IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search

Author:
Patrick Henriksen

Supervisor:
Alessio Lomuscio

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (AI) of Imperial College London

September 2019

Abstract

Neural networks have over the last years become an essential technique for solving regression and classification problems with complex data. While these networks often achieve impressive results, the empirical methods commonly used to measure their performance have their limitations. We propose an efficient algorithm based on symbolic interval propagation for formal verification of large neural networks with high-dimensional input data. Our approach extends present state-of-the-art algorithms with three significant novel contributions. We use an adaptive node refinement, aiming to split the nodes with the most significant impact on the output bounds first. Furthermore, we use a gradient descent-based local search around spurious results produced by the LP-solver to substantially improve our algorithms ability to find valid counterexamples. Finally, we derive the necessary linear relaxations to support s-shaped activation functions such as the Sigmoid and Tanh. We have implemented the algorithm in a toolkit, VeriNet. Compared to present state-of-the-art algorithms, VeriNet achieves a speed-up of about an order of magnitude for safe cases and more than three orders of magnitude for unsafe-cases. [1]

[1] Prior to this project, I did a literature review of complete verification algorithms as an independent study option (ISO) under the supervision of Professor Alessio Lomuscio. Some of the algorithms covered in the ISO are also covered in the background section of this report. However, the background section significantly extends the literature review of the ISO with new material, and the presentation of similar topics is tailored towards the goal of this project. The resulting report differs substantially from the ISO, and I consider all parts of it to be independent of the ISO. However, for completeness, we have listed all similar topics in appendix B.

Acknowledgements

Most of all, I would like to express my sincere gratitude to my supervisor Alessio Lomuscio, for excellent guidance through the project. I would also like to thank my fellow students at Imperial and the University of Oslo for a great working environment during the past five years. A special thanks goes to Andreas Zinonos, Kendeas Theofanous, and Steven Boo at Imperial College and Vemund S. Schøyen and Sejla Ackar at the University of Oslo for a lot of fun during long hours at the labs. Finally, I would like to thank Aker Scholarships for the financial aid making this year at Imperial College possible.

Contents

Chapter 1

Introduction

Neural networks have had an enormous impact in the field of AI over the last years, and the rapid development has opened up a new world of possibilities in automation and analysis of complex high-dimensional data. However, the high degree of non-linearity exhibited by these networks has traditionally limited us to empirical performance metrics. This, combined with the fact that neural networks have been proven to be susceptible to adversarial examples (Szegedy et al., 2013), significantly limits their usefulness, especially in safety-critical applications.

Researchers have recently started focusing on formal verification algorithms for neural networks, and promising advances have already been made. Several formal verification tools (Ehlers, 2017; Tjeng et al., 2018; Katz(B) et al., 2017; Katz et al., 2019; Wang et al., 2018b,a) can prove important safety properties for small feed-forward networks, such as the ACASXu network (Kochenderfer et al., 2012) designed to avoid aircraft collisions. Other tools (Singh et al., 2018, 2019; Zhang et al., 2018) are also able to prove safety properties for medium-sized networks with thousands of ReLU nodes; however, these algorithms are usually incomplete, as discussed later. Furthermore, the work of (Akintunde et al., 2018, 2019) introduced verification tools for agent-environment systems, where the agents are controlled by feed-forward networks and recurrent networks, respectively.

We categorise formal verification algorithms into complete algorithms and sound but incomplete algorithms. Sound but incomplete methods can usually verify properties for relatively large networks with high-dimensional input, and several of them can also handle non-piecewise linear activation functions. Sound algorithms guarantee that properties verified as safe are actually safe; however, they may overestimate the number of unsafe properties or be unable to solve problems in a finite amount of time. Notable algorithms include (Gehr et al., 2018; Zhang et al., 2018; Singh et al., 2018, 2019).

A complete algorithm can always solve a given verification problem in a finite amount of time. However, most of these algorithms are in practice limited to either small networks or low-dimensional inputs due to their computational complexity. Complete algorithms are also limited to piecewise linear activation functions, and most of them only support the ReLU activation function. The algorithm from (Wang et al., 2018a) stands out among the complete algorithms as the only algorithm proven to be able to do verification on medium-sized networks with thousands of nodes and high-dimensional input. Other notable algorithms are (Ehlers, 2017; Katz(B) et al., 2017; Tjeng et al., 2018; Katz et al., 2019). Complete algorithms are always sound.

Complete verification algorithms can further be divided into three subcategories. The first category is the SMT-based algorithms, using boolean satisfiability solvers combined with an LP-solver to support piecewise-linear activation functions. Secondly, we have the MILP-based algorithms utilising binary and real-valued constraints to encode neural networks with piecewise linear activation functions. Finally, we have the symbolic interval propagation-based algorithms, using interval arithmetic to bound the behaviour of the individual nodes in the network. We explore different verification approaches, current state-of-the-art algorithms, and essential concepts and definitions in more detail in chapter 2.

The primary goal of this project was to design a complete verification algorithm able to verify properties of larger and more realistic neural networks than the current state-of-the-art algorithms. The main concern was significantly increasing the verification speed, so our algorithm can handle networks with tens of thousands of nodes and high-dimensional input. We used the symbolic interval propagation approach of (Wang et al., 2018a) as a foundation for our algorithm.

Our work introduces two significant novel techniques to increase the scalability of our algorithm. We employ a gradient descent based local search around spurious outputs from the LP-solver to substantially improve our algorithms ability to find valid counterexamples. Moreover, we use a new adaptive splitting strategy, aiming to always split the node with the most impact on the output first. Our new splitting strategy does especially well on the cone-shaped convolutional neural networks often used in computer vision tasks.

Furthermore, we added support for batch normalisation (Ioffe and Szegedy, 2015) and s-shaped activation functions such as the Sigmoid and Tanh. To the best of our knowledge, our approach is the first verification algorithm with iterative refinement supporting these functions. Our proposed algorithm and novel contributions are described in detail in chapter 3. Following, chapter 4 discusses some important algorithmic design choices.

We implemented our algorithm as a python toolkit, VeriNet. VeriNet utilises highly optimised libraries, such as OpenBLAS (OpenBLAS, 2019), to speed-up vectorised calculations. Furthermore, we exploit the highly parallel nature of our algorithms' refinement phase using multi-processing to achieve state-of-the-art performance. The details of VeriNet are discussed in chapter 5, while chapter 6 contains a complexity analysis focusing on computational and memory bottlenecks.

We used VeriNet to compare the performance of our proposed algorithm against present state-of-the-art verification algorithms. The experiments show that VeriNet is significantly faster than the current state-of-the-art complete verification algorithm, Neurify (Wang et al., 2018a). We usually see a speed-up between $\times 5$ and $\times 30$ for non-trivial safe cases. VeriNet also found all cases proven to be unsafe in less than one second. Neurify timed out after 3600 seconds for several of the unsafe cases, resulting in a lower bound for the speed-up of $\times 3977.5$. These experiments are described in detail in chapter 7, while chapter 8 concludes and outlines possible future work.

Chapter 2

Background

In this chapter, we introduce definitions, concepts, and algorithms important to our proposed verification algorithm. Furthermore, we have a look at current state-of-the-art verification algorithms focusing on complete methods. The reader is expected to have a working knowledge of neural networks and the necessary mathematics, primarily linear algebra.

2.1 Neural networks

Neural networks come in a wide variety of architectures and designs. This complicates the verification task, and a general verification algorithm for all network architectures is out of the scope for this project. Most state-of-the-art verification approaches are designed for feed-forward neural networks, and these networks are also the focus for our project.

A feed-forward neural network (FFNN) is made up of an input layer, an output layer, and one or more hidden layers. Each layer has one or more neurons, and each neuron has an activation function, $\sigma^i : \mathbb{R} \to \mathbb{R}$. All input values to layer i only depend on output values from previous layers, j with $j < i$. We use $\boldsymbol{z}^i$ to denote the input to layer i, $\boldsymbol{y}^i$ to denote the output, and m^i to denote the number of neurons in layer i. $\boldsymbol{z}^i$ and $\boldsymbol{y}^i$ are also referred to as the pre- and post-activation values of layer i.

Feed-forward neural networks include both fully-connected and convolutional networks and are extensively used in problems with high-input dimensions, such as computer vision tasks. The standard definition of FFNN's mentioned above also includes networks with skip-connections (He et al., 2016); however, these are not considered in this project. So, we always assume that the input to one layer is calculated as a linear combination from the output of only the previous layer. The most common layer types in FFNN's are fully-connected layers and convolutional layers.

Definition 2.1.1. ***A fully connected layer,*** *i, has a weight matrix, $W^i \in \mathbb{R}^{m^{i-1} \times m^i}$, a bias vector $\boldsymbol{b}^i \in \mathbb{R}^{m^i}$ and an activation function $\sigma^i : \mathbb{R} \to \mathbb{R}$. The output of layer i*

is $\boldsymbol{y}^i = \sigma^i(\boldsymbol{z}^i)$ with $\boldsymbol{z}^i = W^i\boldsymbol{y}^{i-1} + \boldsymbol{b}^i$ where σ^i is applied element-wise. Networks with only fully connected layers are called fully connected neural networks.

Definition 2.1.2. ***A convolutional layer****, i, has a kernel, $K^i \in \mathbb{R}^{k_1 \times k_2}$, a bias vector $\boldsymbol{b}^i \in \mathbb{R}^{m^i}$ and an activation function $\sigma^i : \mathbb{R} \to \mathbb{R}$. The output of layer i is $\boldsymbol{y}^i = \sigma(\boldsymbol{z}^i)$ with $\boldsymbol{z}^i = K^i * \boldsymbol{y}^{i-1} + \boldsymbol{b}^i$, where $*$ is the convolution operator and σ^i is applied element-wise. Networks with only convolutional layers are called fully convolutional networks.*

Each layer has an activation function used to introduce non-linearities to the network. The most common activation functions are the ReLU, Sigmoid, and Tanh functions.

$$ReLU(z) = max(0, z)$$
$$Sigmoid(z) = \frac{1}{1 + e^{-z}}$$
$$Tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

Notice that activation functions are applied element-wise to each $z_j \in \boldsymbol{z}$, we omit the indexing when this is clear from the context. Furthermore, we always assume that the output layer does not have any activation functions. This is not a huge limitation since activation functions at the output layer often only have a normalising effect. For example, a classification network using the Softmax at the output, does not change its predicted class if we remove the Softmax.

In addition to the standard layers and activation functions, our algorithm also supports batch-normalisation (Ioffe and Szegedy, 2015). These layers are used to normalise the data at different locations in the network to improve gradient flow and accelerate learning by reducing covariate shifts.

A batch normalisation layer, i, always has the same number of neurons as the previous layer, $i - 1$. During evaluation, batch normalisation acts as a linear transformation on the output of the previous layer. So $\boldsymbol{y}^i = a\boldsymbol{y}^{i-1} + b$ for constants a, b. Note that this is not true during training; the operation is highly non-linear in this phase; however, we are only interested in the behaviour during evaluation.

Several other architectures, such as RNN's, and layers, such as Dropout and Pooling, exist. While it should be possible to add support for most of these in our verification algorithm, this is out of the scope for this project.

2.2 The verification problem

In this report, we treat verification problems with concrete bounds on the input and LP-constraints on the output, formally:

Definition 2.2.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network and $\mathcal{X} = \{\boldsymbol{x'} \in \mathbb{R}^n | x^l_i \leq x'_i \leq x^u_i\}$ be a set of valid inputs for some concrete lower and upper bounds $\boldsymbol{x}^l, \boldsymbol{x}^u \in \mathbb{R}$. Given* *a set of linear constraints on the output $\psi_{\boldsymbol{y}}$, let $\mathcal{Y} = \{\boldsymbol{y} | \psi_{\boldsymbol{y}}\}$ be the set of outputs that fulfils $\psi_{\boldsymbol{y}}$. The verification problem is to determine if $\boldsymbol{x'} \in \mathcal{X} \implies f(\boldsymbol{x'}) \in \mathcal{Y}$ or find a counterexample, $\boldsymbol{x'} \in \mathcal{X}$, such that this is not true.*

So, given the constraints on the input and output, we either want to prove that no valid input fulfils the output constraints or find an input that does. If no input fulfils the output constraints, we say that the given property is "safe", otherwise it is "unsafe", and the corresponding input is a counterexample. Most complete algorithms, including (Katz(B) et al., 2017; Ehlers, 2017; Katz et al., 2019), are limited to verification problems with these types of constraints. However, a few algorithms support special cases of other constraints on the input, such as (Tjeng et al., 2018) supporting l_1, l_2 constraints and (Wang et al., 2018a) supporting l_1, l_2, brightness and contrast constraints on the input.

Another key verification problem for neural networks concerns determining the robustness of classification networks. A classification network is said to robust for an input, if small perturbations of the input do not lead to misclassification, or more formally:

Definition 2.2.2. *Let $\boldsymbol{x} \in \mathbb{R}^n$ be an input to a classification neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m > 1$ is the number of classes and $f(\boldsymbol{x}) = \boldsymbol{y}$. Let c be the correct class of $\boldsymbol{x}$ and let $C_{\boldsymbol{x}} = \{\boldsymbol{x'} \in \mathbb{R}^N | x_i^l \leq x_i' \leq x_i^u\}$ for some concrete lower and upper bounds $\boldsymbol{x}^l, \boldsymbol{x}^u$.* ***The targeted robustness verification problem*** *for the input $\boldsymbol{x}$ given a target class $t \neq c$ is to determine if $\boldsymbol{x'} \in C_{\boldsymbol{x}} \implies f(\boldsymbol{x'})_c > f(\boldsymbol{x'})_t$, or find a $\boldsymbol{x'}$ such that this is not true.*

Definition 2.2.3. *Let $\boldsymbol{x} \in \mathbb{R}^n$ be an input to a classification neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m > 1$ is the number of classes and $f(\boldsymbol{x}) = \boldsymbol{y}$. Let c be the correct class of $\boldsymbol{x}$ and let $C_{\boldsymbol{x}} = \{\boldsymbol{x'} \in \mathbb{R}^N | x_i^l \leq x_i' \leq x_i^u\}$ for some concrete lower and upper bounds $\boldsymbol{x}^l, \boldsymbol{x}^u$.* ***The general robustness verification problem*** *for the input $\boldsymbol{x}$ is to determine if $\boldsymbol{x'} \in C_{\boldsymbol{x}} \implies f(\boldsymbol{x'})_c > f(\boldsymbol{x'})_t$ for all classes $t \neq c$, or find a $\boldsymbol{x'}$ such that this is not true.*

The targeted and general robustness problems determine if changes to an input, described by $\boldsymbol{x}^l$ and $\boldsymbol{x}^u$, can lead to a change in classification to class t, or any other class, respectively. The targeted robustness problem can be expressed with linear bounds on the output and is a sub-problem of the general verification problem from definition 2.2.1. The general robustness problem, on the other hand, can not directly be encoded with linear constraints. However, this can be solved by treating the problem as a targeted problem for each output, or by allowing MILP-constraints on the output.

2.3 Algorithm types

Verification algorithms are usually categorized using the concepts of soundness and completeness. A complete algorithm, given enough time, always determines if a given property is safe or not. A sound but incomplete algorithm only returns that a property is safe if the property actually is safe; however, it may overestimate the number of unsafe cases. We are going to define soundness and completeness more formally, after developing the necessary notation, in section 3.6. We also have the traditional empirical approaches which are neither sound nor complete; these are not considered in our report.

This categorisation of verification algorithms does not always provide the full picture. Most sound but incomplete algorithms for neural network verification perform a ”one-shot approximation” by implementing a linear approximation of the non-linearities and using these approximations for verification. Complete algorithms combine this with an iterative refinement stage by splitting the networks hidden nodes, providing a better estimation for each split. For piecewise linear activation functions, this refinement can remove all overestimation in a final number of splits, resulting in completeness. However, there is also a type of verification algorithms using a refinement stage to guarantee any arbitrary overestimation > 0 in a finite number of steps, without being complete. An example of this is (Wang et al., 2018b). Our algorithm is complete for networks using only piecewise linear activation functions and implements a refinement stage for other networks.

2.4 Traditional solvers

Most state-of-the-art verification algorithms use traditional solvers as part of their approach. We are going to give a short introduction into some of the most important solvers; more information can be found in standard textbooks on the subject.

2.4.1 Linear programming (LP) solvers

LP-solvers minimise or maximise a real-valued objective function given some real-valued linear constraints. Verification algorithms utilising LP-solvers usually make a satisfiability call without an objective function to find a valid assignment to a set of variables under linear constraints. However, some methods also use an objective function for different optimisation purposes. LP-solvers can directly be used to efficiently solve verification problems for neural networks with only linear layers; however, they do not support non-linear activation functions.

2.4.2 Mixed integer linear programming (MILP) solvers

MILP solvers extend LP solvers by also supporting integer-valued variables. These integer variables can be used to encode piecewise linear activation functions, such as the ReLU. Therefore, networks with only piecewise linear activation functions can directly be encoded as MILP-systems; however, solving MILP systems with many integer constraints is computationally expensive. This limits the size of networks that can directly be verified by MILP solvers.

2.4.3 Satisfiability (SAT) solver

SAT solvers are used to find valid assignments for boolean expressions. A naïve SAT solver works by iteratively assigning values to the boolean variables and backtracking when a conflict is detected. Modern solvers extend this approach with techniques such as unit propagation and advanced conflict clauses to improve the performance.

2.4.4 Satisfiable modulo theory (SMT) solvers

SMT solvers combine SAT solvers with other theories, such as LP Solvers, to handle more complex expressions. Encoding verification problems for piecewise linear neural networks into an SMT problem is usually straightforward; however, this approach is also typically too computationally expensive for all but the smallest networks.

2.5 Important algorithms and concepts

In this section, we introduce some essential algorithms and concepts used as a foundation for our algorithm.

2.5.1 Linear relaxation

Due to the high degree of non-linearity exhibited by neural networks, complete verification algorithms often use a simpler initial approximation of the network. In Ehlers (2017), the authors introduced linear relaxations of piecewise-linear activation functions to approximate the network. A linear relaxation is a set of linear constraints on the input and output of a node, such that the valid outputs, with respect to the linear relaxations, overestimates the real output space.

Definition 2.5.1. *Let $\sigma : \mathcal{Z} \rightarrow \mathbb{R}$ be an activation function and $\psi_{z,y}$ be a set of linear constraints on the output variable, y. Furthermore, let $\mathcal{Y}_{\psi_{z,y}} = \{y \in \mathbb{R} | \psi_{z,y}, z \in \mathcal{Z}\}$ be the set of valid outputs. $\psi_{z,y}$ is a linear relaxation of σ iff $y = \sigma(z) \implies y \in \mathcal{Y}_{\psi_{z,y}}$ for all $z \in \mathcal{Z}$.*

For a ReLU function with lower and upper bounds on the input, z_l, z_u, Ehlers (2017) introduces a linear relaxation with three constraints:

$$\psi_{z,y} = \{y \geq 0, y \geq z, y \leq \frac{z_u(z - z_l)}{z_u - z_l}\}$$

These constraints are optimal in the sense that they minimise the overestimation-area in the xy-plane. This is illustrated in figure 2.1. However, to reduce the computational complexity, a relaxation with two constraints might be the better choice. The two-constraint relaxation used in (Wang et al., 2018a) is:

$$\psi_{z,y} = \{y \geq \frac{z_u z}{z_u - z_l}, y \leq \frac{z_u(z - z_l)}{z_u - z_l}\}$$

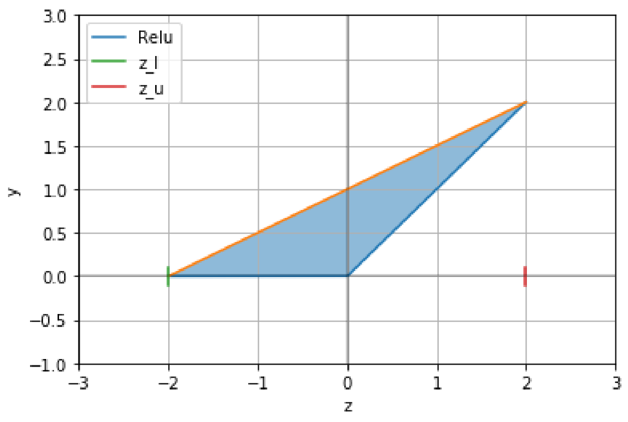


Figure 2.1: Linear relaxation of ReLU as used in (Ehlers, 2017). The shaded blue area is the relaxation, z_l and z_u are the lower and upper bounds on the input respectively.

We refer to the bounding lines of this relaxation as the upper linear relaxation, $r_u(z)$, and lower linear relaxation, $r_l(z)$. Notice that both relaxations require a lower and upper bound on the input, z_l, z_u. These bounds can be obtained through interval propagation, covered later in this chapter.

