b}, \boldsymbol{A}\boldsymbol{V}, \mathcal{P} \rangle$
      for $\mathcal{H} \in \mathit{halfspaces}$ **do**
        $\Theta' \leftarrow \Theta' \cap \mathcal{H}$
      R.`append`(Θ')

Algorithm 4:
Approximate Activation Function Application

input : Star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Dimension j,
List of cases C
output : Star Θ'

$l_j \leftarrow c_j + \min\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
$u_j \leftarrow c_j + \max\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
for $(\mathsf{P}, \mathcal{B}, \mathit{info}) \in C$ **do**
  if $\mathsf{P}(l_j, u_j) = \top$ **then**
    if $\mathcal{B} = \top$ **then**
      `// Exact computation`
      $(\boldsymbol{A}, \boldsymbol{b}) \leftarrow \mathit{info}$
      $\Theta' \leftarrow \langle \boldsymbol{A}\boldsymbol{c} + \boldsymbol{b}, \boldsymbol{A}\boldsymbol{V}, \mathcal{P} \rangle$
    else
      `// Approximation`
      $\mathcal{P}' \leftarrow \mathcal{P}$
      for $\varphi \in \mathit{info}$ **do**
        $\mathcal{P}' \leftarrow \mathcal{P}' \wedge \varphi$
      $\Theta' \leftarrow \left\langle \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}, \begin{pmatrix} \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V} & \boldsymbol{e}_j^{(n)} \end{pmatrix}, \mathcal{P}' \right\rangle$

is the result of this process with Θ as input. For all $\boldsymbol{x} \in [\![\Theta']\!]$, the value x_j is fully represented by α_{m+1} of the corresponding predicate variable $\boldsymbol{\alpha} \in \mathcal{P}'$. Thus, $x_j = \alpha_{m+1}$. Note, that the constraints describing $\mathcal{P}$ are also part of $\mathcal{P}'$. The additional variable α_{m+1} of $\mathcal{P}'$ only appears in the new constraints describing the relaxation of the activation function.

This over-approximation is only applied, if the direct application of the activation function does not preserve convexness. Otherwise, the over-approximating approach computes with the exact method. This means that the over-approximating approach always produces a single star instead of the list in the exact calculation. This process is described in Algorithm 2 using the same notation as before and `approximateAct` for the approximating application of a specific activation function.

The way a specific activation function is applied to a dimension depends on the set of values represented by a star in this dimension. More specifically, the segments of the activation function these values are elements of. Depending on these segments the exact approach applies splits and the over-approximation adds constraints. To determine the relevant segments, it is sufficient to check the upper and lower bounds of the dimension the activation function is applied to. This follows from the convexness of stars. The bounds can be computed as presented in Proposition 2.2.5.

The exact and over-approximating algorithms for the reachability analysis of FNN presented in Algorithms 1 to 4 are aggregations of the algorithms for specific activation functions presented in [TMLM+19, AMÁ23]. For these aggregations, the relevant information for the application of the activation functions is represented as a list of cases C. These cases correspond to splits and constraints for exact or over-approximated approach. For the exact approach, a case in C has the form

$$(\mathsf{P}, [(\boldsymbol{A}^{(1)}, \boldsymbol{b}^{(1)}, [\mathcal{H}_{1,1}, \ldots, \mathcal{H}_{1,k_1}]), \ldots, (\boldsymbol{A}^{(k)}, \boldsymbol{b}^{(k)}, [\mathcal{H}_{k,1}, \ldots, \mathcal{H}_{k_k}])]).$$

$\mathsf{P} : \mathbb{R}^2 \to \{\top, \bot\}$ is a predicate indicating the segments of the activation function. The intended input for P are thus the lower bound l_j and upper bound u_j in dimension j

of the star. P maps to $\top$, if the bounds correspond to the case, and false otherwise. Note, that for l_j, u_j exactly one predicate in the list of cases is satisfied.

If the bounds of a star satisfy a predicate, the activation function is applied according each element of the list that is the second entry of the case. Each of these elements corresponds to a split and subsequent activation function application. Thus, an additional star represents the reachable set after the application of the activation function for each element $(\boldsymbol{A}, \boldsymbol{b}, [\mathcal{H}_1, \ldots, \mathcal{H}_k])$ in this list. For a given (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, the star resulting from this element has the form

$$\Theta' = \langle \boldsymbol{A}\boldsymbol{c} + \boldsymbol{b}, \boldsymbol{A}\boldsymbol{V}, \mathcal{P} \rangle \cap \bigcap_{i=1}^{k} \mathcal{H}_i$$

with $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^n$, and halfspaces $\mathcal{H}_i \subseteq \mathbb{R}^n$ for $i \in \{1, \ldots, k\}$ and $k \in \{0, 1, 2\}$. Note, that the restriction of k is solely based on the activation functions we used. If $k \in \{1, 2\}$ then the exact computation splits the star. If otherwise $k = 0$, the bounds of the star fall within one segment of the activation function definition.

For the over-approximating method, a case in C has the following form

$$(\mathsf{P}, \mathcal{B}, info)$$

where P is the same as before and $\mathcal{B} \in \{\top, \bot\}$ indicates the information contained in $info$. As explained before, the over-approximating approach applies exact computation, if no splitting is required. Otherwise, the activation function is applied in the form of a convex relaxation. If $\mathcal{B} = \top$, $info$ contains information to apply the exact reachability method. Thus, $info = (\boldsymbol{A}, \boldsymbol{b})$ for a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$. The resulting star $\Theta' = \langle \boldsymbol{A}\boldsymbol{c} + \boldsymbol{b}, \boldsymbol{A}\boldsymbol{V}, \mathcal{P} \rangle$ has therefore the same form as in the exact computation if no halfspace intersection is necessary.

If otherwise $\mathcal{B} = \bot$ over-approximation is used. To this end, the information has the form $info = [\varphi_1, \ldots, \varphi_k]$ for $k \in \{2, 3, 4\}$ where each φ_i corresponds to a constraint for the convex relaxation. For an (n, m)-dimensional star Θ as before, the resulting $(n, m+1)$-dimension star has the following form

$$\Theta' = \left\langle \boldsymbol{M}_{[0]}^{(n)} \boldsymbol{c}, \begin{pmatrix} \boldsymbol{M}_{[0]}^{(n)} \boldsymbol{V} & \boldsymbol{e}_j^{(n)} \end{pmatrix}, \mathcal{P} \wedge \bigwedge_{i=1}^{k} \varphi_i \right\rangle.$$

In Algorithms 1 and 2 `exactAct` and `approximateAct` refer to the exact and over-approximating algorithms in Algorithms 3 and 4. For these function calls, the list of cases C is implicitly replaced by the information corresponding to the application of a specific activation function to dimension j. The information for these lists is given in Tables 2.1 to 2.5. In these tables, the new constraints for the approximate case often contain a variable x_j. This variable refers to all values x_j of dimension j represented by an input star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$. As mentioned before x_j can be represented as $c_j + \sum_{i=1}^{m} v_{i,j} \alpha_i$ based the variables of the predicate $\mathcal{P}$. Additionally, a halfspace $\left\{ \boldsymbol{x} \in \mathbb{R}^n \;\middle|\; \boldsymbol{h}^T \boldsymbol{x} \leq g \right\}$ is represented by the linear constraint $\sum_{i=1}^{n} h_i x_i \leq g$ corresponding to $\boldsymbol{h}^T \boldsymbol{x} \leq g$.

For the ReLU activation function, the three cases for splitting and approximation are presented in Table 2.1. In the first case, all values represented by a star in dimension j are non-negative. The second case essentially represents the opposite. All values have to be negative except for the upper bound which can 0. The reason for this upper is the definition of the ReLU activation function. It is arbitrary in which

$\mathsf{P}(l_j, u_j)$		Exact				Approximate	
l_j	u_j	Θ_i	$\boldsymbol{A}$	$\boldsymbol{b}$	Halfspaces	$\mathcal{B}$	Constraints
$[0, \infty)$	$[0, \infty)$	Θ_1	$\boldsymbol{I}^{(n)}$	$\mathbf{0}^{(n)}$	-	$\top$	-
$(-\infty, 0)$	$(-\infty, 0]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	-	$\top$	-
$(-\infty, 0)$	$(0, \infty)$	Θ_1	$\boldsymbol{I}^{(n)}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (0 \leq x_j)$	$\bot$	$\varphi_1 = 0 \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (x_j \leq 0)$		$\varphi_2 = x_j \leq \alpha_{m+1}$
							$\varphi_3 = \alpha_{m+1} \leq \frac{u_j(x_j - l_j)}{u_j - l_j}$

Table 2.1: Cases for the reachability algorithms for the ReLU activation function. In the column labeled $\mathsf{P}(l_j, u_j)$, the predicate corresponds to $l_j \in \mathsf{I}_l \wedge u_j \in \mathsf{I}_u$ for I_l the interval below l_j and I_u the interval below u_j. Note, that the columns $\boldsymbol{A}$ and $\boldsymbol{b}$ under Exact are also used for the exact computation in over-approximate reachability algorithms.

segment 0 is included as both segments map it to 0 again. In the third case, values included in both subregions are represented by the star.

Example 2.3.1. *Based on Table 2.1, the exact and over-approximating reachability algorithms for the ReLU activation function can be rewritten as presented in Algorithms 5 and 6. For this, we first extract the lists of cases from Table 2.1. For the exact algorithm, the list contains the following tuples describing the cases as defined before:*

- $((0 \leq l_j \wedge 0 \leq u_j), [(\boldsymbol{I}^{(n)}, \mathbf{0}^{(n)}, [])])$
- $((l_j < 0 \wedge u_j \leq 0), [(\boldsymbol{M}^{(n)}_{[0]}, \mathbf{0}^{(n)}, [])])$
- $((l_j < 0 \wedge 0 < u_j), [(\boldsymbol{I}^{(n)}, \mathbf{0}^{(n)}, [(0 \leq x_j)]), (\boldsymbol{M}^{(n)}_{[0]}, \mathbf{0}^{(n)}, [(x_j \leq 0)])])$

Analogously, the list of cases for the over-approximating approach can be extracted as follows:

- $((0 \leq l_j \wedge 0 \leq u_j), \top, (\boldsymbol{I}^{(n)}, \mathbf{0}^{(n)}))$
- $((l_j < 0 \wedge u_j \leq 0), \top, (\boldsymbol{M}^{(n)}_{[0]}, \mathbf{0}^{(n)}))$
- $((l_j < 0 \wedge 0 < u_j), \bot, [0 \leq \alpha_{m+1}, x_j \leq \alpha_{m+1}, \alpha_{m+1} \leq \frac{u_j(x_j - l_j)}{u_j - l_j}])$

Instead of looping through all the cases in these list to apply the one with a satisfied predicate (first element), we can write these cases as a sequence of `if`*-cases. In both the exact and over-approximating algorithm, the* `if`*-cases in lines* 3, 6 *and* 9 *correspond to the cases in the list in the same order. Since the first two cases of the ReLU activation function do not require a split, both algorithms perform the same actions in the* `if`*-cases starting in lines* 3 *and* 6*, besides the difference in output representation. Regarding the last case, the exact and over-approximating algorithm differ.*

Algorithm 5:
Exact ReLU Application

input : Star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Dimension j

output : List of stars R

1 $l_j \leftarrow c_j + \min\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
2 $u_j \leftarrow c_j + \max\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
3 **if** $0 \leq l_j$ **then**
4 $\quad \Theta \leftarrow \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$
5 $\quad R \leftarrow [\Theta]$
6 **if** $l_j < 0 \wedge u_j \leq 0$ **then**
7 $\quad \Theta \leftarrow \left\langle \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}, \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V}, \mathcal{P} \right\rangle$
8 $\quad R \leftarrow [\Theta]$
9 **if** $l_j < 0 \wedge 0 < u_j$ **then**
10 $\quad \Theta' \leftarrow \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle \cap (0 \leq x_j)$
11 $\quad \Theta'' \leftarrow \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle \cap (x_j \leq 0)$
12 $\quad \Theta'' \leftarrow \left\langle \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}, \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V}, \mathcal{P}' \right\rangle$
13 $\quad R \leftarrow [\Theta', \Theta'']$

Algorithm 6:
Approximate ReLU Application

input : Star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ with
Predicate $\mathcal{P} = \{\boldsymbol{\alpha} \mid \boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d}\}$
Dimension j

output : Star Θ'

1 $l_j \leftarrow c_j + \min\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
2 $u_j \leftarrow c_j + \max\{\boldsymbol{V}_j\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$
3 **if** $0 \leq l_j$ **then**
4 $\quad \Theta' \leftarrow \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$
5
6 **if** $0 < l_j \wedge u_j \leq 0$ **then**
7 $\quad \Theta' \leftarrow \left\langle \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}, \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V}, \mathcal{P} \right\rangle$
8
9 **if** $l_j < 0 \wedge 0 < u_j$ **then**
10 $\quad \boldsymbol{A}' \leftarrow \begin{pmatrix} & & & 0 \\ & \boldsymbol{A} & & \vdots \\ & & & 0 \\ 0 & \cdots & 0 & -1 \\ & \boldsymbol{V}_j & & -1 \\ & -\frac{u_j}{u_j - l_j}\boldsymbol{V}_j & & 1 \end{pmatrix}$
11 $\quad \boldsymbol{\alpha}' \leftarrow \begin{pmatrix} \boldsymbol{\alpha} \\ \alpha_{m+1} \end{pmatrix}$
12 $\quad \boldsymbol{d}' \leftarrow \begin{pmatrix} \boldsymbol{d} \\ 0 \\ -c_j \\ \frac{u_j \cdot (c_j - l_j)}{u_j - l_j} \end{pmatrix}$
13 $\quad \boldsymbol{c}' \leftarrow \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{c}$
14 $\quad \boldsymbol{V}' \leftarrow \begin{pmatrix} \boldsymbol{M}_{[0]}^{(n)}\boldsymbol{V} & \boldsymbol{e}_j^{(n)} \end{pmatrix}$
15 $\quad \Theta' \leftarrow \langle \boldsymbol{c}', \boldsymbol{V}', \{\boldsymbol{\alpha}' \mid \boldsymbol{A}'\boldsymbol{\alpha}' \leq \boldsymbol{d}'\} \rangle$

Figure 2.10: Algorithms for the reachability analysis for FNN using only ReLU activation functions.

In Algorithm 5 the star is split into two according to the halfspaces in the second element of the list. Afterwards, the activation function can be applied in the same way as the previous two cases. In Algorithm 6, the convex relaxation of the ReLU activation function is applied by adding the three constraints from the list in the third element of the tuple. These constraints are added to the predicate of the input star. Afterwards the center and generator matrix are changed as described before including the addition of a new column to the generator. For this new column, a new constraint variables is added in line 11*. The constraints correspond to the changes to the predicate in lines* 10 *and* 12*. For this, the constraints from the list are first transformed such that* $\boldsymbol{A}'$ *contains the factors for the constraint variables in* $\boldsymbol{\alpha}'$ *in the last three rows. Thus,* $\boldsymbol{d}'$ *contains the remaining constants. For the transformation,* $x_j = \boldsymbol{V}_j\alpha + c_j$ *is used.*

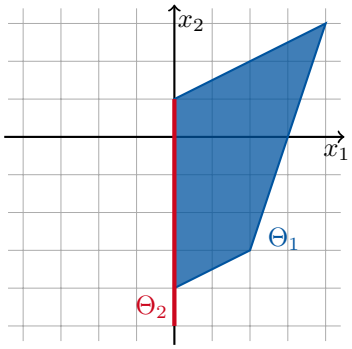


Figure 2.11: Result of the application of the exact ReLU to the first dimension.

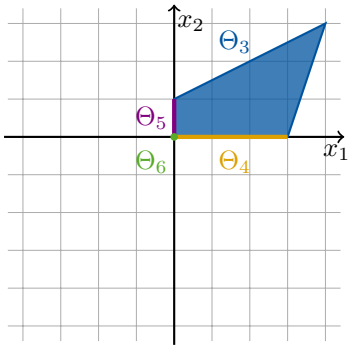


Figure 2.12: Result of the application of the exact ReLU to the second dimension.

Example 2.3.2. *Assume the same star Θ derived from a polytope $\mathcal{P}$ introduced in Example 2.2.2 and a two layer FNN F. The first layer of an FNN always corresponds to the input. Thus, only the second layer, which is the output layer in this case, applies any operations to the input. Assume that the affine mapping of this layer is applied with the weight matrix $\boldsymbol{W}^{(2)} = \boldsymbol{W}$ and bias vector $\boldsymbol{b}^{(2)} = \boldsymbol{b}$ as introduced for the affine mapping in Example 2.2.3. Also, assume that this layer applies the ReLU activation function.*

In the following, Algorithm 1 using Algorithm 5 for `exactAct` *will be applied to the input star Θ for the FNN F focusing on the representation of the stars computed. The first change to Θ is the application of the affine mapping with matrix $\boldsymbol{W}$ and vector $\boldsymbol{b}$ in line 5 of Algorithm 1 which is explained in Example 2.2.3 in detail. On the result Θ' of this, Algorithm 5 is applied for dimension $j = 1$. Based on the lower bound $l_1 = -2$ and the upper bound $u_1 = 4$, the third case of the exact ReLU application is executed. This results in the following two stars represented in Figure 2.11:*

$$\Theta_1 = \left\langle \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -2 & -1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right\rangle$$

$$\Theta_2 = \left\langle \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} \right\rangle$$

Note that the calculation for Θ_1 is presented in detail in Example 2.2.4.

Since F has only two layers, the missing steps in calculating the reachable set of F with the exact algorithm is the application of ReLU to the second dimension for both stars Θ_1 and Θ_2. Based on the bounds of both Θ_1 and Θ_2 this results in computation

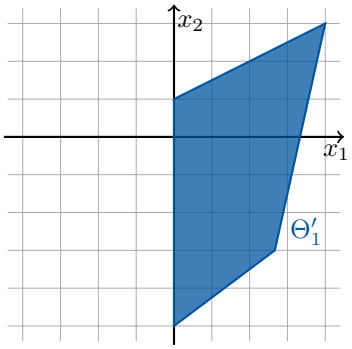


Figure 2.13: Result of the application of the over-approximating ReLU to the first dimension.

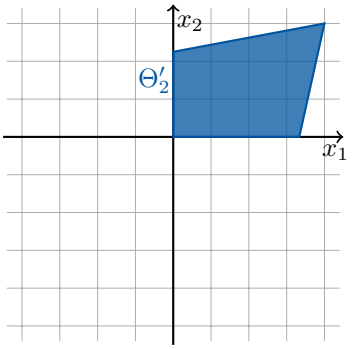


Figure 2.14: Result of the application of the over-approximating ReLU to the second dimension.

of the third case. The following stars result from Θ_1*:*

$$\Theta_3 = \left\langle \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -2 & -1 \\ -1 & -3 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} \right\rangle$$

$$\Theta_4 = \left\langle \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -2 & -1 \\ 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right\rangle$$

and the following stars from Θ_2*:*

$$\Theta_5 = \left\langle \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 2 & 1 \\ -1 & -3 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \\ -1 \end{pmatrix} \right\rangle$$

$$\Theta_6 = \left\langle \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 2 & 1 \\ 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \\ 1 \end{pmatrix} \right\rangle$$

All of these are represented in Figure 2.12.

Example 2.3.3. *Assume the same star Θ and two-layer FNN F introduced in the previous Example 2.3.2.*

In the following, Algorithm 2 using Algorithm 6 for `approximateAct` *will be applied to the input star Θ for the FNN F focusing on the representation of the stars computed. Thus, similar to the previous example applying the exact reachability analysis, the approximate reachability analysis for ReLU is performed. The first step is the application of the same affine mapping as before. On the result Θ' of this, Algorithm 6 is applied for dimension $j = 1$ and the third case is applied as to before.*

Using the lower bound $l_1 = -2$ and upper bound $u_1 = 4$, this results in the following star represented in Figure 2.13:

$$\Theta'_1 = \left\langle \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 1 & 3 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ -2 & -1 & 0 \\ 0 & 0 & -1 \\ 2 & 1 & -1 \\ -\frac{4}{3} & -\frac{2}{3} & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \\ 2 \end{pmatrix} \right\rangle$$

Like before, the missing step in the calculation of the reachable set with the approximate algorithm is the application of ReLU to the second dimension of Θ'_1. Since Θ'_1 has a lower bound $l_2 = -5$ and an upper bound $u_2 = 3$ the computation of the third case is performed once more resulting in the following star:

$$\Theta'_2 = \left\langle \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 2 & 1 & -1 & 0 \\ -\frac{4}{3} & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 3 & 0 & -1 \\ -\frac{3}{8} & -\frac{9}{8} & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \\ 2 \\ 0 \\ 1 \\ \frac{3}{2} \end{pmatrix} \right\rangle$$

The LeakyReLU activation function can be applied using the splitting and approximation presented in Table 2.2 for any constant $\gamma \in (0, 1)$. Besides the additional constant, this activation function behaves the same way as ReLU.

The UnitStep activation function introduces three constants $R_{min}, R_{max}, v \in \mathbb{R}$. The Table 2.3 describing the corresponding cases for the algorithms uses $R_{min} = p$ and $R_{max} = q$ for better readability. This activation function appears similar to ReLU and LeakyReLU at first as all of them have a definition with two segments. This similarity appears for the exact computation as there are only three real cases. The third and fourth rows in the table define the same split. Both cases are still necessary due to the definition of the constraints for the over-approximating algorithm in the third case. These require division by $u_j - v$ which can become 0 if $u_j = v$ is not excluded. The last case dealing with this would require a third constraint $x_j \leq v$

$\mathsf{P}(l_j, u_j)$		Exact				Approximate	
l_j	u_j	Θ_i	$\boldsymbol{A}$	$\boldsymbol{b}$	Halfspaces	$\mathcal{B}$	Constraints
$[0, \infty)$	$[0, \infty)$	Θ_1	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	-	$\top$	-
$(-\infty, 0)$	$(-\infty, 0]$	Θ_1	$\boldsymbol{M}^{(n)}_{[\gamma]}$	$\boldsymbol{0}^{(n)}$	-	$\top$	-
$(-\infty, 0)$	$(0, \infty)$	Θ_1	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	$\mathcal{H}_1 : (0 \leq x_j)$	$\bot$	$\varphi_1 = \gamma x_j \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[\gamma]}$	$\boldsymbol{0}^{(n)}$	$\mathcal{H}_1 : (x_j \leq 0)$		$\varphi_2 = x_j \leq \alpha_{m+1}$
							$\varphi_3 = \alpha_{m+1} \leq \frac{x_j(u_j - \gamma l_j) + u_j l_j (\gamma - 1)}{u_j - l_j}$

Table 2.2: Cases for the reachability algorithms for the LeakyReLU activation function. In the column labeled $\mathsf{P}(l_j, u_j)$, the predicate corresponds to $l_j \in \mathsf{I}_l \wedge u_j \in \mathsf{I}_u$ for I_l the interval below l_j and I_u the interval below u_j. Note, that the columns $\boldsymbol{A}$ and $\boldsymbol{b}$ under Exact are also used for the exact computation in over-approximate reachability algorithms.

to define a bounded over-approximation. It is not necessary to add this additional constraint because $v = u_j$ already implies it. Note, that cases three and four for the over-approximation in Table 2.3 only apply if $p < q$. For $p > q$ the direction of all inequalities needs to be inverted. If otherwise $p = q$, the first or second case fully define the exact and over-approximate computation.

$\mathsf{P}(l_j, u_j)$		Exact				Approximate	
l_j	u_j	Θ_i	$\boldsymbol{A}$	$\boldsymbol{b}$	Halfspaces	$\mathcal{B}$	Constraints
$[v, \infty)$	$[v, \infty)$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	-	$\top$	-
$(-\infty, v)$	$(-\infty, v)$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	-	$\top$	-
$(-\infty, v)$	(v, ∞)	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	$\mathcal{H}_1 : (x_j \leq v)$	$\bot$	$\varphi_1 = p \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	$\mathcal{H}_1 : (v \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq q$
							$\varphi_3 = \frac{(q-p)x_j + u_j p - vq}{u_j - v} \leq \alpha_{m+1}$
							$\varphi_4 = \alpha_{m+1} \leq \frac{(q-p)x_j + vp - l_j q}{v - l_j}$
$(-\infty, v)$	$[v, v]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	$\mathcal{H}_1 : (x_j \leq v)$	$\bot$	$\varphi_1 = p \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	$\mathcal{H}_1 : (v \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq \frac{(q-p)x_j + vp - l_j q}{v - l_j}$

Table 2.3: Cases for the reachability algorithms for the UnitStep activation function. In the column labeled $\mathsf{P}(l_j, u_j)$, the predicate corresponds to $l_j \in \mathsf{I}_l \wedge u_j \in \mathsf{I}_u$ for I_l the interval below l_j and I_u the interval below u_j. Note, that the columns $\boldsymbol{A}$ and $\boldsymbol{b}$ under Exact are also used for the exact computation in over-approximate reachability algorithms.

$\mathsf{P}(l_j, u_j)$		Exact				Approximate	
l_j	u_j	Θ_i	$\boldsymbol{A}$	$\boldsymbol{b}$	Halfspaces	$\mathcal{B}$	Constraints
$(-\infty, p)$	$(-\infty, p)$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	-	$\top$	-
$[p, q]$	$[p, q]$	Θ_1	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	-	$\top$	-
(q, ∞)	$(q, \infty]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	-	$\top$	-
$(-\infty, p)$	$[p, q]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	$\mathcal{H}_1 : (x_j \leq p)$	$\bot$	$\varphi_1 = x_j \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	$\mathcal{H}_1 : (p \leq x_j)$		$\varphi_2 = p \leq \alpha_{m+1}$
							$\varphi_3 = \alpha_{m+1} \leq \frac{(u_j - p)x_j - (l_j - p)u_j}{u_j - l_j}$
$[p, q]$	$(q, \infty]$	Θ_1	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	$\mathcal{H}_1 : (x_j \leq q)$	$\bot$	$\varphi_1 = \alpha_{m+1} \leq q$
		Θ_2	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	$\mathcal{H}_1 : (q \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq x_j$
							$\varphi_3 = \frac{(l_j - p)x_j - (q - u_j)l_j}{u_j - l_j} \leq \alpha_{m+1}$
$(-\infty, p)$	(q, ∞)	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[p]}$	$\mathcal{H}_1 : (x_j \leq p)$	$\bot$	$\varphi_1 = p \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{I}^{(n)}$	$\boldsymbol{0}^{(n)}$	$\mathcal{H}_1 : (p \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq q$
					$\mathcal{H}_2 : (x_j \leq q)$		$\varphi_3 = \frac{(p - q)x_j - (q - u_j)p}{p - u_j} \leq \alpha_{m+1}$
		Θ_3	$\boldsymbol{M}^{(n)}_{[0]}$	$\boldsymbol{v}^{(n)}_{[q]}$	$\mathcal{H}_1 : (q \leq x_j)$		$\varphi_4 = \alpha_{m+1} \leq \frac{(p - q)x_j - (l_j - p)q}{q - l_j}$

Table 2.4: Cases for the reachability algorithms for the HardTanh activation function. In the column labeled $\mathsf{P}(l_j, u_j)$, the predicate corresponds to $l_j \in \mathsf{I}_l \wedge u_j \in \mathsf{I}_u$ for I_l the interval below l_j and I_u the interval below u_j. Note, that the columns $\boldsymbol{A}$ and $\boldsymbol{b}$ under Exact are also used for the exact computation in over-approximate reachability algorithms.

In the definition of the HardTanh activation function two values $V_{min} \leq V_{max} \in \mathbb{R}$ are introduced. For the purpose of readability, Table 2.4 describing the operations for the HardTanh activation functions use $V_{min} = p$ and $V_{max} = q$, similarly to UnitStep. Since this activation function is defined over three segments, the application of the reachability algorithm has to differentiate between six cases. In the first three cases, the representation of the star is a subset of one of the subregions. The representation of the star is subset of two of the segments in the fourth and fifth cases and subset of all in the last one. Due to the convexness of stars, an additional seventh case in which a set is included in the two not connected subregions can be disregarded.

The HardSigmoid activation function behaves similarly to HardTanh and also uses values $V_{min} \leq V_{max} \in \mathbb{R}$ once again written as p and q in Table 2.5. Additionally, we set $s = \frac{1}{q-p}$ and $t = \frac{p}{p-q}$.

Based on these tables, the complexity of the exact and over-approximating algorithms can be discussed. The worst-case runtime of reachability algorithms and consecutive verification of an FNN depend on the FNN itself and the constructed stars. More specifically, the number of neurons N in the FNN and the amount of generators and constraints of the predicate of the input star influence the runtime. The specific

$\mathsf{P}(l_j, u_j)$		Exact				Approximate	
l_j	u_j	Θ_i	$\boldsymbol{A}$	$\boldsymbol{b}$	Halfspaces	$\mathcal{B}$	Constraints
$(-\infty, p]$	$(-\infty, p]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	-	$\top$	-
(p, q)	(p, q)	Θ_1	$\boldsymbol{M}^{(n)}_{[s]}$	$\boldsymbol{v}^{(n)}_{[t]}$	-	$\top$	-
$[q, \infty)$	$[q, \infty]$	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	-	$\top$	-
$(-\infty, p]$	(p, q)	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (x_j \leq p)$	$\bot$	$\varphi_1 = 0 \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[s]}$	$\boldsymbol{v}^{(n)}_{[t]}$	$\mathcal{H}_1 : (p \leq x_j)$		$\varphi_2 = \frac{x_j + p}{q - p} \leq \alpha_{m+1}$
							$\varphi_3 = \alpha_{m+1} \leq \frac{(u_j - p)(x_j - l_j)}{(q-p)(u_j - l_j)}$
(p, q)	$[q, \infty]$	Θ_1	$\boldsymbol{M}^{(n)}_{[s]}$	$\boldsymbol{v}^{(n)}_{[t]}$	$\mathcal{H}_1 : (x_j \leq q)$	$\bot$	$\varphi_1 = \alpha_{m+1} \leq 1$
		Θ_2	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (q \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq \frac{x_j - p}{q - p}$
							$\varphi_3 = 1 + \frac{(q - l_j)(x_j - u_j)}{(q-p)(u_j - l_j)} \leq \alpha_{m+1}$
$(-\infty, p)$	(q, ∞)	Θ_1	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (x_j \leq p)$	$\bot$	$\varphi_1 = 0 \leq \alpha_{m+1}$
		Θ_2	$\boldsymbol{M}^{(n)}_{[s]}$	$\boldsymbol{v}^{(n)}_{[t]}$	$\mathcal{H}_1 : (p \leq x_j)$		$\varphi_2 = \alpha_{m+1} \leq 1$
					$\mathcal{H}_2 : (x_j \leq q)$		$\varphi_3 = \frac{x_j - l_j}{q - l_j} \leq \alpha_{m+1}$
		Θ_3	$\boldsymbol{M}^{(n)}_{[0]}$	$\mathbf{0}^{(n)}$	$\mathcal{H}_1 : (q \leq x_j)$		$\varphi_4 = \alpha_{m+1} \leq \frac{x_j - p}{u_j - p}$

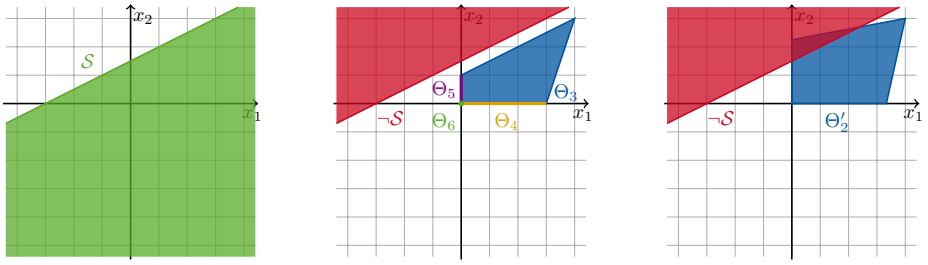
Table 2.5: Cases for the reachability algorithms for the HardSigmoid activation function. In the column labeled $\mathsf{P}(l_j, u_j)$, the predicate corresponds to $l_j \in \mathsf{I}_l \wedge u_j \in \mathsf{I}_u$ for I_l the interval below l_j and I_u the interval below u_j. Note, that the columns $\boldsymbol{A}$ and $\boldsymbol{b}$ under Exact are also used for the exact computation in over-approximate reachability algorithms.

worst-case values for these metrics depend on the activation function used and can be found in Table 2.6 which is based on [TMLM+19, Mas23].

In general, the amount of stars representing the reachable set using the exact algorithm is exponential in regard to N in the worst-case, because the exact algorithm can split an input star into multiple stars for each neuron regardless of the specific activation function. Despite this exponential worst-case number of stars, this number is often significantly lower in practice which allows the exact algorithm to be used.

At the same time, the amount of constraints only grows linearly regarding N and the dimension of the stars remains unchanged. The clear advantage of the over-approximating method lies in the representation of the reachable set of an FNN with only one star. To facilitate this, the dimension, specifically the amount of generators m, increases by a linear amount regarding N. The worst-case amount of added constraints also grows linearly regarding N with the over-approximating algorithm.

The last topic of this section is the verification and thus safety of FNNs based on reachability analysis. For this a safety specification or property is used. This safety specification has to be satisfied by the reachable set of an FNN for a given input set.



(a) Safety specification (b) Exact computation (c) Over-approximation

Figure 2.15: Figure 2.15a illustrates the safety specification of Example 2.3.4. Figure 2.15b shows the exact reachable set computed in Example 2.3.2 and unsafe set of Example 2.3.4 with an empty intersection. The over-approximated reachable set computed in Example 2.3.3 is presented in Figure 2.15c intersecting the unsafe set of Example 2.3.4.

Considering the reachability algorithms described before, the safety specification has to hold for all stars representing the reachable set. For the purpose of this thesis, safety is defined in the following way:

Definition 2.3.2 (Safety [TMLM+19])**.** *Assume an FNN F with $L \in \mathbb{N}_{\geq 2}$ layers and an input set $\mathcal{I} \subseteq \mathbb{R}^{\langle 1 \rangle}$ for F.*
A safety specification or property $\mathcal{S}$ for F is a union of $k \in \mathbb{N}_+$ convex polytopes with $\boldsymbol{A}^{(i)} \in \mathbb{R}^{s_i \times \langle L \rangle}$ and $\boldsymbol{d}^{(i)} \in \mathbb{R}^{s_i}$ for some $s_i \in \mathbb{N}_{\geq 1}$ and all $i \in \{1, \ldots, k\}$:

$$\mathcal{S} = \bigcup_{i=1}^{k} \left\{ \boldsymbol{y} \in \mathbb{R}^{\langle L \rangle} \;\middle|\; \boldsymbol{A}^{(i)} \boldsymbol{y} \leq \boldsymbol{d}^{(i)} \right\}$$

The unsafe set $\neg\mathcal{S}$ is defined as $\neg\mathcal{S} := \mathbb{R}^{\langle L \rangle} \setminus \mathcal{S}$.
F is called safe regarding an input set $\mathcal{I} \subseteq \mathbb{R}^{\langle 1 \rangle}$ if and only if the intersection with the unsafe set is empty:

$$\neg\mathcal{S} \cap \mathcal{R}_L = \emptyset$$

Example 2.3.4. *Consider the reachable sets computed in Examples 2.3.2 and 2.3.3 and the following safety specification represented in Figure 2.15a:*

$$\mathcal{S} = \left\{ \boldsymbol{y} \in \mathbb{R}^2 \;\middle|\; \begin{pmatrix} -1 & 2 \end{pmatrix} \boldsymbol{y} \leq 3 \right\}$$

Using the exact algorithm and checking the safety with the resulting reachable set represented by stars, requires checking the emptiness of four intersections with the unsafe set. All of these are empty as represented in Figure 2.15b. Thus, the FNN is safe regarding the input set.

Despite this actual safety, when using the over-approximated reachable set to check the safety of the FNN, the intersection is non-empty and thus the FNN would be falsely regarded as unsafe. This is represented in Figure 2.15c and shows the incompleteness of the over-approximating method.

If an FNN is unsafe, then the intersection of the unsafe set and the reachable set of the FNN is nonempty. The elements $\boldsymbol{z} \in \neg\mathcal{S} \cap \mathcal{R}_L$ in this intersection are

	Exact			Approximate		
	Stars	Constraints	Variables	Stars	Constraints	Variables
ReLU	$\mathcal{O}(2^N)$	$\mathcal{O}(p+N)$	m	1	$\mathcal{O}(p+3N)$	$\mathcal{O}(m+N)$
LeakyReLU	$\mathcal{O}(2^N)$	$\mathcal{O}(p+N)$	m	1	$\mathcal{O}(p+3N)$	$\mathcal{O}(m+N)$
HardTanh	$\mathcal{O}(3^N)$	$\mathcal{O}(p+2N)$	m	1	$\mathcal{O}(p+4N)$	$\mathcal{O}(m+N)$
HardSigmoid	$\mathcal{O}(3^N)$	$\mathcal{O}(p+2N)$	m	1	$\mathcal{O}(p+4N)$	$\mathcal{O}(m+N)$
UnitStep	$\mathcal{O}(2^N)$	$\mathcal{O}(p+N)$	m	1	$\mathcal{O}(p+4N)$	$\mathcal{O}(m+N)$

Table 2.6: The table contains worst-case amounts of stars, constraints of these and variables of the predicate. For these an FNN with N neurons and an (n, m)-dimensional input star with a predicate containing p linear constraints is assumed. The amounts for the application of the exact reachability algorithm is provided in the three columns labeled with Exact. The corresponding amounts for the over-approximating reachability algorithm is given under Approximate.

called counterexamples and establish the starting point for our reachability method introduced in the following chapter. All counterexamples $\boldsymbol{z}$ originate from the input set $\mathcal{I}$ when the exact method is used to compute the reachable set of an FNN F. This means that for all elements $\boldsymbol{z} \in \neg\mathcal{S} \cap \mathcal{R}_L$ an element $\boldsymbol{x} \in \mathcal{I}$ exists such that $F(\boldsymbol{x}) = \boldsymbol{z}$.

Counterexample of this kind also appear using the over-approximating method. In addition, *spurious* counterexamples may be introduced by an over-approximated activation function application. In this case, the counterexample does not originate from the input set. Therefore, finding a spurious counterexample neither implies the unsafety of the FNN nor its safety.

For the over-approximating computation it is not easy to differentiate a spurious from a non-spurious counterexample. For the exact method on the other hand, an input resulting in the counterexample can be computed based on the predicates of the stars representing the reachable set of the FNN. This is possible because the only modifications of the predicate of the input star are additional constraints. Thus, the predicates of stars representing reachable sets are always subsets of the predicate of the input star.

Proposition 2.3.1 (Counter Input [TMLM+19])**.** *Assume an FNN F with $L \in \mathbb{N}_{>2}$ layers, an input set $\mathcal{I} \subseteq \mathbb{R}^{\langle 1 \rangle}$ with $\mathcal{I} = [\![\langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle]\!]$ and a safety specification $\mathcal{S} \subseteq \mathbb{R}^{\langle L \rangle}$. Let $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P}' \rangle$ represent a reachable set of F with input set $\mathcal{I}$ computed using the exact algorithm and let $\boldsymbol{z} \in \neg\mathcal{S} \cap [\![\Theta']\!]$ be a counterexample. Then $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha}$ for some $\boldsymbol{\alpha} \in \mathcal{P}'$ and $F(\boldsymbol{x}) = \boldsymbol{z}$ for the counter input $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$.*

Based on the safety definition, the complexity of the verification of FNNs based on reachability analysis can be discussed. For a safety specification $\mathcal{S}$ defined by k polytopes with $s = \max\{s_i \mid i \in \{1, \ldots, k\}\}$ it is necessary to check the safety of each star representing a reachable set of an FNN F. This requires checking k feasibility problems [TMLM+19]. Assume F has N neurons and the input for the reachability analysis is a star with a predicate with p linear constraints. If the stars representing reachable sets are (n, m)-dimensional, these feasibility problems have a linear amount of constraints dependent on N, p, and s. The amount of variables is linear dependent on m. Note, that the k feasibility problem for all sets defining the safety specification

can be combined into one check for each star.

A more refined description of the amount of linear constraints can be derived from Table 2.6 based on the amount of constraints appearing in the worst-case for a specific activation function. The same holds for the amounts of variables for the feasibility problems and the amount of variables described in the table. Due to the need of solving the feasibility problem for each star in the reachable set, an exponential amount of these problems has to be solved in the worst-case when using the exact reachability algorithm for safety verification. This exponential amount of feasibility problems is the reason for using the over-approximating approach despite the incompleteness of the resulting verification, since the reachable set is fully represented by one star with a convex representation containing the exact solution.

2.4 Solver Z3

Z3 [dMB08] is the solver we use to solve the feasibility problems in the context of stars and the refinement algorithms introduced in the next chapter. This solver is more specifically a *satisfiability modulo theory* (SMT) solver made for the verification and analysis of software. This means that *Z3* is capable of checking the feasibility of first-order logic formulas over a variety of theories. Of these theories, we only use real arithmetic fragments. In most cases, linear real arithmetic formulas are sufficient for our requirements.

Definition 2.4.1 (Linear Arithmetic Formulas [DdM06])**.** *Let $n \in \mathbb{N}_+$. A linear arithmetic formula is a boolean combination of atoms or constraints of the following form*

$$\left(\sum_{i=1}^{n} a_i x_i\right) \circ d$$

for constants $a_1, \ldots, a_n, d \in \mathbb{Q}$, variables $x_1, \ldots, x_n$, and $\circ \in \{\leq, <, =, >, \geq\}$. A linear arithmetic formula is a linear real arithmetic formulas if the domains for the assignments of all variables are the real numbers. Analogously, a linear arithmetic formula is a linear integer arithmetic formula, if the domains for the assignments of all variables are the integers.

A linear real arithmetic formula can be used to represent any polytopes $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d}\}$ for $m, p \in \mathbb{N}_+$, $\boldsymbol{A} \in \mathbb{R}^{p \times m}$, and $\boldsymbol{d} \in \mathbb{R}^p$. The corresponding formula consists of p constraints. Each constraint matches a row in the matrix $\boldsymbol{A}$ and vector $\boldsymbol{d}$. To derive constraint $i \in \{1, \ldots, p\}$, the calculation $\boldsymbol{A}\boldsymbol{\alpha}$ is performed symbolically and represented by an atom. Thus, $\sum_{j=1}^{m} a_{i,j}\alpha_j \leq d_i$ describes the i-th constraint defining $\mathcal{P}$. Since an element of $\mathcal{P}$ needs to satisfy $\boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d}$ for all dimensions $i \in \{1, \ldots, p\}$, the conjunction $\psi_{\mathcal{P}} := \bigwedge_{i=1}^{p} \sum_{j=1}^{m} a_{ij}\alpha_j \leq d_i$ of all constraints needs to be satisfied by elements of $\mathcal{P}$. In the following, $\psi_{\mathcal{P}}$ describes this formula for any polytope $\mathcal{P}$. Note, that this formula is also used for the predicates of stars in Section 2.2 and can be used to describe a singular polytope in the union defining a safety specification.

In addition to linear arithmetic formulas, we require non-linear integer arithmetic formulas for the method described in Section 3.2.2 in the following chapter.

Definition 2.4.2 (Non-Linear Arithmetic Formulas [CGI+18])**.** *Let $n, m \in \mathbb{N}$. For constants $a_1, \ldots, a_n, d \in \mathbb{Q}$ as well as $b_{1,1}, \ldots, b_{1,m}, \cdots, b_{n,m} \in \mathbb{N}$, relation $\circ \in$*

$\{\leq, <, =, >, \geq\}$, *and variables* $x_1, \ldots, x_m$ *a non-linear arithmetic formula is a boolean combination of atoms or constraints of the following form*

$$\left(\sum_{i=1}^{n} \left(a_i \cdot \prod_{j=1}^{m} x_j^{b_{i,j}} \right) \right) \circ d$$

For non-linear real arithmetic formulas, the domains for the assignments of all variables are the real numbers. For non-linear integer arithmetic formulas, the domains for the assignments of all variables are the integers.

Z3 is capable of identifying a model for a given formula of the described types as long as the formula is satisfiable. For the formulas we use, such a model is an assignment of values to all variables such that the formula is satisfied. These values are represented by infinite precision arithmetic [BdMNW]. This means a value $x \in \mathbb{Q}$ is represented by two integers $n, d \in \mathbb{Z}$ of arbitrary size such that $x = \frac{n}{d}$. By representing numbers this way, exact computation and thus soundness of the computation can be assured. The price for this precision are potentially large representations which decreases performance. Regardless of this disadvantage we still use this a number representation for the implementation of all algorithms presented in this work.

For unsatisfiable formulas, *Z3* can produce an (unsatisfiable) core instead of a model. Such a core is a subset of the constraints of a given formula that is already unsatisfiable. Note, that these subsets are not minimal in general.

Chapter 3

Verification with CEGAR

In this chapter, our verification method for FNN using *counterexample-guided abstraction refinement* (CEGAR) is introduced. This method combines the over-approximating and exact reachability analysis introduced in the previous chapter to gain the advantages of both approaches. Thus, our verification method achieves the completeness of the exact approach. In addition, the number of resulting reachable sets is reduced. This leads to a reduced amount of safety checks for the verification of FNNs. This combination is achieved by using over-approximating computation whenever possible and exact computation whenever necessary. As introduced in the previous chapter, the only over-approximated operation of the reachability analysis is the application of activation functions. This is thus the only type of operation for which either exact or over-approximating computation needs to be chosen. Affine mappings are always computed efficiently with the exact method introduced before.

As stated before, the over-approximating approach is applied whenever possible. Thus, the initial abstraction corresponds to the over-approximate reachability analysis. This abstraction is then refined until either the safety or unsafety of the FNN is proven. If the initial abstraction includes an unsafe reachable set, we identify the reason for this unsafety. This search is based on the potentially spurious counterexample found during the safety check. The counterexample is traced to an origin, which is either the input set or an over-approximated activation function application. If the counterexample originates from the input set, we find a counter input and thus the actual unsafety of the FNN. Otherwise, the counterexample is introduced by over-approximation and thus spurious. Therefore, the abstraction needs to be refined. For this, the over-approximated operation that is responsible for the identified introduction of the counterexample is replaced by the corresponding exact operation. This process of refinement is repeated until either the unsafety of the FNN can be proven with a counter input or all resulting reachable sets satisfy the safety specification. This satisfaction implies the safety of the FNN.

An overview of this refinement process is provided in Figure 3.1. In this figure, a reachability tree is constructed based on reachability analysis. This data structure is introduced in Section 3.1 and allows access to all reachable sets computed during the analysis. Our methods for the identification of counterexamples corresponds to safety checking and are introduced in Section 3.2. Based a reachability tree and a spurious counterexample, the tracing methods introduced in Section 3.3 identify an operation responsible for the counterexample. If the counterexample, is not spurious

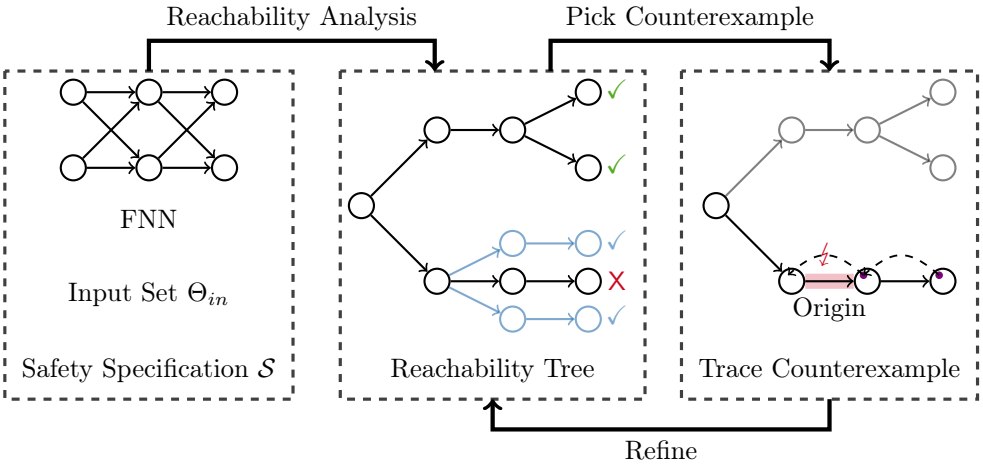


Figure 3.1: Verification with CEGAR

these methods instead provide a counter input.

The refinement algorithms introduced in Section 3.4 combine reachability analysis, safety checking, and tracing into our verification method. Due to the cyclic nature of this process, all of these methods can use the information gained during previous cycles. This is also visualized in Figure 3.1 where the blue subtrees of the central reachability tree are the result of refinement. Because multiple methods and heuristics for all of these steps are introduced in their respective sections, the motivation for some them depends on the methods introduced at a later point. This motivation will still be explained.

3.1 Reachability Trees

The *reachability tree* is the data structure we use as an abstraction for a reachability analysis of an FNN. The nodes in this tree contain the reachable sets that are computed during this analysis. The root of such a tree thus represents the input to the reachability analysis. Reachable sets are still represented as stars. Therefore, we sometimes use nodes and the stars they contain interchangeably.

The structure of a reachability tree corresponds to the structure of an FNN. More specifically, the operations in the FNN are applied to reachable sets in nodes to calculate the children of that node. These operations are the affine mappings and the dimension-wise activation functions applications. For the algorithms in this chapter and the definition of reachability trees, we define these operations on stars. In the following definitions, an operation application returns a set of stars according to reachability analysis. The operation that applies the affine mapping corresponds to line 5 in Algorithms 1 and 2.