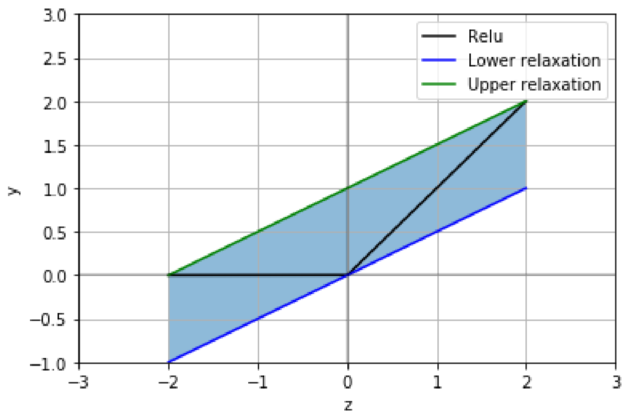


Figure 2.2: The linear relaxation of the ReLU used in (Wang et al., 2018a)

By defining linear relaxations for all non-linearities in a network, we get a linear approximation of the network. Most sound algorithms use some sort of relaxations to infer properties of the behaviour of the actual network. Complete algorithms also use relaxations; however, this is usually combined with a refinement phase, which iteratively decreases the overestimation introduced by the relaxations.

2.5.2 Interval propagation

In this section, we introduce interval arithmetic and explain how it can be used to calculate bounds for the nodes in a neural network.

Naïve interval propagation

Naïve interval propagation uses interval arithmetic to propagate concrete bounds through a network. The calculated bounds at the output layer can then be used for verification purposes. The input bounds, $\boldsymbol{z}_l^i$, $\boldsymbol{z}_u^i$, to a layer i are calculated from the output bounds, $\boldsymbol{y}_l^{i-1}$, $\boldsymbol{y}_u^{i-1}$, of the previous layer as:

$$\boldsymbol{z}_l^i = W^{i+}\boldsymbol{y}_l^{i-1} + W^{i-}\boldsymbol{y}_u^{i-1} + \boldsymbol{b}^i$$
$$\boldsymbol{z}_u^i = W^{i-}\boldsymbol{y}_l^{i-1} + W^{i+}\boldsymbol{y}_u^{i-1} + \boldsymbol{b}^i$$

Where W^{i+} is the matrix with all positive elements from the weight matrix W^i, and all other elements set to 0, more formally:

$$W_{k,h}^{i+} = \begin{cases} W_{k,h}^i & W_{k,h}^i > 0 \\ 0 & else \end{cases}$$

Analogously W^{i-} is the matrix with the negative elements of W^i. The input bounds to layer i are then propagated through the activation functions, $\sigma^i : \mathbb{R} \rightarrow \mathbb{R}$, to get the output bounds:

$$\boldsymbol{y}_l^i = \sigma^i(\boldsymbol{z}_l^i)$$
$$\boldsymbol{y}_u^i = \sigma^i(\boldsymbol{z}_u^i)$$

This approach can be visualised as propagating the bounds in the same manner as in a standard network forward phase, except that we switch the lower and upper bound each time we multiply by a negative weight. The process is repeated for all layers until we reach the output layer.

Example 2.5.1. *Let our network be as in figure 2.3 with ReLU activations for all hidden nodes and the input node bounded by* $[-1, 1]$*. The input bounds to the first layer are still* $[-1, 1]$ *since the weight is* 1*. Next, we propagate these bounds through the ReLU, zeroing out the negative lower bound. The input bounds to* h_{21} *are the same as the output bounds from* h_{11}*. However, for* h_{22} *we have to multiply the bounds by the weight,* -1*, and switch the lower and upper bound since it is negative. Finally, we propagate these bounds through the ReLU's in the second layer, multiply by the weights and sum the resulting bounds to get bounds on the output.*

The naïve interval propagation has one major drawback. The nodes of a neural network usually cannot all reach their minima or maxima at the same time, and these conditional dependencies are not accounted for. This leads to overestimating intervals, and the problem increases with more complex networks.

Symbolic interval propagation

To reduce the overestimation of naïve interval propagation, (Wang et al., 2018b) introduced symbolic interval propagation for neural network verification. This symbolic approach propagates linear equations instead of concrete bounds, as illustrated

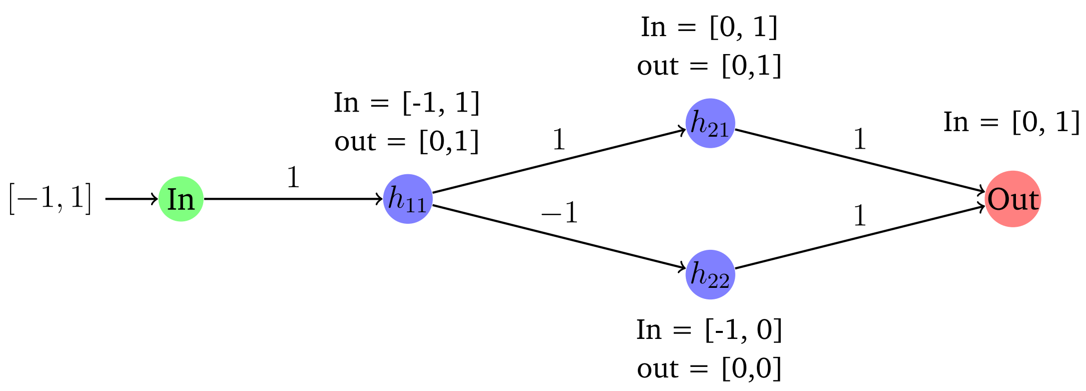


Figure 2.3: Naïve interval propagation a ReLU network

in figure 2.4. These equations can then be minimised and maximised to determine the concrete lower and upper bounds, respectively. The symbolic lower and upper input bounds of a fully connected layer i, $eq^i_{low,in}(\boldsymbol{x})$ and $eq^i_{up,in}(\boldsymbol{x})$, are calculated from the symbolic output bounds of layer $i-1$, $eq^{i-1}_{low,out}(\boldsymbol{x})$ and $eq^{i-1}_{up,out}(\boldsymbol{x})$, using the formula:

$$\begin{aligned} eq^i_{low,in}(\boldsymbol{x}) &= W^{i+}eq^{i-1}_{low,out}(\boldsymbol{x}) + W^{i-}eq^{i-1}_{up,out}(\boldsymbol{x}) \\ eq^i_{up,in}(\boldsymbol{x}) &= W^{i-}eq^{i-1}_{low,out}(\boldsymbol{x}) + W^{i+}eq^{i-1}_{up,out}(\boldsymbol{x}) \end{aligned}$$

Where W^{i+} and W^{i-} are the matrices with positive and negative weights respectively, as in naïve interval propagation.

Compared to naïve interval arithmetic, there is one challenge. Propagating the symbolic bounds through the activation functions results in non-linear bounds. This is solved in (Wang et al., 2018a) by defining a linear relaxation with two constraints, as explained in section 2.5.1. Instead of propagating the symbolic bounds through the activation function, the upper bound is propagated through the upper linear relaxation, and the lower bound through the lower linear relaxation.

The concrete input bounds for the linear relaxation, z_l, z_u, are calculated from the symbolic input bounds. For a linear equation $eq(\boldsymbol{x}) = \sum_i a_i x_i$ where each x_i is bounded by $x^l_i \leq x_i \leq x^u_i$ the maxima and minima can be calculated as:

$$\begin{aligned} z_l = \min(eq(\boldsymbol{x})) &= \sum_{i|a_i>0} a_i x^l_i + \sum_{i|a_i<0} a_i x^u_i \\ z_u = \max(eq(\boldsymbol{x})) &= \sum_{i|a_i>0} a_i x^u_i + \sum_{i|a_i<0} a_i x^l_i \end{aligned}$$

Since two equations bound the input to each node, one lower and one upper symbolic bound, this is done twice. The concrete bounds from the lower symbolic bounds are used to calculate the lower linear relaxation, r_l, and the concrete bounds from the upper symbolic bound are used to calculate the upper linear relaxation, r_u.

Finally, we propagate the symbolic equations through the linear relaxations to calculate the output equations of node k in layer i:

$$eq^i_{low,out}(\boldsymbol{x})_k = r^i_{l,k}(eq^i_{low,in}(\boldsymbol{x})_k)$$
$$eq^i_{up,out}(\boldsymbol{x})_k = r^i_{u,k}(eq^i_{up,in}(\boldsymbol{x})_k)$$

Example 2.5.2. *Let our network be as illustrated in figure 2.4 with the input $x \in [-1, 1]$ and ReLU activations for all hidden nodes. Since it should be clear from the context, we will skip the layer and node indexing, k and i, for the equations in this example. The lower and upper input equation to the first node, h_{11}, are x, since the weight is 1. The concrete lower and upper bounds are still -1 and 1, and the corresponding linear relaxations are $r_l(z) = 1/2z$ and $r_u(z) = 1/2z + 1/2$ respectively. The resulting output equations for h_{11} are:*

$$eq_{low,out}(x) = r_l(eq_{low,out}(x))) = 1/2x$$
$$eq_{up,out}(x) = r_u(eq_{up,out}(x))) = 1/2x + 1/2$$

The next step is to calculate the input equations for the second hidden layer. For h_{21}, the weight is 1, and the input equations are the same as the output of h_{11}. Node h_{22} has a weight of -1, so we have to multiply the lower and upper output equations of h_{11} with -1 and switch the lower and upper bound since the weight is negative.

The input equations to h_{21} are then used to calculate the corresponding concrete bounds. Minimising and maximising $eq_{low,in}(x) = 1/2x$ results in the concrete bounds for the lower equation, $[z_{ll}, z_{lu}] = [-1/2, 1/2]$. The corresponding linear relaxations are $r_l(z) = 1/2z$. The same calculations for the upper linear relaxation results in $[z_{ul}, z_{uu}] = [-1/2, 1]$ and $r_u(z) = 2/3z + 1/3$. The output equations are:

$$eq_{low,out}(x) = r_l(eq_{low,in}(x))) = 1/2(1/2x) = 1/4x$$
$$eq_{up,out}(x) = r_u(eq_{up,in}(x))) = 2/3(1/2x + 1/2) + 1/3 = 1/3x + 2/3$$

For the h_{22} we get the linear relaxations $r_l(z)) = 1/3z$ and $r_u(z) = 1/2z + 1/2$ and output equations $eq_{low,out}(x) = -1/6x - 1/6$ and $eq_{up,out}(x) = -1/4x + 1/2$

Finally, we multiply the output equations of the second hidden layer by their weights and add them to get the equations at the output layer. The resulting concrete lower and upper bound at the output layer, $-1/4$ and $5/4$, are calculated by minimising and maximising the lower and upper equation respectively.

Notice that the output bounds of the previous example are actually worse than the output bounds from the naïve interval propagation. This can be understood by realizing that the naïve method is a special case of symbolic interval propagation with constant upper and lower relaxations (Wang et al., 2018a). As we see illustrated in figure 2.5, the overestimation area of the symbolic linear relaxation is smaller

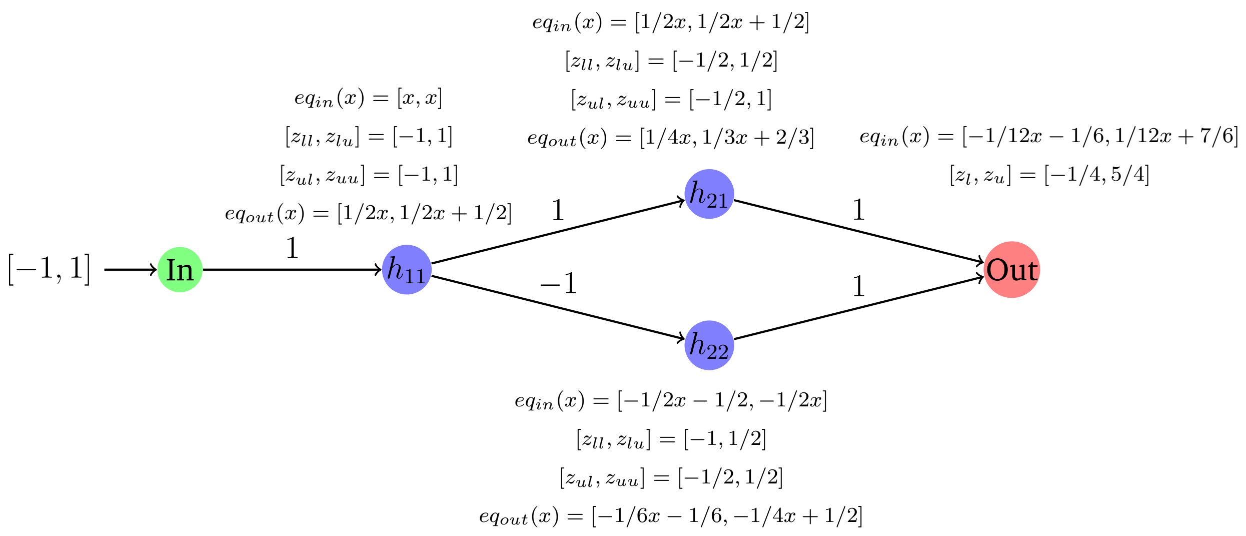

$eq_{in}(x) = [1/2x, 1/2x + 1/2]$
$[z_{ll}, z_{lu}] = [-1/2, 1/2]$
$[z_{ul}, z_{uu}] = [-1/2, 1]$
$eq_{out}(x) = [1/4x, 1/3x + 2/3]$
$eq_{in}(x) = [x, x]$
$[z_{ll}, z_{lu}] = [-1, 1]$
$[z_{ul}, z_{uu}] = [-1, 1]$
$eq_{out}(x) = [1/2x, 1/2x + 1/2]$
$eq_{in}(x) = [-1/12x - 1/6, 1/12x + 7/6]$
$[z_l, z_u] = [-1/4, 5/4]$
$[-1, 1]$
In
1
h_{11}
1
-1
h_{21}
h_{22}
1
1
Out
$eq_{in}(x) = [-1/2x - 1/2, -1/2x]$
$[z_{ll}, z_{lu}] = [-1, 1/2]$
$[z_{ul}, z_{uu}] = [-1/2, 1/2]$
$eq_{out}(x) = [-1/6x - 1/6, -1/4x + 1/2]$

Figure 2.4: Symbolic interval propagation for a ReLU network

than for the naïve relaxation; yet, the symbolic relaxation is not contained in the naïve relaxation. For this reason, there is no guarantee that the symbolic interval propagation results in better bounds than the naïve method. However, in practical applications, symbolic interval propagation usually has significantly better results for all but the smallest networks.

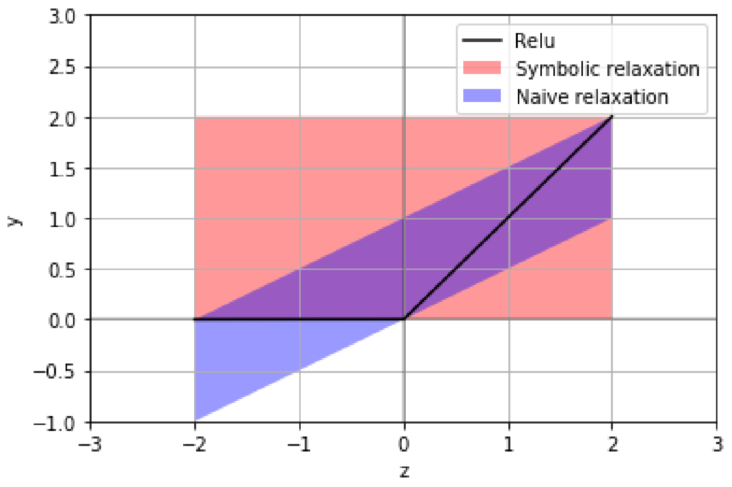


Figure 2.5: The relaxation used for the naïve proapagation(red) and symbolic propagation (blue). (Wang et al., 2018a)

Even though symbolic interval propagation usually is a considerable improvement over the naïve method, it does not entirely solve the conditional dependency issue. Calculating the bounds of one node might implicitly use both the lower and upper relaxation of a preceding node at the same time. The next section introduces a method to reduce this problem of untracked conditional dependencies.

Error-based symbolic interval propagation

To address the challenges of symbolic interval propagation, the latest versions of ReluVal and Neurify, implementations of Wang et al. (2018b) and Wang et al. (2018a), use a slight variation of the standard symbolic interval propagation.

The error-based symbolic interval propagation only propagates one equation, instead of one for the lower and one for the upper bound. This equation is always propagated through the lower linear relaxation. The resulting error from not using the upper linear relaxations are calculated as concrete values and propagated together with the equations.

At the input to each layer, i, the errors are represented by a matrix $E^i_{in} \in \mathbb{R}^{m_i \times m'_i}$ where m_i is the number of nodes in layer i, and m'_i is the total number of nodes in all previous layers. An element $(E^i_{in})_{k,h}$ represents how much the input equation for node k in layer i would change if the equation had been propagated through the upper bound of node h instead of the lower. If $eq^i_{in}(x)$ are the input equations to layer i, the concrete lower and upper bounds to node k, $z^i_{l,k}$ and $z^i_{u,k}$, are calculated as:

$$z^i_{l,k} = \min(eq^i_{in}(x)_k) + \sum_{h|\ (E^i_{in})_{k,h}<0} (E^i_{in})_{k,h}$$
$$z^i_{u,k} = \max(eq^i_{in}(x)_k) + \sum_{h|\ (E^i_{in})_{k,h}>0} (E^i_{in})_{k,h}$$

These concrete bounds, $z^i_{l,k}$ and $z^i_{u,k}$, are used to calculate the nodes lower and upper linear relaxations, $r^i_{l,k}(z)$ and $r^i_{u,k}(z)$. Both the error matrix and the equations are propagated through the lower linear relaxation:

$$(\hat{E}^i_{out})_{k,:} = r^i_{l,k}((E^{i+1}_{in})_{k,:})$$
$$eq^i_{out}(x)_k = r^i_{l,k}(eq^i_{in}(x)_k)$$

The new error introduced from only using the lower relaxation at node k is:

$$\epsilon^i_k = \max_{z^i \in [z_{l,k}, z^i_{u,k}]} (r^i_{u,k}(z) - r^i_{l,k}(z))$$

The resulting output error matrix is the concatenation of the propagated errors and the new errors, $E^i_{out} = [(\hat{E}^i_{out}), diag(\boldsymbol{\epsilon}^i)]$. Finally, the input errors and equations to the next layer are calculated by propagating them through the affine layer:

$$E^i_{in} = W^i E^i_{out}$$
$$eq^i_{in}(x) = W^i eq^i_{out}(x) + \boldsymbol{b}$$

Since each nodes error is tracked separately, we always know if previous nodes had to operate at the lower or upper relaxation. This is not the case for the standard symbolic interval propagation from the previous section, and nodes might implicitly have to operate at both relaxations simultaneously to achieve the calculated bounds.

Algorithm 1 Error based symbolic interval propagation

```
//eqMin(X), and eqMax(X) are functions calculating the min/max values of linear
equations with coefficients stored in X. The variables of the equations are assumed
to be bounded.

$m \leftarrow inputSize$
$X \leftarrow [eye(m), \mathbf{0}]$ // The equation coefficient matrix
$E \leftarrow matrix(m, 0)$ // The error matrix
for i in layers do

    //Calculate input equations to layer i
    $X \leftarrow W_i \times X$
    $X[:, -1] \leftarrow X[:, -1] + \boldsymbol{b}_i$
    $E \leftarrow W_i \times X$

    //Calculate concrete bounds and relaxations
    $\boldsymbol{lowBound} \leftarrow eqMin(X) + sum(E < 0, axis = 1)$
    $\boldsymbol{upBound} \leftarrow eqMax(X) + sum(E > 0, axis = 1)$
    $\boldsymbol{a}_l, \boldsymbol{b}_l, \boldsymbol{a}_u, \boldsymbol{b}_u \leftarrow relaxations(\boldsymbol{lowBound}, \boldsymbol{upBound})$

    //Calculate output equations for layer i
    $X \leftarrow X \times \boldsymbol{a}_l$ //Rows of X times rows (1 element) of $\boldsymbol{a}_l$
    $X[:, -1] \leftarrow X[:, -1] + \boldsymbol{b}_l$
    $ENew \leftarrow diag(\boldsymbol{b}_u - \boldsymbol{b}_l)$ // Assuming $\boldsymbol{a}_l = \boldsymbol{a}_u$
    $E \leftarrow concatenate(E, ENew)$
end for
```

Example 2.5.3. *Let our network be as illustrated in figure 2.6 with the input $x \in [-1, 1]$ and ReLU activations for all hidden nodes. Since it should be clear from the context, we will skip the layer and node indexing, k and i, for the equations in this example. The input to the first node, h_{11} is the equation x, since the weight is 1. The corresponding lower and upper linear relaxations are $r_l(z) = 0.5z$ and $r_u(z) = 0.5z + 0.5$. Propagating the equation through the lower linear relaxation results in the output equation $eq_{out}(x) = 1/2x$, and the resulting error is $\epsilon^{11}_{out} = r_u(z) - r_l(z) = 1/2$.*

The weight for node h_{21} is 1 so the input equation and error are the same as the output of node h_{11}. Minimising $eq_{in}(x) = 1/2x$ results in $z_l = -1/2$. Note that the error, ϵ^{11}_{out} is positive, so it does not affect the lower bound. Maximising the equation results in $1/2$, however this time we have to add the positive error, so $z_u = 1/2+1/2 = 1$. The resulting lower and upper linear relaxations are $r_l(z) = 2/3z$ and $r_u(z) = 2/3z + 1/3$. The error from this node is $\epsilon^{21}_{out} = r_u(z) - r_l(z) = 1/3$. Finally, we have to propagate the equation and error from h_{11} through the linear relaxation to get $eq_{out}(x) = r_l(eq_{in}(x)) = 1/3x$

and $\epsilon^{11}_{out} = r_l(\epsilon^{11}_{in}) = 1/3$.