Definition 3.1.1 (Affine Mapping Operation)**.** *Consider a weight* $\boldsymbol{W}^{(i)} \in \mathbb{R}^{\langle i \rangle \times \langle i-1 \rangle}$ *and bias* $\boldsymbol{b}^{(i)} \in \mathbb{R}^{\langle i \rangle}$ *for layer* $i \in \{2, \ldots, L\}$ *for some FNN with* $L \in \mathbb{N}_{\geq 2}$ *layers. Let* Θ *further be a* $(\langle i-1 \rangle, m)$*-dimensional star with* $m \in \mathbb{N}_+$ *representing a reachable set. Then the operation* $op(\cdot)$ *corresponding to the affine mapping in the FNN applied to* Θ

is defined as

$$op(\Theta) = \left\{ \boldsymbol{W}^{(i)}\Theta + \boldsymbol{b}^{(i)} \right\}$$

were $\boldsymbol{W}^{(i)}\Theta + \boldsymbol{b}^{(i)}$ *is the typical application of an affine mapping to a star.*

The operation for the application of activation functions corresponds to Algorithms 3 and 4.

Definition 3.1.2 (Exact Activation Function Operation)**.** *Consider an activation function application to neuron* $j \in \{1, \ldots, \langle i \rangle\}$ *for* $i \in \{2, \ldots, L\}$ *for an FNN with* $L \in \mathbb{N}_{\geq 2}$ *layers. Let* Θ *further be a* $(\langle i \rangle, m)$*-dimensional star with* $m \in \mathbb{N}_+$ *representing a reachable set. Then the operation* $op(\cdot)$ *corresponding to the exact application of the activation function in the FNN applied to* Θ *is defined as*

$$op(\Theta) = \texttt{exactAct}(\Theta, j)$$

were `exactAct` *refers to Algorithm 3 with an implicit input of a list of cases corresponding to the activation function. The output of* `exactAct` *is a list of stars. For this definition, the list is interpreted as a set of stars instead.*

Definition 3.1.3 (Over-Approximated Activation Function Operation)**.** *Consider an activation function application to neuron* $j \in \{1, \ldots, \langle i \rangle\}$ *for* $i \in \{2, \ldots, L\}$ *for an FNN with* $L \in \mathbb{N}_{\geq 2}$ *layers. Let* Θ *further be a* $(\langle i \rangle, m)$*-dimensional star representing a reachable set for* $m \in \mathbb{N}_+$*. Then the operation* $op(\cdot)$ *corresponding to the over-approximated application of the activation function in the FNN applied to* Θ *is defined as*

$$op(\Theta) = \{\texttt{approximateAct}(\Theta, j)\}$$

were `approximateAct` *refers to Algorithm 4 with an implicit input of a list of cases corresponding to the activation function.*

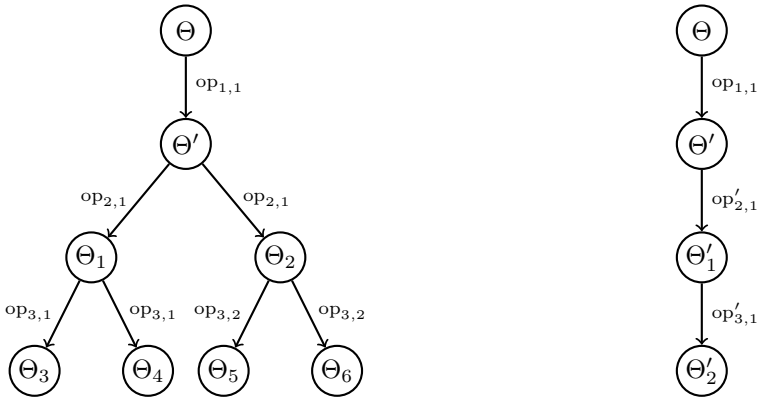
The operations are applied based on the order in the FNN. Thus, for a single layer, the first operation is an affine mapping. This is followed by the application of the activation function to each neuron in ascending order. The overall order of layers corresponds to the order of operations in the whole FNN which defines an order of all operations. Note, that a single operation corresponding to an activation function application can be either exact or over-approximating.

In an abuse of notation, we also identify these operations with the corresponding function applications in the FNN. An operation $\text{op}(\cdot)$ can thus be applied to a value $\boldsymbol{x}$ directly. In this case we write $\text{op}(\boldsymbol{x}) = \boldsymbol{y}$. While this notation does use stars or sets of stars anymore, the output is still well-defined. $\boldsymbol{x}$ can be represented by a star containing only this value. In this case, the operation would return a set only containing the star that has $\boldsymbol{y}$ as its only element.

Based on these operations we can now define a reachability tree.

Definition 3.1.4 (Reachability Tree)**.** *Let* F *be a FNN with* $L \in \mathbb{N}_{\geq 2}$ *layers and* $N = \sum_{l=2}^{L} \langle l \rangle$ *neurons after the input layer, let* $\mathcal{I} = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ *represent the input set for the verification of* F*, let* r_i *be the amount of stars representing the reachable set after the application of* $i \in \{1, \ldots, L + N - 1\}$ *operations, and let* $op_{i,j}(\cdot)$ *be an instance of the* i*-th operation applied in* F *for* $j \in \{1, \ldots, r_{i-1}\}$*. Then a full reachability tree* $T = (\mathsf{V}, \mathsf{E})$ *is defined with a vertex set* $\mathsf{V} = \bigcup_{i=0}^{L+N-1} \mathsf{V}_i$ *for* $\mathsf{V}_0 = \{\Theta_{0,1}\} := \{\mathcal{I}\}$ *and*

$$\mathsf{V}_i = \{\Theta_{i,1}, \ldots, \Theta_{i,r_i}\} := \bigcup_{j=1}^{r_{i-1}} op_{i,j}(\Theta_{i-1,j})$$



(a) Exact reachability tree (b) Over-approximating reachability tree

Figure 3.2: Figure 3.2a is a representation for the exact reachability analysis in Example 2.3.2. Figure 3.2b is a representation for the over-approximating reachability analysis in Example 2.3.3.

and a set of edges

$$\mathsf{E} := \bigcup_{i=1}^{N+L-1} \left\{ (\Theta_{i-1,j}, \Theta') \in \mathsf{V}_{i-1} \times \mathsf{V}_i \;\middle|\; \Theta' \in op_{i,j}(\Theta_j), j \in \{1, \ldots, r_{i-1}\} \right\}$$

A (partial) reachability tree $T' = (\mathsf{V}', \mathsf{E}')$ *is defined as a subtree of a full reachability tree* (V, E) *such that* T' *is a tree,* $\mathsf{V}' \subseteq \mathsf{V}$*,* $\mathsf{V}_0 \subseteq \mathsf{V}'$*, and* $\mathsf{E}' = \mathsf{E} \cap (\mathsf{V}' \times \mathsf{V}')$*. Additionally, if* $\Theta, \Theta' \in \mathsf{V}'$ *with* $\Theta \in op_{i,j}(\Theta')$ *for any* $i \in \{1, \ldots, N+L\}$ *and* $j \in \{1, \ldots, r_{i-1}\}$*, then* $op_{i,j} \subseteq \mathsf{V}'$*.*

Note, that an instance of an operation $\mathrm{op}_{i,j}(\cdot)$ always refers to the same affine mapping, if the i-th operation of the FNN is an affine mapping. For activation function application, this operation either refers to an exact or over-approximating application. Reachability trees corresponding to the exact reachability analysis only contain exact activation function operations, while reachability trees for over-approximating reachability analysis only contain over-approximated operations. A reachability tree for the method introduced in this chapter may contain both exact and over-approximated activation function operations. This means, that for $i \in \{1, \ldots, N+L-1\}$ and $j, k \in \{1, \ldots, r_{i-1}\}$, operation $\mathrm{op}_{i,j}$ may be an exact activation function application, while operation $\mathrm{op}_{i,k}$ is over-approximating.

Example 3.1.1. *The reachability analysis described in Examples 2.3.2 and 2.3.3 are represented by the reachability trees in Figure 3.2. The FNN used for this analysis consists only of an input and output layer. Therefore, the operations applied to the input set are an affine mapping followed by two ReLU activation function operations. Thus, operation* $op_{1,1}$ *is the same affine mapping operation in both trees.*

In the reachability tree represented in Figure 3.2a, operation $op_{2,1}$ *is the exact application of the ReLU activation function to the first dimension* $j = 1$*. As seen in the example describing this reachability analysis, this computation results in two sets representing the reachable set of the FNN up to this point. Thus, the node* Θ' *has two children representing one of these sets each. Corresponding to the FNN, operations*

$op_{3,1}$ and $op_{3,2}$ are both exact applications of the ReLU activation function to the second dimension $j = 2$. The remaining nodes in the tree result from the same process as before.

The over-approximated reachability analysis from Example 2.3.3 described by the reachability tree in Figure 3.2b applies over-approximated ReLU operations in both dimensions instead of the exact computation before. Therefore, operation $op'_{2,1}$ is this application in dimension $j = 1$ and $op'_{3,1}$ is the application in dimension $j = 2$.

In a full reachability tree, each leaf represents a subset of the reachable set $F(\mathcal{I})$ of an FNN F for an input set $\mathcal{I}$. This representation is, in general, an over-approximation, if at least one operation on the path from the root of the reachability tree to the leaf is over-approximated. The union of all leaves in a full reachability tree corresponds to a reachable set $F(\mathcal{I})$ or an over-approximation thereof, because all operations of the FNN are applied to the input. In a partial reachability tree, leaves that do not correspond to the reachable set $F(\mathcal{I})$ can occur, if not all operations are applied to a specific path from the root. Leaves representing such a partial computation are referred to as non-final leaves, while leaves resulting from the application of all operations of the FNN are called final.

As final leaves contain stars representing $F(\mathcal{I})$, these leaves are directly relevant for safety checking. If the set contained in a final leaf is safe regarding some specification, we consider the leaf to be safe. If all leaves reachable from a node in the reachability tree are final and safe, this node is considered safe. The safety of the root node then implies the safety of the FNN corresponding to the reachability tree. Note, that while the safety of an FNN can only be proven by a full reachability tree, it is sufficient to find a single final leaf to identify a counter input proving its unsafety.

3.2 Safety and Counterexamples

In this section, we once more look into safety as defined in Definition 2.3.2 and the process to derive a counterexample from this. More specifically, the safety of reachable sets of an FNN represented by the final leaves of a reachability tree will be explored. To this end, we assume an FNN F, a corresponding safety specification $\mathcal{S}$, and a reachable set Θ represented by a final leaf of a reachability tree corresponding to F. As described in Definition 2.3.2, the safety of Θ regarding $\mathcal{S}$ can be derived by checking the intersection of Θ and $\neg\mathcal{S}$ for emptiness. Equivalently, the unsafety of Θ is proven, if a counterexample $\boldsymbol{z} \in (\llbracket\Theta\rrbracket \cap \neg\mathcal{S})$ exists. We use this second option and obtain a counterexample by checking the feasibility of the following formula

$$\psi_{\mathcal{S},\Theta} \equiv \boldsymbol{z} \in \llbracket\Theta\rrbracket \wedge \boldsymbol{z} \notin \mathcal{S} \tag{3.1}$$

If a value $\boldsymbol{z}$ that satisfies $\psi_{\mathcal{S},\Theta}$ exists, then $\boldsymbol{z}$ is a counterexample. Since the verification with CEGAR generally uses a combination of exact and over-approximating computation, this counterexample can be spurious. Our method to determine the type of counterexample is the topic of the next section.

The abstract formula $\psi_{\mathcal{S},\Theta}$ needs to be transformed into an equivalent real arithmetic formula to allow us to check its feasibility. The formula we use will even be linear. To transform $\psi_{\mathcal{S},\Theta}$ we make use of the definition of an element in the representation of a star introduced in Definition 2.2.1. Based on this definition, we search for an element of the predicate of Θ corresponding to the counterexample.

Algorithm 7: Safety Checking Algorithm

input : (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Safety specification $\mathcal{S} = \bigcup_{i=1}^{k} S_i$
output : Pair of counterexample and predicate value $(\boldsymbol{z}, \boldsymbol{\alpha})$
Or a pair of empty tuples $((), ())$

```
1 ψ_P ← constructFormula(P)
2 for S_i ∈ S do
3 |  ψ_{S_i} ← constructFormula(S_i)
4 α ← checkFeasibility(ψ_P(α) ∧ ⋀_{i=1}^{k} ¬ψ_{S_i}(c + Vα))
5 if α = () then // [[Θ]] ⊆ S
6 |  return ((), ())
7 else // (c + Vα) ∈ ([[Θ]] ∩ ¬S)
8 |  return (c + Vα, α)
```

Proposition 3.2.1 (Counterexamples)**.** *Assume an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and a safety specification $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i \subseteq \mathbb{R}^n$. Let $\psi_{\mathcal{P}}, \psi_{\mathcal{S}_1}, \ldots, \psi_{\mathcal{S}_k}$ be the linear real arithmetic formulas corresponding to $\mathcal{P}, \mathcal{S}_1, \ldots, \mathcal{S}_k$ as introduced in Section 2.4. Then all values $\boldsymbol{\alpha} \in \mathbb{R}^m$ satisfying the following formula correspond to counterexamples $\boldsymbol{z} \in [\![\Theta]\!] \cap \neg\mathcal{S}$ with $\boldsymbol{z} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$:*

$$\psi_{\mathcal{S},\Theta}(\boldsymbol{\alpha}) := \psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{k} \neg\psi_{\mathcal{S}_i}(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha})$$

Based on this proposition, finding a satisfying assignment $\boldsymbol{\alpha}$ for $\psi_{\mathcal{S},\Theta}$ proves the unsafety of the reachable set Θ. In addition, such an assignment is a predicate value $\boldsymbol{\alpha} \in \mathcal{P}$ corresponding to a counterexample $(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}) \in [\![\Theta]\!]$ for $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$. Both the counterexample and the predicate value are relevant for tracing the counterexample to its origin. If $\psi_{\mathcal{S},\Theta}$ is unsatisfiable, no counterexamples exist which implies the safety of Θ regarding $\mathcal{S}$. The feasibility of this formula provides the primary method of safety checking for our verification method.

For the safety of a star Θ regarding a safety specification $\mathcal{S}$, this process is captured in Algorithm 7. The algorithm starts by constructing the formula $\psi_{\mathcal{S},\Theta}$ as described in Proposition 3.2.1. To this end, the function `constructFormula` takes a convex bounded polytope and returns the corresponding formula. Then the function `checkFeasibility` is used to check the feasibility of $\psi_{\mathcal{S},\Theta}$. This function returns the empty tuple (), if the formula is infeasible and a model $\boldsymbol{\alpha}$ of $\psi_{\mathcal{S},\Theta}$ otherwise. This check relies on the solver introduced in Section 2.4. Afterwards the function either returns $((), ())$ to indicate the absence of a counterexample and thus the safety of Θ or the counterexample $\boldsymbol{z}$ and the corresponding predicate value $\boldsymbol{\alpha}$.

3.2.1 Safety Checking with Previous Counterexamples

During the refinement process explained in the following Section 3.4 the safety of multiple reachable sets is checked and typically multiple of them are unsafe. If this heuristic is applied, the algorithm keeps track of all counterexamples that are found during safety checks. Then it is checked whether any previous counterexample is included in the current reachable set as a first step. If a previous counterexample is

Algorithm 8: Algorithm for Safety Checking with Previous Counterexamples

input : (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Safety specification $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i$,
Set of previous counterexample Z

output : Pair of counterexample and predicate value $(\boldsymbol{z}, \boldsymbol{\alpha})$
Or a pair of empty tuples $((), ())$

$\psi_{\mathcal{P}} \leftarrow$ `constructFormula`$(\mathcal{P})$
for $\boldsymbol{z} \in Z$ **do**
  for $i \leftarrow 1$ **to** n **do**
    $\varphi_i \leftarrow$ `constructFormulaEq`$(\Theta, \boldsymbol{z}, i)$
  $\boldsymbol{\alpha} \leftarrow$ `checkFeasibility`$(\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \neg\varphi_i(\boldsymbol{\alpha}))$
  if $\boldsymbol{\alpha} \neq ()$ **then** // $\boldsymbol{z} \in [\![\Theta]\!]$
    return $(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}, \boldsymbol{\alpha})$ // $\boldsymbol{z} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$
return `checkSafety`$(\Theta, \mathcal{S})$

still element of the current reachable set, it is also a counterexample for it. Thus, the reachable set is unsafe. If none of the previous counterexamples are element of the current reachable set, the safety of the set is checked using the previous method.

The predicate and its dimension may differ based on the combination of exact and over-approximated operations applied to obtain the reachable set. Thus, it is not sufficient to check if the previous predicate value $\boldsymbol{\alpha}$ is still in the predicate of the current reachable set. Instead, a feasibility problem has to be solved for a formula that indicates this containment. This requires finding an element of the predicate that is equal to the counterexample after application of the affine mapping defining the star. For a (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ with a predicate value $\boldsymbol{\alpha} \in \mathcal{P}$ and a value $\boldsymbol{x} \in \mathbb{R}^n$, the equality of the value in the star based on $\boldsymbol{\alpha}$ and $\boldsymbol{x}$ in dimension $i \in \{1, \ldots, n\}$ is described by the following formula based on Definition 2.2.1:

$$\varphi_i(\boldsymbol{\alpha}) := \left(c_i + \sum_{j=1}^{m} v_{ij}\alpha_j = x_i \right) \tag{3.2}$$

Based on this formula, the following proposition defines a formula for the inclusion of a value $\boldsymbol{x}$ in a star.

Proposition 3.2.2 (Elements of Stars)**.** *Assume an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ with predicate $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \psi_{\mathcal{P}}(\boldsymbol{\alpha})\}$ and a value $\boldsymbol{x} \in \mathbb{R}^n$. Then $\boldsymbol{x} \in [\![\Theta]\!]$ if and only if the following formula using is feasible*

$$\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \varphi_i(\boldsymbol{\alpha})$$

for $\varphi_i(\boldsymbol{\alpha})$ defined according to Formula 3.2.

The formula introduced in Proposition 3.2.1 checks the non-containment of the counterexample. This property of the counterexample does not change for a different reachable set. Thus, if a previous counterexample is included in the reachable set, it is a counterexample. Further checks are not required.

Algorithm 8 describes this process in the form of an algorithm. The input is an extension of the input for the previous safety checking algorithm. In addition to the reachable set and safety specification, a set of all previous counterexample is required. Based on this input, the formulas for the containment check of a previous counterexample are constructed using the function calls `constructFormula` and `constructFormulaEq`. `constructFormula` is the same function as before, while `constructFormulaEq` constructs Formula 3.2. The feasibility check with the function call `checkFeasibility` works in the same way as before. Thus, if this check finds feasibility, a previous counterexample is also a current counterexample which is thus returned. Otherwise, the algorithm continues with checking the other previous counterexamples. Once all of them are checked and none are current counterexample, the algorithm calls Algorithm 7 for the basic safety checking method which is referenced with the function name `checkSafety`.

3.2.2 Reducing Representation Size

The counterexamples produced by the previous methods are represented in an exact manner as introduced in Section 2.4. This representation can become very large which reduces the performance of our algorithms. In our early experiments, we encountered counterexamples with hundreds of digits in their representation. The two methods to derive counterexamples described in the following are our attempts to reduce this number of digits. Instead of increasing performance both attempts resulted in a performance decrease with such significant that verification becomes infeasible. These methods are thus not used for verification.

The basic idea for both methods is to minimize the sum of the absolute values of the numerators and denominators of all variable values. Finding a satisfying assignment to a formula that is minimized regarding an affine transformation of the variables is possible with *Z3*. Absolute values on the other hand cannot be used for minimization. In addition, the numerators and denominators of the representation of variables cannot be accessed. Thus, the formulas typically used to check safety need to be changed to allow this behavior.

To this end, a real variable α_i is represented by $\sigma_i \cdot \frac{\eta_i}{\delta_i}$ for integer variables σ_i, η_i, and δ_i. In this representation, σ_i represents the sign of α_i and is thus restrained to 1 or -1. By restraining the numerators η_i to the non-negative integers and the denominators δ_i to the positive integers, these values are equal to their absolute value. The final hurdle for this representation is that *Z3* does not support division. Thus, the representation $\sigma_i \cdot \frac{\eta_i}{\delta_i}$ of α_i needs to be transformed in the final formulas.

The first method to reduce the representation size of counterexamples, adds the new representation on top of the formula $\psi_{\mathcal{S},\Theta}$ presented in Proposition 3.2.1. This means, that for all variables α_i the equality $\alpha_i \cdot \delta_i = \sigma_i \cdot \eta_i$ and the aforementioned constraints on the new variables σ_i, η_i, and δ_i need to be satisfied in addition. The resulting formula is thus a non-linear arithmetic formula with mixed integer and real variables. For an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and a safety specification $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i$ the resulting formula is of the following form:

$$\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{k} \neg\psi_{S_i}(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{m} (\alpha_i \cdot \delta_i = \sigma_i \cdot \eta_i)$$
$$\wedge \bigwedge_{i=1}^{m} ((\sigma_i = 1 \vee \sigma_i = -1) \wedge \eta_i \geq 0 \wedge \delta_i \geq 1)\,.$$

The second method substitutes the variables α_i in $\psi_{\mathcal{S},\Theta}$ with the new representation $\sigma_i \cdot \frac{\eta_i}{\delta_i}$ instead. For a polytope $\{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d}\}$ with $\boldsymbol{A} \in \mathbb{R}^{m \times p}$ and $\boldsymbol{d} \in \mathbb{R}^p$ the

following formula results from this substitution:

$$\bigwedge_{i=1}^{p} \sum_{j=1}^{m} \left(a_{ij} \cdot \frac{\sigma_j \cdot \eta_j}{\delta_i} \right) \leq d_i.$$

Since division cannot be represented directly by *Z3*, it is necessary to use the following equivalent formula:

$$\bigwedge_{i=1}^{p} \sum_{j=1}^{m} \left(a_{ij} \cdot \sigma_j \cdot \eta_j \cdot \prod_{\substack{k=1 \\ k \neq j}}^{m} \delta_k \right) \leq d_i \prod_{k=1}^{m} \delta_k$$

This second method thus checks feasibility based on $\psi_{\mathcal{S},\Theta}$ and these non-linear integer arithmetic formulas for the polytopes $\mathcal{P}$, $\mathcal{S}_1, \ldots, \mathcal{S}_k$.

Checking the feasibility for either of these types of formulas is less efficient than checking the feasibility of linear real arithmetic formulas. In fact, checking safety with these formulas is so inefficient that even for very fast verifications no assignment or lack thereof can be found in a reasonable time frame. With this result, we stopped looking for further options to reduce the number representation.

3.3 Tracing Counterexamples

The last step before the refinement algorithms performing verification can be introduced is the identification of an origin of a counterexample. Such a counterexample is derived during the safety checks of final leaves. The origin of a counterexample is either a counter input as defined in Proposition 2.3.1 or an over-approximated operation during the reachability analysis. In the first case, the counterexample is non-spurious which implies the unsafety of the FNN. Finding such a counter input requires tracing the counterexample to the root of the reachability tree. Identifying an operation as the origin of a counterexample also relies on tracing the counterexample on the shortest path to the root of the corresponding reachability tree. In both cases, our tracing method relies on finding values that lead to the counterexample. Such a value is called a source of the counterexample. This phrase is based on the following general definition for a source.

Definition 3.3.1 (Sources)**.** *Let T be a reachability tree with a root node Θ_0, a final leaf Θ_t, and a path $(\Theta_0, \ldots, \Theta_t)$ in the reachability tree. Let further $op_1(\cdot), \ldots, op_t(\cdot)$ be the sequence of operations such that $\Theta_i \in op_i(\Theta_{i-1})$ for all $i \in \{1, \ldots, t\}$. Then an element $\boldsymbol{y} \in [\![\Theta_{i-1}]\!]$ with $op_j(op_{j-1}(\cdots op_i(\boldsymbol{y}) \cdots)) = \boldsymbol{y'} \in [\![\Theta_j]\!]$ is called a source of $\boldsymbol{y'}$ in Θ_j for $i \in \{1, \ldots, t\}$ and $j \in \{i, \ldots, t\}$.*

Based on this definition, a source of a counterexample is a value in a reachable set which is mapped to the counterexample by the remaining computations of the FNN. Note, that the computation for an operation on singular values is always performed in an exact manner. This means, that values that are only in a reachable set due to over-approximation do not necessarily have a source in each prior node of the reachability tree. Based on this definition of sources, origins of counterexamples can be defined as follows.

Definition 3.3.2 (Origin of a Counterexample)**.** *Let T be a reachability tree with a root node Θ_0, a final leaf Θ_t with a counterexample $\boldsymbol{z} \in [\![\Theta_t]\!]$, and a path $(\Theta_0, \dots, \Theta_t)$ in the reachability tree. Let further $op_1(\cdot), \dots, op_t(\cdot)$ be the sequence of operations such that $\Theta_i \in op_i(\Theta_{i-1})$ for all $i \in \{1, \dots, t\}$. An origin of a counterexample is either a counter input $\boldsymbol{x} \in [\![\Theta_0]\!]$ with $F(\boldsymbol{x}) = \boldsymbol{z}$ or an operation $op_i(\cdot)$ for $i \in \{1, \dots, t\}$ such that a source $\boldsymbol{y} \in [\![\Theta_i]\!]$ of $\boldsymbol{z}$ exists and no element $\boldsymbol{y}' \in [\![\Theta_{i-1}]\!]$ exists that is a source of $\boldsymbol{y}$.*

In the following, we introduce different methods to trace a counterexample to its origin based on a single value. This begins with a basic method that iteratively finds sources in parent nodes until an origin is found in Section 3.3.1. This quite inefficient method is then improved by tracing a source of a counterexample through sequences of activation function operations. In Section 3.3.2 this will be done with a binary search based method. This process is then amended by testing if previous origins are still origins of counterexamples in Section 3.3.3. Afterwards, we change our method of search in Section 3.3.4. This new method makes use of the information we gain from feasibility checks during the tracing process.

3.3.1 Basic Tracing

As mentioned before, this basic tracing method identifies an origin of a counterexample by finding a source in one node after the other. These nodes form a path from an unsafe final leaf to the root of a reachability tree. This, a new source of the original counterexample contained in the parent node is identified in the form of a source for the already known source. This process stops once an origin is identified. Such an origin is an over-approximated activation function operation if no new source can be identified. If a source for the counterexample is traced to the root, the origin is a counter input.

The process to identify a new source in the parent node either utilizes the star representation of the nodes in a reachability tree or a solution to a feasibility problem. This feasibility problem describes the revers application of the operation with which the node containing the known source was computed. Any new source found in either way is then a source of the counterexample contained in a node further on the path to the root.

The utilization of the representation will always produce a new source, but only works for affine operations. The feasibility problem on the other hand can become infeasible indicating that the known source does not have a new source in the parent node. In this case, the operation in between the current pair of child node and parent node is an origin of the counterexample. It is necessary to differentiate between affine operations and activation function operations to describe this process in more detail. It is not necessary to also distinguish exact and over-approximated activation function operations.

Given a child node with a known source that is the result of an affine operation, a source in the parent node can be identified in the same way counter inputs are derived for the exact reachability analysis following Proposition 2.3.1. Thus, a source can be calculated by simply using the same predicate value as before.

Proposition 3.3.1 (Tracing through Affine Operations)**.** *Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ such that $\Theta' \in op(\Theta)$ for an affine operation $op(\cdot)$. For an element $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} \in [\![\Theta']\!]$ with $\boldsymbol{\alpha} \in \mathcal{P}$, the element $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \in [\![\Theta]\!]$ is a source of $\boldsymbol{z}$.*

Thus, no feasibility problem needs to be solved to trace a counterexample through this operation type, if the predicate value is kept track of. The use of the same predicate value relies on the lack of change to the predicate of a star when an affine operation is applied. Note, that a parent node computing a child node via an affine operation always contains at least one source for each element of the reachable set represented by the child due to the exactness of this operation. On the other hand, multiple sources for the child node's source can generally be contained in the parent node.

The tracing process requires solving a feasibility problem for each activation function operation. Because activation functions are applied to only one dimension at a time, the remaining dimensions remain unchanged. This needs to be represented by the formula for feasibility checking. For the changing dimension, the formula represents the application of the activation function to a potential new source. In all other dimensions, this new source needs to equal the known source. Such a formula is thus similar to the formula introduced for checking if a value is an element of a star introduced in Proposition 3.2.2.

The only change in these formulas regards the values in dimension k to which the activation function is applied. Equivalently to the description before, this dimension is defined by the inverse application of the activation function. To this end, the value in the star is compared to a constant that can be derived from the known source and the specific activation function. To calculate this constant, a univariate affine transformation is applied to the relevant dimension of the known source. Then, the value of the new source in dimension k is compared to this constant using an operator $\circ_k \in \{\leq, <, =, >, \geq\}$ which depends on the same information as the constant. In special cases, the restriction to this dimension can also be removed completely.

The specific affine transformation and comparison depend on the segments of the activation function the source is the result of. For a specific activation function operation $\mathrm{op}(\cdot)$ applied to dimension $k \in \{1, \ldots, n\}$, an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, and a source $\boldsymbol{z} \in [\![\Theta']\!]$ of a counterexample for $\Theta' \in \mathrm{op}(\Theta)$, any solution $\boldsymbol{\alpha} \in \mathbb{R}^m$ to formula of the following form

$$\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \varphi_i \tag{3.3}$$

corresponds to a new source $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \in [\![\Theta]\!]$. The subformula $\psi_{\mathcal{P}}(\boldsymbol{\alpha})$ describes the inclusion of the vector $\boldsymbol{\alpha}$ in the predicate of the star. The subformulas φ_i are the constraints on the values z'_i of a new source $\boldsymbol{z}' \in [\![\Theta]\!]$ where each dimension i is defined based on the predicate values $\boldsymbol{\alpha}$. Therefore, these subformulas are of the following form

$$\varphi_i := \left(c_i + \sum_{j=1}^{m} v_{ij} \alpha_j \right) \circ_i (a_i z_i + b_i) \tag{3.4}$$

with $\circ_i$ set to the equality $=$, $a_i = 1$, and $b_i = 0$ for $i \in \{1, \ldots, k-1, k+1, n\}$ and $\circ_k \in \{\leq, <, =, >, \geq\}$ and $a_k, b_k \in \mathbb{Q}$ for the dimension the activation function is applied to.

With this definition, a formula φ_i for a dimension the activation function is not applied to is equivalent to the subformula of the containment check introduced as Formula 3.2. Thus, the conjunction of these formulas for all dimensions describes the equality of $\boldsymbol{z}$ and $\boldsymbol{z}'$ in all dimensions $i \neq k$ and add a single constraint on the value

Case	$\circ$	a	b
$z_k = 0$	$\leq$	0	0
$z_k > 0$	$=$	1	0

(a) ReLU

Case	$\circ$	a	b
$z_k < 0$	$=$	$\frac{1}{\gamma}$	0
$z_k \geq 0$	$=$	1	0

(b) LeakyReLU

Case	$\circ$	a	b
$z_k = R_{min}$	$<$	0	v
$z_k = R_{max}$	$\geq$	0	v

(c) UnitStep

Case	$\circ$	a	b
$z_k = V_{min}$	$\leq$	0	V_{min}
$z_k = V_{max}$	$\geq$	0	V_{max}
$V_{min} < z_k < V_{max}$	$=$	1	0

(d) HardTanh

Case	$\circ$	a	b
$z_k = 0$	$\leq$	0	V_{min}
$z_k = 1$	$\geq$	0	V_{max}
$0 < z_k < 1$	$=$	$V_{max} - V_{min}$	V_{min}

(e) HardSigmoid

Table 3.1: Constants for the tracing formula for all introduced activation functions

z'_k. As mentioned before, the subformula φ_k may also be set to $\top$ or equivalently not be contained at all.

Note, that the Formula 3.3 requires additional checks on the input before construction. These checks could be removed by completely encoding the activation function in the formula. To reduce the complexity of the feasibility check, we instead apply this approach.

Since Formula 3.3 is meant to be feasible if and only if there is a new source of the known source in the parent node, we assume that the known source is the result of an exact activation function application. The set of all potential values in dimension k of a new source is therefore the union of all preimages of the value in dimension k of the known source. Due to the definition of piecewise linear functions over multiple segments, the image of an activation function is the union of the images of the affine mappings for each segment with the segment as domain. Thus, the value in dimension k of a known source is an element of some images of the affine mappings defining the activation function.

If an affine mapping of a segment is a constant function, any value in the segment corresponds to a source. Otherwise, an affine mapping $f : \mathsf{X} \to f(\mathsf{X}), x \mapsto ax + b$ on a segment $\mathsf{X} \subseteq \mathbb{R}$ is invertible with $f^{-1} : f(\mathsf{X}) \to \mathsf{X}, x \mapsto \frac{x}{a} - \frac{b}{a}$. Thus, $f^{-1}(z_k)$ is the only value in dimension k for a new source based on a known source $\boldsymbol{z}$ and the specific segment.

Generally, the preimage of a piecewise linear function for a single value is some subset of the real numbers. For the activation functions introduced in Section 2.1, the preimage of any value is an interval with only one bound, with only one value, or without any bounds. The intervals with a singular bound can thus be described by formulas φ_k with an operator $\circ_k \in \{<, \leq, \geq, >\}$. The operator $=$ can be used to describe the single valued intervals and the interval without bounds corresponds to the aforementioned removal of φ_k. A compilation of the values $a, b \in \mathbb{Q}$ and $\circ \in \{<, \leq, =, \geq, >\}$ for all activation functions introduced in Section 2.1 can be found in Table 3.1. The derivation of these values based on each activation function will be described in the following. To increase the readability, $c_k + \sum_{j=1}^{m} v_{kj}\alpha_j$ is abbreviated with z'_k for the description of the formulas.

The ReLU activation function is defined over the segments $(-\infty, 0)$ and $[0, \infty)$.

The affine mapping applied to elements of $(-\infty, 0)$ is the constant function mapping to 0. In the other interval, the affine mapping is the identity. Thus, the image of the ReLU activation function consists of the images of these segments which are $\{0\} = [0, 0]$ and $[0, \infty)$ respectively. The only overlapping value of these images is 0. If a new source $\boldsymbol{z}$ is the result of a ReLU operation in dimension k and $z_k = 0$, the set of all possible values which result in 0 are $(-\infty, 0)$ from the first segment and $[0, 0]$ from the second. Therefore, a source for $\boldsymbol{z}$ has a value in $(-\infty, 0]$ in the k-th dimension which can be described as $z'_k \leq 0$. Otherwise, $z_k > 0$ which means that z_k is the result of the identity function. Thus, $z'_k = z_k$. These values are represented in Table 3.1a. Note, that any value $a \in \mathbb{R}$ leads to the same result for the case $z_k = 0$ as a is multiplied by z_k.

The LeakyReLU activation function behaves similarly to ReLU and is defined over the same segments. While the affine mapping applied to values in $[0, \infty)$ is the same, the affine mapping applied to values in $(-\infty, 0)$ is $x \mapsto \gamma x$ for $\gamma \in (0, 1)$. Thus, the image for this interval is $(-\infty, 0)$. Since $(-\infty, 0)$ and $[0, \infty)$ partition the real numbers and both affine mappings defining the activation function are invertible, the searched for values are unique. For a source for $\boldsymbol{z}$ which is the result of a LeakyReLU operation in dimension k, the same formula as for ReLU is created to find new source values if $z_k \geq 0$. If $z_k < 0$, the source $\boldsymbol{z}'$ has the value $z'_k = \frac{1}{\gamma} z_k$ in dimension k. The values for the formula φ_k that result from this are represented in Table 3.1b.

The HardTanh activation function is defined over three segments. For two of these, the corresponding affine mapping is the constant function mapping to V_{min} and V_{max} respectively. Thus, the image for the segment $(-\infty, V_{min})$ is $[V_{min}, V_{min}]$ and the image for (V_{max}, ∞) is $[V_{max}, V_{max}]$. The remaining segment $[V_{min}, V_{max}]$ once more corresponds to the identity function, where the image is the same as the segment. The images of either of the segments with constant functions are subsets of the image of the segment with the identity function. Thus, we derive formulas φ_k similarly to the ReLU activation function, if $V_{min} < V_{max}$. If a known source $\boldsymbol{z}$ is the result of a HardTanh operation in dimension k, formula φ_k is constructed for a new source $\boldsymbol{z}'$ as $z'_k \leq V_{min}$ if $z_k = V_{min}$ and as $z'_k \geq V_{max}$ if $z_k = V_{max}$. For the remaining case $V_{min} < z_k < V_{max}$ the formula is once more equivalent to $z'_k = z_k$. If $V_{min} = V_{max}$, the HardTanh activation function becomes the constant function that maps to $V_{min} = V_{max}$. In this case, all values in $(-\infty, \infty)$ are mapped to the same value which means that any value z'_k satisfies the condition to be a source. Thus, the formula φ_k is not included in the overall feasibility problem. The values for the formula in the case of $V_{min} < V_{max}$ are represented in Table 3.1d.

The HardSigmoid activation function is also defined over three segments of which two once more correspond to constant functions. The image of the segment $(-\infty, V_{min}]$ is $[0, 0]$ and the image of the segment $[V_{min}, \infty)$ is $[1, 1]$. The image of the remaining segment (V_{min}, V_{max}) is $(0, 1)$ which corresponds to the affine mapping $x \mapsto \frac{x}{V_{max} - V_{min}} - \frac{V_{min}}{V_{max} - V_{min}}$. Since $[0, 0], (0, 1), [1, 1]$ are pairwise disjoint, finding a new source can be done for each segment individually. Because the segments corresponding to constant functions are defined with closed intervals, the formula φ_k for the cases $z_k = 1$ and $z_k = 0$ are the same as the formulas for the corresponding cases for HardTanh. For the remaining segment, the defining affine mapping needs to be inverted similarly to the negative case of LeakyReLU. The inverse of the affine mapping for this segment is $x \mapsto (V_{max} - V_{min})x + V_{min}$. Thus, the formula φ_k is equivalent to $z'_k = (V_{max} - V_{min}) z_k + V_{min}$ for a known source V_{max} and a new source z'_k. These values are once more collected in Table 3.1e.

Algorithm 9: Basic Tracing Algorithm

input : Reachability tree T,
(n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Operation op,
Known source of a counterexample $\boldsymbol{z}$ contained in a child of Θ,
Corresponding predicate value $\boldsymbol{\alpha}$
output : Counter input $\boldsymbol{z}'$ or operation op$'$

```
   /* Compute the predicate value of the next source      */
 1 if isAffine(op) then
 2 |  α' ← α // Continue in line 12
 3 else // Trace z to Θ
 4 |  k ← getDimension(op)
 5 |  ψ_P ← constructFormula(P)
 6 |  for i ← 1 to n do
 7 |  |  if i = k then
 8 |  |  |  φ_i ← constructFormula(z, op, i, Θ)
 9 |  |  else
10 |  |  |  φ_i ← constructFormulaEq(z, i, Θ)
11 |  α' ← checkFeasibility(ψ_P ∧ ⋀_{i=1}^{n} φ_i)
   /* Return an origin                                    */
12 if α' = () then
   |  // Return an over-approximated operation
13 |  return op
14 else if isRoot (T, Θ) then
   |  // Return a counter input
15 |  return c + Vα'
16 else
   |  // Continue tracing
17 |  Θ' ← getParent(T, Θ)
18 |  op' ← getOperation(T, Θ')
19 |  return basicTracing (T, Θ', op', c + Vα', α')
```

The final activation function UnitStep introduced in Section 2.1 is defined over two segments which both correspond to constant functions mapping to R_{min} and R_{max} respectively. Thus, the image of segment $(-\infty, v)$ is $[R_{min}, R_{min}]$ and the image of segment $[v, \infty)$ is $[R_{min}, R_{min}]$ similar to the constant function before. If $R_{min} \neq R_{max}$, these images are disjoint. Therefore, the formula φ_k for a source $z_k = R_{min}$ corresponds to $z'_k < v$ and for $z_k = R_{max}$ to $z'_k \geq v$. Since there are only two segments in the definition of this activation function, no further cases need to be considered if the initial condition holds. These two cases are shown in Table 3.1c. If $R_{min} = R_{max}$ on the other hand, this activation function behaves the same way the HardTanh activation function behaved in the case of $V_{min} = V_{max}$. Thus, the formula φ_k is not included for feasibility checking since all values map to $R_{min} = R_{max}$.

These formulas are the template for the construction of the feasibility problem that allows tracing in Algorithm 9 which is referenced as `basicTracing` in this algorithm. For this algorithm a (partial) reachability tree $T = (\mathsf{V}, \mathsf{E})$ with a star $\Theta \in \mathsf{V}$ is assumed.

Additionally, a source for a counterexample $\boldsymbol{z}$ and the corresponding predicate value $\boldsymbol{\alpha}$ are parameters. It is assumed, that the counterexample is an element of a child of Θ and that this child was computed using the operation $\mathrm{op}(\cdot)$ which is the final part of the input.

Based on this input, the algorithm computes an origin. Therefore, the algorithm either returns an element of the root or an over-approximated activation function operation as this origin. This return can also be based on a recursive call. In this call, the algorithm is applied to the parent node of the input star. Therefore, one application of this algorithm either terminates or reduces the distance to the root by one. Since all reachability trees have a root, this algorithm terminates once the original input has been traced to the root at the latest.

The algorithm first differentiates between affine and activation function operations with the if-statement. For this the method `isAffine` results in true, when applied to an affine operation, and false otherwise. Tracing of a source through an affine operation is then simply handled via the predicate value. The activation function operation is handled in lines 4 to 11. Since, we aim to trace through this operation, we need to know the dimension it is applied to. The function `getDimension` returns this dimension. For the tracing process, Formula 3.3 is now constructed using methods `constructFormula` and `constructFormulaEq`. When the first one is applied to a polytope, a corresponding formula is produced. This application behaves the same way it does in Algorithm 7. A method with the same name is also used with a source, an operation, a dimension, and a star as input. In this case, the resulting formula corresponds to Formula 3.4 where the variables are replaced by constants corresponding to the specifics of the input as explained before. Similarly, the method `constructFormulaEq` returns a formula for the equality of the values of stars in one dimension. Thus, this algorithm constructs all subformulas of Formula 3.3 during the loop starting in line 6. Finally, the instance of this formula is checked for feasibility in line 11 with the method `checkFeasibility` that behaves as before. Note, that feasibility corresponds to the return of the empty tuple and infeasibility to the return of a satisfying assignment.

Lines 12 to 19 describe the different ways this algorithm continues based on a new predicate value $\boldsymbol{\alpha}'$. If this value is the empty tuple, the feasibility problem is infeasible. Thus, the algorithm returns the identified origin operation in line 13. Otherwise, a new predicate value exists which either corresponds to a counter input or the new source for further tracing. A counter input is found if the input star is the root of the reachability tree which is checked with the method `isRoot`. If neither the current operation nor the current star are an origin, the tracing process continues. To this end, the algorithm obtains the parent of the input star using the method `getParent` and the operation by which the input star was computed from the parent using the method `getOperation`.

3.3.2 Binary Search Based Tracing

The basic tracing method introduced in the previous section ensures the identification of an origin of a counterexample. The disadvantage of this method is the high number of feasibility problems that need to be solved to trace the counterexample through sequences of activation function operations. This high number is one of the main causes of low efficiency. To reduce this amount, we introduce a method that allows the tracing of a counterexample through such a sequence by solving only a singular

feasibility problem.

This feasibility problem for tracing through sequences of activation function operations is a modification of Formula 3.3 introduced for basic tracing. The previous method traces a known source through an activation function application to one dimension k based on a single subformula φ_k not representing equality. This new method aims to trace a known source through the application of activation functions to a sequence of dimensions. For all these dimensions k, the subformulas φ_k is constructed based on the activation function and known source values as introduce in the previous section. Thus, the formulas φ_k for all of these dimensions are derived based on Table 3.1 and the remaining formulas φ_i describe the equality as before. Note, that the special cases where a formula φ_k is set to $\top$ can still occur for any of the dimensions.

The structure of reachability trees for reachability analysis is based on FNNs. Therefore, the operations in the tree are singular affine operation and sequences of activation function operations corresponding to the layers of the FNN. Based on the new formula, a counterexample or known source can be traced to a new source through the operations corresponding to a layer in the FNN with only a single feasibility check. Tracing through the complete sequence of activation function operations is thus possible by constructing formulas φ_k for all dimensions of known source. For the affine operation, this method relies on the same process that was used for the basic tracing method. A known source is only traced through the affine operation, if the initial tracing through the sequence of activation function operation is successful. Otherwise, an origin for the counterexample can be found in the sequence of activation function operations.

The searching process for such an origin gives this method its name. We introduce a method based on binary search that finds the origin in such a sequence using the new tracing method. For this, the initial tracing attempt through the complete sequence provides a bound at the end of the sequence in the form of a reachable set that does not include any new sources for the known source. The other bound is given by the reachable set with the known source at the beginning of the sequence. This method then repeatedly attempts to find a source in the node that is in the middle of both bounds while updating the bounds depending on success and failure. These updates reduce the search space for the origin by halve with each tracing attempt. The repetition ends once the bounds correspond to parent and child node in the reachability tree. Such bounds identify the operation in between to be an origin of the counterexample.

This faster tracing process is defined in Algorithm 10. This algorithm is referred to as `binaryTracing` for recursive calls. Since this algorithm is meant to replace Algorithm 9, the input and output behave the same way. Additionally, both algorithms contain similar parts, including the functions used. All methods that are used in both algorithms behave the same way as before.

Once again, the first step is the distinction between an affine operation and activation function operation. The first case of the if-statement in lines 2 to 7 corresponds to the handling of affine operations. Despite the difference in format it actually behaves the same way Algorithm 9 dealt with affine operations. If the input star is the root, the algorithm returns a counter input. Otherwise, the tracing process is continued in the parent node.

The else-case of the if-statement beginning in line 9 corresponds to the tracing through an activation function sequence or the identification of an origin in the

Algorithm 10: Binary Search Based Tracing Algorithm

input : Reachability tree T,
(n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Operation op,
Known source of a counterexample $\boldsymbol{z}$ contained in a child of Θ,
Corresponding predicate value $\boldsymbol{\alpha}$

output : Counter input $\boldsymbol{z}'$ or Operation op′

if `isAffine`(op) **then**
  if `isRoot`(T, Θ) **then** // Return counter input
    return $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$ // $\boldsymbol{\alpha} = \boldsymbol{\alpha}'$
  else // Continue tracing
    $\Theta' \leftarrow$ `getParent`(T, Θ)
    $\mathrm{op}' \leftarrow$ `getOperation`(T, Θ')
    return `binaryTracing`$(T, \Theta', \mathrm{op}', \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}, \boldsymbol{\alpha})$
else
  /* Attempt to trace through the whole sequence */
  $\mathrm{op}_n, \ldots, \mathrm{op}_1 \leftarrow$ `getOperationSequence`(T, Θ)
  $\Theta_n, \ldots, \Theta_0 \leftarrow$ `getStarSequence`(T, Θ) // Let $\Theta_i = \langle \boldsymbol{c}_i, \boldsymbol{V}_i, \mathcal{P}_i \rangle$
  for $i \leftarrow 1$ **to** n **do** // Trace $\boldsymbol{z} \in [\![\Theta_n]\!]$ to Θ_0
    $\varphi_i \leftarrow$ `constructFormula`$(\boldsymbol{z}, \mathrm{op}_i, i, \Theta_0)$
  $\psi_{\mathcal{P}_0} \leftarrow$ `constructFormula`$(\mathcal{P}_0)$
  $\boldsymbol{\alpha}' \leftarrow$ `checkFeasibility`$(\psi_{\mathcal{P}_0} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
  if $\boldsymbol{\alpha}' = ()$ **then**
    /* An origin is in the sequence */
    $\nu \leftarrow n, \quad \iota \leftarrow \lfloor \frac{n}{2} \rfloor, \quad \kappa \leftarrow 0$
    $\boldsymbol{z}' \leftarrow \boldsymbol{z}$
    while $\nu \neq \kappa + 1$ **do**
      for $i \leftarrow 1$ **to** n **do** // Trace $\boldsymbol{z} \in [\![\Theta_\nu]\!]$ to Θ_ι
        if $\iota < i \leq \nu$ **then**
          $\varphi_i \leftarrow$ `constructFormula`$(\boldsymbol{z}', \mathrm{op}_i, i, \Theta_\iota)$
        else
          $\varphi_i \leftarrow$ `constructFormulaEq`$(\boldsymbol{z}', i, \Theta_\iota)$
      $\psi_{\mathcal{P}_\iota} \leftarrow$ `constructFormula`$(\mathcal{P}_\iota)$
      $\boldsymbol{\alpha}'' \leftarrow$ `checkFeasibility`$(\psi_{\mathcal{P}_\iota} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
      if $\boldsymbol{\alpha}'' = ()$ **then** // An origin is an operation after op_ι
        $\kappa \leftarrow \iota, \quad \iota \leftarrow \lfloor \frac{\nu+\kappa}{2} \rfloor$
      else // An origin is an operation before op_ι
        $\nu \leftarrow \iota, \quad \iota \leftarrow \lfloor \frac{\nu+\kappa}{2} \rfloor$
        $\boldsymbol{\alpha}' \leftarrow \boldsymbol{\alpha}'', \quad \boldsymbol{z}' \leftarrow \boldsymbol{c}_\nu + \boldsymbol{V}_\nu \boldsymbol{\alpha}'$
    return op_ν // Origin identified
  else // $(\boldsymbol{c}_0 + \boldsymbol{V}_0 \boldsymbol{\alpha}') \in [\![\Theta_0]\!]$
    /* Continue tracing */
    $\Theta' \leftarrow$ `getParent`(T, Θ_0)
    $\mathrm{op}' \leftarrow$ `getOperation`(T, Θ')
    return `binaryTracing` $(T, \Theta', \mathrm{op}', \boldsymbol{c}_0 + \boldsymbol{V}_0 \boldsymbol{\alpha}', \boldsymbol{\alpha}')$

sequence. To this end, the algorithm starts by identifying the sequence. The method `getOperationSequence` returns the sequence of activation function operation $\mathrm{op}_n, \ldots, \mathrm{op}_1$. The index of these operations corresponds to the dimension the operation is applied to. Similarly, the method `getStarSequence` returns the sequence of stars $\Theta_n, \ldots, \Theta_0$ where Θ_i is the result of the application of op_i to Θ_{i-1}. Θ_0 is the first node in the reachability tree that does not result from this sequence of activation function applications to all dimension of the input star. Note, that op_n and Θ_{n-1} correspond to the input and that the source that is part of the input is an element of Θ_n.