For node h_{22} *the input error is* $\epsilon^{11}_{in} = -0.5$*, since the weight is* -1*. Since the error is negative we have to add it to the lower concrete input bound and we get* $z_l, z_u = -1, 1/2$*. The same calculations as before result in* $eq_{out}(x) = -1/6x$*,* $\epsilon^{11}_{out} = -1/6$*, and* $\epsilon^{22}_{out} = 1/3$*.*

Finally, at the output layer, both weights are 1 *so the resulting input errors are* $\epsilon^{11}_{in} = 1/3 - 1/6$*,* $\epsilon^{21}_{in} = 1/3$*, and* $\epsilon^{22}_{in} = 1/3$*. The input equation is* $eq_{in}(x) = 1/3x - 1/6x = 1/6x$*. Since all errors are positive, the lower concrete input bound is the minimum of the equation,* $z_l = -1/6$*. The upper concrete input bound is the maximum of the equation plus all positive errors* $z_u = 1/6 + 1/6 + 1/3 + 1/3 = 1$

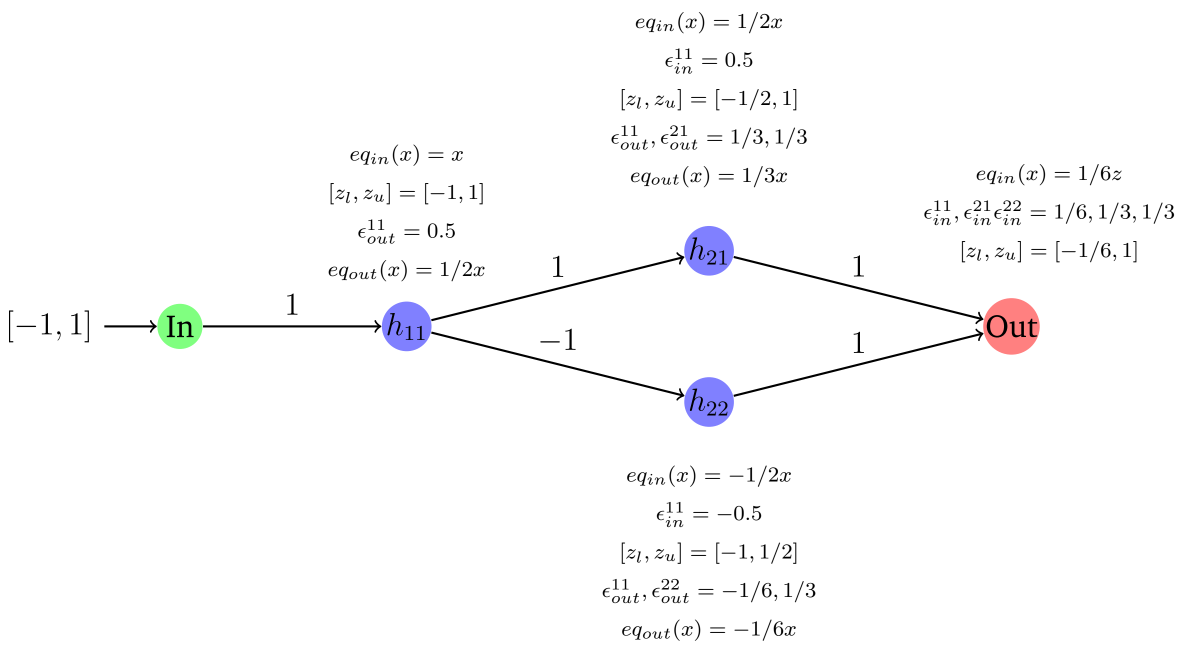


Figure 2.6: Error-based interval propagation for a ReLU network

The calculations for the output layer in the previous example is were we can see the real advantage of the error-based symbolic interval propagation. Notice that if we use the upper relaxation for node h_{11}, the value of h_{21} increases (positive error) and h_{22} decreases (negative error). However, these effects do to some degree cancel out at the output layer, as the errors are summed, and the final error e^{11}_{in} is positive. This positive error tells us that using the upper linear relaxation for node h_{11} increase the value at the output layer.

In contrast, the standard symbolic interval propagation from the previous chapter does not provide us with this information. Actually, in example 2.5.2 we used the lower relaxation of h_{11} to calculate the upper bound of h_{22} and the upper relaxation of h_{11} to calculate the upper bound h_{21}. Finally, we used the upper relaxations of both h_{21} and h_{22} to calculate the upper bound of the output node. So the calculated upper bound of the output node requires h_{11} to operate at its lower and upper relaxation simultaneously, which clearly is not possible and a conditional dependency issue.

In summary, the error-based symbolic interval propagation reduces the conditional dependency issue compared to standard symbolic interval propagation by tracking an individual error for each node. This comes at the cost of always using parallel lower and upper relaxations, which might result in worse relaxations than the standard symbolic interval propagation. However, in practice, we usually see significantly better bounds with the error-based interval propagation.

2.5.3 Abstract interpretation

Abstract interpretation is a technique used in several sound but incomplete verification algorithms (Gehr et al., 2018; Singh et al., 2018, 2019). The key idea is to represent the network vectors in an abstract domain, and each network operation as an abstract transformer. The abstract transformers are sound, so if the input to an abstract transformer overestimates the input to its corresponding operation in the network, the output of the abstract transformer overestimates the output of the network operation.

Using the notation from (Gehr et al., 2018), the abstract domain, A^m, encodes shapes as logical formulas to describe sets of vectors from the network domain, $\mathcal{P}(\mathbb{R}^m)$. An abstraction function, $\alpha : \mathcal{P}(\mathbb{R}^m) \rightarrow A^m$, is used to convert sets of vectors from the network domain to the abstract domain. Analogously, a concretisation function $\gamma : A^m \rightarrow \mathcal{P}(\mathbb{R}^m)$ converts abstract elements back to the network domain. The abstraction and concretisation functions are sound, so for all $\mathcal{X} \in \mathcal{P}(\mathbb{R}^m)$, we have $\mathcal{X} \subseteq \gamma(\alpha(\mathcal{X}))$. Finally, sound abstract transformers, $T_{L_i} : A^m \rightarrow A^n$, are defined for the corresponding layers, $L_i : \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n)$, in the network.

Given a set of valid inputs, $\mathcal{X}$, the abstraction function α is used to calculate the corresponding abstract element $\alpha(\mathcal{X})$. The abstract set is propagated through the abstract transformers, calculating the abstract output $a_{\bar{\mathcal{Y}}}$. Since all operations are sound, the final set $\bar{\mathcal{Y}} = \gamma(a_{\bar{\mathcal{Y}}}))$ is a sound approximation of the true output set $\mathcal{Y}$ and can be used for verification purposes. This is illustrated in figure 2.7.

Notice that the interval propagation from the previous section can be considered in an abstract interpretation framework. The symbolic intervals correspond to the abstract domain, while the linear relaxations correspond to the abstract transformers. We explore the fact that some of the abstract interpretation based verifiers are very similar to the symbolic interval propagation-based verifiers in more detail in section 2.6.4.

Example 2.5.4. *Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ be a very simple network, with only the input layer, output layer and both weights equal to $1/2$. Let the input set be $\mathcal{X} = \{(0,1),(1,2)\}$. Since we only have two points, we can calculate the networks true output*

$\mathcal{Y} = \{(0, 1/2), (1/2, 1)\}$.

For this simple problem, we will use the box-domain where each coordinate in the abstract domain is represented by a upper and lower bound. So an abstract element $a \in A^m$ is represented by $a = \{x_1^l \leq x_1 \leq x_1^u, ..., x_m^l \leq x_m \leq x_m^u\}$, where x_l^i and x_u^i are concrete lower and upper bounds for element i. We use the abstraction function $\alpha(\mathcal{X}) = \{x_1^l \leq x_1 \leq x_1^u, ..., x_n^l \leq x_n \leq x_n^u\}$, where x_i^l and x_i^u are the minimum and maximum i'th element of the vectors in the input set $\mathcal{X}$. For an abstract element $a \in A^m$, the concretisation function calculates the set of valid vectors, $\gamma(a) = \{(x^1, ..., x^n) | x_l^i \leq x^i \leq x_u^i \forall i \in \{1, 2...m\}\}$. Finally, the abstract transformer used for the multiplication $f(x) = cx$ is given by $T_f(a) = \{cx_1^l \leq x_1 \leq cx_1^u, ..., cx_m^l \leq x_m \leq cx_m^u\}$.

With this framework we have:

$$\begin{aligned}
\alpha(\mathcal{X}) &= \{0 \leq x^1 \leq 1, 1 \leq x^i \leq 2\} \\
T_f(\alpha(\mathcal{X})) &= \{0 \leq x^1 \leq 1/2, 1/2 \leq x^i \leq 1\} \\
\gamma(T_f(\alpha(\mathcal{X}))) &= \{(x^1, x^2) | 0 \leq x^1 \leq 1/2, 1/2 \leq x^2 \leq 1\}
\end{aligned}$$

We see that $\mathcal{Y} \subset \gamma(T_f(\alpha(\mathcal{X})))$, so the output of the abstract transformer is sound as expected.

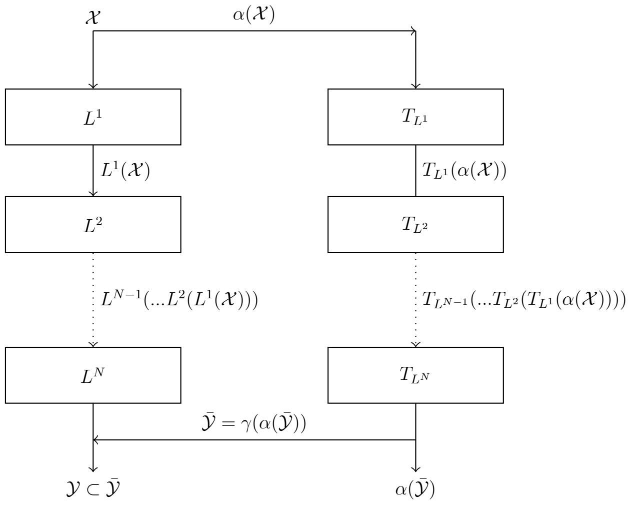

$\mathcal{X}$
$\alpha(\mathcal{X})$
L^1
T_{L^1}
$L^1(\mathcal{X})$
$T_{L^1}(\alpha(\mathcal{X}))$
L^2
T_{L^2}
$L^{N-1}(...L^2(L^1(\mathcal{X})))$
$T_{L^{N-1}}(...T_{L^2}(T_{L^1}(\alpha(\mathcal{X}))))$
L^N
T_{L^N}
$\bar{\mathcal{Y}} = \gamma(\alpha(\bar{\mathcal{Y}}))$
$\mathcal{Y} \subset \bar{\mathcal{Y}}$
$\alpha(\bar{\mathcal{Y}})$

Figure 2.7: Illustration of abstract interpretation for network verification. The left path illustrates the input set $\mathcal{X}$ propagated through the layers of the network, while the right path illustrates the abstracted input sent through the corresponding abstract transformers. The final output-set of the abstract transformers overestimates the corresponding output-set of the network.

2.5.4 Adversarial examples and attacks

As already mentioned, neural networks are susceptible to adversarial examples (Szegedy et al., 2013). An adversarial example is a small perturbation to an input leading to a significant output change. For image-classification networks, this is usually defined as a perturbation, small enough to be unnoticeable by humans, leading to misclassification. These adversarial examples can be obtained through methods known as adversarial attacks. Adversarial attacks are usually gradient descent based with a loss function designed to maximise the output change. An adversarial example obtained by our algorithm is illustrated in figure 2.8.

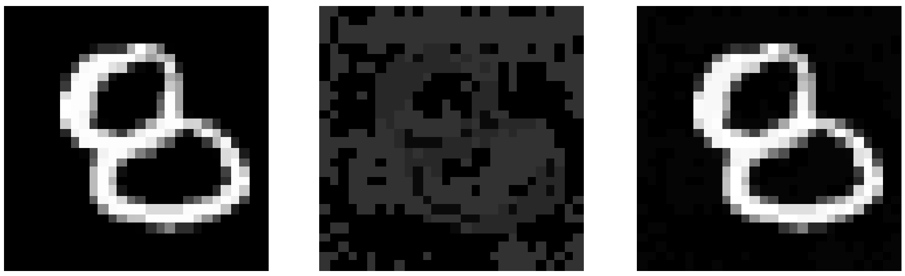


Figure 2.8: Adversarial example found by our algorithm. The image on the left is classified as the digit 8. After adding the noise in the middle (scaled x 10 to make it visible), the image is classified as a 6.

A significant amount of research has been done on adversarial attacks and a detailed review is out of the scope for this report. However, we are going to introduce a simple gradient-based adversarial attack, as this is essential for our verification algorithm. The gradient descent is very similar to the back-propagation phase in a neural network, but instead of updating the network parameters, it updates the input.

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a neural network with $f(\boldsymbol{x}) = \boldsymbol{y}$. Let y_i indicate the confidence for class i. Assume that the network classifies $\boldsymbol{x}$ as class k, meaning $y_k > y_j$ for all other classes j. To find a counterexample, $\boldsymbol{x'}$ a gradient descent is performed on $\boldsymbol{x}$. Let l be our target class, so the goal is to find $\boldsymbol{x'}$ such that $f(\boldsymbol{x'})$ results in $y_l > y_k$. With $\boldsymbol{x}_0 = \boldsymbol{x}$, $\boldsymbol{x'} = \boldsymbol{x_n}$, and loss function $L(\boldsymbol{x}) = y_k - y_l$, the naïve gradient descent update is:

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} - \gamma \times \frac{d}{d\boldsymbol{x}} L(\boldsymbol{x}),\ i = \{0, 1..., n\}$$

This naïve algorithm does work; however, it tends to adjust some pixels a lot more than others, creating noise noticeable by humans. This can be solved by clipping the resulting $\boldsymbol{x_i}$ vector to predetermined bounds after each gradient update. Several other more advanced attacks exist, however, we do not cover them in this report.

2.6 State-of-the-art verification algorithms

In this section, we introduce some state-of-the-art verification algorithms. In particular, we are going to focus on algorithms using symbolic interval propagation since our algorithm builds on these approaches. However, for completeness, we are also going to mention a few other approaches.

2.6.1 Symbolic interval propagation based algorithms

The algorithm from (Wang et al., 2018b) was, to our knowledge, the first algorithm using symbolic interval propagation for neural network verification. It uses the standard symbolic interval propagation from section 2.5.2, except that it does not use linear relaxation for the activation functions. Instead, the algorithm only propagates equations through the ReLU nodes operating in the linear area. For nodes operating in the non-linear area, it calculates concrete minima and maxima values from the equations and propagates those values instead, similar to the naïve interval propagation. The refinement phase is implemented by splitting the input nodes, as illustrated in figure 2.9. The algorithm uses an interval-based gradient analysis to determine which input node to split.

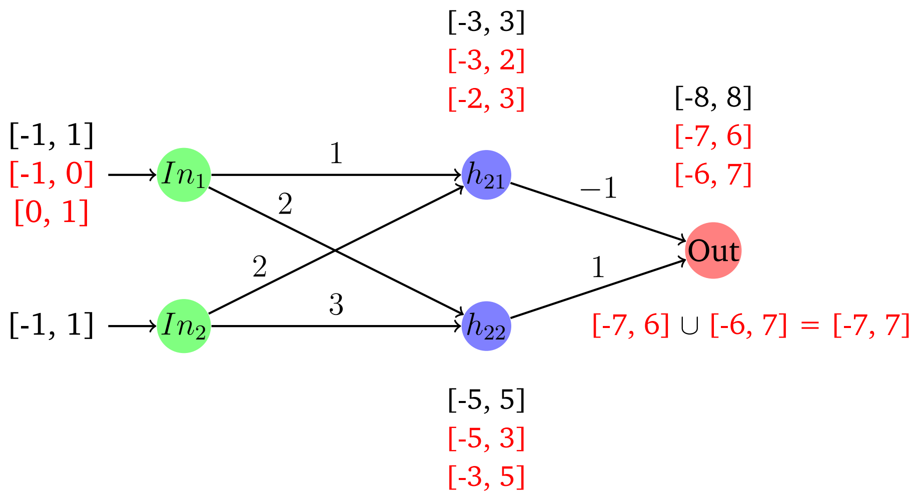


Figure 2.9: Example of refinement through input bisection with naïve interval propagation. The black intervals are done without bisection, while the red intervals are the results from bisecting In_1. The intersection of the bisection intervals at the output is tighter than without bisection.

This approach has some challenges compared to other complete verification algorithms. First of all, since the splitting is done in the input domain and not the ReLU nodes, there is no guarantee that the overestimation ever reaches 0 and the algorithm is technically not complete. However, the authors prove this strategy reaches an arbitrary overestimation > 0 in a finite number of splits. Secondly, propagating concrete values instead of equations through ReLU nodes results in lost information about the conditional dependencies. Finally, the algorithm searches for counterexamples by picking random values from the input space and checking if they are valid counterexamples, this can take a very long time.

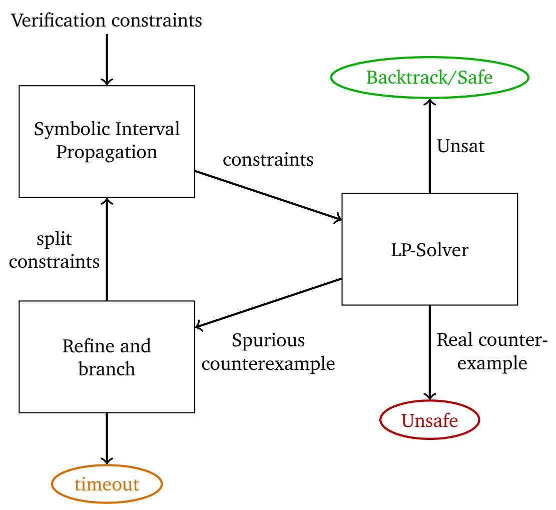


Figure 2.10: The pipeline from (Wang et al., 2018a)

The previous approach is improved upon in (Wang et al., 2018a), which is also closer to our proposed algorithm described in the next chapter. This method starts with symbolic interval propagation to calculate bounds on the output nodes. These bounds, together with the constraints from the verification problem, are then used as constraints in a satisfiability call to the LP-solver. If the LP-solver returns UNSAT, no counterexample exists, and the branch is safe. If the LP-solver returns a potential counterexample, and it is valid, the property is unsafe. However, if it is a spurious counterexample, the algorithm starts a refinement stage by splitting the input to a ReLU node at 0 and branching. So, one branch explores the case that the input to the ReLU node is larger than 0, and the other that the input is smaller than 0. The splitting is implemented by adding the relevant split-constraints to the symbolic interval propagation and the LP-solver. The process repeats until the property is proven to be safe, unsafe, or a time-out criterion is reached. The pipeline is illustrated in figure 2.10.

This algorithm uses linear relaxations in the symbolic interval propagation phase, which should result in reduced conditional dependency issues compared to the first approach. Furthermore, Splitting ReLU nodes ensures that all overestimation is removed in a finite number of branches, which makes the algorithm complete.

Both of the algorithms mentioned above originally used the standard symbolic interval propagation. However, the latest implementations (Wang et al., 2019b,a) have switched to the error-based symbolic interval propagation described in section 2.5.2.