The next step of tracing a source to its origin is the attempt to trace through the whole sequence. Thus, Formula 3.3 is constructed similarly to Algorithm 9, but with Formula 3.4 adapted to tracing for all dimension as describe before. If this tracing attempt succeeds, a new source is identified in Θ_0. The algorithm thus calls itself in line 35 on the parent Θ_0 and the corresponding operation to continue the tracing process. Checking, if Θ_0 is the root is not necessary due to the structure of FNNs.

If the attempt to trace through the sequence fails, an origin of the counterexample can be found in the sequence. To identify this origin, a binary search based approach is used. For this, the variable ν indicates the star Θ_ν with the smallest $\nu \in \{\kappa + 1, \ldots, n\}$ that contains a known source. The variable $\kappa \in \{0, \ldots, \nu - 1\}$ indicates the opposite. Θ_κ is the star with the largest κ that does not contain any source.

The last variable ι points to the star Θ_ι to which tracing is attempted next. Since Θ_ι is the result of the activation function application in dimension ι and a source $\boldsymbol{z} \in \Theta_\iota$ is searched for, this source is traced through the operations and corresponding dimensions starting with $\iota + 1$ and stopping at ν. This tracing attempt is represented in lines 19 to 25 and works similar to Algorithm 9. The difference is that Formula 3.3 is used for tracing for all dimensions $i \in \{\iota + 1, \ldots, \nu\}$. The equality version of the formula is still used in the remaining dimensions.

Dependent on the success or failure of this tracing attempt the variables ν, ι, and κ are adjusted in lines 26 to 30. If Θ_ι contains a new source, a star containing a source of an initial counterexample is identified with $\iota \leq \nu$. Therefore, the star ν points to is changed to Θ_ι. In addition, the known source is updated with the result of the successful feasibility check. If Θ_ι does not contain any new source, κ is updated instead. In both case the star ι refers to is updated to be in the middle between ν and κ again.

These updates reduce the distance between ν and κ with each attempted tracing. Once ν and κ point to a parent and child node, an origin of the counterexample is identified. With the end of the tracing loop in line 31, the operation between this parent and child is thus the origin.

3.3.3 Tracing with Remembered Origins

The remembering tracing method extends the binary search based tracing method by making use of previous results. During the refinement process typically multiple counterexamples are identified and traced to an origin. It is common that the same operation is identified as an origin during more than one of these iterations. For the purpose of this method, two operation are considered the same if they correspond to the same step in the calculation of an FNN. The primary reason for reoccurring origins is the refinement process which is explained in Section 3.4 in more detail. During this process an origin that is replaced by an exact activation function operation can

be replaced by an over-approximated operation again during a later cycle. This can reintroduce already removed counterexamples.

The previous origins are then used to increase the performance of tracing by checking if they are origins of a new counterexample once more. This check is performed once a sequence of activation functions is identified that contains an origin. This heuristic is thus inserted between lines 16 and 17 in Algorithm 10 after the unsuccessful attempt to trace a new source through the sequence of activation functions. To be able to check these previous origins they need to be tracked. This happens during the tracing and refinement algorithms but is not explicitly included in the algorithms introduced in this thesis. For the purpose of the application of this method, it is assumed that a list of the indices of the previous origins in each activation function sequence is kept track of. Based on this list, the previous origins can be checked iteratively. If any of the previous origins are an origin again, this origin is the result of the tracing process. Otherwise, the binary search based tracing method is applied using the information gained from the attempted tracings.

The identification of an origin requires one successful and one unsuccessful tracing attempt. Thus, checking each previous origin would require double the number of previous origins in feasibility checks. This number can be reduced by sorting the list in ascending order and tracing each previous origin at most once. The failure or success of these attempts can then be used to reduce the number of feasibility checks similar to the bounds ν and κ used in binary tracing. If an attempt succeeds the node to which the corresponding previous origin is applied contains a source. Thus, an origin of the counterexample exists between this and the last previous origin to which tracing failed. If no such previous origin exists, the method defaults to binary search between the successfully traced previous origin and the end of the sequence. After failed tracing attempts to previous origins, this algorithm simply continues with the next element of the list.

Algorithm 11 describes this process. This algorithm is meant to initialize the variables in lines 16 and 17 of Algorithm 10 with its output. The names of output and variables correspond to each other. The algorithm takes the reachability tree, the sequences of $n+1$ stars and n operations generated in lines 9 and 10 of Algorithm 10, a source of a counterexample, and the aforementioned ordered list $L = [l_0, \ldots, l_\lambda]$ of previous origins with $1 \leq l_0 < \cdots < l_\lambda \leq n$ in the form of indices as input.

The algorithm then initialize the variables ν, ι, κ and $\boldsymbol{z}'$ for the binary search based tracing method. This ensures correct behavior, if no previous origins are known. Afterwards, the algorithm iterates through these origins with the loop beginning in line 3. In lines 4 to 11 the counterexample is traced to the reachable set to which the previous origin operation is applied to. If the current index in the list is not the last and the tracing attempt fails, the loop continues with the next iteration after updating κ in the same way as in Algorithm 10. For the last element of the list, the variable ι for the next tracing attempt is also updated to l_λ. This ensures the binary tracing method checks if the previous origin that is the last element of the list is an origin again before continuing with the typical method.

If it is possible to find a new source, then the previous origin cannot be an origin again. In lines 17 to 23 the starting position is updated to the node containing the new source. Only the previous origin checked before the current iteration of the loop can be the origin once a tracing attempt is successful. It is necessary to trace a source contained in the star corresponding to the first element of the list through the stars of all other elements in the list. Therefore, no previous origin is an origin again, if

Algorithm 11: Tracing Algorithm with Previous Origins

input : Reachability tree T,
Sequence of stars $\Theta_n, \ldots, \Theta_0$,
Sequence of activation function operation $\mathrm{op}_n, \ldots, \mathrm{op}_1$,
Known source of a counterexample $\boldsymbol{z} \in [\![\Theta_n]\!]$,
Ascending list of previous origins $L = [l_0, \ldots, l_\lambda]$,
output : Values $\nu, \iota, \kappa \in \mathbb{N}$ for binary tracing
And a new source $\boldsymbol{z}'$

```
1  ν ← n,   ι ← ⌊n/2⌋,   κ ← 0
2  z' ← z
3  for j ← 0 to λ do
4  |  ι ← l_j − 1
5  |  for i ← 0 to n do // Trace z ∈ [[Θ_ν]] to Θ_ι
6  |  |  if ι < i ≤ ν then
7  |  |  |  φ_i ← constructFormula(z', op, i, Θ_ι)
8  |  |  else
9  |  |  |  φ_i ← constructFormulaEq(z', i, Θ_ι)
10 |  ψ_{P_ι} ← constructFormula(P_ι)
11 |  α' ← checkFeasibility(ψ_{P_ι} ∧ ⋀_{i=1}^{n} φ_i)
12 |  if α' = () then
13 |  |  κ ← ι
14 |  |  if j = λ then // l_j is the last element
15 |  |  |  ι ← l_j  // Check if op_{l_j} is the origin
16 |  else
17 |  |  ν ← ι
18 |  |  z' ← c_ν + V_ν α'
19 |  |  if j > 0 then
20 |  |  |  ι ← l_{j−1}  // Find an origin between Θ_{l_j} and Θ_{l_{j−1}}
21 |  |  else
22 |  |  |  ι ← ⌊(ν+κ)/2⌋
23 |  |  break // Continue with line 24
24 return ν, ι, κ, z'  // Use binary search based tracing
```

tracing to this first element is successful. The variable ι is set according to the process in binary search based tracing in this case. Otherwise, if the current element of the list is not the first, an origin can be found between this node and the last node that was checked. This also includes the previous origin that was checked before. Thus, ι is set such that the binary tracing method checks if this previous origin is an origin again, before continuing with the standard algorithm.

3.3.4 Core based Tracing

In this section, an alternative method to binary search based tracing introduced in Section 3.3.2 is presented. This method also aims to reduce the number of feasibility checks required to identify an origin in a sequence of activation function operation. Instead of halving the search space in each step, this method uses information gained from unsuccessful tracing attempts. This is facilitated by the core that the *Z3* solver

provides for unsatisfiable formulas. Such a core represents a subformula that is already unsatisfiable.

For the purpose of this method, we are specifically interested in which of the formulas φ_i corresponding to Formula 3.4 are present in the core. These formulas constrain the values in dimension i of new sources in a sequence of activation function operations. For this heuristic we assume, that a formula φ_i that is present in the core indicates that the corresponding operation is more likely to be an origin.

The idea of this method, is to trace iteratively to a dimension i with a formula φ_i in the core. We use the largest dimension for which a corresponding formula is part of the core. In addition, only dimensions decreasing the search space for an origin are considered. This search space is once again bounded by the star that contains the known source and by a star that contains no source.

A reachable set of an FNN for a nonempty input set is never empty. Thus, any star corresponding to a node in the reachability tree is also nonempty. Based on Proposition 2.2.1, the predicates $\mathcal{P}$ of such stars are nonempty which means that $\psi_{\mathcal{P}}$ the formula corresponding to $\mathcal{P}$ is always satisfiable. Thus, at least one formula φ_i is present in any core. This does however not prove that this formula would reduce the search space. In the case that no formula φ_i that satisfies the aforementioned conditions is part of the core, we simply trace to $\kappa + 1$. This ensures that the search space is reduced with every tracing attempt.

Besides the difference in deciding the next star in the sequence, the method generally works similar to the binary search based tracing method. After identifying a sequence of activation function operations that contains an origin, the method keeps track of the star Θ_ν containing the current source and the star Θ_κ not containing any source. The identification of the sequence already requires a feasibility check. This check provides the first core and thus the first goal for tracing as explained before.

If a tracing attempt succeeds, the value of ν and thus the star corresponding to ν as well as the source and corresponding predicate value are updated. The next tracing goal is set to Θ_κ. For unsuccessful tracing attempts, the core is used to derive the next goal. In addition, Θ_ν is adjusted to the current goal. This process is repeated until an origin is identified. Since the search space is reduced in each iteration, this condition is always fulfilled at some point.

This process is described in more detail in Algorithm 12. As mentioned before, this algorithm behaves very similarly to Algorithm 9. Since it is a tracing algorithm, it has the same input and output as the previous tracing algorithms. In addition, only lines 15 to 35 differ from the binary tracing method because only the approach of tracing in a sequence of activation function operations that contains an origin changes for this method. This also means, that the function calls in the method are mostly the same as before. The exception to this, are the recursive call with the name `coreTracing` and the new method `getCore`. This method returns a set of all indices of formulas φ_i that are part of the unsatisfiable core of the input formula. Such an input formula is always a formula similar to Formula 3.3.

As mentioned before this core is used to define the next target for a tracing attempt. The initial target is set after a failed attempt to trace a source through the whole sequence of activation functions. The core corresponding to this attempt is then used to find the next target with the method described before.

Lines 20 to 26 correspond to this and all other tracing attempts in the sequence. If such an attempt succeeds, a new predicate value corresponding to a new source is returned by the feasibility check. In this case, the starting point of tracing attempts is

Algorithm 12: Core Based Tracing Algorithm

input : Reachability tree T,
(n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Operation op,
Known source of a counterexample $\boldsymbol{z}$ contained in a child of Θ,
Corresponding predicate value $\boldsymbol{\alpha}$

output : Counter input $\boldsymbol{z}'$ or Operation op$'$

if `isAffine`(op) **then**
  if `isRoot`(T, Θ) **then** `// Return counter input`
    return $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$ `//` $\boldsymbol{\alpha} = \boldsymbol{\alpha}'$
  else `// Continue tracing`
    $\Theta' \leftarrow$ `getParent`(T, Θ)
    op$' \leftarrow$ `getOperation`(T, Θ')
    return `coreTracing`$(T, \Theta', \text{op}', \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}, \boldsymbol{\alpha})$
else
  $\text{op}_n, \ldots, \text{op}_1 \leftarrow$ `getOperationSequence`(T, Θ)
  $\Theta_n, \ldots, \Theta_0 \leftarrow$ `getStarSequence`(T, Θ) `// Let` $\Theta_i = \langle \boldsymbol{c}_i, \boldsymbol{V}_i, \mathcal{P}_i \rangle$
  for $i \leftarrow 1$ **to** n **do** `// Trace` $\boldsymbol{z} \in [\![\Theta_n]\!]$ `to` Θ_0
    $\varphi_i \leftarrow$ `constructFormula`$(\boldsymbol{z}, \text{op}_i, i, \Theta_0)$
  $\psi_{\mathcal{P}_0} \leftarrow$ `constructFormula`$(\mathcal{P}_0)$
  $\boldsymbol{\alpha}' \leftarrow$ `checkFeasibility`$(\psi_{\mathcal{P}_0} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
  if $\boldsymbol{\alpha}' = ()$ **then** `// An origin is in the sequence`
    core $\leftarrow$ `getCore`$(\psi_{\mathcal{P}_0} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
    $\nu \leftarrow n, \quad \iota \leftarrow \max((\text{core} \cap (\kappa, \nu)) \cup \{\kappa + 1\}), \quad \kappa \leftarrow 0$
    $\boldsymbol{z}' \leftarrow \boldsymbol{z}$
    while $\top$ **do**
      for $i \leftarrow 1$ **to** n **do** `// Trace` $\boldsymbol{z} \in [\![\Theta_\nu]\!]$ `to` Θ_ι
        if $\iota < i \leq \nu$ **then**
          $\varphi_i \leftarrow$ `constructFormula`$(\boldsymbol{z}', \text{op}_i, i, \Theta_\iota)$
        else
          $\varphi_i \leftarrow$ `constructFormulaEq`$(\boldsymbol{z}', i, \Theta_\iota)$
      $\psi_{\mathcal{P}_\iota} \leftarrow$ `constructFormula`$(\mathcal{P}_\iota)$
      $\boldsymbol{\alpha}'' \leftarrow$ `checkFeasibility`$(\psi_{\mathcal{P}_\iota} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
      if $\boldsymbol{\alpha}'' \neq ()$ **then** `// New source is identified`
        $\nu \leftarrow \iota, \quad \iota \leftarrow \kappa$
        $\boldsymbol{\alpha}' \leftarrow \boldsymbol{\alpha}'', \quad \boldsymbol{z}' \leftarrow \boldsymbol{c}_\nu + \boldsymbol{V}_\nu \boldsymbol{\alpha}'$
      else if $\nu = \iota + 1$ **then**`// Origin identified`
        return op_ι
      else `// No new source exists in` Θ_ι
        $\kappa \leftarrow \iota$
        core $\leftarrow$ `getCore`$(\psi_{\mathcal{P}} \wedge \bigwedge_{i=1}^{n} \varphi_i)$
        $\iota \leftarrow \max((\text{core} \cap (\iota, \nu)) \cup \{\kappa + 1\})$
  else `//` $(\boldsymbol{c}_0 + \boldsymbol{V}_0 \boldsymbol{\alpha}') \in [\![\Theta_0]\!]$
    $\Theta' \leftarrow$ `getParent`(T, Θ_0)
    op$' \leftarrow$ `getOperation`(T, Θ')
    return `coreTracing` $(T, \Theta', \text{op}', \boldsymbol{c}_0 + \boldsymbol{V}_0 \boldsymbol{\alpha}', \boldsymbol{\alpha}')$

updated to the current goal. In addition, the next tracing attempt will be performed with the goal of the star closest to the start of the sequence to which tracing is not possible. This attempt always fails, but provides a new goal based on the core.

Such a core is used in lines 32 to 35 to set this goal in the same manner as before. Since a core only exists, if the formula is infeasible, this computation is only performed for failed tracing attempts. Besides setting new tracing goals, an origin can only be returned after a failed tracing attempt from a child to a parent node.

3.4 Refinement Methods

In this section, our approaches to the verification of FNN using CEGAR are introduced. These combine the reachability tree data structure with the reachability analysis from Section 2.3 and the safety checking and tracing algorithms presented in the previous sections of this chapter. When a non-specific method from these sections is referenced, generally any of the introduce methods can be used.

This section will start with two general verification algorithms. Both of them, iteratively construct and refine reachability trees until either safety or unsafety of the FNN can be proven. Construction refers to the use of the reachability methods introduced in Section 2.3 to generate the reachable sets that the reachability tree is composed of. The actual refinement process begins once final leaves are computed and starts with the safety checking of these leaves. If an unsafe final leaf is found, the resulting counterexample and predicate value are used to trace this counterexample to an origin. This origin then either allows refinement or proves the unsafety of the FNN. In the second case, this origin is a counter input. Otherwise, the reachability tree and the corresponding reachability analysis are refined by exchanging the over-approximated operation that is the origin with the corresponding exact operation. This replacement of an operation removes the source of the spurious counterexample that is used to identify the origin as specified in the following proposition.

Proposition 3.4.1 (Refinement)**.** *Let $\Theta_o \in op_o(\Theta)$ for two stars Θ_o and Θ in a reachability tree and op_o an over-approximated activation function operation.*

If $\boldsymbol{z} \in [\![\Theta_o]\!]$ is a source of a counterexample such that op_o is an origin of this counterexample, then $\boldsymbol{z} \notin \bigcup_{i=1}^{k} [\![\Theta_i]\!]$ for $op_e(\Theta) = \{\Theta_1, \ldots, \Theta_k\}$ where op_e is the exact activation function operation corresponding to op_o.

The replacement of the origin with an exact operation then requires recomputation of the reachability subtree. This recomputation is the first step for the next iteration of the refinement algorithms.

The way reachability trees are constructed is the primary difference between refinement algorithms. The first algorithm introduced in Section 3.4.1 always computes full reachability trees before continuing with the next steps of refinement. The second algorithm which is presented in Section 3.4.2 instead computes partial reachability trees that have one more final leaf in each iteration and then performs the subsequent steps on this leaf. After a deeper introduction to these algorithms, multiple heuristics are presented in Sections 3.4.3 to 3.4.5. These heuristics add to the basic algorithms by storing different information during each refinement loop and then modify the construction and verification of subsequent reachability trees.

3.4.1 Full Refinement

As mentioned before, this refinement method always constructs a full reachability tree as a first step. During this construction the activation function operations of the FNN are applied in an over-approximated manner. Once all leaves in the reachability tree are final, the safety of the leaves is checked iteratively. For safe final leaves no further computation is necessary. If a final leaf is unsafe on the other hand, the identified counterexample is traced to its origin. If the origin is a counter input, the FNN is unsafe and the algorithm terminates. Otherwise, the origin is an over-approximated activation function operation. This operation is then replaced by its exact counterpart. For this, the subtree with the star resulting from the application of the operation as its root is removed. Afterwards, the exact activation function operation is applied instead of the over-approximated operation. All resulting stars are then added to the reachability tree accordingly. This exchange of exact and over-approximated activation function operation is the core of the refinement process.

The described process up to this point represents one cycle in the refinement loop. The next cycle of the loop and thus the next steps are the completion of the reachability analysis to construct a full reachability tree once again. This, starts with the identification of all non-final leaves. Previously, this step was implied in the construction of the initial fully over-approximated reachability tree. The actual initialization of the reachability tree for this algorithm creates a new partial reachability tree that only consists of the input set as the root. This root is also a non-final leaf. From the partial reachability tree, a full tree is computed by applying the corresponding operations of the FNN to the reachable sets represented by non-final leaves until all leaves are final. The correct operation can be identified by the length of the path from a leaf to the root. This length corresponds to the number of the operation.

This algorithm identifies the unsafety of an FNN once a single counter input is calculated. If all final leaves of the reachable set are safe, the union of them represents an over-approximation of the reachable set of the FNN. Because none of the reachable sets in this over-approximation is unsafe, the union is safe. Therefore, the FNN is also safe. The algorithm thus determines the safety of an FNN once all final leaves are safe.

This refinement method is described by Algorithm 13. For the verification of an FNN, this algorithm performance reachability analysis and safety checks. Therefore, the input for the algorithm consists of an FNN and an input set in the form of a star for the reachability analysis and a safety specification to check. Note, that a bounded convex polytope could be used as an input set in the same way it could be used for reachability analysis. Such a polytope would be transformed into a star with an application of Proposition 2.2.2. Based on this input, this algorithm determines whether the FNN is safe regarding the input star and the safety specification and returns $\top$ for safety and a counter input to indicate unsafety.

After initializing a reachability tree T with input star Θ and a boolean variable to false. The refinement loop begins in line 3 and only stops with the termination of the algorithm. This can either happen, if a counter input is identified in line 17, or if all final leaves are safe.

In lines 4 to 7, a full reachability tree is constructed form T according to F. To identify at which point T is a full reachability tree, the method `isFull` is used. If T is not full, a non-final leave exists in T. One of these is returned by the method `getNonFinalLeaf`. Note, that checking if a leave is final or if a reachability tree is full generally requires access to the FNN corresponding to the reachability tree. Once

Algorithm 13: Full Refinement Algorithm

input : FNN F,
Input set as a star $\mathcal{I} = \Theta$,
Safety specification $\mathcal{S}$
output : Safe ($\top$) or unsafe with a counter input o

```
1  T = (V, E) ← ({Θ}, ∅)
2  isSafe ← ⊥
3  while isSafe = ⊥ do
       /* Construction of a full reachability tree                    */
4      while not isFull (T, F) do
5          Θ′ ← getNonFinalLeaf(T, F)
6          op ← getOperation(T, F, Θ′) // affine or approximate
7          T = (V, E) ← (V ∪ op(Θ′), E ∪ {(Θ′, Θ″) | Θ″ ∈ op(Θ′)})
       /* Refinement of the reachability tree                          */
8      isSafe ← ⊤
9      leaves ← getFinalLeaves(T, F)
10     for Θ′ ∈ leaves do
11         (z, α) ← checkSafety(Θ′, S)
12         if z ≠ () then // z is a counterexample
13             Θ″ ← getParent(T, Θ′)
14             op ← getOperation(T, Θ″)
15             o ← trace(T, Θ″, op, z, α)
16             if o ∈ ℝ^⟨1⟩ then // The origin is a counter input
17                 return o
               /* Refine by replacing the origin                      */
18             Θ″ ← getNode(T, Θ′, o)
19             V′ ← V \ {Θ ∈ V | Θ is a descendant of Θ″}
20             E′ ← E ∩ V′ × V′
21             op_e ← getExact(o)
22             T = (V, E) ← (V′ ∪ op_e(Θ″), E′ ∪ {(Θ″, Θ‴) | Θ‴ ∈ op_e(Θ″)})
23             isSafe ← ⊥
24             break // Continue with the next loop after line 3
25 return ⊤
```

a non-final leave is chosen, the function `getOperation` is used to get the operation that should be applied to the leaf. This operation is either an exact affine mapping or an over-approximated activation function application.

Once a full reachability tree is constructed the safety of the final leaves is determined. The final leaves that have not been checked for safety are returned by the method `getFinalLeaves`. For this the safety of each leaf is check using any of the methods introduced in Section 3.2. These methods are represented by the function named `checkSafety`. The call to this function is represented with the basic input. If further input is required for the safety check this would be added to this call. The algorithm returns $((), ())$, if a leaf is safe and a pair of a counterexample and a corresponding predicate value otherwise. If all leaves are safe, the refinement loop ends and the algorithm returns $\top$ and thus the safety of F.

Otherwise, the counterexample is traced to an origin. For this, the input for a tracing algorithm is prepared. These algorithms require the initial goal for tracing in the form of a star as well as the operation that is applied to calculate the star containing the counterexample. The star and operation are derived using the methods `getParent` and `getOperation` which behave the same way as in the algorithm in Section 3.3. `trace` refers to any of the tracing methods introduced the aforementioned section. The same note regarding different inputs that applied to safety checking methods also applies to the different tracing methods.

The output of tracing is either a counter input or an over-approximated activation function operation. In the first case, the algorithm returns the unsafety of F in line 17. If an over-approximated activation function operation is identified, the reachability tree is refined. To this end, the node Θ'' in T is identified to which the operation is applied using the method `getNode`. Then, in lines 19 and 20 the subtree starting with the child of Θ'' is removed from T. Note, that Θ'' has exactly one child due to the over-approximated manner in which the operation is applied.

With the application of the method `getExact` the exact operation corresponding to the origin is obtained. This exact operation is then applied to Θ''. All results are added as leaves to T. Afterwards the refinement cycle ends, and the algorithm continues with the next loop. Note, that the leaves added to T may be final or non-final leaves depending on the operation that is replaced. This does not introduce problems as long as `getFinalLeaves` is defined correctly and not solely based on a leave being final or non-final.

3.4.2 Avoidant Refinement

This refinement method works similarly to the method introduced in the previous section. Like before, this method takes non-final leaves and applies over-approximated reachability analysis until final leaves are computed. Different from before, only a single non-final leaf is chosen to compute a path to a single final leaf. For this non-final leaf, the reachability analysis is applied like before to compute a reachability tree in the form of a path ending in a final leaf.

Instead of applying reachability analysis to any other non-final leaves, the next steps in the refinement loop are applied to the computed final leaf. Thus, the next step is the verification of safety again. If the reachable set in the final node is safe, the next cycle begins. Otherwise, the counterexample is traced to its origin which is treated as before. An over-approximated operation is replaced with the corresponding exact operation, or the algorithm terminates with the unsafety of the FNN.

The exception to this procedure occurs if the identified origin is the final operation of an FNN. This means that the leaves computed by the exact operation are final. In this case, all of these leaves need to be checked for safety before continuing with the next refinement cycle. Otherwise, the safety of these leaves would remain unchecked as the reachability trees are constructed based on only non-final leaves. If any of these final leaves are unsafe, the typical process of refinement is applied to the first unsafe one. Thus, the counterexample is traced to its origin which is either refined or causes the termination of the algorithm. Note, that if one unsafe leaf is found and refined, the other leaf is discarded in this process because the last operation is already exact and cannot be an origin in this step. Therefore, these leaves are only checked until the first unsafe one is identified.

The previous algorithm terminates once a counter input is found or all final leaves

Algorithm 14: Avoidant Refinement Algorithm

input : FNN F,
Input set as a star $\mathcal{I} = \Theta$,
Safety specification $\mathcal{S}$
output : Safe ($\top$) or unsafe with a counter input o

```
1  T = (V, E) ← ({Θ}, ∅)
2  while not isFull(T, F) do
       /* Compute a new final leaf                                  */
3      Θ′ ← getNonFinalLeaf(T, F)
4      while not isFinal(T, F, Θ′) do
5          op ← getOperation(T, F, Θ′) // affine or approximate
6          {Θ″} ← op(Θ′)
7          T = (V, E) ← (V ∪ {Θ″}, E ∪ {(Θ′, Θ″)})
8          Θ′ ← Θ″
       /* Refinement of the reachability tree                       */
9      (z, α) ← checkSafety(Θ′, S)
10     if z ≠ () then // z is a counterexample
11         o ← trace(T, Θ′, z, α)
12         if o ∈ ℝ^⟨L⟩ then // The origin is a counter input
13             return o
           /* Refine by replacing the origin                        */
14         Θ″ ← getNode(T, Θ′, o)
15         V′ ← V \ {Θ ∈ V | Θ is a descendant of Θ″}
16         E′ ← E ∩ V′ × V′
17         op_e ← getExact(o)
18         R ← op_e(Θ″)
19         T = (V, E) ← (V′ ∪ R, E′ ∪ {(Θ″, Θ‴) | Θ‴ ∈ R})
           /* Refinement if new leaves in R are final               */
20         for Θ_e ∈ R do
21             if isFinal(T, F, Θ_e) then
22                 (z, α) ← checkSafety(Θ_e, S)
23                 if z ≠ () then
24                     o ← trace(T, Θ′, z, α)
25                     if o ∈ ℝ^⟨L⟩ then // The origin is a counter input
26                         return ⊥
27                     Θ′_e ← getNode(T, Θ_e, r)
28                     V′ ← V \ {Θ ∈ V | Θ is a descendant of Θ′_e}
29                     E′ ← E ∩ V′ × V′
30                     op_e ← getExact(o)
31                     T = (V, E) ← (V′ ∪ op_e(Θ_e), E′ ∪ {(Θ_e, Θ′_e) | Θ′_e ∈ op_e(Θ_e)})
32                     break // Continue after line 2
33             else
34                 break // Continue after line 2
35 return ⊤
```

are safe. This algorithm behaves the same way. The unsafety is identified with a counter input. The safety of all final leaves is detected when the reachability tree is full at the beginning of a cycle. In this case, all leaves in the tree are final and safe leaves. If such a leaf would not be safe, the algorithm would have refined the origin of this unsafety and thus created non-final leaves. This contradiction also applies to the special case where the last operation is an origin since these leaves are checked for safety and refined if necessary.

This process is captured by Algorithm 14. It uses the same input and output as Algorithm 13 and thus verifies an FNN F based on an input star Θ for a safety specification $\mathcal{S}$. For this process, it also utilizes the same calls to methods.

First the initial reachability tree is constructed consisting only of Θ as its root. This tree is than extended in lines 4 to 8 based on the application of reachability analysis until a single final leaf is computed. To this end, a non-final leaf in the reachability tree is picked. We use depth-first search to find such a leaf. In the first cycle, this leaf is Θ. Based on the leaf, a subtree is computed using only affine and over-approximated activation function operation. This process is analog to the construction of the full reachability tree in the Algorithm 13.

Once a final leaf is computed, the algorithm behaves the same way as the previous refinement algorithm. Thus, the safety of the final leaf is determined. If the leaf is safe, the algorithm continues with the next non-final leaf or returns the safety of F for the given input. Otherwise, the counterexample found during the safety check is traced to its origin which is refined.

After this refinement, this algorithm once more differs from the previous one. As mentioned before, the last operation in an FNN can be an origin. If such an operation is identified as an origin, refinement in lines 14 to 19 adds final leaves to the reachability tree. These leaves need to be checked for safety directly because the algorithm would not otherwise do so. This is done using exactly the same process as before.

3.4.3 Refinement with Exact Origins

This method is the first heuristic we introduce for the refinement process. During each loop, the origins of counterexamples are kept track of. The activation function operations corresponding to these origins are then always computed in the exact manner during later cycles. This ensures that each operation can only be an origin at most once. Using more exact operations has the disadvantage of an increased amount of reachable sets that need to be checked for safety.

This method works with both the full and avoidant refinement algorithms introduced in the previous section. In both cases, the previous origins need to be kept track of. This can be done with a set to which origins are added once they are identified after tracing.

Since the full refinement algorithm always constructs full reachability trees, only the `getOperation` method used in line 6 of Algorithm 13 needs to be changed to implement this heuristic. Instead of always returning an over-approximated activation function operation, the method returns the corresponding exact operation, if the operation is part of the set of previous origins.

The construction of reachability trees in the avoidant refinement algorithm needs to be changed to a greater degree. The reason for this change is the assumption that only a single leaf is added in each iteration of the construction loop which is not true for exact activation function operation in general. To make the methods compatible,

Algorithm 15: Reachability Tree Construction with Exact Origins

input : FNN F,
Reachability tree T,
Star Θ,
Set of previous origins L
output : Updated reachability tree T' and star Θ_0

1 op $\leftarrow$ `getOperation`(T, F, Θ) `// affine or approximate`
2 **if** $op \in \mathsf{L}$ **then**
3 $\quad$ op $\leftarrow$ `getExact`(op)
4 $\mathsf{R} \leftarrow \text{op}(\Theta)$ `//` `Let` $\mathsf{R} = \{\Theta_0, \ldots, \Theta_k\}$
5 $T' = (\mathsf{V}, \mathsf{E}) \leftarrow (\mathsf{V} \cup \mathsf{R}, \mathsf{E} \cup (\{\Theta\} \times \mathsf{R}))$
6 **return** T', Θ_0

one of these leaves is chosen for further computation as presented in Algorithm 15.

This algorithm is meant to replace lines 5 and 6 in Algorithm 14. In addition, the reachability tree and star that are set in lines 7 and 8 should be set corresponding to the output of this algorithm. The methods used here behave the same way as in the previous sections. This includes the `getOperation` method.

The algorithm ensures that exact operations are used instead of over-approximated once for previous origin. In addition, a newly computed leaf with which the remaining refinement algorithm can continue is picked. Note, that the algorithm could also be used to replace the analog computation in Algorithm 13 instead of the change to the `getOperation` method. For this the information of the next leaf would simply be discarded.

This heuristic can in principle be used in conjunction with any of the tracing and safety checking methods introduced in the previous sections. In practice, a similar problem to the safety checking method that uses previous counterexamples introduced in Section 3.2.1 is solved. While it is theoretically possible that a counterexample remains after an origin is removed, it is unlikely in practice. This checking method thus unnecessarily increases the computational effort required for the safety checks. Even less useful, but still possible, is the combination of this heuristic with the tracing method that keeps track of origins to check if they remain sources introduced in Section 3.3.3. Since this heuristic ensures all previous origins are computed in an exact manner and can thus not be origins again, this method just adds an unnecessary overhead by checking these operations again.

3.4.4 Refinement with Remembered Sources

This refinement heuristic also changes the reachability analysis used for the construction of the reachability tree. For this it keeps track of previous origins and the last source that leads to the identification of this origin. Assume an over-approximated activation function operation op is an origin and applied to a reachable set Θ resulting in $\Theta' \in \text{op}(\Theta)$. This method then keeps track of a source $\boldsymbol{z} \in [\![\Theta']\!]$ and the operation op.

During the construction of the reachability tree, the method then first constructs new nodes of the reachability tree using over-approximated activation function operations similar to the construction in the unmodified full or avoidant refinement algorithm. Once a previous origin is computed in the over-approximated manner, it is

Algorithm 16: Reachability Tree Construction with Remembered Sources

input : FNN F,
Reachability tree $T = (\mathsf{V}, \mathsf{E})$,
(n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$,
Set of previous sources and origins $\mathsf{L} = \{(\mathrm{op}_{l_1}, \boldsymbol{z}^{(1)}), \ldots, (\mathrm{op}_{l_\lambda}, \boldsymbol{z}^{(\lambda)})\}$
output : Updated reachability tree T and star Θ'

```
1  op ← getOperation(T, F, Θ) // affine or approximate
2  {Θ'} ← op(Θ) // Let Θ' = ⟨c', V', P'⟩
3  if not isAffine(op) then
4  |  for i = 1 to λ do
5  |  |  if op_{l_i} = op then
6  |  |  |  for j = 1 to n do
7  |  |  |  |  φ_j ← constructFormulaEq(z^(i), j, Θ')
8  |  |  |  ψ_{P'} ← constructFormula(P')
9  |  |  |  α ← checkFeasibility(ψ_{P'} ∧ ⋀_{i=1}^{n} φ_i)
10 |  |  |  if α ≠ () then // z^(i) ∈ [[Θ']]
11 |  |  |  |  op ← getExact(op)
12 |  |  |  |  R ← op(Θ) // Let R = {Θ_0, ..., Θ_k}
13 |  |  |  |  T = (V, E) ← (V ∪ R, E ∪ ({Θ} × R))
14 |  |  |  |  return T, Θ_0
15 T = (V, E) ← (V ∪ {Θ'}, E ∪ ({Θ} × {Θ'}))
16 return T, Θ'
```

checked whether the resulting node contains the same source again. If this is the case, the resulting node is discarded and the corresponding exact operation is used instead.

This process is described by Algorithm 16 in more detail using methods that are all introduced previously. This algorithm is meant to be applied similarly to Algorithm 15. Thus, the algorithm for this heuristic is used to replace lines 5 and 6 in Algorithms 13 and 14 and set the variables in lines 7 and 8 to its output. This output is a reachability tree to which the stars resulting from the operation are added and one of these stars which allows further computation for avoidant refinement method.

To facilitate this computation, the input consists of an FNN, a reachability tree, and a star similar to the input of Algorithm 15. In addition, a set L containing all pairs of previous origins and the corresponding source as described above is a parameter for this method. These origins and sources need to be added during the tracing process.

The algorithm for the computation of the next set in the reachability tree starts the same as before by applying the next operation op to the input star. Assume that Θ' results from this computation. If this operation is affine, the algorithm continues with the typical update to the reachability tree as described for the full and avoidant refinement methods.

For activation function operations, it is checked whether op is a previous origin in the loop in lines 3 to 16. This is the case, if any pair in L contains the operation. Once a previous origin is identified, it is checked whether the corresponding source to the operation is contained in Θ' using the formula introduced in Proposition 3.2.2. This formula is feasible, if a predicate value $\boldsymbol{\alpha}$ exists that corresponds to the source in Θ'.

The feasibility check with `checkFeasibility` returns a nonempty tuple in this case. If such a tuple is found, the exact operation corresponding to op is used instead

of the over-approximation to construct the next nodes in the reachability tree. This process works the same way as in Algorithm 15. If no source with an operation corresponding to op is contained in Θ', the algorithm adds Θ' as child of Θ to the reachability tree.

Using this heuristic has a similar advantage to the previous heuristic, but reduces the amount of safety checks that need to be performed compared to the previous method. The disadvantage of this method are the additional containment checks which are necessary to decide whether exact computation should be used. This method is useable with all heuristics and methods introduced for tracing and safety checking. The same reduction of performance the previous heuristic has in combination with the tracing method introduce in Section 3.3.3 and the safety checking method introduced in Section 3.2.1 also occur with this method as similar problems are solved.

3.4.5 Refinement with Safe Histories

This heuristic can be used on top of the other refinement heuristics introduced before. It relies on the idea, that once a set in a node of a reachability tree is identified to be safe all subsets of this set represented by a node resulting from the same amount of operation of the corresponding FNN are also safe.

Proposition 3.4.2 (Safe Subtrees)**.** *Let Θ_i be a star resulting from the application of the first $i \in \{1, \ldots, L + N - 1\}$ operations in an FNN F with $L \in \mathbb{N}_{\geq 2}$ layers and $N = \sum_{i=2}^{L} \langle i \rangle$ non-input neurons.*

If the subtree of a reachability tree corresponding to F with Θ_i as root is safe, then all subtrees of reachability trees corresponding to F with a root represented by a star Θ'_i resulting from the first i operations of F such that $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$ are also safe.

Based on this proposition, it seems possible to simply keep track of sets represented by safe roots of subtrees and then check if other nodes calculated with the same number of operations are subsets of the known safe sets. This method is not applicable because testing whether a star is the subset of another star is computationally very expensive.

Nonetheless, the idea behind this heuristic can be applied by using the properties of reachability trees and reachability analysis to derive the necessary subset relation of two stars. The foundation for this is that an over-approximated activation function operation applied to a star results in a superset of the union of stars resulting from the corresponding exact operation applied to the same star.

Proposition 3.4.3 (Safe Paths)**.** *Let F be an FNN with $L \in \mathbb{N}_{\geq 2}$ layers and let $\Theta_0, \ldots, \Theta_i$ and $\Theta'_0, \Theta'_1, \ldots, \Theta'_i$ be two paths in reachability trees corresponding to F with the root $\Theta_0 = \Theta'_0$. If for all $j \in \{1, \ldots, i\}$ the following conditions hold, then $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$:*

- *If Θ_j is the result of an exact activation function operation applied to Θ_{j-1}, then Θ'_j is computed from Θ'_{j-1} using the same calculations. This means intersection with the same halfspaces and the application of the same affine mapping.*
- *If Θ_j is the result of an over-approximated activation function operation applied to Θ_{j-1}, then Θ'_j is the result of a corresponding exact or over-approximated operation applied to Θ'_{j-1}.*

- *If Θ_j is the result of an affine operation applied to Θ_{j-1}, then Θ'_j is the result of the same affine operation applied to Θ'_{j-1}.*

By keeping track of the operations applied to a node in a reachability tree, it is thus possible to identify the roots of safe subtrees. For this to work, it is necessary to not only keep track of the operation, but also the specific halfspace intersections and affine mappings that are applied to compute the node in the case of an exact activation function application. Once such a root is identified, no further reachability analysis or safety checking is necessary to ensure the safety of the corresponding subtree. Thus, the time for the construction of the reachability tree and the amount of final leaves for which safety needs to be checked can be reduced by applying this method. Note, that due to time constraints an implementation of this heuristic was not possible.

Chapter 4

Experimental Results

In this chapter, the different heuristics we implemented for the verification of FNN using CEGAR will be evaluated and compared with each other. Additionally, the best combination of these heuristics is then compared to the existing method of verification using exact or over-approximated reachability analysis. To this end, we first evaluate the drones benchmark with different combinations of heuristics for verification with CEGAR. Afterwards, the resulting best combination of heuristics is compared to the previous verification methods on the drones benchmark. This is followed by a similar evaluation on the sonar benchmark which is based on a FNN mapping complex inputs to a binary output. Lastly, we evaluate the verification with CEGAR on the thermostat benchmark.

All these evaluations were performed on a computer with an Intel Core i7-9700K CPU ($3.6GHz$, 8 cores) and $32GB$ RAM. In addition, computation was aborted after $1h$ of runtime.

4.1 Drone Benchmark

The drone benchmark [Dem23] consists of a set of eight FNNs designed to allow a drone to hover at a chosen altitude autonomously. The first four of these FNN are made up of two hidden layers. The remaining four have an additional hidden layer. In both groups of four, the size of the hidden layers increases from one FNN to the next, while decreasing with each consecutive layer. For all these FNNs, the size of each layer can be found in Table 4.1. In each hidden layer a ReLU activation function is applied. The output layer maps to $\mathbb{R}$ and does not apply any activation function, despite the definition for FNN provided in Section 2.1.

For the verification, input sets of $\mathcal{P} = \left\{\boldsymbol{x}' \in \mathbb{R}^{12} \;\middle|\; \|\boldsymbol{x} - \boldsymbol{x}'\|_\infty \leq \delta\right\}$ for constants $\boldsymbol{x} \in \mathbb{R}^{12}$ and $\delta \in \{0.01, 0.1\}$ are used. Such a set represents a 12-dimensional hyperbox around $\boldsymbol{x}$ and can be represented by a polytope $\mathcal{P} = \left\{\boldsymbol{\alpha} \in \mathbb{R}^{12} \;\middle|\; \psi_{\mathcal{P}}(\boldsymbol{\alpha})\right\}$ with

$$\psi_{\mathcal{P}}(\boldsymbol{\alpha}) := \bigwedge_{i=1}^{12} \left(x_i - \delta \leq \alpha_i \wedge \alpha_i \leq x_i + \delta\right).$$

This then corresponds to a star $\Theta = \left\langle \mathbf{0}^{(12)}, \boldsymbol{I}^{(12)}, \mathcal{P} \right\rangle$. For this benchmark safety is specified by a single polytope for each FNN.

	⟨1⟩	⟨2⟩	⟨3⟩	⟨4⟩	⟨5⟩
AC1	12	32	16	1	-
AC2	12	64	32	1	-
AC3	12	128	64	1	-
AC4	12	256	128	1	-
AC5	12	32	16	8	1
AC6	12	64	32	16	1
AC7	12	128	64	32	1
AC8	12	256	128	64	1

Table 4.1: The sizes of layers (including input and output layers) of all FNN in the benchmark.

In the following, this benchmark is used to first determine the most performative combination of heuristics for the verification of FNN using CEGAR presented in the previous chapter. Afterwards, this method is compared to the previous verification methods using exact and over-approximated reachability analysis.

4.1.1 Comparison of Tracing Methods

We first compare the verification of the drones benchmark for the tracing methods introduced in Section 3.3. To ensure the same conditions for the different methods, the basic safety checking method corresponding to Algorithm 7 and the avoidant refinement strategy corresponding to Algorithm 14 are used for verification with all tracing methods. The resulting verification time and the amount of tracing attempts during the verification can be found in Table 4.2 for the input set with $\delta = 0.01$. In this

	Time (ms)				Tracing Attempts			
	Basic	Binary	Origin	Core	Basic	Binary	Origin	Core
AC1	28480	14422	**13687**	14082	59	21	19	**15**
AC2	16311	11646	11012	**8953**	76	24	21	**13**
AC3	197442	104604	104765	**86490**	37	**8**	**8**	**8**
AC4	-	-	-	**2642829**	-	-	-	**17**
AC5	12643	3942	3986	**3365**	80	15	15	**9**
AC6	56321	**7509**	7574	7606	179	19	19	**10**
AC7	2926778	657925	**556049**	603130	572	87	75	**48**
AC8	-	585452	**492057**	534406	-	125	96	**68**

Table 4.2: Verification time in milliseconds and amount of tracing attempts using CEGAR with different tracing methods for the drones benchmark with $\delta = 0.01$. The bold number in each row indicates the method with the best performance. Dashes indicate a timeout.

table, columns are labeled with the tracing method used to verify the FNN specified by the row. For all tables in this section, Basic refers to the tracing method introduced in Section 3.3.1. The tracing method labeled as Binary refers to the method from Section 3.3.2. Origin is used to identify the heuristics used on top of the binary tracing method introduced in Section 3.3.3. Finally, the label Core is used for the method introduced in Section 3.3.4.

These results, confirm that the basic tracing method is significantly slower than the other introduced methods. In addition, the heuristics keeping track of previous origins increases the performance or maintains it compared to the binary search based methods. This seems especially impactful for large networks as can be seen for AC7 and AC8. The reason a similar performance increase does not occur for AC3 is the specific characteristics of the verification. Only a single origin is refined, before termination. Thus, the heuristics does not have any impact besides a slight overhead for this verification.

The final tracing method using the unsatisfiable cores of failed tracing attempts increases performance in half of cases. In all cases, the amount of tracings attempts performed decreases or stays the same. The higher verification time for AC1, AC7, and AC8 can be explained by the overhead of computing a core. Overall, the tracing methods introduced in Sections 3.3.3 and 3.3.4 achieve the best performance.

Note, that the networks of the drones benchmark were also tested for the second input set with $\delta = 0.1$. In this case, all verification attempts were aborted due to the time limit of one hour.

4.1.2 Comparison of Refinement Methods

Based on the previous section, we now compare the different refinement algorithms introduced in Section 3.4 with both high performing tracing methods. Like before, the basic safety checking method is used. The full and avoidant refinement methods introduced in Sections 3.4.1 and 3.4.2 are referenced with corresponding labels in the

	Time (ms)				Tracing Attempts			
	Full	Avoidant	E-Origin	Source	Full	Avoidant	E-Origin	Source
AC1	14114	13687	**11700**	14721	19	19	**15**	19
AC2	11121	11012	**10192**	11494	21	21	**17**	21
AC3	**103026**	104765	105016	105559	**8**	**8**	**8**	**8**
AC4	-	-	**3124603**	-	-	-	**21**	-
AC5	4742	**3986**	4014	3993	**15**	**15**	**15**	**15**
AC6	8871	7574	**7471**	7478	**19**	**19**	**19**	**19**
AC7	587647	556049	**368918**	615233	75	75	**46**	75
AC8	781904	492057	**304737**	533738	96	96	**57**	96

Table 4.3: Verification time in milliseconds and amount of tracing attempts using CEGAR with the tracing method introduced in Section 3.3.3 and different refinement methods for the drones benchmark with $\delta = 0.01$. The bold number in each row indicates the method with the best performance. Dashes indicate a timeout.

	Safety Checks				Final Leaves			
	Full	Avoidant	E-Origin	Source	Full	Avoidant	E-Origin	Source
AC1	7	6	6	6	3	3	4	3
AC2	7	6	6	6	3	3	4	3
AC3	3	3	3	3	2	2	2	2
AC4	6	6	6	6	3	3	4	3
AC5	3	2	2	2	2	2	2	2
AC6	3	2	2	2	2	2	2	2
AC7	19	14	19	14	4	4	14	4
AC8	23	12	5	12	4	4	4	4

Table 4.4: Amount of safety checks performed, and final leaves computed using CEGAR with the tracing methods introduced in Sections 3.3.3 and 3.3.4 and different refinement methods for the drones benchmark with $\delta = 0.01$. Note, that no data exists for the tracing method introduced in Section 3.3.3 for AC4 with all refinement methods except E-Origins.

tables in this section. The label E-Origin is used to indicate the algorithms using exact activation function operation for all identified origins introduced in Section 3.4.3. Finally, Source is used to refer to the heuristic described in Section 3.4.4.

In Table 4.3 the verification time and the amount of tracing attempts performed are compared. The verifications were performed with the tracing method introduced in Section 3.3.3 and all refinement methods. Most of the refinement methods perform similarly for the networks in this benchmark. The full refinement method performs slightly worse compared to the avoidant refinement method for most networks, and significantly worse for AC8. This cannot be explained by a different amount of tracing attempts like in the previous section. The reason for this difference in performance is the amount of safety checks illustrated in Table 4.4. These are higher for full refinement because full reachability trees are constructed.

The good performance of the full refinement method for AC3 can once more be explained by the specific verification. The all refinement methods perform the same computation. The initial fully over-approximated reachability analysis is refined exactly once resulting in a safe reachability tree. The slightly better performance can thus be explained by a reduced amount of overhead.

Compared to either method, the refinement heuristic introduced in Section 3.4.3 is faster or performs similarly. Because the avoidant refinement method performs slightly better, this heuristic is used on top of it instead of the full refinement method. The performance increase and the variance in the degree of this increase is explained by the reduced amount of tracing attempts. The opposed performance impact of an increased amount of required safety checks and final leaves does not change this. With this performance, this refinement method is the most efficient in this combination.

The final refinement heuristic is also used on top of the avoidant refinement method. Instead of increasing performance the overhead of this method reduces it. The reason for the decrease in performance is that the method is not actually applied. Thus, this benchmarks it is not conclusive whether this method could have an impact on the correct FNN.

	Time (ms)				Tracing Attempts			
	Full	Avoidant	E-Origin	Source	Full	Avoidant	E-Origin	Source
AC1	14555	14082	**12036**	14604	15	15	**11**	15
AC2	9256	8953	**7864**	9082	13	13	**9**	13
AC3	86972	**86490**	87434	87918	**8**	**8**	**8**	**8**
AC4	2577470	2642829	**1555611**	3094790	17	17	**13**	17
AC5	4205	**3365**	3592	3366	**9**	**9**	**9**	**9**
AC6	8978	7606	7529	**7391**	**10**	**10**	**10**	**10**
AC7	634857	603130	**314285**	685440	48	48	**26**	44
AC8	803029	534406	**286673**	668121	68	68	**32**	68

Table 4.5: Verification time in milliseconds and amount of tracing attempts using CEGAR with the tracing method introduced in Section 3.3.4 and different refinement methods for the drones benchmark with $\delta = 0.01$. The bold number in each row indicates the method with the best performance.