2.6.2 SMT-based algorithms

SMT-based complete verification algorithms (Ehlers, 2017; Katz(B) et al., 2017; Katz et al., 2019) use LP-solvers to evaluate the satisfiability of relaxed versions of the verification problems. If no conclusion can be drawn from the relaxed problems, they refine the problem by splitting nodes with piecewise linear activation functions using an SAT-solver.

The first approach (Ehlers, 2017) relaxes ReLU nodes with the three-constraint linear relaxation from section 2.5.1. The algorithm also supports the piecewise linear max-pooling layer using a similar relaxation. These relaxations are used to encode the problem as an LP-system, and all possible splits resulting in the nodes operating in linear areas are encoded as a boolean formula in the SAT-solver.

The method starts by linearly relaxing all ReLU and max-pooling nodes operating in the non-linear area. The algorithm iteratively splits the nodes and branches. For each branch, there are two possible scenarios. If the resulting constraints are determined to be unsatisfiable by the LP-solver, the algorithm backtracks to another branch in the SAT-solver. If no more branches exist, there is no valid assignment to the variables, and the verification-property is proven safe. The second scenario is that the resulting system is satisfiable, and the algorithm continues the splitting on a new node. If a valid assignment is found after splitting all nodes, there is no more overestimation, so the assignment is a valid counterexample, and the property is unsafe.

This algorithm is combined with several steps to guide the search. Each time a branch is determined to be unsatisfiable, the algorithm adds a conflict clause to the SAT-solver, ruling out as many branches as possible with elastic filtering (Chinneck and Dravnieks, 1991). The approach also utilises the objective function of the LP-solver to force the outputs of the linear relaxations towards the true ReLU function, proving satisfiability for several branches at once. Furthermore, naive interval propagation is used to continuously improve the lower and upper input bounds to each node, reducing overestimation from the relaxations.

The second algorithm, (Katz(B) et al., 2017), extends the Simplex algorithm for solving linear programs to support non-linear ReLU constraints. The idea is to describe each ReLU node using two variables, the forward and backward facing variable, v_b and v_f, as illustrated in figure 2.11. While running the Simplex algorithm, Reluplex treats these two variables as independent variables. After Simplex finds a valid assignment to the linear system, the algorithm enforces the ReLU constraints, $v_f = max(0, v_b)$. This can lead to an invalid assignment for the Simplex equations, in which case the process is repeated. If a ReLU constraint is enforced too many times, the algorithm branches by splitting the node. If a valid assignment is found, it is a counterexample, and the property is unsafe. Otherwise, if the simplex system or the ReLU constraints are unsatisfiable, the property is proven safe. Similar to (Ehlers, 2017), this algorithm also continuously refines the lower and upper input bounds to each node.

Finally, (Katz et al., 2019) is a extension of (Katz(B) et al., 2017). Most notably, they add a divide and conquer step, allowing the algorithm to also split on input nodes,

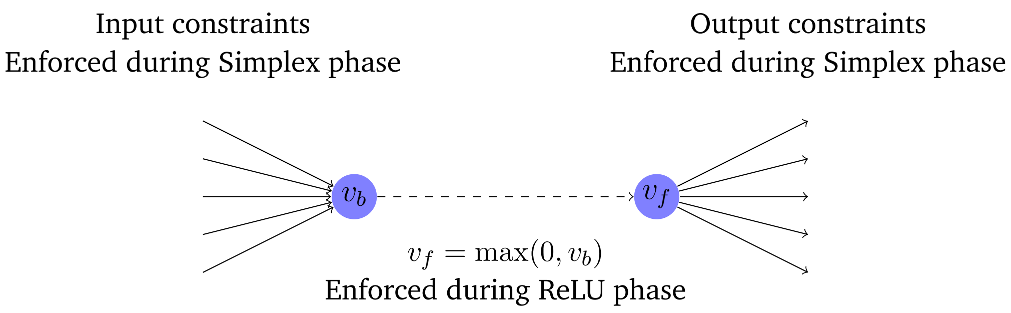

Input constraints
Enforced during Simplex phase
Output constraints
Enforced during Simplex phase
v_b
v_f
$v_f = \max(0, v_b)$
Enforced during ReLU phase

Figure 2.11: Illustration of the ReLU node split into forward- and backward-facing variable used in (Katz(B) et al., 2017)

and they add a symbolic interval propagation step used to calculate tighter bounds improving the performance of the Simplex step.

2.6.3 MILP-based algorithms

(Tjeng et al., 2018) uses a MILP solver to solve verification problems for neural networks. Piecewise linear functions can be directly encoded as MILP constraints using real-valued and binary variables. The paper provides MILP encodings for ReLU functions and max-pooling layers. The 5 constraints used to encode a ReLU node are:

$$(\sigma(z) \geq 0), (\sigma(z) > z), (\sigma(z) \leq a \times u), (\sigma(z) \leq z - l(1-a)), (a \in \{0, 1\})$$

Where, z is the input to the node, $\sigma(z)$ is the output, l, u are the lower and upper bounds on the input, and a is a binary variable used to encode the non-linear part of the ReLU. The algorithm uses either naïve interval arithmetic or linear programming, maximising and minimising each nodes inputs layer-wise to find the lower and upper bounds on the input nodes.

2.6.4 Sound but incomplete algorithms

Since our algorithm is complete for networks using only piecewise linear functions, we primarily focus on complete algorithms in this report. However, our algorithm also supports s-shaped activation functions such as the Sigmoid, at the cost of completeness. So in this section, we will give an introduction to some state-of-the-art sound but incomplete algorithms supporting s-shaped activation functions.

The algorithm from (Singh et al., 2018) uses an abstract interpretation based approach for sound verification of networks using ReLU, Sigmoid, or Tanh activation functions. The algorithm uses the Zonotope abstractions, where each variable x_j of a vector $\boldsymbol{x}$ is represented by: [1]

[1] Note that the actual algorithm in (Singh et al., 2018) uses intervals instead of concrete values for all coefficients to achieve soundness with respect to floating-point arithmetic. We have not considered this here for simplicity.

$$\hat{x}_j = \alpha_{j,0} + \sum_{i=1}^{m} \alpha_{j,i}\epsilon_i$$
$$\alpha_{j,0}, \alpha_{j,i}, \in \mathbb{R}, \epsilon_i \in [-1, 1]$$

Assuming that variable x_j is bounded by a concrete interval $[x_j^l, x_j^u]$, we can encode x_j as $\hat{x}_j = \alpha_{j,0} + \alpha_{j,j}\epsilon_j$ where $\alpha_{j,0} = (x_j^l + x_j^u)/2$ and $\alpha_{j,j} = (x_j^l - x_j^u)/2$. The next step is to define abstract transformers for the affine operations of the neural network. For a fully-connected layer with weight matrix $W \in \mathbb{R}^{m\times n}$ and bias vector $\boldsymbol{b} \in \mathbb{R}^n$, we can use the exact abstract transformer:

$$\hat{z}_j = T(\hat{\boldsymbol{y}})_j = (\boldsymbol{b}_j + \sum_i W_{j,i}\alpha_{i,}) + \sum_{i=1}^{n}(\sum_{k=1}^{m} W_{j,k}\alpha_{k,i})\epsilon_i$$

The noise symbols are shared for all input variables, capturing the conditional dependency between variables. Finally, we need abstract transformers for the activation functions. The transformer used in (Singh et al., 2018) for the ReLU is:

$$\lambda = \frac{z_u}{z_u - z_l}$$
$$\mu = \frac{z_u z_l}{2(z_u - z_l)}$$
$$\hat{y} = \lambda\hat{z} + \mu + \mu\epsilon_{new}$$

Where z_l and z_y are the lower and upper bounds on the input and ϵ_{new} is a new noise variable. Interestingly, by substituting ϵ_{new} with its minima and maxima, -1 and 1, we get the upper and lower linear relaxation used in the symbolic interval propagation approach of (Wang et al., 2018a). The abstract transformer for the fully connected layer is also exactly the same as used in the symbolic interval propagation.

It actually turns out that the approach described in this section is almost equivalent to the error-based symbolic interval propagation from section 2.5.2. The main difference is the ReLU transform. The abstract transformation described here is the same as we would get if we propagated our equation through a line in the middle of the lower and upper linear relaxation instead of the lower linear relaxation. This would require us to keep track of a lower and upper error, instead of just an upper error. The new noise variable, ϵ_{new} represents this by varying from -1, corresponding to the lower relaxation, to $+1$, corresponding to the upper. The second difference is that the bounds on the input variables to the network are only used when calculating concrete upper and lower bounds in the symbolic interval propagation. In this approach, they are directly encoded into the Zonotopes instead. However, both of these methods should result in precisely the same bounds on the output of the network.

This duality between the Zonotope approach and the symbolic interval propagation indicates that we can use the abstract transformers for the s-shaped activation functions as inspiration to define linear relaxations. The abstract transformers used for the Sigmoid in (Singh et al., 2018) are equivalent to the lower and upper linear relaxations:

$$\begin{aligned} \lambda &= \min(\sigma'(z_l), \sigma'(z_u)) \\ r_l(z) &= \sigma(z_l) + \lambda(z - z_l) \\ r_u(z) &= \sigma(z_u) + \lambda(z - z_u) \end{aligned}$$

Where z_l and z_u are the lower and upper input-bounds respectively. The relaxation is illustrated in figure 2.12. While these relaxations are valid, the large projected area in the xy-plane indicates that they might not be optimal. This is discussed in more detail in chapter 3.

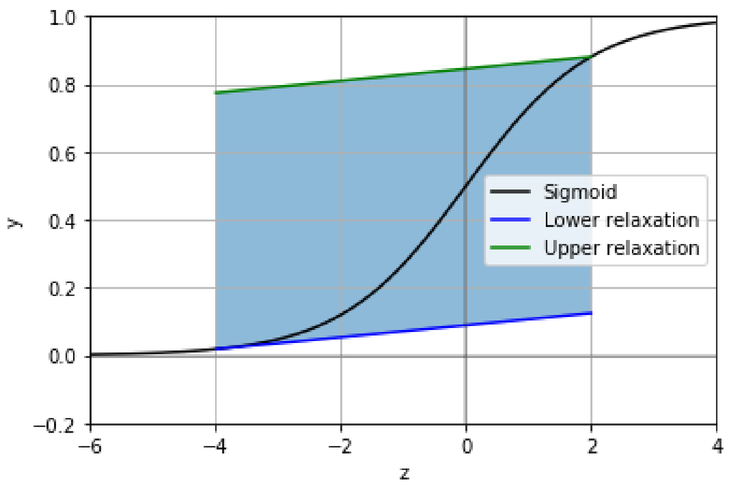


Figure 2.12: Zonotope abstraction for Sigmoid with lower and upper input bound -4 and 2 respectively.

The Zonotpe approach of (Singh et al., 2018) was further improved in (Singh et al., 2019). Most notably, the algorithm introduces a new abstract domain with a lower and upper bounding equation for each variable. This also opens the possibility for abstract transformers with non-parallel lower and upper bounds. For Sigmoid activation functions with $z_l < 0$ and $z_u > 0$, this algorithm uses almost the same abstraction as (Singh et al., 2018); except that the lower and upper relaxation are calculated separately. So the upper relaxation is given by $r_u(z) = \sigma(z_u) + \sigma'(z_u)(z - z_u)$ and the lower relaxation is $r_l(z) = \sigma(z_l) + \sigma'(z_l)(z - z_l)$. If $z_l \geq 0$ or $z_u \leq 0$ the line intercepting $\sigma(z_l)$ and $\sigma(z_u)$ is used for the lower or upper bound respectively instead. The experimental results show that this approach achieves better precision than the pointwise zonotopes; however, the runtime is somewhat longer for some networks.

The previous approaches have been further improved for ReLU networks by adding a refinement step in (Singh et al., 2019). This algorithm uses the abstract transformers from (Singh et al., 2018), but refines the calculated bounds by encoding a few of the early layers as MILP or LP-systems. Since MILP problems are computationally expensive, the MILP-solver is only used to calculate bounds for some nodes in early layers. In later layers, an LP-solver is used instead with the linear relaxations from (Ehlers, 2017). Finally, in the last layers, only abstract interpretation is used. If the MILP-solver is used for all nodes operating in non-linear areas, the approach is complete and similar to 2.6.3. The difference is that abstract interpretations are used to calculate concrete bounds for the MILP constraints instead of interval and LP analysis.

Finally, the verification algorithm from (Zhang et al., 2018) also supports s-shaped activation functions. This paper focuses on finding minimal adversarial distances using a symbolic interval propagation based approach. The minimal adversarial distance is defined as the minimal ϵ, such that there are adversarial examples for inputs in a ball, $\mathbb{B}_p(\boldsymbol{x}, \epsilon)$. Since we do not focus on minimal adversarial distances in this report, we wont cover the algorithm here. However, the paper introduces relaxations for s-shaped activation functions, and we build on this approach in chapter 3.

For the Sigmoid, the paper uses three different relaxations, depending on the lower and upper input bounds, z_l and z_u. If $z_u \leq 0$, the Sigmoid operates in the convex area, and they use the line intercepting both endpoints, $\sigma(z_l)$ and $\sigma(z_u)$, as an upper linear relaxation. If $z_l \geq 0$ the Sigmoid is concave, and any tangent to the Sigmoid can be used as an upper relaxation. However, the paper does not specify which tangent they use. Finally, if $z_l < 0$ and $z_u > 0$, they use the line that intercepts $\sigma(z_l)$ and is a tangent to $\sigma(z_t)$ for some $z_t > 0$. The value of z_t is determined using a binary search. The lower relaxation and the Tanh are done analogously.

2.7 Ethical consideration

Our project is a software project and does not involve any experiments on humans or animals and does not pose any physical or environmental safety hazards. We do not collect or use any personal or sensitive data. All data used for experiments are from large open-source datasets commonly used for benchmarking in machine learning projects.

We do not see any direct applications for our project in military use. However, the nature of machine learning algorithms is general in the sense that the same algorithms can be applied to a wide range of problems, also in military use. Our project could be used to analyse and improve the design and performance of neural networks, which again might have military applications. We do, however, not consider this potential to be larger than for any other machine learning-based research.

Our software uses several open-source libraries. We have made sure to credit the authors were this applies. We also use Gurobi, which is proprietary software with a free academic license. We do not distribute any parts of Gurobi with our software; instead, we have included a guide on how to obtain Gurobi through their website.

2.8 Summary

In this chapter, we introduced feed-forward neural networks, a layered neural network design where the pre-activation values of each node are only dependent on post-activation values from nodes in previous layers. The most common verification problem for these networks is to prove that the output stays within some bounds, given concrete lower and upper bounds on the input. We reviewed complete verification algorithms based on symbolic interval arithmetic, SMT-solvers, and MILP-solvers able to solve these types of problems for ReLU networks. Finally, we introduced a few sound but incomplete algorithms based on abstract interpretation also able to solve verification problems for networks with Sigmoid and Tanh activation functions.

Chapter 3

Contribution

In this chapter, we present our proposed algorithm for efficient verification of neural networks. The algorithm builds on the error-based symbolic interval propagation approach, and a complete description is given in the first section. The following sections describe our novel contributions in detail, and the last section contains proofs for soundness and completeness.

3.1 Algorithm overview

The algorithm handles the general verification problem from definition 2.2.1, and we represent this problem as a tuple $\langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle$. $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ is an FFNN, $\boldsymbol{x_l}, \boldsymbol{x_u} \in \mathbb{R}^m$ are concrete lower and upper bounds on the input, and $\psi_{\boldsymbol{y}}$ is a set of LP constraints on the output. $L : \mathbb{R}^m \rightarrow \mathbb{R}$ is a loss function used for local search which we cover in more detail later. If the output constraints, $\psi_{\boldsymbol{y}}$, are proven to be unsatisfiable, our algorithm returns "safe", if not we return "unsafe" and a counterexample $\boldsymbol{x'}$ such that $f(\boldsymbol{x'})$ violates $\psi_{\boldsymbol{y}}$.

Given a verification problem, $\langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle$, the algorithm initialises by running the error-based symbolic interval propagation described in section 2.5.2, producing bounds on the output, $\psi_{y,eq(\boldsymbol{x})}$. These bounds, the concrete lower and upper inputs, $\boldsymbol{x_l}, \boldsymbol{x_u}$, and the output constraints $\psi_{\boldsymbol{y}}$ are used as constraints in a satisfiability call to the LP-solver. If the solver determines that the problem is unsatisfiable, it is provably safe for the current branch and the algorithm backtracks. Otherwise, the solution from the solver, $\boldsymbol{x'}$, is checked for validity by determining if $f(\boldsymbol{x'})$ violates any constraints in $\psi_{\boldsymbol{y}}$. If the counterexample does not violate any constraint, the property is proven unsafe, and the algorithm terminates returning $\boldsymbol{x'}$. Otherwise, a local search is initiated around the spurious result, $\boldsymbol{x'}$, as described below. If still no valid counterexample is found, the refinement phase is launched. This phase adaptively splits the input to the most influential node by adding split-constraints to the LP-solver and symbolic interval propagation. Finally, both branches are explored by restarting from the symbolic interval propagation. The algorithm terminates when a valid counterexample is found, all branches have been proven safe, or a timeout criterion is reached. The whole pipeline is illustrated in figure 3.1 and implemented in a python library, called VeriNet. The rest of this chapter covers our novel contributions while the implementation details of the individual steps are covered in chapter 5.

Our algorithm introduces three major changes compared to (Wang et al., 2018a).

- We use an **adaptive splitting** strategy, always splitting the most influential node independent of its location in the network. This is in contrast to Neurifys (Wang et al., 2018a) hierarchical splitting strategy, starting at the first fully-connected layer, and only moving on to the next layer after splitting all nodes in the previous.
- We implement a **local search** phase, using spurious counterexamples from the LP-solver as a starting point for an adversarial attack in an attempt to find valid counterexamples.
- We add support for **more activation functions**. (Wang et al., 2018a) only supports the ReLU activation function, while we extend our algorithm to also support s-shaped activation functions and batch normalisation layers.

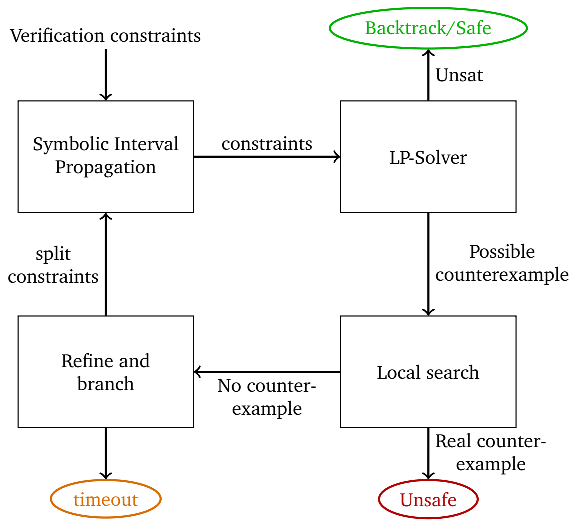


Figure 3.1: The pipeline for our algorithm. The main difference from (Wang et al., 2018a) is the local search and the implementation of the refine and branch step.

3.2 Adaptive splitting vs hierarchical splitting

For fully connected networks, Neurify (Wang et al., 2018a) uses a hierarchical splitting strategy, starting at the first layer and moving to the next layer after splitting all nodes in the previous layer. Splitting in the first layers has the advantage of improving bounds for later nodes, which improves the relaxations for these nodes. However, there are some significant disadvantages to this approach. First of all, there is no guarantee that all nodes operating in the non-linear area have a significant impact on the output. This may lead to splitting on relatively unimportant nodes, before reaching nodes with more influence on the output. Furthermore, after each split the equations for all succeeding nodes change; so the symbolic interval propagation has to be re-run from the layer where the split happens. So, splitting nodes in earlier layers is more computationally expensive. Finally, our experiments indicate that splitting in layers with few nodes is a significant advantage. Many neural networks, especially convolutional networks, have a non-uniform architecture with significantly fewer nodes in some layers. Hierarchical splitting might have to split a significant amount of nodes in larger layers before reaching more impactful nodes in smaller layers.

To avoid these limitations, our algorithm uses an adaptive splitting strategy instead. This strategy chooses split-nodes by evaluating their impact on the output, disregarding their location in the network, thus solving the limitations of the hierarchical approach. However, the adaptive method also introduces a new challenge. During hierarchical splitting, we can add and remove the latest split-constraints, safely assuming that no other constraints are affected. This is not true for adaptive splitting, since we already might have split-constraints in layers after the adjusted node, and the symbolic equations for these constraints change. This requires us to modify the LP-solver constraints, which is a computationally expensive task. However, even with this added complexity, we have seen some improvements for networks with uniform layers and significant improvements for larger cone-shaped convolutional neural networks. The LP-constraints we use for splitting in our implementation are described in chapter 5.

3.3 Splitting heuristic

The algorithm from (Wang et al., 2018a) uses a heuristic based on interval gradient analysis to determine the next split-node. Our approach instead utilizes the error-matrix calculated in the forward pass to define a heuristic. Remember that if $E^m_{k,h}$ is the error matrix at the output layer, the bounds for output node k are calculated as:

$$z^i_{l,k} = \min(eq^i_{in}(x)_k) + \sum_{h|\ (E^i_{in})_{k,h}<0} (E^i_{in})_{k,h}$$

$$z^i_{u,k} = \max(eq^i_{in}(x)_k) + \sum_{h|\ (E^i_{in})_{k,h}>0} (E^i_{in})_{k,h}$$

We use this to define an intuitive splitting heuristic. For a classification problem with correct class c, we either want to increase the lower bound of output node c by removing negative errors from the error matrix $E^m_{c,h}$, or decrease the upper bound of potential counterexample-classes, t, by removing positive errors from $E^m_{t,h}$. So for each hidden node h we calculate the impact-score as $s(h) = \gamma_c max(E^m_{c,h}, 0) - \sum_{t \neq c} \gamma_t min(E^m_{t,h}, 0)$, where γ is a weighing factor. For classes that have been proven safe in previous branches, we use $\gamma = 0$, for the correct class we use $\gamma = n$ where n is the number of potential counterexample classes, and for the rest, we use $\gamma = 1$. The reasoning for weighing the correct class more is that increasing the lower bound of the correct class might help prove all other classes safe. We have not observed very significant performance differences in our heuristic vs the gradient-based heuristic. However, our heuristic is less complex since the error-matrix is calculated in the forward pass, while gradient calculations need a separate back-propagation phase.

Notice that both the gradient and the error heuristic have their limitations. Most noticeably, both of them only measure the direct effect on the output by splitting a node. They do not account for the fact that splitting a node in earlier layers will also indirectly improve the output bounds by improving the relaxations of all succeeding nodes. So, our heuristic undervalues the effect of splitting nodes in the early layers. Secondly, none of the heuristics take into account that splitting in later layers is computationally advantageous since we only have to redo interval propagation from the layer where we split. This effect results in our heuristics undervaluing splits in later layers. These two effects could to some degree cancel out, however further research into this area is required.

3.4 Local search

In Wang et al. (2018a), potential counterexamples from the LP-solver, $\boldsymbol{x'}$, are tested for validity by calculating $\boldsymbol{y'} = f(\boldsymbol{x'})$ and checking if $\boldsymbol{y'}$ violates any constraints in $\psi_{\boldsymbol{y}}$. If a counterexample is spurious, this approach skips straight to the refinement phase. Our algorithm instead introduces a new local search phase before refinement. The intuition is that a true counterexample may be ”close” to the spurious results. This is because the LP-constraints are designed to approximate the true behaviour of the network.

The local search is implemented as a gradient descent with the spurious counterexample as a starting point. The loss function, $L(\boldsymbol{x})$, depends on the verification problem and LP-solver output. For classification problems with a correct class c that potentially can be misclassified as class t, the loss function is $L(\boldsymbol{x}) = y_c - y_t$ where y_c and y_t are the outputs of the correct and potential class respectively. The potential counterexample classes, t, are the classes having a larger upper bound than the correct class's lower bound in the symbolic interval propagation. In the case of multiple potential counterexample classes, t, the gradient descent is run once for each class. After each gradient descent step, the result, $\boldsymbol{x}_i$, is clipped to the input bounds. The gradient descent is terminated if the loss changes less than a given fraction, or after a specified number of steps.

3.5 Supported activation functions

While (Wang et al., 2018a) only supports ReLU activation functions, our implementation natively supports the Sigmoid and Tanh activation functions and batch-normalisation layers. The only requirement for adding more activation functions is that we define their linear relaxations. In this section, we define linear relaxations for all s-shaped activation functions, including the Sigmoid and Tanh. For the ReLU activation function, we use the same linear relaxation as Wang et al. (2018a), illustrated in figure 2.2.

3.5.1 S-shaped activation functions

Most of the activation functions used in neural networks can be divided into two families. The first family contains the piecewise linear activations such as the ReLU and MaxPool. Secondly, we have the s-shaped activation functions such as the Sigmoid and Tanh. [1]

Definition 3.5.1. *A continuous function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is a s-shaped activation function iff:*

1. $\sigma''(z) \begin{cases} > 0 \textit{ if } z < 0 \\ < 0 \textit{ if } z > 0 \\ = 0 \textit{ if } z = 0 \end{cases}$
2. $\sigma'(z) \geq 0$ *for all z.*
3. $\sigma'(z) = \sigma'(-z)$ *for all z.*
4. $\sigma''(z)$ *is differentiable for all z.*

This definition is somewhat more restricted than the usual definition of s-shaped functions. Notice especially that we require three times differentiability and that the shift from convex to concave happens at $z = 0$. The three times differentiability is important in the calculations of optimal solutions, but not for calculating valid solutions. The condition $\sigma''(z) = 0$ iff $z = 0$ is only for computational convenience, and our algorithm can easily be extended to support s-shaped functions with $\sigma''(z) = 0$ at some other z.

S-shaped activation functions are only supported by a few verification algorithms, most notably in (Zhang et al., 2018; Singh et al., 2018, 2019) as discussed in section 2.6.4. All of these algorithms have defined valid relaxations for both the Sigmoid and Tanh; however, none of them have focused on finding optimal relaxations, which we will do in this section. In this report, we only derive the upper linear relaxation; the lower relaxation is done analogously. Formal definitions of a upper linear relaxation and optimal upper linear relaxation are given below.

Definition 3.5.2. *A linear function, $r_u : [z_l, z_u] \rightarrow \mathbb{R}$, is a upper linear relaxation of $\sigma : [z_l, z_u] \rightarrow \mathbb{R}$ iff $r_u(z) > \sigma(z) \forall z \in [z_l, z_u]$.*

[1] A few common activation functions do not fit in either family, such as the exponential-linear unit (ELU). Those functions are not covered in this report; however, they can be implemented by defining valid linear relaxations.

Definition 3.5.3. *An upper linear relaxation, $r_u : [z_l, z_u] \rightarrow \mathbb{R}$ of $\sigma : [z_l, z_u] \rightarrow \mathbb{R}$ is optimal iff it is the upper linear relaxation that minimises $\int_{z_l}^{z_u} r(z) - \sigma(z)dz$.*

So, our definition of optimal linear relaxations minimises the projected relaxation area in the xy-plane. Treating this as a standard optimization problem results in non-linear constraints because of the s-shaped activation function. Solving a non-linear optimisation problem for each node in a large neural network is infeasible, so we have to find another approach. For the rest of this section let $\sigma(z)$ be an s-shaped activation function and let z_l and z_u be the lower and upper bound on the input. We start by proving two important lemmas for the optimal linear relaxation.

Lemma 3.5.1. *Let $\sigma : [z_l, z_u] \rightarrow \mathbb{R}$ be an s-shaped activation function. If the line $r_u(z)$ intercepting both endpoints,$\sigma(z_l)$ and $\sigma(z_u)$, is a valid upper linear relaxation, it is optimal.*

Proof. Let $l(z)$ be another line such that $\int_{z_l}^{z_u} l(z) - \sigma(z)dz < \int_{z_l}^{z_u} r(z) - \sigma(z)dz$. This requires $l(z') < r_u(z')$ for at least one $z' \in [z_l, z_u]$. If $z' = z_l$, then $l(z_l) < r_u(z_l) = \sigma(z_l)$ and $l(z)$ is not an upper linear relaxation. If $z' \in (z_l, z_u]$ and $l(z_l) \geq r_u(z_l)$ then $l'(z) < r_u'(z)$ and $l(z_u) < r(z_u) = \sigma(z_u)$. So $l(z)$ can't be a upper relaxation, and $r_u(z)$ is the optimal upper linear relaxation. □

Lemma 3.5.2. *Let $\sigma[z_l, z_u] \rightarrow \mathbb{R}$ be an s-shaped activation function. If the line $r_u : [z_l, z_u] \rightarrow \mathbb{R}$ intercepting both endpoints, $\sigma(z_l)$ and $\sigma(z_u)$, is not a valid upper linear relaxation, then the optimal linear relaxation is a tangent line to σ for a tangent point $z' \in [z_l, z_u]$.*

Proof. It is obvious that for an optimal upper linear relaxation, $r_u(z)$ we have $r_u(z') = \sigma(z')$ for at least one point $z' \in [z_l, z_u]$. Since $r_u(z) \geq \sigma(z)$ for all $z \in [z_l, z_u]$ we know that z' is a either tangent point or z' is one of the endpoints. If z' is a tangent point we are done.

Else, assume for contradiction that r_u is not a tangent to σ and $r_u(z_u) = \sigma(z_u)$. Since r_u is a valid upper linear relaxation it can't intercept σ for any $z \in (z_l, z_u)$, and it can't intercept z_l from the assumptions. So $r_u(z) > \sigma(z)$ for all $z \in [z_l, z_u)$. Since $r_u(z)$ is strictly larger than $\sigma(z)$ on $[z_l, z_u)$, there is a line $l(z)$ intercepting $\sigma(z_u)$ with $\sigma(z) \leq l(z) < r_u(z)$. So $l(z)$ is a valid upper linear relaxation, smaller than $r_u(z)$ for all $z \in [z_l, z_u)$ meaning r_u is not be optimal, and we have our contradiction.

If $r_u(z_l) = \sigma(z_l)$ the proof is analogous. □

These two lemmas prove that if the line intercepting both endpoints is a valid upper linear relaxation, it is the optimal upper linear relaxation. If this line is not valid, the optimal linear relaxation is a tangent to σ for a tangent point in $[z_l, z_u]$.

For the rest of the chapter, we split the input domain into three cases and analyse them separately. Let $S^+ = \{(z_l, z_u) | z_l \geq 0\}$, $S^- = \{(z_l, z_u) | z_u \leq 0\}$, and $S^{+/-} =$

$\{(z_l, z_u) | z_l < 0, z_u > 0\}$. This is the same segmentation is used in (Zhang et al., 2018). If $z_l = z_u$ there is no need for a relaxation, so for the rest of the chapter assume $z_l \neq z_u$.

Negative upper bound, $(\mathbf{z_l}, \mathbf{z_u}) \in \mathbf{S}^-$.

The case where $(z_l, z_u) \in S^-$ is simple, $\sigma(z)$ is convex for $z \leq 0$ so the line intercepting both endpoints is a valid upper relaxation and optimal. This is illustrated for the Sigmoid in figure 3.2a and is the same approach used in (Zhang et al., 2018). In this case, $r_u(z)$ is:

$$r_u(z) = \frac{\sigma(z_u) - \sigma(z_l)}{z_u - z_l}(z - z_u) + \sigma(z_u)$$

Positive lower bound, $(\mathbf{z_l}, \mathbf{z_u}) \in \mathbf{S}^+$.

The line intercepting the both endpoints is not a valid upper relaxation when $(z_l, z_u) \in S^+$. However, since $\sigma(z)$ is concave for $z \geq 0$ we can use any tangent to $\sigma(z_t)$ for $z_t \geq 0$ (Zhang et al., 2018). The next theorem provides us with the optimal tangent point:

Theorem 3.5.1. *Let $\sigma : [z_l, z_u] \to \mathbb{R}$ be an s-shaped activation function. If $0 \leq z_l < z_u$, the optimal upper linear relaxation, $r_u(z)$ is the tangent at $z_t = \frac{z_u^2 - z_l^2}{2(z_u - z_l)}$.*

Proof. Let $A : [z_l, z_u] \to \mathbb{R}$ be defined by $A(z_t) = \int_{z_l}^{z_u} r_u(z) - \sigma(z)dz$. From theorem 3.5.2, the optimal upper linear relaxation, r_u, is a tangent to $\sigma(z)$, so $r_u(z)) = \sigma'(z_t)(z - z_t) + \sigma(z_t)$. To find z_t, we find the minimise $A(x_t)$:

$$\begin{aligned}
A(z_t) &= \int_{z_l}^{z_u} \sigma'(z_t)(z - z_t) + \sigma(z_t)dz - \int_{z_l}^{z_u} \sigma(z)dz \\
&= \left[\frac{z^2}{2}\sigma'(z_t) - z_t z\sigma'(z_t) + z\sigma(z_t)\right]_{z_l}^{z_u} - (\Lambda(z_u) - \Lambda(z_l)) \\
A'(z_t) &= \left[\frac{z^2}{2}\sigma''(z_t) - z\sigma'(z_t) - z_t z\sigma''(z_t) + z\sigma'(z_t)\right]_{z_l}^{z_u} \\
&= \left[z\sigma''(z_t)(\frac{z}{2} - z_t)\right]_{z_l}^{z_u} \\
&= \sigma''(z_t)(z_u(\frac{z_u}{2} - z_t) - z_l(\frac{z_l}{2} - z_t)) = 0
\end{aligned}$$

This equation has two solutions. Firstly, $\sigma''(z_t) = 0$, secondly we have:

$$z_u(\frac{z_u}{2} - z_t) - z_l(\frac{z_l}{2} - z_t) = 0 \implies z_t = \frac{z_u^2 - z_l^2}{2(z_u - z_l)}$$

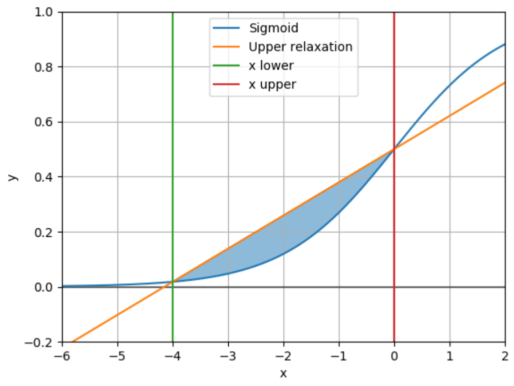
From the definition of σ, we know that $\sigma''(z_t) = 0 \implies z_t = 0$. Since $z_l \geq 0$, and $z_t \in [z_l, z_u]$, we know that $\sigma''(z_t) = 0 \implies z_t = z_l$.

This leaves us with three candidate tangent points for the minima, the endpoints z_l, z_u and $\frac{z_u^2 - z_l^2}{2(z_u - z_l)}$. The derivative at z_u is:

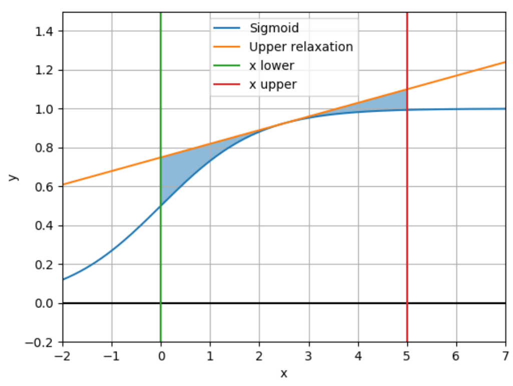
$$\begin{aligned} A'(z_u) &= \sigma''(z_u)(z_u(\frac{z_u}{2} - z_u) - z_l(\frac{z_l}{2} - z_u)) \\ &= \sigma''(z_u)(\frac{2z_l z_u - z_u^2 - z_l^2}{2})) \\ &= -\sigma''(z_u)(\frac{(z_u - z_l)^2}{2})) \end{aligned}$$

Since $\sigma''(z_u) < 0$, we know that $A'(z_u) > 0$ and this can't be a minima. Analogously, $A'(z_l) < 0$, so the tangent at z_l can't be a minima either. Finally, A is defined on a closed interval so it has a minima and the only remaining candidate is the tangent at $z_t = \frac{z_u^2 - z_l^2}{2(z_u - z_l)}$. □

This solution is illustrated in figure 3.2b.



(a) Upper relaxation intercepting endpoints



(b) Tangent as upper relaxation

Mixed positive and negative bounds, $(\mathbf{z_l}, \mathbf{z_u}) \in \mathbf{S}^{+/-}$.

For $(z_l, z_u) \in S^{+/-}$, the optimal linear relaxation depends on the exact values of z_l and z_u. From lemma 3.5.1 we know that if the line intercepting $\sigma(z_l)$ and$\sigma(z_u)$ is a valid upper relaxation, it is optimal, and the following theorem provides us with a test to check if this line is a valid.

Theorem 3.5.2. *Let $\sigma : [z_l, z_u] \to \mathbb{R}$ be an s-shaped activation function with $z_l \leq 0$ and $z_u > 0$. Furthermore, let $r_u : [z_l, z_u] \to \mathbb{R}$ be the line intercepting $\sigma(z_l)$ and $\sigma(z_u)$. $r_u(z)$ is a valid upper relaxation iff $r_u'(z_u) \leq \sigma'(z_u)$.*

Proof. First assume that r_u is not a valid upper linear relaxation, so $r_u(z^*) < \sigma(z^*)$ for some $z^* \in [z_l, z_u]$. If $z^* \in [0, z_u]$ then $r_u(z) < \sigma(z)$ for all $z \in [z^*, z_u]$ since $r_u(z)$ intercepts $\sigma(z_u)$ and we are in the concave part of $\sigma(z)$. As $r_u(z) < \sigma(z)$ before z_u and intercepts $\sigma(z_u)$ we have $r_u'(z_u) > \sigma'(z_u)$. If $z^* \in [z_l, 0]$, we also have $r_u(0) < \sigma(0)$ since $r_u(z)$ intercepts the lower endpoint, $\sigma(z_l)$, and $\sigma''(z) > 0$ for $z < 0$. This combined with the first argument shows that $r'_u(z_u) > \sigma'(z_u)$. So, all invalid linear relaxations have $r'_u(z_u) > \sigma'(z_u)$, and $r_u(z)$ is valid if $r'_u(z_u) \leq \sigma'(z_u)$.

Finally, assume that $r_u(z)$ is a valid upper linear relaxation. Then $r_u(z) \geq \sigma(z)$ for all $z \in [z_l, z_u]$, and since $r_u(z_u) = \sigma(z_u)$ we have $r'_u(z_u) \leq \sigma'(z_u)$. □

Theorem 3.5.2 lets us use the condition $r'_u(z_u) \leq \sigma'(z_l)$ to check if the line intercepting both endpoints is a valid upper relaxation. If this is not the case, we know from lemma 3.5.2 that the optimal relaxation is a tangent. From theorem 3.5.1, the optimal tangent point is:

$$z_t = \frac{z_u^2 - z_l^2}{2(z_u - z_l)}$$

However, this time this tangent is not automatically a valid upper relaxation since σ is not concave for $z_l < 0$. The next theorem provides a simple test to check if a tangent is a valid upper relaxation.

Theorem 3.5.3. *Let $\sigma : [z_l, z_u] \to \mathbb{R}$ be an s-shaped activation function with $z_l \leq 0$ and $z_u > 0$. Furthermore, let $r_u(z)$ be the tangent line at $\sigma(z_t)$ for $z_t \geq 0$. $r_u(z)$ is a valid upper relaxation iff $r_u(z_l) \geq \sigma(z_l)$.*

Proof. Assume that $r_u(z_l) \geq \sigma(z_l)$. Since $r_u(z)$ is a tangent line in the concave part of $\sigma(z)$ we know that $r_u(z) \geq \sigma(z)$ for $z \geq 0$. Since $\sigma(z)$ is convex for $z \leq 0$, $r_u(z_l) > \sigma(z_l)$ and $r_u(z)$ intercepts both $\sigma(z_l)$ and $r_u(0) \geq \sigma(0)$ we know that $r_u(z) > \sigma(z)$ for all $z_l \leq z \leq 0$ and $r_u(z)$ is a valid upper linear relaxation.

If $r_u(z)$ is a valid upper linear relaxation, $r_u(z_l) \geq \sigma(z_l)$ follows directly from the definition. □

So, we can use the condition $r_u(z_l) \geq \sigma(z_l)$ to test if the optimal tangent line is a valid upper relaxation. If this is not the case, we have one last possibility.