In Table 4.5, analog results for the verification with the tracing method using the unsatisfiable core are presented. The different refinement methods behave similarly to before with the method introduced in Section 3.4.3 performing the best.

Additionally, the low performance of the refinement heuristic introduced in Section 3.4.4 is consolidated. The effect of the overhead of this method seems to be especially impactful for longer computation based on the difference in computation to the avoidant refinement method for AC4. For AC7, the heuristic actually changes the computation. This reduces the number of tracing attempts slightly, but does not increase the performance compared to the avoidant refinement method.

Thus, the refinement method labeled E-Origin performs the best for both tracing methods. Comparing the performance between the tracing methods for this refinement method, the tracing method based on unsatisfiable cores either performs similarly or better.

Note, that once more the benchmarks for the input set with $\delta = 0.1$ did not finish within the time limit of $1h$.

4.1.3 Comparison of Safety Checking Methods

Based on the results of the previous sections, the combination of the unsatisfiable core based tracing method and the refinement method using exact operations for previous origins is the most performant. The last comparison of methods regards the safety checks. In Tables 4.4 and 4.5, this combination of refinement and tracing method is used with the basic tracing method. Table 4.6 holds the corresponding data for the safety checking method using previous counterexamples introduced in Section 3.2.1.

When comparing the verification times of these approaches, the basic safety checking method performs better or similar. The reason for this is, that each safety check resulting in a new counterexample can increase the computational cost of subsequent safety checks because the inclusion of these counterexamples is checked. The chosen refinement method ensures that each identified origin cannot be an origin in a later refinement cycle again. It is thus less likely, that a counterexample can be used more

	Time (ms)		Tracing Attempts		Safety Checks		Final Leaves	
	Basic	Counter	Basic	Counter	Basic	Counter	Basic	Counter
AC1	**12036**	12824	11	11	6	6	4	4
AC2	**7864**	7973	9	9	6	6	4	4
AC3	**87434**	92531	8	8	3	3	2	2
AC4	1555611	**1537637**	13	13	6	6	4	4
AC5	**3592**	3383	9	9	2	2	2	2
AC6	7529	**7353**	10	10	2	2	2	2
AC7	**314285**	376592	**26**	31	**19**	30	**14**	22
AC8	286673	**283308**	32	32	5	5	4	4

Table 4.6: Verification time in milliseconds and the amount of tracing attempts, safety checks, and final leaves using CEGAR with the tracing method introduced in Section 3.3.4, the refinement methods introduced in Section 3.4.3, and the safety checking heuristic introduced in Section 3.2.1 for the drones benchmark with $\delta = 0.01$. The bold number in each row indicates the method with the best performance.

than once. Therefore, the overhead of this method is greater than the computational benefit.

As seen in AC7, the reuse of a counterexample can in addition lead to an increased amount of tracings, safety checks, and final leaves. Because this is a singular result, and we do not introduce any methods to find counterexamples with any specific properties, it is inconclusive whether reusing counterexamples always leads to a performance decrease or only in some cases.

4.1.4 Comparison with Previous Methods

To allow for a better comparison between verification with exact reachability analysis, over-approximated reachability, and reachability analysis using CEGAR, the algorithms Algorithms 1 and 2 were implemented for reachability analysis with reachability trees. These algorithms perform non-parallelized reachability analysis and construct the corresponding reachability tree. For the exact method, this tree can be constructed using a breadth-first (BF) or a depth-first (DF) approach. This differentiation is not necessary for over-approximated reachability analysis. Every node in such a reachability tree has either exactly one child node or is a leaf. Additionally, these implementations terminate as soon as an unsafe final leaf is computed.

All results of both exact approaches for the drone benchmark are complied in Table 4.7. In this table the exact computation with BF is label E-BF and the DF-based one with E-DF. In addition, the label Exact is used for both methods if the results are the same. Besides significant differences in the computation time of both algorithms for the unsafe FNNs, both approaches behave relatively similar for safe FNNs. Since the order of final leaves differs between the BF approach and the DF approach, the difference in computation time for the unsafe FNNs does not directly indicate the superiority of one method over the other. It is however noticeable, that the DF approach produces a result faster or in comparable time. This occurs for unsafe FNN where the same, a lesser, or a higher number of final leaves are computed before the

		Time (ms)			Final Leaves		Safety	
	δ	E-BF	E-DF	Approximate	E-BF	E-DF	Exact	Approximate
AC1	0.1	2245071	2276649	477151	2660	2660	safe	unknown
AC1	0.01	13145	13129	2398	25	25	safe	unknown
AC2	0.1	-	-	1738191	-	-	-	unknown
AC2	0.01	3403	3568	2432	5	5	safe	unknown
AC3	0.1	-	-	-	-	-	-	-
AC3	0.01	161836	159360	24334	81	81	safe	unknown
AC4	0.1	-	-	-	-	-	-	-
AC4	0.01	-	-	104998	-	-	-	unknown
AC5	0.1	-	-	-	-	-	-	-
AC5	0.01	2131	1467	1659	3	3	unsafe	unknown
AC6	0.1	-	-	-	-	-	-	-
AC6	0.01	5263	5275	3531	1	4	unsafe	unknown
AC7	0.1	-	-	-	-	-	-	-
AC7	0.01	140856	139500	43185	88	88	safe	unknown
AC8	0.1	-	-	-	-	-	-	-
AC8	0.01	361246	94180	77222	13	3	unsafe	unknown

Table 4.7: In , the verification time in milliseconds, the amount of final leaves, and the resulting safety of the drone benchmark FNNs derived using exact computation with breadth-first (BF) or depth-first (DF) is displayed. In , the verification time in milliseconds and the resulting knowledge regarding the safety of the FNN using over-approximated computation is presented for the drones benchmark.

unsafety is found. The likely reason for this behavior is the difference in computation. The BF approach calculates all non-final leaves, before the first final leaf is computed. On the other hand, a final leaf is produced after computing as little non-final leaves as possible with the DF approach. Thus, DF exact computation is used to compare verification performance with CEGAR in the following sections.

The benchmark results for the over-approximated verification are also presented in Table 4.7 under the label Approximate. This approach terminates before the exact algorithm in almost all cases except for AC6 with $\delta = 0.01$, where the exact computation with DF also only requires a single safety check. Compared to any CEGAR verification, the over-approximated computation always terminates faster, because this computation is the first step of CEGAR. Different from both the CEGAR-based and exact verification, over-approximated reachability analysis may result in an unknown safety of the FNN. This is the case for all FNNs in the drone benchmark. Thus, the over-approximate analysis cannot provide a conclusive answer for this benchmark despite its efficiency.

Comparing our best CEGAR-based verification method presented in Table 4.8 with the exact approach, CEGAR is only faster for three of the eight FNNs that terminate within the $1h$ time limit. Only for two of these, the performance increase is significant.

		Time (ms)		Checks		Variables		Constraints	
	δ	Exact	CEGAR	Exact	CEGAR	Exact	CEGAR	Exact	CEGAR
AC1	0.1	**2276649**	-	**2660**	-	12	[31, 36]	[28, 41]	[83, 96]
AC1	0.01	13129	**12036**	25	**17**	12	[15, 18]	[28, 29]	[35, 42]
AC2	0.1	-	-	-	-	12	[39, 39]	[30, 48]	[105, 105]
AC2	0.01	**3568**	7864	**5**	15	12	[13, 15]	[25, 27]	[29, 33]
AC3	0.1	-	-	-	-	12	-	[35, 50]	-
AC3	0.01	159360	**87434**	81	**11**	12	[19, 20]	[27, 32]	[46, 48]
AC4	0.1	-	-	-	-	12	-	[32, 41]	-
AC4	0.01	-	**1555611**	-	**19**	12	[21, 23]	[29, 35]	[53, 57]
AC5	0.1	-	-	-	-	12	-	[32, 49]	-
AC5	0.01	**1467**	3592	**3**	11	12	[14, 15]	[25, 27]	[31, 33]
AC6	0.1	-	-	-	-	12	-	[29, 49]	-
AC6	0.01	**5275**	7529	**1**	12	12	[13, 14]	[26, 26]	[28, 30]
AC7	0.1	-	-	-	-	12	-	[34, 49]	-
AC7	0.01	**139500**	314285	88	**45**	12	[13, 20]	[29, 32]	[31, 48]
AC8	0.1	-	-	-	-	12	-	[37, 49]	-
AC8	0.01	**94180**	286673	**13**	37	12	[18, 22]	[28, 30]	[45, 54]

Table 4.8: Comparison of the best combination of heuristics for CEGAR-based and exact verification with corresponding label for columns. For the verification time in milliseconds and the amount of checks the bold number in each row indicates the method with the better performance. Dashes indicate a timeout. Note, that data prior to timeouts was still recorded. The dashes in columns Variables and Constraints thus indicate unfinished reachability analysis.

Whether CEGAR is faster or slower is primarily dependent on the amount of feasibility checks performed and the complexity of these checks. This is enough of a reason for the verification of FNNs AC2, AC5, AC6, and AC8 with $\delta = 0.01$ to be slower with CEGAR than with the exact computation. The amount of stars checked for safety is lower with exact calculation than the sum of tracing attempts and safety checks. Both of these values are contained in the column labeled Checks. Note, that affine tracing does not require feasibility checks and that these attempts are still counted towards the sum of tracing attempts and safety checks.

For AC1 with $\delta = 0.1$ and AC7 with $\delta = 0.01$, this is not a sufficient reason as the amount of feasibility checks for the exact computation is higher than the aforementioned sum. The higher number of safety checks in the exact computation is still performed faster because the amount of constraints in the corresponding feasibility checks is lower than in the checks for tracing and safety checking with CEGAR. For AC1, the amount of constraints in the predicates of all stars represented by final leaves is between 28 and 41 for the exact computation. All of these constraints have 12 variables. For the same computation with CEGAR before termination due to time out, the number of variables ranged between 36 and 31 and the amount of constraints between 83 and 96. Similarly, the stars checked for safety during verification of AC7 had 12 variables and 28 to 40 constraints each, while the computation with CEGAR

Input	Over-Approximated	CEGAR	Exact
9	7556	7984	**6638**
27	10180	**10907**	11033
48	7230	1038998	**6800**
70	10774	11386	**10461**
88	14307	15050	**14060**
94	13353	14227	**12563**
97	8481	8956	**8346**
98	15377	**15826**	16782
167	7998	8596	**7418**
176	8893	9495	**8846**

Table 4.9: Comparison of verification times in milliseconds for relevant inputs of the sonar benchmark. Between the CEGAR-based and exact method, the faster verification times are indicated by bold numbers.

worked with stars with 13 to 20 variables and 34 to 48 constraints. The ranges for these amounts of variables and constraints are presented in Table 4.8 with corresponding names.

The remaining verifications of FNNs where thus faster with CEGAR than with the exact approach, when the amount of tracing attempts and safety checks was low enough to compensate for the increased cost of feasibility checks.

4.2 Sonar Benchmark

The Connectionist Bench (Sonar, Mines vs. Rock) [SG88], or sonar benchmark in short, provides data for training an FNN. This FNN can than classify whether a sonar signal has bounced of rocks or metal cylinders (mines). To this end, the FNN takes a 60-dimensional vector in $[0, 1]^{60}$ of a frequency modulated chirps as input. Similar to the previous benchmark, the input set is defined as $\mathcal{I} := \left\{ \boldsymbol{x}' \in \mathbb{R}^{60} \mid \|\boldsymbol{x} - \boldsymbol{x}'\|_\infty \leq 0.0001 \right\}$ for a vector $\boldsymbol{x}$ corresponding to one of the aforementioned signals. The expected output is then a singleton set containing only the single value 0 (rock) or 1 (metal).

The FNN performing this computation consists of four layers, where the first two have size 60 and all others have size 1. Note, that the affine mappings applied between layers of size 1 are all the identity. The activation function corresponding to the second layer is ReLU. Afterwards, a HardSigmoid activation function is applied with $V_{min} = -2.5$ and $V_{max} = 2.5$. Finally, a UnitStep activation function is used to classify the output to rocks or metal cylinders with $v = 0.5, R_{min} = 0$, and $R_{max} = 1$.

The benchmark provides 208 inputs $\boldsymbol{x}$. Of these, the first 97 are expected to be classified as rock and the remaining chirps correspond to metal. For most of these, the exact and over-approximated reachability analysis performs the same calculations. A comparison of verification time for the 10 inputs for which different computation occurs is presented in Table 4.9.

Only for numbers 9 and 48, the over-approximation is not able to identify the safety of the FNN. In both cases, reason for result is the actual unsafety of the FNN.

Input	Safety Specification	Over-Approximated	CEGAR	Exact
$([22, 23], \text{ON})$	OFF	199	193	291
$([22, 23], \text{ON})$	ON	192	451	291
$((-\infty, 17], \text{OFF})$	OFF	262	256	444
$((-\infty, 17], \text{OFF})$	ON	267	235	474

Table 4.10: Comparison of verification times in milliseconds for the thermostat benchmark.

For these inputs, the CEGAR approach only traces counterexamples identified in the output of the over-approximation. For input 9, a counter input is found after 4 tracing attempts within a reasonable timeframe. This is not the case for input 48 even though the number of tracing attempts required is the same. The difference between these inputs is, that the output for 9 is computed using exact calculations only, while the output for 48 is computed using one over-approximation for a HardSigmoid activation function application.

Comparing the verification times for these inputs, the exact method is only slower than the over-approximated one for inputs 27 and 98. In these cases, CEGAR is also faster than the exact method. Otherwise, CEGAR is slower due to the already slower over-approximation. It is unclear, why CEGAR performs this bad for input 27.

4.3 Thermostat Benchmark

The thermostat benchmark [Jia23] consists of an FNN designed to control the temperature in a room by activating and deactivating a heater. The FNN input consists of the current temperature of the room and the state of the heater. This state is indicated by 1 (ON) and 0 (OFF). Based on this, the benchmark provides two input sets. The first set corresponds to an active heater and a room temperature between 22 and 23. The second one to an inactive heater and a room temperature below 17.

Based on these inputs the FNN is meant to calculate whether the heater should stay active or not. This is indicated by a 1-dimensional output that is 0, if the heater should be turned off, and 15 otherwise. The FNN is expected to behave according to the following function

$$F : \mathbb{R} \times \{0, 1\} \to \{0, 15\}, (x, m) \mapsto \begin{cases} 0 & \text{, if } 22 \leq x \\ 15 & \text{, if } x \leq 18 \\ m \cdot 15 & \text{, otherwise.} \end{cases}$$

Thus, the safety specification consists of a single polytope corresponding to 0 or 15 with a tolerance of 0.1.

The given FNN is composed of 4 layers. Both hidden layers have ten neurons and apply the ReLU activation function. The output layer only has size 1 and applies a UnitStep activation function with $v = 7.5, V_{min} = 0$, and $V_{max} = 15$.

This benchmark is intended to verify the combination of input and safety specification in rows 1 and 4 in Table 4.10. We still decided to use this benchmark with combination for data. Only the combination of the input corresponding to a room

temperature in $(-\infty, 17]$ and with an activated heater (ON) is safe. This safety is already identified by the over-approximated method. This is thus the reason for the better performance of CEGAR compared to the exact method for this input and safety combination.

Because the other combinations are unsafe, the result of the over-approximation is otherwise not meaningful. Using CEGAR for verification still provides a higher performance in two of the other cases. In these, the amount of tracings and safety checks is once again low. For the slower case, the amount of these operations is high making the exact computation method faster as expected.

Chapter 5

Conclusion

In this thesis, a new approach for the verification of FNN using five widely-used piecewise linear activation functions is introduced. This approach is based on CEGAR and pre-existing methods for star-based reachability analysis of such FNNs [TMLM$^+$19, AMÁ23]. We first summarized the pre-existing reachability algorithms for exact and over-approximated analysis for the individual activation functions. Based on these algorithms, the CEGAR based reachability analysis is introduced. This analysis is composed of sub-algorithms for safety checking, tracing of counterexamples, and the refinement of the analysis. For all of these, sub-algorithms multiple approaches and heuristics are introduced. All of these heuristics except for the refinement heuristic introduced in Section 3.4.5 are implemented in HyPro [SÁMK17].

Based on the performance of different combinations of heuristics discussed in Chapter 4, the fastest combination for verification of FNN with CEGAR is identified. This method uses the basic safety checking method introduced in Section 3.2 which simply identifies a counterexample using a feasibility check with *Z3*. In addition to this safety method, tracing based on unsatisfiable cores and refinement with application of exact activation functions operations for all identified origins is used. The corresponding algorithms are introduced in Sections 3.3.4 and 3.4.3.

Despite partially relying on over-approximation for reachability analysis, it is still a complete method for verification. This completeness is the desired advantage of the CEGAR-based verification compared to verification with over-approximated reachability analysis. The second part of the initial goal for CEGAR-based verification is an increased performance compared to verification with exact reachability analysis.

This goal is only partially satisfied by the method introduced in this thesis. Verification of FNN with CEGAR sometimes leads to decreased verification time compared to the exact method and sometimes not. Whether the method with CEGAR is faster depends on the specific FNN and input set. Specifically, on the number of resulting reachable sets with exact approach compared to the amount of feasibility problems used for safety checking and tracing. In addition to the amount, the complexity of these problems impacts the verification time of CEGAR. Generally, the complexity of the formulas checked on an over-approximation heavy reachability analysis is higher than for a reachability analysis focused on exact computation.

5.1 Future work

In this thesis, various heuristics and methods for the verification of FNN with CEGAR are introduced. Part of future work are the introduction and implementation of further heuristic to increase performance further. A first unimplemented heuristic is already proposed in Section 3.4.5. The implementation of this heuristic requires additional tracking of the specific computations performed to generate reachable sets during reachability analysis. Once implemented, this approach likely reduces the time invested in reachability analysis and the amount of both tracing attempts and safety checks for a rather low overhead, if the tracking is implemented correctly.

In addition to this refinement heuristic, specifying the counterexample derived during the safety checking might lead to better verification results. Counterexamples with specific properties might lead to origins that have a greater impact on the reachable sets of an FNN. Finding such origins first would then reduce the amount of necessary refinement loops and thus also safety checks and tracing attempts. An interesting property of counterexamples could be the relative position in regard to the bounds of the unsafe subset of the reachable set. Specifically the distance of the counterexamples to safe values in the reachable set might impact the verification algorithms.

All heuristics and methods for verification with CEGAR introduced in this thesis use one concrete counterexample to find a single origin using a sequence of sources. An implementation of safety checking and tracing that relies on a symbolic representation of counterexamples and sources might lead to a performance increase. Such an implementation would allow to always find the first origin or all origins of a counterexample. This could thus reduce the required amount of refinement cycles.

The exact reachability analysis and thus verification using it is easily parallelizable. Our approaches using CEGAR is less straight forward to parallelize. The primary reason for this is the refinement process which repeatedly discards and reconstructs subtrees of reachability trees. Multiple parallel process performing reachability analysis, tracing, or safety checks thus need to be implemented facilitating this property of the refinement process.

Bibliography

[AÁM25] László Antal, Erika Ábrahám, and Hana Masara. Generalizing neural network verification to the family of piece-wise linear activation functions. *Science of Computer Programming*, 243:103269, 2025.

[AMÁ23] László Antal, Hana Masara, and Erika Ábrahám. Extending neural network verification to a larger family of piece-wise linear activation functions. In *EPTCS*, volume 395, pages [30]–68. NICTA, 2023.

[Bak21] Stanley Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods*, pages 19–36, Cham, 2021. Springer International Publishing.

[BD17] Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 401–420, Cham, 2017. Springer International Publishing.

[BdMNW] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. Programming Z3. `https://z3prover.github.io/papers/programmingz3.html#sec-references`. Accessed: 12-03-2025.

[CGI+18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Experimenting on solving nonlinear integer arithmetic with incremental linearization. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 383–398, Cham, 2018. Springer International Publishing.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 81–94, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Dem23] Stefano Demarchi. *Experimenting with constraint programming techniques in artificial intelligence: Automated system design and verification of neural networks.* Phd thesis, Università degli studi di Genova, 2023. Available at `https://hdl.handle.net/11567/1117675`.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[DV16] Parasara Sridhar Duggirala and Mahesh Viswanathan. Parsimonious, simulation based verification of linear systems. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 477–494, Cham, 2016. Springer International Publishing.

[Gus22] Murilo Gustineli. A survey on recently proposed activation functions for deep learning, 2022.

[Jia23] Ruoran Gabriela Jiang. *Verifying AI-controlled hybrid systems.* Master's thesis, RWTH Aachen, March 2023. Available at `https://ths.rwth-aachen.de/wp-content/uploads/sites/4/master_thesis_jiang.pdf`.

[KBD+17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[LMSR08] Matilde P. Legua, Isabel Morales, and Luis M. Sánchez Ruiz. The heaviside step function and matlab. In Osvaldo Gervasi, Beniamino Murgante, Antonio Laganà, David Taniar, Youngsong Mun, and Marina L. Gavrilova, editors, *Computational Science and Its Applications – ICCSA 2008*, pages 1212–1221, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Maa13] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models, 2013.

[Mas23] Hana Masara. Star set-based reachability analysis of neural networks with differing layers and activation functions. Bachelor's thesis, RWTH Aachen University, Aachen, Germany, 2023.

[Mei72] William S. Meisel. Chapter vii piecewise linear discriminant functions. In *Computer-Oriented Approaches to Pattern Recognition*, volume 83 of *Mathematics in Science and Engineering*, page 120. Elsevier, 1972.

[SÁMK17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. Hypro: A c++ library of state set representations for hybrid systems reachability analysis. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods*, pages 288–294, Cham, 2017. Springer International Publishing.

[SG88] Terry Sejnowski and R. Gorman. Connectionist Bench (Sonar, Mines vs. Rocks). UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C5T01Q.

[SGPV19] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[TAH+20] Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, Krunal Patel, and Juan Pablo Vielma. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification, 2020.

[TMLM+19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-based reachability analysis of deep neural networks. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 670–686, Cham, 2019. Springer International Publishing.

[TPL+21] Hoang-Dung Tran, Neelanjana Pal, Diego Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. Verification of piecewise deep neural networks: a star set approach with zonotope pre-filter. *Form. Asp. Comput.*, 33(4–5):519–545, August 2021.

[XLD+20] Jin Xu, Zishan Li, Bowen Du, Miaomiao Zhang, and Jing Liu. Reluplex made more practical: Leaky relu. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2020.

[YAT+20] Yongbin Yu, Kwabena Adu, Nyima Tashi, Patrick Anokye, Xiangxiang Wang, and Mighty Abra Ayidzoe. Rmaf: Relu-memristor-like activation function for deep learning. *IEEE Access*, 8:72727–72741, 2020.