Theorem 3.5.4. *Let $\sigma : [z_l, z_u] \to \mathbb{R}$ be an s-shaped activation function with $z_l \leq 0$ and $z_u > 0$, and let z_t^* be the minimal z such that the tangent at $\sigma(z_t^*)$ is a valid upper relaxation. Assume that the line intercepting both endpoints and the tangent from theorem 3.5.1 are not a valid upper relaxation. Then the tangent at $\sigma(z_t^*)$ is the optimal valid upper relaxation.*

Proof. Since the line intercepting both endpoints is not a valid upper relaxation, the optimal valid upper relaxation is a tangent. A tangent is a valid upper relaxation iff $z_t^* \leq z_t \leq z_u$. We have shown that $A'(z_t)$ only has two zeros. The first solution $z_t = 0$ is a maximum and the second solution is assumed to be an invalid upper relaxation. So $A(z_t)$ has no minima's for $z_t^* < z_t < z_u$. This means that the minima is achieved at either $z_t = z_t^*$ or $z_t = z_u$. In the proof of theorem 3.5.1, we did show that

$A'(z_u) > 0$ so $z_t = z_u$ is not minimal. This only leaves us with $A(z_t^*)$ as minimal. $\square$

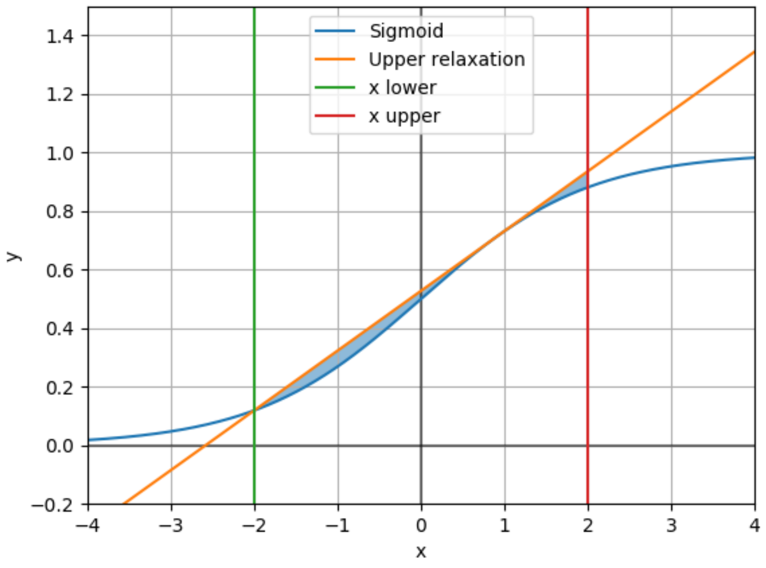


Figure 3.3: Minimal tangent point z_t^*

The minimal tangent point, z_t^* is illustrated in figure 3.3. The only thing that remains is to determine z_t^*. Since the tangent at $\sigma(z_t^*)$ intercepts $\sigma(z_l)$. the slope, a, of the linear relaxation, $r_u(z)$, is given by:

$$a = \frac{\sigma(z_t^*) - \sigma(z_l)}{z_t^* - z_l}$$

Since the slope is equal to the derivative at z_t^* we have:

$$\frac{\sigma(z_t^*) - \sigma(z_l)}{z_t^* - z_l} = \sigma'(z_t^*)$$

Unfortunately, we can't solve this equation analytically. Instead, we propose an efficient iterative method to find $\hat{z}_t^* \approx z_t^*$ in the next section.

Iterative minimal tangent point approximation

As discussed in the last section, we need a method to find z_t^* such that z_t^* is the minimal z where the tangent is a valid upper relaxation. We propose an algorithm to estimate $\hat{z}_t^* \approx z_t^*$. We also make sure that $z_t^* \leq \hat{z}_t^* \leq z_u$, such that the tangent for $\sigma(z)$ at $\hat{z}_t^*$ is a valid upper relaxation. We are only interested in z_t^* when we cannot use a line intercepting both endpoints, so assume that $z_u > z_t^*$. In the analytical approach, we tried to find the coordinate z_t^* such that the slope of the line from z_l to z_t^* is equal to the gradient of $\sigma(z)$ at z_t^*. Instead, we start by setting $z_0 = z_u$ and find the point z_1, where the gradient of $\sigma(z)$ is equal to the slope of the line between z_l and z_0. z_i can be found by solving:

$$\frac{\sigma(z_{i-1}) - \sigma(z_l)}{z_{i-1} - z_l} = \sigma'(z_i)$$

We have solved this equation for z_i for the Sigmoid and Tanh functions at the end of this section. Since $\sigma'(z) = \sigma'(-z)$ these equation will always have two solutions, one for $z_i < 0$ and one for $z_i > 0$ corresponding to lower and upper relaxations. The first step of the algorithm is illustrated in figure 3.4. We will prove that $z_t^* < z_i < z_u$ for all i, so the tangent at $\sigma(z_i)$ is a valid upper relaxation. Furthermore, we are also going to prove that $z_i < z_{i-1}$ so z_i converges closer to the optimal solution z_t^* each iteration.

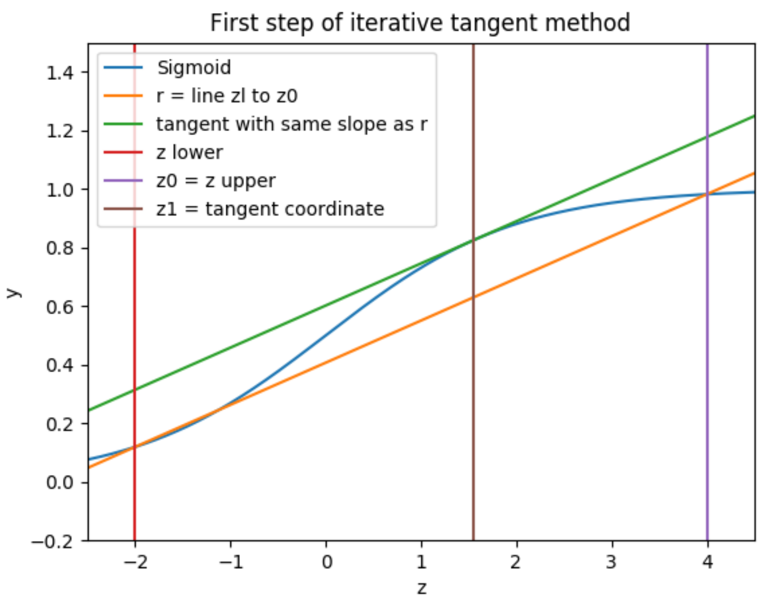


Figure 3.4: First step of the iterative optimal tangent coordinate algorithm. Calculate the line $r_0(z)$ intercepting z_l and $z_0 = z_u$. Find the tangent at $\sigma(z_t)$ with the same slope as $r_0(z)$ and set $z_1 = z_t$

Theorem 3.5.5. *Let $\sigma : [z_l, z_u] \to \mathbb{R}$ be an s-shaped activation function with $z_l \leq 0$ and $z_u > 0$ and let z_i be the positive solution of $\frac{\sigma(z_{i-1})-\sigma(z_l)}{z_{i-1}-z_l} = \sigma'(z_i)$. Furthermore, let z_t^* be the minimal z such that the tangent line at $\sigma(z_t^*)$ is a valid upper relaxation and let $z_{i-1} > z_t^*$, then $z_i \in (z_t^*, z_{i-1})$.*

Proof sketch. Since the line intercepting both endpoints, $l : \mathbb{R} \to \mathbb{R}$, is not a valid upper relaxation, the line has to be smaller than $\sigma(z)$ at some points in the concave area, $z > 0$. This combined with the fact that the line intercepts $\sigma(z_{i-1})$ means that $l(z) < \sigma(z)$ right before z_{i-1} and $l(z) > \sigma(z)$ right after. So the derivative of the line, a, is larger than

$\sigma'(z_i)$. Since z_{i-1} is in the concave area of $\sigma(z)$, we know that z_i, where $\sigma'(z_i) = a$, has to be smaller than z_{i-1}. So $z_i < z_{i-1}$.

Finally, from the definition of z_t^*, the derivative, a, has to be smaller than $\sigma'(z_t^*)$. Since z_t^* also is in the concave area of $\sigma(z)$, we know that the point $z_i > 0$ where $\sigma'(z_i) = a$ has to be larger than z_t^*. So $z_i \in (z_t^*, z_{i-1})$. $\square$

This theorem proves that with $z_0 = z_u$, the tangent at z_i is a valid upper relaxation since $z_i \geq z_t^*$, and that it is a better upper relaxation than the tangent at z_u since $z_i < z_u$. We usually got acceptable results in very few steps, and we use a maximum of 2 iterations in our implementation.

The relaxations presented in this section are optimal in the sense that they minimise the area between them in the xy-plane. However, our error-based symbolic interval propagation does not take full advantage of this. The concrete error is always calculated as the maximal distance between the relaxations, so for our algorithm, there might be better performing parallel relaxations. Even so, the relaxations proposed here performed significantly better than naïve solutions, as evident by the experiments in the end of this report.

Sigmoid update rule:

$$\begin{aligned}
\frac{\sigma(z_{i-1}) - \sigma(z_l)}{z_{i-1} - z_l} &= \sigma(z_i)(1 - \sigma(z_i)) \\
\sigma(z_i) &= \frac{1 \pm \sqrt{1 - 4(\sigma(z_{i-1}) - \sigma(z_l))/(z_{i-1} - z_l)}}{2} \\
z_i &= -\log(1/\sigma(z_i) - 1)
\end{aligned}$$

Tanh update rule:

$$\begin{aligned}
\frac{\sigma(z_{i-1}) - \sigma(z_l)}{z_{i-1} - z_l} &= 1 - \sigma(z_i)^2 \\
\sigma(z_i) &= \pm\sqrt{1 - (\sigma(z_{i-1}) - \sigma(z_l))/(z_{i-1} - z_l)} \\
z_i &= \frac{1}{2}\log\left(\frac{1 + \sigma(z_i)}{1 - \sigma(z_i)}\right)
\end{aligned}$$

3.5.2 Batch normalization

Batch normalisation layers perform a normalisation of the input by subtracting a running mean and dividing with a running standard deviation. This technique is used to reduce covariate shift, enabling us to train deeper networks. The running mean and standard deviation are updated during the networks training phase using the batch variance and mean per-dimension. Usually, batch normalisation also includes two learnable parameters, $\boldsymbol{\gamma}$, and $\boldsymbol{\beta}$. If layer i is a batch normalisation layer, and $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ are the running mean and variance respectively, the output of the batch normalisation during evaluation is:

$$\hat{\boldsymbol{y}}^i = \frac{\boldsymbol{y}^{i-1} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$$
$$\boldsymbol{y}_i = \boldsymbol{\gamma}_i \hat{\boldsymbol{y}}_i + \boldsymbol{\beta}_i$$

Since the running mean and variance do not change during evaluation, this is a linear transformation and implementing it is trivial; we just add the transformation to the symbolic interval propagation phase.

Similar to batch normalisation, dropout layers also operate linearly during the evaluation phase. Our current implementation does not support dropout; however, we could implement it the same way as batch normalisation.

3.6 Sound and complete

In this section, we prove that the algorithm proposed in this chapter is sound for all networks, and complete for ReLU networks. We start by formalizing the notions of soundness and completeness. Remember that our verification problem is represented by a tuple $\langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle$, where $f : \mathbb{R}^m \to \mathbb{R}^n$ is an FFNN, $\boldsymbol{x_l}, \boldsymbol{x_u} \in \mathbb{R}^m$ are concrete lower and upper bounds on the input, and $\psi_{\boldsymbol{y}}$ is a set of LP constraints on the output.

Definition 3.6.1. *Let $\boldsymbol{v} \in \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle$ be our verification problem, $\mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^m | \ x_{l,i} \leq x_i \leq x_{u,i}\}$, and $\mathcal{Y} = f(\mathcal{X})$. A verification algorithm $\mathcal{A} : \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle \to \{"safe", "unsafe"\}$ is sound iff $\mathcal{A}(\boldsymbol{v}) = "safe" \implies \mathcal{Y} \cap \{\boldsymbol{y} | \psi_{\boldsymbol{y}}\} = \emptyset$*

Definition 3.6.2. *Let $\boldsymbol{v} \in \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle$ be our verification problem, $\mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^m | \ x_{l,i} \leq x_i \leq x_{u,i}\}$, and $\mathcal{Y} = f(\mathcal{X})$. A verification algorithm $\mathcal{A} : \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}, L\rangle \to \{"safe", "unsafe"\}$ is complete iff it is sound and $\mathcal{Y} \cap \{\boldsymbol{y} | \psi_{\boldsymbol{y}}\} = \emptyset \implies \mathcal{A}(\boldsymbol{v}) = "unsafe"$.*

So, an algorithm is sound if it only returns "safe" when no valid input leads to an output fulfilling the constraints $\psi_{\boldsymbol{y}}$. Furthermore, it is complete if also all unsafe cases are identified correctly. To prove soundness for our algorithm, we first have to prove that the output bounds calculated by the symbolic interval propagation are overestimating. Proving this requires some preliminary results. Remember that the error-based symbolic interval propagation always propagates equations through the lower linear relaxations. The following lemma tells us how a nodes input equation would change if we used the upper linear relaxation instead of the lower for a previous node.

Lemma 3.6.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network with parallel linear relaxations, $r_l, r_u : \mathbb{R} \to \mathbb{R}$, for each node. Furthermore, let $eq_{in}^i(\boldsymbol{x})_k$ be the input equation and $(E_{in}^i)_{k,h}$ be the error for node k in layer i as calculated by the error-based symbolic interval propagation. If we were to use the upper linear relaxation instead of the lower for one node, h, in layer $j < i$, the resulting equation of the error-based symbolic interval propagation at node k in layer i would be $\widehat{eq_{in}^i(\boldsymbol{x})}_k = eq_{in}^i(\boldsymbol{x})_k + (E_{in}^i)_{k,h}$.*

Proof sketch. Assume that this theorem holds for all nodes up to layer $i-1$ an let h be in layer $j < i-1$. From the inductive hypothesis, the input equation to a node s in layer $i-1$ is $eq_{in}^{i-1}(\boldsymbol{x})_s + (E_{out}^{i-1})_{s,h}$. Applying the lower linear relaxation results in the output equation:

$$\widehat{eq_{out}^{i-1}(\boldsymbol{x})}_s = r_{l,s}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_s + (E_{in}^{i-1})_{k,h})$$

The input equations to layer i are calculated by performing the layers affine transformation on the outputs of layer $i-1$. Let $W \in \mathbb{R}^{m_{i-1}\times m_i}$ be the weight matrix, $\boldsymbol{b}^i \in \mathbb{R}_i^m$ be the bias.

$$\begin{aligned}
\widehat{eq_{in}^{i}(\boldsymbol{x})}_k &= W_{k,:}\widehat{eq_{out}^{i-1}(\boldsymbol{x})} + \boldsymbol{b}_k^i \\
&= W_{k,:}r_{l,k}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x}) + \sum_k (E_{in}^{i-1})_{k,h})) + \boldsymbol{b}_k^i \\
&= W_{k,:}r_{l,k}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})) + \boldsymbol{b}_k^i \\
&+ W_{k,:}r_{l,k}^{i-1}(\sum_k (E_{in}^{i-1})_{k,h}))) \\
&= eq_{in}^{i}(\boldsymbol{x})_k + (E_{in}^{i})_{k,h}
\end{aligned}$$

Where $eq_{in}^{i}(\boldsymbol{x})_k$ and $(E_{in}^{i})_{k,h}$ are as defined in the error-based symbolic interval propagation, proving that the theorem holds for $j < i-1$. For the next part let node h be node number t in layer $i-1$. Since the linear relaxations are parallel, we have:

$$\begin{aligned}
(E_{out}^{i-1})_{t,h} &= \max_{\boldsymbol{x}}(r_{u,t}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t)) - r_{l,t}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t))) \\
&= r_{u,t}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t)) - r_{l,t}^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t)) \;\forall \boldsymbol{x}
\end{aligned}$$

The output equation of node t in layer $i-1$ from using the upper linear relaxation is:

$$\begin{aligned}
\widehat{eq_{out}^{i}(\boldsymbol{x})}_t &= r_u^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t) \\
&= r_l^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t) + r_u^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t) - r_l^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t) \\
&= r_l^{i-1}(eq_{in}^{i-1}(\boldsymbol{x})_t) + (E_{out}^{i-1})_{t,h}
\end{aligned}$$

Applying the affine transform to this equation results in $eq_{in}^{i}(\boldsymbol{x})_k + (E_{in}^{i})_{k,h}$, proving the theorem. □

The previous lemma requires parallel linear relaxations, and we have not explicitly restricted our relaxations to be parallel. However, we use the maximum distance between the relaxations, which implicitly is the same as using parallel relaxations. The next corollary states that the effect of using the upper relaxation for more than one node.

Corollary 3.6.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a neural network with parallel linear relaxations, $r_l, r_u : \mathbb{R} \to \mathbb{R}$, for each node. Furthermore, let $eq_{in}^{i}(\boldsymbol{x})_k$ be the input equation and $(E_{in}^{i})_{k,h}$ be the error for node k in layer i as calculated by the error-based symbolic interval propagation. If we were to use the upper linear relaxation instead of the lower* *for all nodes in a set H, the resulting input equation at node k in layer i would be $eq^i_{in}(\boldsymbol{x})_k + (\sum\limits_{h\in H} E^i_{out})_{k,h}$.*

Proof sketch. Can be proven inductively from lemma 3.6.1. □

Using this corollary, we can prove that the error-based symbolic interval produces overestimating bounds.

Theorem 3.6.1. *Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be an FFNN, and let l^i_k, u^i_k be the true lower and upper bound of node k in layer i given input $\mathcal{X}$, so $l^i_k \leq z^i_k(\boldsymbol{x}) \leq u^i_k$ for all $\boldsymbol{x} \in \mathcal{X}$. If $eq^i(x)_k$ is the nodes equation and $E^i_{k,:}$ the errors, as calculated by the error-based symbolic interval propagation, then:*

$$l^i_k \geq \min_{\boldsymbol{x}}(eq^i(x)_k) + \sum_{k|\ E^i_{k,h}<0} E^i_{k,h}$$
$$u^i_k \leq \max_{\boldsymbol{x}}(eq^i(x)_k) + \sum_{h|\ E^i_{k,h}>0} E^i_{k,h}$$

Proof sketch. Since each linear relaxation is overestimating, it is clear that propagating an equation through the network and always choosing the lower or upper relaxation depending on which maximises the equation at node k in layer i produces a valid upper bounding equation at this node. It follows directly from Corollary 3.6.1, that the equation from always choosing the maximising relaxation is $eq^i(x)_k + \sum_{h|\ E^i_{k,h}>0} E^i_{k,h}$. The lower bound is analogous, by always choosing the minimising equation. □

With the previous theorem, the soundness and completeness of our proposed algorithm follows. We assume that the LP-solver is sound.

Theorem 3.6.2. *The algorithm proposed in this chapter, $\mathcal{A} : \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}\rangle \to \{"safe", "unsafe"\}$, is sound.*

Proof sketch. Let $\mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^m |\ x_{l,i} \leq x_i \leq x_{u,i}\}$ and $\mathcal{Y} = f(\mathcal{X})$. Let $\widehat{\mathcal{Y}}$ be the valid outputs given the bounding equations on the output from the error-based symbolic interval arithmetic. From theorem 3.6.1, these bounding equations are overestimating, so $\mathcal{Y} \subseteq \widehat{\mathcal{Y}}$.

$\mathcal{A}(\boldsymbol{v}) = "safe"$ only happens if a satisfiability call to the LP-solver with the constraints $\{\boldsymbol{y} \in \widehat{\mathcal{Y}}, \psi_{\boldsymbol{y}}\}$ is unsatisfiable, or equivalently $\{\widehat{\mathcal{Y}} \cap \{\boldsymbol{y}|\psi_{\boldsymbol{y}}\} = \emptyset$. Since $\mathcal{Y} \subseteq \widehat{\mathcal{Y}}$, we also have $\mathcal{Y} \cap \{\boldsymbol{y}|\psi_i\} = \emptyset$. This combined with the fact that the branching exhaustively explores all possibilities proves that our algorithm is sound. □

Finally, we have completeness for ReLU networks.

Theorem 3.6.3. *The algorithm proposed in this chapter, $\mathcal{A} : \langle f, \boldsymbol{x_l}, \boldsymbol{x_u}, \psi_{\boldsymbol{y}}\rangle \to \{"safe", "unsafe"\}$, is complete for networks using only ReLU activation functions.*

Proof sketch. This follows from theorem 3.6.2 and the fact that after splitting a ReLU node, it operates linearly in each branch, removing all overestimation for that node. In a network with no overestimation, the output bounds produced by the symbolic interval propagation are clearly exact, so the problem is solved by exploring a maximum of 2^N branches where N is the number of nodes. □

3.7 Summary

In this chapter, we proposed an efficient algorithm for verification of neural networks based on the symbolic interval propagation approach of (Wang et al., 2018a). The novel local search phase uses an adversarial attack-based technique to find valid counterexamples from the LP-solvers spurious counterexamples. We also employ an adaptive splitting strategy, compared to the hierarchical strategy used in (Wang et al., 2018a). The adaptive approach prioritizes refinement of the most significant nodes, generalizing better to the non-uniform architectures commonly seen in convolutional networks. Finally, we derived the necessary theory for optimal relaxations of s-shaped activation functions and proved that our algorithm is sound for all networks, and complete for ReLU networks.

Chapter 4

Analysis of algorithmic design choices

State-of-the-art complete verification algorithms are characterised by their choice of underlying solver and splitting domain. In this chapter, we discuss our choice of the symbolic interval propagation approach compared to SMT and MILP-based methods. Furthermore, we analyse the implications of splitting nodes during refinement instead of the input domain.

4.1 Symbolic interval propagation

Symbolic interval propagation has several advantages over SMT and MILP-based approaches for complete verification. First of all, the symbolic interval propagation-based approaches of (Wang et al., 2018b) and (Wang et al., 2018a) are currently the only complete verification algorithms with a proven ability to verify problems with large neural networks and high input dimensionality. Furthermore, the computationally expensive parts of the algorithm are mostly standard linear algebra operations, so their implementations can benefit from one of several highly optimised linear algebra libraries. Finally, as shown in chapter 3, symbolic interval propagation generalises well to non-piecewise linear activation functions.

However, there is one major challenge with symbolic interval propagation. The coefficient matrices require a significant amount of memory. This problem is further aggravated since we store most intermediate calculations for efficient branching. As our experiments at the end of this report show, both our implementation and Neurify (Wang et al., 2018a) are memory-bound for large neural networks. We address this challenge in more detail in chapter 6. Current algorithms based on SMT and MILP solvers, don't seem to face the same memory requirements, and with future computational optimisation they might scale better to larger networks. However, at this point, the advantages of symbolic interval propagation seem to outweigh the disadvantages.

4.2 Splitting domain

Refinement phases of verification algorithms are usually implemented by splitting either the input domain or the networks hidden nodes. The first modern complete verification algorithms (Ehlers, 2017; Katz(B) et al., 2017) used node-splitting. This strategy has the advantage that piecewise linear activation functions can be split into linear sub-domains in a finite number of branches, resulting in completeness.

Later, ReluVal (Wang et al., 2018b) introduced splitting on the input domain instead of the node domain. ReluVal has a substantial speed-up over Reluplex on the most popular benchmark networks, the ACAS-Xu (Kochenderfer et al., 2012) neural networks. This approach was again improved in Neurify (Wang et al., 2018a). The paper returned the focus to splitting on ReLU nodes and managed to verify properties of significantly larger high-dimensional input networks than before. However, the implementation used for experiments on ACAS Xu did still split the input domain, and it is very similar to the algorithm used by ReluVal. Finally, Marabou (Katz et al., 2019) was introduced as an improvement of Reluplex. The maybe most notable improvement is the divide-and-conquer part of the algorithm introducing input-domain splitting in addition to ReLU splitting. Marabou achieves speeds on the ACAS Xu networks comparable to ReluVal.

It seems like algorithms have moved more towards splitting the input domain, or a combination of input and nodes, instead of just splitting the node domain. Judging solely on the improvements for the ACAS Xu networks, this might seem like the correct choice. However, we believe that the ACAS Xu dataset does not necessarily provide a good indication of the performance with other network architectures and datasets. This would make the current trend of ACAS Xu being overrepresented in experiments unfortunate.

Our experiments indicate that splitting nodes in small layers is significantly more efficient than splitting in larger layers. Following this intuition, it is reasonable to assume that if the dimension of the input layer is small, it might be more efficient to split the input domain than the nodes. The input of the ACAS Xu networks is only six-dimensional, much smaller than the 50-node hidden layers. This could explain why the input-domain splitting algorithms have the best performance on this dataset.

However, one of the main strengths of neural networks is their ability to handle high-dimensional data, such as images, and none of the algorithms splitting on the input domain have achieved notable results on this kind of data with the usual l_∞ based verification problems. The networks used for processing high-dimensional data usually have several layers with significantly fewer nodes than the input dimension. We believe that splitting in these smaller layers is essential for efficient verification and our algorithm achieves state-of-the-art performance on these kinds of problems, supporting the assumption.

From the discussion so far, it might seem like we should split the input domain for networks with relatively small input dimensions and on the node domain else. However, it's not that simple. The choice of splitting domain also depends on the verification problem. Most of the verification algorithms focus on problems where each input has a concrete upper and lower bound, and can vary independently within those bounds. However, Neurify also handles verification problems based on contrast and brightness changes in images. Given a γ, the brightness problem is to find an $|\epsilon| \leq \gamma$ such that for an input, $\boldsymbol{x}$, $\boldsymbol{x} + \epsilon$ is misclassified, or prove that no such ϵ exists. The contrast problem is exactly the same, but with $\boldsymbol{x}\epsilon$ instead of $\boldsymbol{x} + \epsilon$. Since each input pixel changes by the same amount (or factor for contrast), we can actually split more than one input at a time. Neurify splits all inputs in the middle of their current valid interval at each branch. We have not found any experiments comparing this approach with splitting on the nodes; however, it is reasonable to assume that splitting all inputs at once in each branch is a huge advantage compared to splitting one node at a time.

In summary, the splitting domain problem is complex, depending on the input data, network architecture, and verification problem. A lot more research is required to determine optimal splitting strategies. However, the main goal of this project was to solve verification problems for high dimensional input data with concrete upper and lower bounds on the inputs. Current research seems to indicate that splitting the node domain is the best strategy for these types of problems.

Chapter 5

Implementation

The large computational and memory demands for our algorithm requires some carefully considered implementations choices. In this chapter, we first present an overview of the implementation, and then discuss some important design choices, focusing on choices we made to reduce the impact of computational and memory intensive parts of the code.

5.1 Overview

We implemented the algorithm presented in this paper as a Python toolkit, VeriNet. VeriNet takes as input a neural network, input bounds, and a verification objective containing the output constraints and the loss function for the local search. The resulting output is either ”unsafe” together with a counterexample, or ”safe” if the system is proven to have no counterexamples. A counterexample is defined as an input within the given input bounds, resulting in an output violating the given output constraints. Due to floating point precision, ”underflow” may also be returned, indicating that no counterexample was found, and the system was not proven safe after splitting all nodes. Since our code has to handle several different network architectures, we have focused on writing modular, object-oriented code. A sketch of the class dependencies is illustrated in figure 5.1.

The VeriNet class contains the multiprocessing functionality and spawns a pool of VeriNetWorkers running our algorithms main verification loop. Each VeriNetWorker depends on an NNBounds object, which contains the functionality related to the symbolic interval propagation. The linear relaxations used for this phase are represented by the abstract class ActivationFunctionAbstraction.

Furthermore, the workers also depend on a verification problem, represented by the abstract class VerificationFunctionAbstraction. This class contains the verification problem-specific logic, including the loss function for gradient descent, and the LP-solver constraints dependent on the verification problem. Finally, Branch objects store information about unexplored branches, and the LP-solver class functions as a wrapper around Gurobi.

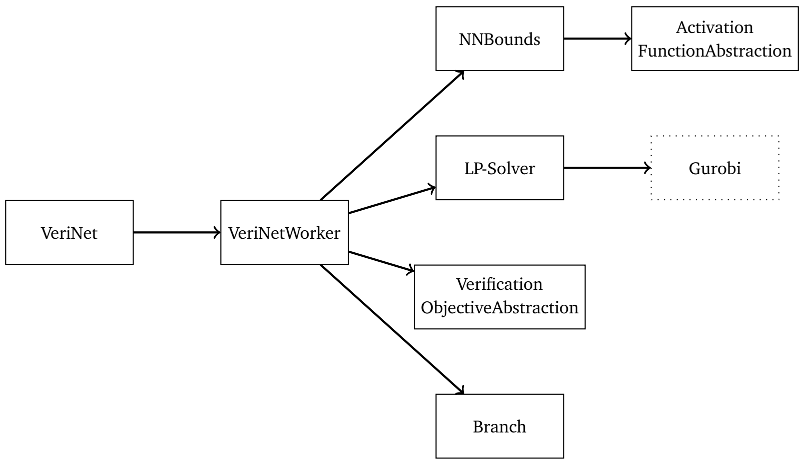

NNBounds
Activation FunctionAbstraction
LP-Solver
Gurobi
VeriNet
VeriNetWorker
Verification ObjectiveAbstraction
Branch

Figure 5.1: The class dependencies of VeriNet

VeriNet has several third-party dependencies including, Numpy (Numpy, 2019) with the OpenBLAS-backend-(OpenBLAS, 2019) for vectorised calculations, Numba (Numba, 2019) for jit-compilation, Pytorch (PyTorch, 2019) for neural network computations, and Gurobi (Gurobi Optimization, 2019) for the LP-solver phase. Furthermore, our algorithm uses Pythons multiprocessing module for parallelisation.

5.2 Symbolic interval propagation

This phase is mostly a straightforward vectorized implementation of the error-based symbolic interval propagation from section 2.5.2. However, this is the most time-consuming part of our algorithm, so we cover some essential aspects in detail. First, we discuss how to minimise the number of calculations after each split, followed by a look at rounding errors and how to reduce this problem through outward rounding. Finally, we are going to see how we can store all the information necessary for branching in a memory-efficient way.

5.2.1 Minimising split-calculations

This part should be apparent from the algorithm description and is essential for an efficient implementation. After each split, we do not need to recalculate all symbolic bounds. Since we are working with feed-forward neural networks, only the symbolic bounds of the node we split, and nodes in later layers are affected. Our algorithm tends to split in later layers, especially for the cone-shaped networks typically seen in deep convolutional networks.

Almost the same is true when back-tracking; however, at this stage, we might remove more than one split constraint. So, we have to recalculate the bounds from the node in the earliest layers where we removed a split-constraint.

During our experiments, we clearly saw how important this part is. For our largest network, exploring the first branch usually took more than 60 seconds. If we recalculated all bounds at every branch, each process should only be able to explore a maximum of 360 branches within the one-hour time-out. However, in practice, each process was able to explore closer to 6000 branches.

5.2.2 Outward rounding

Interval arithmetic on a computer is prone to rounding errors, which can lead to invalid bounds. Technically this problem can be solved by outward rounding, adding and subtracting the maximal rounding error for each floating-point operation to the upper and lower bound. However, this approach is problematic in practice due to two reasons.

First of all, the worst-case scenario is much worse than what can be expected in reality. In one of our experiments, we use a network with two fully connected layers of size 512 and an input of dimension 784. The symbolic equations for the output of layer 1 are represented by a 512×784 matrix. This matrix is multiplied by a 512×512 weight matrix. So, this operation alone requires $512 \times 512 \times 784 \approx 200$ $million$ floating-point multiplications. Even if each worst-case rounding error is in the 10^{-8} range, the outward rounding is around 2 for the lower and upper bound. Moreover, this is only for one operation in one of several layers. So in practice, many problems may be unsolvable with this approach.

Secondly, the maximal errors depend on the values of the floating-point numbers used in the calculations. Numbers with larger magnitudes have a potential for larger rounding errors. So in practice, each floating-point operation has to be considered separately. Most verification algorithms rely on efficient libraries for significant computational speed-up. Our algorithm and Neurify (Wang et al., 2018a) use OpenBLAS to speed up matrix calculations. These libraries do usually not support this kind of outward rounding.

Neurify (Wang et al., 2018a) has the option to do outward rounding when calculating concrete bounds from the symbolic equations. This approach does not give any guarantee of correctness since several floating-point operations, such as matrix multiplication, are performed without outward rounding. However, in most programming languages, rounding is implemented in such a way that the expected rounding error of floating-point operations is 0. This makes it reasonable to assume that performing outward rounding on this subset of operations solves a significant part of the problem. We have also implemented this approach as an option.

5.2.3 Best bounds matrix

In the forward propagation phase, we keep track of a matrix containing the best known concrete upper and lower bounds for each node. This includes bounds calculated in previous branches and bounds resulting from splits. When calculating the concrete bounds in the forward phase, we always use the best bounds from the calculated values or this matrix. This matrix is the only information we need to restart symbolic interval propagation from a given branch, and it is a very memory efficient way of doing this since we only need two concrete values for each node.

5.3 LP-solver

VeriNet uses the the Gurobi (Gurobi Optimization, 2019) solver for the LP phase. Gurobi is a MILP solver; however, for our implementation we only use linear constraints. The solver is used to find possible counterexamples, so the constraints are formulated such that if the solver returns ”UNSAT”, the current branch is safe. The following sections cover the encodings we use for the verification problem and the split-constraints.

5.3.1 Verification problem constraints

In this section, we cover the encodings used for classification networks, and the general robustness adversarial verification problem from definition 2.2.3. A counterexample for this problem is a valid input such that the output of the correct class is smaller than the output of at least one other class. This can directly be encoded as a MILP problem with binary variables, and since Gurobi is a MILP solver we could use this encoding. Assuming we have N classes, $eq^c(\boldsymbol{x})$ is the symbolic equation of the correct class, $eq^t(\boldsymbol{x})$ for all $t \neq c$ are the equations for the potential target classes, y^c_{max} is the concrete maximal value of the correct output, and E is the error matrix at the output layer, a MILP encoding is:

$$
\begin{gathered}
(1-b_t)y^c_{max} + b_t * \left(eq^t(\boldsymbol{x}) + \sum_{\{i|E_{t,i}>0\}} E_{t,i} \right) \geq \left(eq^c(\boldsymbol{x}) + \sum_{\{i|E_{c,i}<0\}} E_{c,i} \right) \\
t \in \{0, 1..., N\} \mid t \neq c \\
b_i \in \{0, 1\} \\
\sum_{i \neq c} b_i \geq 1
\end{gathered}
$$

This works reasonably well, however, there is also another formulation we can use. Instead of using a MILP problem, we can encode this as $N-1$ LP-constraints. Each constraint is on the form:

$$\left(eq^t(\boldsymbol{x}) + \sum_{\{i|E_{t,i}>0\}} E_{t,i} \right) \geq \left(eq^c(\boldsymbol{x}) + \sum_{\{i|E_{c,i}<0\}} E_{c,i} \right)$$

The runtime for all $N-1$ equations on this form is somewhat slower than the MILP encoding. However, this encoding has one huge advantage. Whenever one of these $N-1$ problems is determined to be unsatisfiable for a class t, we know that class t has no counterexample in any succeeding branches. So, we do not have to re-run the LP-solver for this class in any of the later branches. The MILP encoding does not provide us with this information and we noticed a substantial speed-up using this encoding over the MILP.

For the targeted robustness verification problem from definition 2.2.2, we use exactly the same approach. The only difference is that we end up with only one LP-equation.

We can also use MILP constraints on the input variables. However, the current implementation of the symbolic interval propagation only utilises concrete upper and lower bounds on the input when calculating concrete bounds from the symbolic bounds. The resulting concrete bounds are going to be wider than necessary, resulting in sub-optimal linear relaxations, making the problem difficult to solve.

5.3.2 Split constraints

Each split adds split-constraints to the LP-solver constraining the lower and upper symbolic bound to be larger and smaller than the split-value, respectively. If we had the exact input equation to the split node, we would simply constrain this equation to be larger than a split value s. However, we also need to take the errors into account, so our split constraints are:

$$eq_i(\boldsymbol{x}) + \sum_{k|E_{i,k}>0} E_{i,k} \geq s$$
$$eq_i(\boldsymbol{x}) + \sum_{k|E_{i,k}<0} E_{i,k} \leq s$$

Since the ReLU is linear on either side of 0, we use the split-value $s=0$. For Sigmoid and Tanh activation functions, σ, we use the value, s, where $\sigma(s)$ is in the middle of the upper and lower output. So if x_l, x_u are the lower and upper bounds on the input to σ, s is found by solving:

$$\sigma(s) = \frac{\sigma(x_u) + \sigma(x_l)}{2}$$

Adding and removing split-constraints to the LP-solver is a computationally expensive operation. Similarly to how we minimised the calculations of symbolic bounds after splits in section 5.2.1, we make sure only to modify the relevant split constraints during branching and backtracking. Each time we split a node, we have to add the split constraints for that node and modify the constraints for all nodes in later layers since the symbolic equations might change. Analogously, on backtrack, we update the split constraints for all nodes after the earliest layer where we had to remove a split.

5.4 Local search

The local search implementation is a straightforward implementation of the algorithm described in section 3.4. We use the ADAM (Kingma and Ba, 2014) optimiser from PyTorch to do a gradient descent on the input parameters. Updating the input parameters, instead of the weight parameters is natively supported in PyTorch. At each branch, we do a maximum of five iterations of gradient descent, with a relatively large step-size of 0.1, compared to 0.001 in the original paper.

This part of the algorithm turned out to be extremely efficient; almost all counterexamples were found without branching in less than five iterations of gradient descent. However, it does not help at all for safe cases and to reduce the computational demand of this step, we only run it every fifth branch.

5.5 Refinement

The branching part of the algorithm has a natural interpretation as a recursive implementation, and this is how Neurify (Wang et al., 2018a) implements it. However, we noticed that the memory requirements of the algorithm call for a more memory efficient implementation. To achieve this, we instead implement branching with a queue structure. The queue stores unexplored branches represented by a matrix with the current best concrete bounds for each node, and the node-splits done so far. As we mentioned in section 5.2.1, we only recalculate the symbolic bounds and update the LP-solver constraints for nodes after the node we split.

5.6 Multiprocessing

The tree structure from the branching makes our algorithm highly parallelizable, and we use multiprocessing to take advantage of this. Our algorithm starts by doing one run without branching in the main process. For many verification problems, a large amount of the data-points can be verified without branching, and performing this one run in the main process saves us the overhead of initialising child-processes. If the same main process is used for several data-points, we also only have to do the jit-compiling once.

If the first run is undecided, the main process spawns a pool of child-processes. Typically, we found 2 times the number of processor cores to work well. However, for the largest networks, we had to limit the number of processes to not run out of memory. Each process then picks branches out of a multiprocessing branch-queue. When a process picks a new branch from the queue, it has no information about previous symbolic bounds, and it has to recalculate all symbolic bounds and add all LP-solver constraints. These are very time-consuming calculations, so we want to limit the number of times processes have to get branches from the multiprocessing queue.

To reduce the number of times processes have to fetch new branches from the multiprocessing queue, we follow a few rules:

- When a process sends a branch to the multiprocessing queue, it is always the branch with the smallest depth. The reasoning is that shallow branches, which have not been explored much are probably harder and require a longer runtime.

- A process only sends branches to the multiprocessing queue if the depth difference between the current branch and the local branch with the smallest depth is more than a given number. The standard difference is 10. Again, the reasoning is to avoid sending easy branches that can be solved quickly.

- A process only sends branches if the multiprocessing queue has fewer branches than a fraction of the maximal processes. As long as all processes are working, it is better to keep the branches locally, since it is cheaper to backtrack to a local branch than fetching a new branch from the multiprocessing queue. The standard setting for this fraction is 0.2.

We tried some alternative implementations, among them an implementation where processes were spawned in a tree structure by children processes when needed. This approach is theoretically more efficient since we do not spawn any unnecessary processes; however, with a one-hour time-out, the relative time used to spawn processes is minimal. Our implementation is simpler and seems to have a near-linear speed-up in the number of processes for difficult problems, so it is close to optimal.

5.7 Testing

Ensuring that machine learning algorithms work as expected is notoriously difficult, and verification algorithms are no exception. To make sure that our implementation is correct, we wrote unit-tests for low-level functionality such symbolic bound concretisation, propagation functionality and more. However, unit-tests are best suited for testing of small functions that run independently of the rest of the program.

To test the higher-level functionality of the symbolic error propagation, we implemented some brute-force tests for very simple networks. These tests are implemented by calculating lower and upper bounds for the output, and running a wide range of possible inputs through the actual networks checking that all outputs stay within the bounds.

Our local search phase also turned out to provide us with a good sanity check. After running the algorithm with local search, we can disable it and make sure that none of the data points verified as unsafe with the local search are now verified as safe. This helped us identify several bugs during early development.

The main testing was done comparing our implementation to other state-of-the-art verification algorithms. First of all, the calculated bounds of our implementation and Neurify (Wang et al., 2018a) are identical before splitting, indicating that this phase is implemented correctly. Furthermore, our final verification results agreed with other algorithms for all tests. We ran a total of 1350 tests against Neurify (Wang et al., 2018a), 500 test with Marabou (Katz et al., 2019), and 600 tests with DeepPoly (Singh et al., 2019).

5.8 Summary

VeriNet is a modular, object-oriented, implementation of our algorithm. In this chapter, we have discussed some design choices important for an efficient and correct implementation. We discussed how we efficiently store branching information in a queue, how we minimise inter-process communication by keeping as much information as possible local in the processes, and generally how we avoid unnecessary calculations. Finally, we also looked at how we reduce the problem of rounding errors by implementing outward rounding.

Chapter 6

Complexity analysis

Complete verification of neural networks is still infeasible for the largest modern architectures. The computational and memory complexity of our algorithm grows asymptotically faster than standard operations of a neural network. In this chapter, we look at the bottlenecks and explain some of our implementation choices in view of this. Our analysis focuses on bottlenecks we found in practice and to a lesser degree on theoretical worst-case complexity.