Appendix A

Proofs

Proposition A.0.1 (Emptiness [TMLM+19]). *A star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ is empty if and only if $\mathcal{P}$ is empty.*

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star for $n, m \in \mathbb{N}_+$.
If $\mathcal{P} = \emptyset$, then $[\![\Theta]\!] = \{\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\} = \emptyset$.
If $[\![\Theta]\!] \neq \emptyset$, then an $\boldsymbol{x} \in [\![\Theta]\!]$ exists. Because $[\![\Theta]\!] = \{\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\}$, an $\boldsymbol{\alpha} \in \mathcal{P}$ exists such that $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$. Thus, $\mathcal{P} \neq \emptyset$.
Therefore, the proposition holds. □

Proposition A.0.2 (Representing Polyhedra with Star Sets [BD17]). *Any bounded, convex polytope $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{d}\}$ for $n \in \mathbb{N}_+$ can be represented by the (n, n)-dimensional star $\left\langle \mathbf{0}^{(n)}, \boldsymbol{I}^{(n)}, \mathcal{P} \right\rangle$ with $\mathbf{0}^{(n)} \in \mathbb{R}^n$ the origin and $\boldsymbol{I}^{(n)} \in \mathbb{R}^{n \times n}$ the identity matrix.*

Proof. Let $\mathcal{P} \subseteq \mathbb{R}^n$ for $n \in \mathbb{N}_+$ a polytope and $\Theta = \left\langle \mathbf{0}^{(n)}, \boldsymbol{I}^{(n)}, \mathcal{P} \right\rangle$. Then the following holds

$$\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^n \mid \boldsymbol{\alpha} \in \mathcal{P}\} = \left\{ \mathbf{0}^{(n)} + \boldsymbol{I}^{(n)}\boldsymbol{\alpha} \in \mathbb{R}^n \;\middle|\; \boldsymbol{\alpha} \in \mathcal{P} \right\} = [\![\Theta]\!].$$

Therefore, the proposition holds. □

Proposition A.0.3 (Affine Mapping [TMLM+19]). *Let $k, m, n \in \mathbb{N}_+$. For any matrix $\boldsymbol{W} \in \mathbb{R}^{k \times n}$ and any vector $\boldsymbol{b} \in \mathbb{R}^k$ the affine mapping $\{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \mid \boldsymbol{x} \in [\![\Theta]\!]\}$ of a (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, can be represented by a (k, m)-dimensional star $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ where $\boldsymbol{c}' = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b}$ and $\boldsymbol{V}' = \boldsymbol{W}\boldsymbol{V}$.*

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star and $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ a (k, m)-dimensional star for $n, m, k \in \mathbb{N}_+$. Let further $\boldsymbol{c}' = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b}$ and $\boldsymbol{V}' = \boldsymbol{W}\boldsymbol{V}$ for $\boldsymbol{W} \in \mathbb{R}^{k \times n}$ and $\boldsymbol{b} \in \mathbb{R}^k$. Then the following holds

$$\begin{aligned}
\{\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x} \mid \boldsymbol{x} \in [\![\Theta]\!]\} &= \{\boldsymbol{W} \cdot (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}) + \boldsymbol{b} \mid \boldsymbol{\alpha} \in \mathcal{P}\} \\
&= \{(\boldsymbol{W}\boldsymbol{c} + \boldsymbol{b}) + (\boldsymbol{W}\boldsymbol{V})\,\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\} \\
&= \{\boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}\} \\
&= [\![\Theta']\!].
\end{aligned}$$

Therefore, the proposition holds. □

Proposition A.0.4 (Intersection with a Halfspace [TMLM+19]). *Assume $n, m \in \mathbb{N}_+$ for a (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and a halfspace $\mathcal{H} = \left\{ \boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{h}^T \boldsymbol{x} \leq g \right\}$ defined by $\boldsymbol{h} \in \mathbb{R}^n$ and $g \in \mathbb{R}$. Then, the intersection $\mathcal{H} \cap [\![\Theta]\!]$ can be represented by the (n, m)-dimensional star $\Theta' = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \cap \mathcal{P}' \rangle$ with*

$$\mathcal{P}' = \left\{ \boldsymbol{\alpha} \in \mathbb{R}^m \mid (\boldsymbol{h}^T \boldsymbol{V}) \boldsymbol{\alpha} \leq g - \boldsymbol{h}^T \boldsymbol{c} \right\}.$$

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star for $n, m \in \mathbb{N}_+$ and $\mathcal{H} = \left\{ \boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{h}^T \boldsymbol{x} \leq g \right\}$ for $\boldsymbol{h} \in \mathbb{R}^n$ and $g \in \mathbb{R}$. Let further $\mathcal{P} = \{ \boldsymbol{\alpha} \in \mathbb{R}^m \mid \boldsymbol{A}\boldsymbol{\alpha} \leq \boldsymbol{d} \}$ with $A \in \mathbb{R}^{p \times m}$ and $d \in \mathbb{R}^p$ for $p \in \mathbb{N}_+$. Let finally $\Theta' = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \cap \mathcal{P}' \rangle$ for $\mathcal{P}' = \left\{ \boldsymbol{\alpha} \in \mathbb{R}^m \mid (\boldsymbol{h}^T \boldsymbol{V}) \boldsymbol{\alpha} \leq g - \boldsymbol{h}^T \boldsymbol{c} \right\}$. Then the following holds

$$\begin{aligned}
\mathcal{H} \cap [\![\Theta]\!] &= \left\{ \boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{h}^T \boldsymbol{x} \leq g \right\} \cap \{ (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}) \in \mathbb{R}^n \mid \boldsymbol{\alpha} \in \mathcal{P} \} \\
&= \left\{ \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{h}^T (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}) \leq g \wedge \boldsymbol{\alpha} \in \mathcal{P} \right\} \\
&= \left\{ \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{h}^T \boldsymbol{c} + \boldsymbol{h}^T \boldsymbol{V}\boldsymbol{\alpha} \leq g \wedge \boldsymbol{\alpha} \in \mathcal{P} \right\} \\
&= \left\{ \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{h}^T \boldsymbol{V}\boldsymbol{\alpha} \leq g - \boldsymbol{h}^T \boldsymbol{c} \wedge \boldsymbol{\alpha} \in \mathcal{P} \right\} \\
&= \{ \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P}' \wedge \boldsymbol{\alpha} \in \mathcal{P} \} \\
&= [\![\Theta']\!].
\end{aligned}$$

Therefore, the proposition holds. □

Proposition A.0.5 (Bounds [AMÁ23]). *For $n, m \in \mathbb{N}_+$ and any (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$, the upper and lower bounds of the set in dimension $i \in \{1, \ldots, n\}$ can be calculated by:*

- *Lower bound:* $l_i := c_i + \min \{ \boldsymbol{V}_i \boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P} \}$
- *Upper bound:* $u_i := c_i + \max \{ \boldsymbol{V}_i \boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in \mathcal{P} \}$

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star for $n, m \in \mathbb{N}_+$ and $i \in \{1, \ldots, n\}$. Since $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$ for all $\boldsymbol{x} \in [\![\Theta]\!]$ for some $\boldsymbol{\alpha} \in \mathcal{P}$, $x_i = c_i + \boldsymbol{V}_i \boldsymbol{\alpha}$. Because c_i is a constant, the minimal or maximal value in dimension i corresponds to the minimal or maximal value of $\boldsymbol{V}_i \boldsymbol{\alpha}$ for $\boldsymbol{\alpha} \in \mathcal{P}$. Thus, the proposition holds. □

Proposition A.0.6 (Counter Input [TMLM+19]). *Assume an FNN F with $L \in \mathbb{N}_{\geq 2}$ layers, an input set $\mathcal{I} \subseteq \mathbb{R}^{\langle 1 \rangle}$ with $\mathcal{I} = [\![\langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle]\!]$ and a safety specification $\mathcal{S} \subseteq \mathbb{R}^{\langle L \rangle}$. Let $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P}' \rangle$ represent a reachable set of F with input set $\mathcal{I}$ computed using the exact algorithm and let $\boldsymbol{z} \in \neg\mathcal{S} \cap [\![\Theta']\!]$ be a counterexample. Then $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha}$ for some $\boldsymbol{\alpha} \in \mathcal{P}'$ and $F(\boldsymbol{x}) = \boldsymbol{z}$ for the counter input $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$.*

Proof. Let F be an FNN with $L \in \mathbb{N}_{\geq 2}$ layers, $\mathcal{I} = [\![\langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle]\!]$ a $(\langle 1 \rangle, \langle 1 \rangle)$-dimensional input star and $\mathcal{S} \subseteq \mathbb{R}^{\langle L \rangle}$ a safety specification. Let further $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P}' \rangle$ represent a reachable set of F based on the input set $\mathcal{I}$ computed using the exact algorithm. Let finally $\boldsymbol{z} \in (\neg\mathcal{S} \cap [\![\Theta']\!])$ be a counterexample with $\boldsymbol{\alpha} \in \mathcal{P}'$ such that $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha}$.

Because Θ' is the result of the exact reachability analysis, $\boldsymbol{V}', \boldsymbol{c}'$ are the result of $L + N - 1$ for $N = \sum_{i=2}^{L} \langle i \rangle$ affine mappings applied to $\boldsymbol{V}, \boldsymbol{c}$ and $\mathcal{P}'$ is the result of intersecting $\mathcal{P}$ with halfspaces. Thus, $\mathcal{P}' \subseteq \mathcal{P}$ which implies $\boldsymbol{\alpha} \in \mathcal{P}$ and thus $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \in [\![\Theta]\!]$. It also follows that $\boldsymbol{V}' = \boldsymbol{W}^{(L+N-1)} \cdot \ldots \cdot \boldsymbol{W}^{(1)} \cdot \boldsymbol{V}$ and $\boldsymbol{c}' = \boldsymbol{b}^{(L+N-1)} + \boldsymbol{W}^{(L+N-1)} \left(\cdots \left(\boldsymbol{b}^{(1)} + \boldsymbol{W}^{(1)} \cdot \boldsymbol{c} \right) \cdots \right)$ for the affine mapping with matrix $\boldsymbol{W}^{(i)}$ and vector $\boldsymbol{b}^{(i)}$ applied to the input in step i of the FNN computation

for $i \in \{1, \ldots, L+N-1\}$. Since the computation is exact, these affine mappings are thus also applied to derive any value $\boldsymbol{z} \in [\![\Theta']\!]$. Therefore,

$$F(\boldsymbol{x}) = \boldsymbol{b}^{(L+N-1)} + \boldsymbol{W}^{(L+N-1)} \left(\cdots \left(\boldsymbol{b}^{(1)} + \boldsymbol{W}^{(1)} \cdot \boldsymbol{x} \right) \cdots \right)$$

. Since $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$, $\boldsymbol{b}^{(L+N-1)} + \boldsymbol{W}^{(L+N-1)} \left(\cdots \left(\boldsymbol{b}^{(1)} + \boldsymbol{W}^{(1)} \cdot \boldsymbol{x} \right) \cdots \right) = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} = \boldsymbol{z}$.

Thus, the proposition holds. □

Proposition A.0.7 (Counterexamples)**.** *Assume an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and a safety specification $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i \subseteq \mathbb{R}^n$. Let $\psi_{\mathcal{P}}, \psi_{\mathcal{S}_1}, \ldots, \psi_{\mathcal{S}_k}$ be the linear real arithmetic formulas corresponding to $\mathcal{P}, \mathcal{S}_1, \ldots, \mathcal{S}_k$ as introduced in Section 2.4. Then all values $\boldsymbol{\alpha} \in \mathbb{R}^m$ satisfying the following formula correspond to counterexamples $\boldsymbol{z} \in [\![\Theta]\!] \cap \neg\mathcal{S}$ with $\boldsymbol{z} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$:*

$$\psi_{\mathcal{S},\Theta}(\boldsymbol{\alpha}) := \psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{k} \neg\psi_{\mathcal{S}_i}(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}).$$

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star and $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i \subseteq \mathbb{R}^n$ a safety specification for $n, m, k \in \mathbb{N}_+$.

Let $\boldsymbol{\alpha} \in \mathbb{R}^m$ such that $\psi_{\mathcal{S},\Theta}(\boldsymbol{\alpha}) \equiv \top$. Then $\boldsymbol{\alpha} \in \mathcal{P}$ because $\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \equiv \top$. Let $\boldsymbol{z} := \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$. Then $\boldsymbol{z} \in [\![\Theta]\!]$. Because for all $i \in \{1, \ldots, k\}$ it follows that $\psi_{\mathcal{S}_i}(\boldsymbol{z}) \equiv \bot$ which implies $\boldsymbol{z} \notin \mathcal{S}_i$. Therefore, $\boldsymbol{z} \notin \mathcal{S}$ and thus $\boldsymbol{z} \in \neg\mathcal{S}$.

Thus, $\boldsymbol{z} \in ([\![\Theta]\!] \cap \neg\mathcal{S})$ which makes it a counterexample and proves the proposition. □

Proposition A.0.8 (Elements of Stars)**.** *Assume an (n, m)-dimensional star $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ with predicate $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \psi_{\mathcal{P}}(\boldsymbol{\alpha})\}$ and a value $\boldsymbol{x} \in \mathbb{R}^n$. Then $\boldsymbol{x} \in [\![\Theta]\!]$ if and only if the following formula using is feasible*

$$\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \varphi_i(\boldsymbol{\alpha})$$

for $\varphi_i(\boldsymbol{\alpha})$ defined according to Formula 3.2.

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star and $\boldsymbol{x} \in \mathbb{R}^m$ for $n, m \in \mathbb{N}_+$. For checking the containment of $\boldsymbol{x}$ in Θ, $\varphi_i(\boldsymbol{\alpha}) := \left(c_i + \sum_{j=1}^{m} v_{ij}\alpha_j = x_i \right)$ for all $i \in \{1, \ldots, n\}$ and $\psi_{\mathcal{P}}$ is defined such that $\mathcal{P} = \{\boldsymbol{\alpha} \in \mathbb{R}^m \mid \psi_{\mathcal{P}}(\boldsymbol{\alpha})\}$.

If $\boldsymbol{x} \in [\![\Theta]\!]$, then an $\boldsymbol{\alpha} \in \mathcal{P}$ exists such that $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$. Thus,

$$x_i = c_i + \boldsymbol{V}_i\boldsymbol{\alpha} = c_i + \sum_{j=1}^{m} v_{ij}\alpha_j$$

which implies $\varphi_i(\boldsymbol{\alpha}) = \top$ for all $i \in \{1, \ldots, n\}$. Because $\boldsymbol{\alpha} \in \mathcal{P}$, $\psi_{\mathcal{P}}(\boldsymbol{\alpha}) = \top$. Therefore, $\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \varphi_i(\boldsymbol{\alpha})$ is satisfied by $\boldsymbol{\alpha}$ and thus feasible.

Let $\boldsymbol{\alpha} \in \mathbb{R}^m$ such that $\psi_{\mathcal{P}}(\boldsymbol{\alpha}) \wedge \bigwedge_{i=1}^{n} \varphi_i(\boldsymbol{\alpha}) \equiv \top$. Then $\boldsymbol{\alpha} \in \mathcal{P}$ and $x_i = c_i + \sum_{j=1}^{m} v_{ij}\alpha_j = c_i + \boldsymbol{V}_i\boldsymbol{\alpha}$. Therefore, $\boldsymbol{x} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$ and $\boldsymbol{x} \in [\![\Theta]\!]$.

Thus, the proposition holds. □

Proposition A.0.9 (Tracing through Affine Operations)**.** *Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ and $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ such that $\Theta' \in op(\Theta)$ for an affine operation $op(\cdot)$. For an element $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} \in [\![\Theta']\!]$ with $\boldsymbol{\alpha} \in \mathcal{P}$, the element $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha} \in [\![\Theta]\!]$ is a source of $\boldsymbol{z}$.*

Proof. Let $\Theta = \langle \boldsymbol{c}, \boldsymbol{V}, \mathcal{P} \rangle$ be an (n, m)-dimensional star and $\Theta' = \langle \boldsymbol{c}', \boldsymbol{V}', \mathcal{P} \rangle$ an (k, m)-dimensional star with $\Theta' \in \mathrm{op}(\Theta)$ for an affine operation $\mathrm{op}(\cdot)$. Let $\mathrm{op}(\cdot)$ correspond to the affine mapping with $\boldsymbol{W} \in \mathbb{R}^{k \times n}$ and $\boldsymbol{b} \in \mathbb{R}^k$. Thus, $\boldsymbol{c}' = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b}$ and $\boldsymbol{V}' = \boldsymbol{W}\boldsymbol{V}$.

Let $\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} \in [\![\Theta']\!]$ with $\boldsymbol{\alpha} \in \mathcal{P}$. Then, the following holds:

$$\boldsymbol{z} = \boldsymbol{c}' + \boldsymbol{V}'\boldsymbol{\alpha} = \boldsymbol{W}\boldsymbol{c} + \boldsymbol{b} + \boldsymbol{W}\boldsymbol{V}\boldsymbol{\alpha} = \boldsymbol{W}\left(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}\right) + \boldsymbol{b} = \mathrm{op}(\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}).$$

Thus, $\boldsymbol{c} + \boldsymbol{V}\boldsymbol{\alpha}$ is a source of $\boldsymbol{z}$ and the proposition holds. □

Proposition A.0.10 (Refinment)**.** *Let $\Theta_o \in op_o(\Theta)$ for two stars Θ_o and Θ in a reachability tree and op_o an over-approximated activation function operation.*

If $\boldsymbol{z} \in [\![\Theta_o]\!]$ is a source of a counterexample such that op_o is an origin of this counterexample, then $\boldsymbol{z} \notin \bigcup_{i=1}^{k} [\![\Theta_i]\!]$ for $op_e(\Theta) = \{\Theta_1, \ldots, \Theta_k\}$ where op_e is the exact activation function operation corresponding to op_o.

Proof. Let op_o be an over-approximated activation function operation, Θ, Θ_o two stars with $\Theta_o \in \mathrm{op}_o(\Theta)$, $\boldsymbol{z} \in [\![\Theta_o]\!]$ such that $\mathrm{op}_o(\boldsymbol{z}') \neq \boldsymbol{z}$ for all $\boldsymbol{z}' \in [\![\Theta]\!]$, and let op_e be the exact activation function operation corresponding to op_o with $\mathrm{op}_e(\Theta) := \{\Theta_1, \ldots, \Theta_k\}$ for $k \in \mathbb{N}_+$.

Assume $\boldsymbol{z} \in [\![\Theta_i]\!]$ for some $i \in \{1, \ldots, k\}$. Then $\boldsymbol{z}' \in [\![\Theta]\!]$ exists such that $\mathrm{op}_e(\boldsymbol{z}') = \boldsymbol{z}$, because op_e is exact. Because operations applied to single values are always exact, $\boldsymbol{z} = \mathrm{op}_e(\boldsymbol{z}') = \mathrm{op}_o(\boldsymbol{z}')$. This is a contradiction to $\mathrm{op}_o(\boldsymbol{z}') \neq \boldsymbol{z}$ for all $\boldsymbol{z}' \in [\![\Theta]\!]$. Therefore, $\boldsymbol{z} \notin [\![\Theta_i]\!]$ for all $i \in \{1, \ldots, k\}$ and thus $\boldsymbol{z} \notin \bigcup_{i=1}^{k} [\![\Theta_i]\!]$.

Thus, the proposition holds. □

Proposition A.0.11 (Safe Subtrees)**.** *Let Θ_i be a star resulting from the application of the first $i \in \{1, \ldots, L + N - 1\}$ operations in an FNN F with $L \in \mathbb{N}_{\geq 2}$ layers and $N = \sum_{i=2}^{L} \langle i \rangle$ non-input neurons.*

If the subtree of a reachability tree corresponding to F with Θ_i as root is safe, then all subtrees of reachability trees corresponding to F with a root represented by a star Θ_i' resulting from the first i operations of F such that $[\![\Theta_i']\!] \subseteq [\![\Theta_i]\!]$ are also safe.

Proof. Let Θ and Θ' be two stars with $[\![\Theta']\!] \subseteq [\![\Theta]\!]$ that result from the application of the first $i \in \mathbb{N}$ operations in a sufficiently large FNN F. Let T be a reachability tree corresponding to F and let Θ be the root of a safe subtree of T. Because this subtree is safe, the stars resulting from applying exact reachability analysis after Θ are also safe. Let finally $r \in \mathbb{N}$ be the number of remaining operation applications.

If $r = 0$, Θ and Θ' are final leaves and no further computation is necessary. Thus, safety directly corresponds to a safety specification $\mathcal{S}$. Since Θ is safe, $[\![\Theta]\!] \subseteq \mathcal{S}$ which implies $[\![\Theta']\!] \subseteq \mathcal{S}$. Therefore, Θ' is a safe final leave and thus the root of a safe subtree.

For $r + 1$ remaining operations, it is necessary to differentiate between the types of operation that are applied next. Let the next operation be an affine mapping with matrix $\boldsymbol{W}$ and vector $\boldsymbol{b}$ and Θ_a and Θ_a' the resulting stars from the application to Θ and Θ' respectively. For $\boldsymbol{x} \in [\![\Theta']\!]$, $(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \in [\![\Theta_a']\!]$. Because $[\![\Theta']\!] \subseteq [\![\Theta]\!]$, $(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \in [\![\Theta_a]\!]$. Thus, $[\![\Theta_a']\!] \subseteq [\![\Theta_a]\!]$.

Assume now that the next operation $\mathrm{op}(\cdot)$ is an exact activation function application. Let $\mathrm{op}(\Theta) = \{\Theta_1, \ldots, \Theta_k\}$ and $\mathcal{H}_{i,1}, \ldots, \mathcal{H}_{i,k_i}$ the halfspaces that are intersected with Θ to calculate Θ_i for $i \in \{1, \ldots, k\}$, $k_i \in \mathbb{N}$ and $k \in \mathbb{N}_+$. Let additionally $\boldsymbol{W}^{(i)}, \boldsymbol{b}^{(i)}$ be the matrix and vector applied to calculate Θ_i. Because $[\![\Theta']\!] \subseteq [\![\Theta]\!]$, these or less halfspaces and affine mappings are used to calculated $\mathrm{op}(\Theta')$. It holds, that $[\![\Theta']\!] \cap \bigcap_{j=1}^{k_j} \mathcal{H}_{i,j} \subseteq [\![\Theta]\!] \cap \bigcap_{j=1}^{k_j} \mathcal{H}_{i,j}$. The application of the affine mapping to these

intersected stars also preserve the subset property as described for the application of affine operations. Thus, the union of reachable sets resulting from Θ' is a subset of the union of reachable sets resulting from Θ.

Therefore, the proposition holds. □

Proposition A.0.12 (Safe Paths)**.** *Let F be an FNN with $L \in \mathbb{N}_{\geq 2}$ layers and let $\Theta_0, \ldots, \Theta_i$ and $\Theta'_0, \Theta'_1, \ldots, \Theta'_i$ be two paths in reachability trees corresponding to F with the root $\Theta_0 = \Theta'_0$. If for all $j \in \{1, \ldots, i\}$ the following conditions hold, then $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$:*

- *If Θ_j is the result of an exact activation function operation applied to Θ_{j-1}, then Θ'_j is computed from Θ'_{j-1} using the same calculations. This means the same intersections with halfspaces and the application of the same affine mappings.*
- *If Θ_j is the result of an over-approximated activation function operation applied to Θ_{j-1}, then Θ'_j is the result of a corresponding exact or over-approximated operation applied to Θ'_{j-1}.*
- *If Θ_j is the result of an affine operation applied to Θ_{j-1}, then Θ'_j is the result of the same affine operation applied to Θ'_{j-1}.*

Proof. Let $\Theta_0, \ldots, \Theta_i$ and $\Theta'_0, \Theta'_1, \ldots, \Theta'_i$ be two paths in reachability trees corresponding to an FNN F with roots $\Theta_0 = \Theta'_0$ for $i \in \mathbb{N}_+$. Let the conditions in the proposition hold for all $j \in \{1, \ldots, i\}$.

If $i = 0$, then $[\![\Theta'_0]\!] \subseteq [\![\Theta_0]\!]$ because $\Theta'_0 = \Theta_0$.

If otherwise $i+1$ operations are applied to generate Θ_{i+1} and Θ_{i+1} and $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$, the following prove depends on the remaining operation type. Let this operation be affine with matrix $\boldsymbol{W}$ and vector $\boldsymbol{b}$. Because $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$, the following holds

$$[\![\Theta'_{i+1}]\!] = \{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \mid \boldsymbol{x} \in [\![\Theta'_i]\!]\} \subseteq \{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \mid \boldsymbol{x} \in [\![\Theta_i]\!]\} = [\![\Theta_{i+1}]\!]$$

Let the operation now be the same exact activation function operation where Θ_i and Θ'_i are intersected with the halfspaces $\mathcal{H}_1, \ldots, \mathcal{H}_k$ for $k \in \mathbb{N}$ and the affine mapping with matrix $\boldsymbol{W}$ and vector $\boldsymbol{b}$ is applied to them. Because $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$, also $[\![\Theta'_i]\!] \cap \bigcap_{j=1}^{k} \mathcal{H}_j \subseteq [\![\Theta_i]\!] \cap \bigcap_{j=1}^{k} \mathcal{H}_j$. The application of the affine mapping to these new stars preserves the subset property as described for the previous operation. Thus, $[\![\Theta'_{i+1}]\!] \subseteq [\![\Theta_{i+1}]\!]$.

Let $\Theta_{i+1} \in \{\Theta_{i+1}\} = \mathrm{op}_o(\Theta_i)$ be calculated by an over-approximated activation function operation $\mathrm{op}_o(\cdot)$ applied to Θ_i. Let further $\Theta'_{i+1} \in \mathrm{op}_e(\Theta'_i)$ be calculated using an exact activation function application $\mathrm{op}_e(\cdot)$ where the halfspaces $\mathcal{H}_1, \ldots, \mathcal{H}_k$ for $k \in \mathbb{N}$ and the affine mapping with $\boldsymbol{W}$ and $\boldsymbol{b}$ are applied to Θ'_i to calculate Θ'_{i+1}. Let finally $\Theta_e \in \mathrm{op}_e(\Theta_i)$ be the result of the same exact activation function application with halfspace intersections with $\mathcal{H}_1, \ldots, \mathcal{H}_k$ and affine mapping with $\boldsymbol{W}$ and $\boldsymbol{b}$ applied to Θ_i. $\Theta_e \subseteq \Theta_{i+1}$, because Θ_e represents a subset of the exact application of an activation function, while Θ_{i+1} represents the over-approximation of the same function application. Because $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$, it holds that $[\![\Theta'_{i+1}]\!] \subseteq [\![\Theta_e]\!]$ based on the previous case. Therefore, $[\![\Theta'_{i+1}]\!] \subseteq [\![\Theta_{i+1}]\!]$.

Let Θ_{i+1} and Θ'_{i+1} now be calculated based on an over-approximated activation function operation. Because $[\![\Theta'_i]\!] \subseteq [\![\Theta_i]\!]$, the interval of values in the dimension to which the activation function is applied of Θ'_i is a subset of Θ_i. Because we use the tightest convex over-approximation possible, $[\![\Theta'_{i+1}]\!] \subseteq [\![\Theta_{i+1}]\!]$ follows.

Thus, the proposition holds. □