6.1 Symbolic interval propagation

The symbolic interval propagation is currently the bottleneck for very large and deep neural networks. In this section, we do an isolated analysis of one interval propagation instance. Later in this chapter, we analyse the bigger picture when combining this with branching and multiple processes.

6.1.1 Computational complexity

The computational complexity of a standard forward phase in a neural network is dominated by the multiplication of the weight matrix and the output from the previous layer. Assuming we have two layers of size N, we are multiplying a $N \times N$ matrix with a vector of size N. With naïve matrix multiplication; the computational complexity is $O(N^2)$.

With symbolic interval propagation, we multiply the weight matrices by a matrix containing the coefficients of the symbolic equations instead. If the input dimension is M, each symbolic equation has $M + 1$ coefficients. So we are multiplying a $N \times N$ matrix with a $N \times (M + 1)$ matrix. Each of these operations has a complexity of $O(N^2M)$ and since we are working with high-dimensional data M is large, for many layers much larger than N.

Furthermore, we have to multiply the weight and error matrix for each layer. If K is the number of nodes before and including layer i, and layer i is of size N, the error matrix is $N \times K$. If also layer $i + 1$ is of size N the weight matrix is $N \times N$, and the complexity of the weight-error multiplications is $O(KN^2)$. For the later layers, K can be very large. Fortunately, this operation is not so bad in practice for two reasons. Firstly, many networks have a cone-shape where later layers are smaller. So for early layers when N is large, K is small, and for later layers when K is large, N is small. Secondly, for networks with ReLU nodes, nodes operating only in the linear area do not require relaxation and thus do not have an error. In practice, we do not add these nodes to the error matrix, since this would lead to columns with only zeros. So the actual number of columns in the error matrix is usually significantly smaller than K.

Finally, we have one more computationally expensive operation. Calculating concrete upper and lower bounds from the symbolic equations requires one multiplication and addition for each coefficient in each equation. For a layer of size N and input of size M, the computational complexity is $O(NM)$. While this does not seem like a problem compared to the previous complexities, in practice it dominates the run-time for some networks. The reason is that we have to perform an if-test for each coefficient, checking if it is positive or negative. This determines if we multiply the coefficient by the inputs lower or upper bound. The if-test makes the operation unsuitable for direct vectorization. Since our implementation is in python, this is a huge challenge. Pythons loops are far too inefficient to do this the naïve way. We have found two solutions to this problem.

The first solution is a vectorized approach where we make two copies of the coefficient matrix. In the first copy, we set all negative values to 0, and in the second, we set all positive values to zero. These matrices are then multiplied by the lower and upper input bound vectors. The main challenge of this approach is that creating copies of large matrices is expensive. The second approach, which is the currently implemented approach, is to use a python jit (just-in-time) compiler. This approach works reasonably well; however, this means that we do not get the extra speed up from highly optimised linear algebra libraries such as OpenBLAS (the back-end we use for numpy).

In summary, the computational time is in practice dominated by the weight-coefficient matrix multiplication and calculating concrete values from symbolic bounds. The first operation has in practice a cubic complexity with respect to the layer size. The second one is slow since an efficient implementation is difficult. The computational complexity of this phase shows the importance of only re-calculating necessary bounds after branching, as explained in section 5.2.1.

6.1.2 Memory complexity

Our current implementation saves all intermediate information calculated during the forward phase, including symbolic bounds, concrete bounds, linear relaxations, and more for each layer. This simplifies recalculating symbolic bounds for only parts of the network after a split; however it significantly increases the memory complexity. The complexity is dominated by the coefficient matrices storing the symbolic bounds, and the error matrices.

For a layer with size N and an input dimension of M, the memory complexity for storing the symbolic bounds is $O(NM)$. Furthermore, If K is the total number of nodes up to and including a layer of size N, the complexity for storing the error matrix is $O(NK)$. For smaller networks, such as the fully connected MNIST-digit networks we use in our experiments, the memory consumption is neglectable. However, for larger networks, such as the Cifar10 network from our experiments, we have a total of ≈ 55000 nodes, with ≈ 35000 nodes in the first layer. This, combined with an input dimension of ≈ 3000, results in significant memory usage. With 32-bit float, the memory required for storing the symbolic bounds of the first layer alone can be several hundred megabytes.

We might be able to reduce this problem in future implementations by discarding intermediate results from layers where we are unlikely to split. Since the largest layers also the least likely split-layers, this might have a significant impact. However, this comes at the cost of increased computational complexity if we were to split in those layers.

6.2 LP-solver

The LP-solver phase is relatively time-consuming for smaller networks but less significant for larger networks. In our implementation, we are using the Gurobi LP-solver with the Simplex algorithm. The Simplex algorithm has exponential complexity; however, it is usually much faster in practice. The actual runtime for solving LP-systems was generally quite low compared to the total runtime of the algorithm. Profiling usually revealed that less than 10% of the time was spent on solving these systems. The memory usage should also be small compared to symbolic interval propagation.

In practice, the most time-consuming part of the LP-solver was not solving, but adding and removing constraints. In some cases, we even observed close to 40% of the run-time was spent on adding and removing constraints; however, this number was much smaller for the larger convolutional networks. This operation should theoretically be fast since we are only adding and removing parameters from a constraints matrix. The overhead is probably because Gurobi repeatedly has to call its c-backend from python to change these constraints.

6.3 Local search

The local search is implemented through gradient descent, and the complexity for each step is the same as a standard back-propagation step during neural network training. Since we fixed the number of steps at five, and only ran local search every five depths of branching, this part of the algorithm is relatively fast. During our experiments, only a few percents of the run-time was spent on the local search. The computational complexity is dominated by calculating the gradients from the matrix-multiplications, which has a quadratic complexity in the layer size. Again the memory usage is unnoticeable compared to symbolic interval propagation.

6.4 Refinement

The refinement stage is done by branching, and this introduces exponential complexity to our algorithm. While experimenting, we noticed that even small changes to the branching heuristic resulted in considerable changes in performance. Choosing the correct node for branching by using a good branching strategy and heuristic might be the most important part of an efficient implementation. This combined with only recalculating the necessary bounds and LP-constraints at each branch significantly reduces the average runtime.

Another important consideration is that a naïve implementation of branching can result in exponential memory usage. To reduce the memory requirements, we use an efficient queue structure with each branch storing a minimal amount of information, as explained in section 5.2.3. This significantly reduces the memory requirements compared to the naïve solution of storing all symbolic bounds for each branch.

6.5 Multiprocessing

The implemented multiprocessing solution has a close to linear speed-up in the number of processes for difficult verification problems. However, there is some initialisation and communication overhead, leading to a speed-up less than linear. First of all, as explained in 5.6, there is some initialisation delay before all processes get assigned branches. Secondly, each time a process gets a new branch from the multiprocessing queue, it has to recalculate all symbolic intervals and LP-solver constraints instead of just modifying a few.

For larger networks, the main challenge of full CPU-utilisation is the increased memory consumption. The memory requirements do also scale close to linearly with the number of processes. As explained in section 6.1, the memory requirements of the symbolic interval propagation are significant. For the largest networks used in our experiments, we had to limit the number of processes to avoid out of memory exceptions.

6.6 Summary

The symbolic interval propagation phase as an asymptotically larger computational and memory-complexity than standard forward propagation in neural networks.

This, combined with the fact that branching introduces an exponential complexity, underlines the importance of an efficient implementation.

For smaller networks, our algorithm is currently CPU-bound. Fortunately, the worst-case computational behaviour is rarely seen in practice. First of all, through smart branching strategies, and since many nodes often only operate in the linear area, we often only have to branch on a few nodes. Secondly, we do not have to recalculate the whole interval-propagation phase for each branch.

For larger networks, our algorithm is memory-bound. Most of the memory-consumption is a result of storing all intermediate calculations in the symbolic interval propagation phase. It might be possible to significantly improve this situation in the future by not storing intermediate results for layers where we usually do not split nodes, such as large layers.

Chapter 7

Experimental Results

In this chapter, we present the experimental results of our algorithm. Most of our comparisons are made against Neurify (Wang et al., 2018a) since this is the only complete verification algorithm that has proven verification results on large networks with high-dimensional inputs. Neurify only supports ReLU activation functions, so we use the sound but incomplete DeepPoly (Singh et al., 2019) algorithm for comparisons with Sigmoid and Tanh activation functions.

7.1 Choice of experiments

Since this project focuses on verification of neural networks with high-dimensional input, we have chosen networks trained on image datasets for our experiments. The first dataset is the MNIST-digit dataset (LECUN). This set contains one-channel images of hand-written digits from 0 to 9. Each image has 28×28 pixels, so the input dimension is 784.

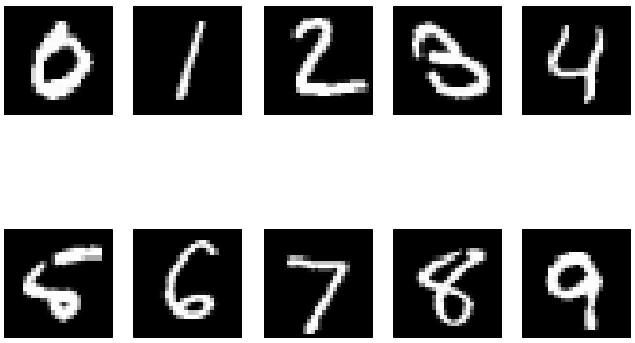


Figure 7.1: Examples of MNIST-digit images

For our MNIST experiments, we used the same networks as in (Wang et al., 2018a). Three of these networks are fully connected, with two layers each and 24, 50, and 512 nodes in each layer for the three networks, respectively. Furthermore, they also used a convolutional network with two convolutional layers and two fully-connected layers. The first convolutional layer has 16 kernels, and the second has 32 kernels, both of size $(4, 4)$. The following fully-connected layers have 100 and 10 nodes. The total number of ReLU nodes in the convolutional network is 4804.

To the best of our knowledge, these networks have been trained without regularisation. Also, no modern techniques such as batch-normalisation, dropout, or skip-connections have been used.

In addition to the MNIST experiments, we trained a new network on the Cifar10 (Krizhevsky et al., 2014) dataset. The Cifar10 dataset is a dataset of 32×32 3-channel colour images and each image belongs to one of ten classes.

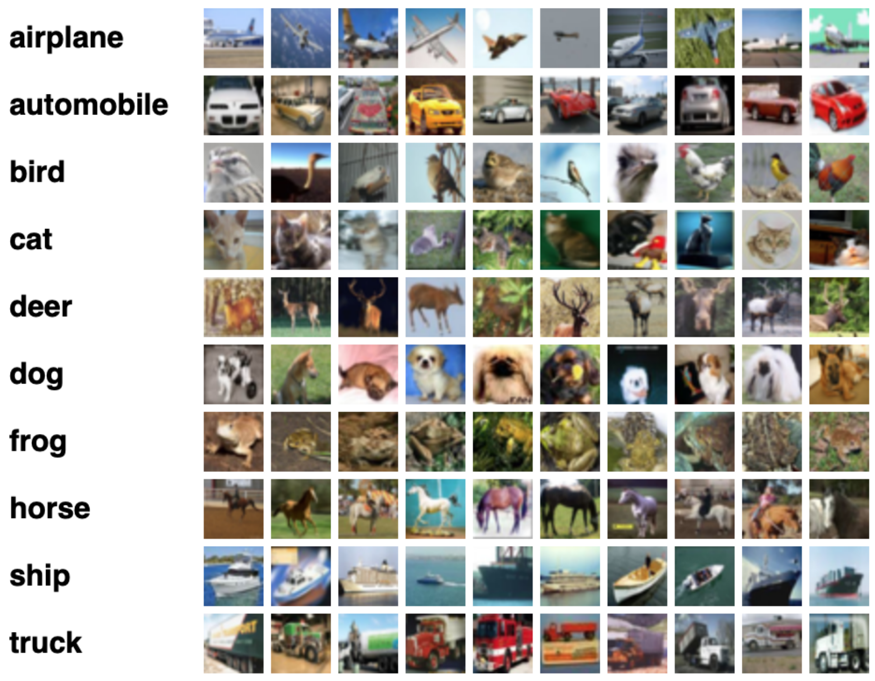


Figure 7.2: Examples of Cifar10 images with classes. https://www.cs.toronto.edu/ kriz/cifar.html

This dataset is considerably more challenging than the MNIST dataset. State of the art networks achieve 95-96% accuracy; however, they often have millions of nodes.

We trained a medium-sized network, with seven convolutional layers, six batch normalisation layers, and a total of 55616 ReLU nodes. We used a small l_1 regularisation of $2 * 10^{-6}$ and a learning rate scheduler with step-sizes $10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$ for $20, 80, 30$ and 20 epochs respectively. Furthermore, we used data augmentation with random flips, cropping, rotation, brightness, and contrast changes. The training was done with the ADAM optimiser (Kingma and Ba, 2014) and our final test set accuracy is 85.56%.

To our knowledge, this is the largest network ever used for experiments with complete verification algorithms. The second-largest image classification network is the convolutional MNIST network with 4 layers and 4804 ReLU nodes previously mentioned. The same paper also did experiments with a 10 276-node convolutional network trained on the DAVE dataset. However, the DAVE dataset is a dataset of images used for steering self-driving cars, not image classification.

7.2 Experimental setup

We ran all experiments with the MNIST fully connected networks on a server with Intel Core i9 9900X 3.5 GHz 10-core CPU and 128 GB ram. Due to the extended run-time of the experiments, we used a second server with a Ryzen 3700X 3.6 GHz 8-core CPU and 64 GB ram for the rest of the experiments.

Both our implementation and Neurify use OpenBLAS as a back-end for linear algebra operations. Neurify has limited OpenBLAS to 1 thread, so we did the same for our implementation. For Neurify, we did not see a significant change in runtime with more threads, and our implementation was actually slower with more OpenBLAS threads.

All experiments are done with $l_\infty \leq \epsilon$ bounds on the input, verifying that the predicted class does not change for any input within those bounds. The timeout is set to 3600-seconds.

7.3 Ground truth

For most of our experiments, we have decided not to use the ground truth targets from the datasets. Instead, we run all inputs through our trained networks, and use the network output as the correct class when doing the verification. We have two reasons for this, first of all, this is how it is done in Neurify (Wang et al., 2018a) and we wanted to do as few changes as possible to Neurify's source code for our comparisons. Secondly, if we use the ground truth labels, we would have to discard all inputs misclassified by the network since finding a counterexample for these inputs would be trivial, we could just return the input as it is already a counterexample.

The exception is for the Sigmoid/ Tanh experiments, where we compare against DeepPoly (Singh et al., 2019). For these experiments, we use the ground-truth labels from the dataset and skip images misclassified by the network.

7.4 Changes made to Neurify source code

We made some small changes to the Neurify source code. Most of these changes are made for convenience, such as commenting out excessive printing during benchmarking. However, we made two changes affecting the results.

First of all, Neurify adds the split constraints: $eq_i(\boldsymbol{x}) > 0$ and $eq_i(\boldsymbol{x}) < 0$ for the upper and lower branch respectively. This is correct for the fully connected networks since Neurify always splits from the first layers. However, for the convolutional network, Neurify skips the first convolutional layers, so the error from the error-based symbolic interval propagation is not 0 at the split nodes. The correct split constraints in this case should be: $eq_i(\boldsymbol{x}) + \sum_{i|\boldsymbol{\epsilon}_i > 0} \boldsymbol{\epsilon}_i \geq s$ and $eq_i(\boldsymbol{x}) + \sum_{i|\boldsymbol{\epsilon}_i \leq s} \boldsymbol{\epsilon}_i < 0$. No official fix for this has been published at the time of writing; however, we have made our best effort to implement this fix into Neurify. Secondly, the timer in Neurify resets after doing the first phase of symbolic interval propagation. We commented out the line resetting the timer.

All changes are well-documented in our repo. At the relevant places in the source code, we have added a comment: ”//Changed code: ....”, explaining the changes and why they are done.

7.5 MNIST Fully connected

The MNIST fully connected experiments are done on the same networks used in (Wang et al., 2018a). We tested all three networks with 100 images from the test-set with $l_\infty \leq \epsilon$ and $\epsilon \in \{1, 2, 5, 10, 15\}$, except for the largest network where we only used 50 images for $\epsilon = \{5, 10, 15\}$. The results for each experiment are given in tables at the end of this section. In figure 7.3, we have plotted the time spent on verification for data points verified before timeout by either Neurify or VeriNet. Neurify failed to terminate after timeout for 6/1350 images; these images have not been included.

The MNIST fully-connected experiments clearly illustrate the advantage of the local search. VeriNet found all unsafe cases Neurify found, and more. Also, VeriNet found all but one of the unsafe cases without branching. The last case was found at depth 5. For our 1344 test cases, VeriNet found a total of 454 unsafe cases with a total runtime of 9.66 seconds. Neurify found 444 unsafe cases in 6022.68 seconds. One of the unsafe cases Neurify did not find reported underflow, the other 9 timed-out. Since the timeout was 3600 seconds, the minimum time it would take Neurify to find the same unsafe cases is 38422.68 seconds. So the lower bound on the speed-up is $\times 3977.5$. Since VeriNet only branched once, almost all of this speed-up is due to the local search.

The effect of adaptive splitting is more ambiguous for these experiments. While it seems like VeriNet finds most cases that require splitting faster, we also have four

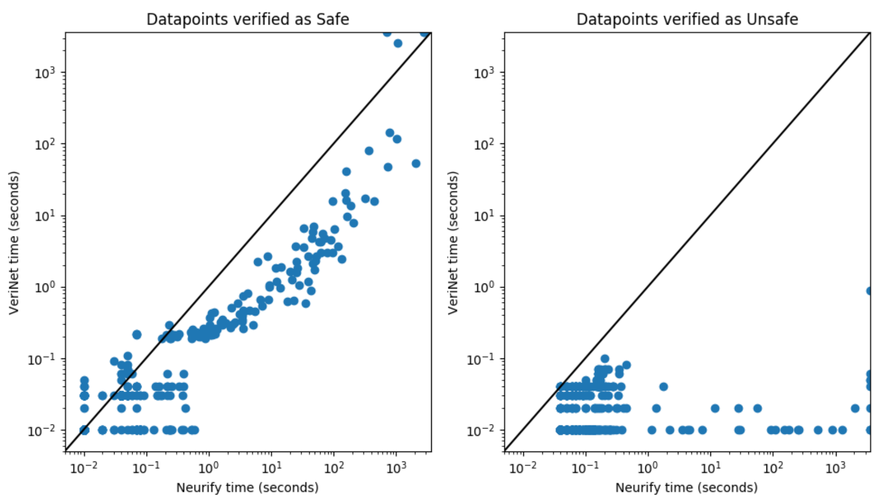


Figure 7.3: Time spent on verification for VeriNet vs. Neurify on the MNIST FC networks. The left plot is of the images verified as safe (808), and the right plot is of the images verified as unsafe (454). Notice the log-scale on the axes. The ”line” patterns seen for the images verified quickly are due to our time-resolution of 1/100 seconds.

difficult cases where Neurify is faster and three of these cases timed-out in VeriNet. It is also challenging to determine how much of the time difference is due to the splitting strategy compared to other factors. However, the real advantage of adaptive splitting becomes evident in the convolutional experiments.

It is interesting to notice that all four outliers have a long runtime. As we explained in section 3.3, our splitting heuristic might underestimate the value of splitting nodes in the early layers, while Neurify always splits from the first layer. However, we expected this effect to be more evenly distributed among data points with different runtimes. Our best guess is that almost all ReLU nodes operating in the non-linear area have to be split for most branches in these difficult problems. If this is the case, splitting from the first layer has the advantage that it improves the bounds of the nodes in the second layer, which might lead to more nodes operating in the linear area without the need for splitting.

For all experiments, VeriNet was able to verify at least as many cases as Neurify and usually more. In the tables below, we measured the average speed-up for all cases were neither VeriNet nor Neurify timed out. For the simplest problems with $\epsilon \in \{1, 2\}$, all 100 datapoints were solved within a few seconds. In these experiments, the algorithm usually did not have to branch, and the speed-up is mainly due to smaller initialisation overhead, not algorithmic differences. For the 48 and 100 ReLU node networks with $\epsilon \in \{5, 10, 15\}$, VeriNet was usually significantly faster, with a speed-up of $\times 12 - 29$. The one exception is the 100 Node ReLU network, with a speed-up of $1.65\times$. For the largest network, most cases are either solved trivially in $<< 1$ second or timed-out.

7.5.1 48 ReLU node network

		VeriNet				Neurify				
ϵ	Total	Safe	Unsafe	Undec	time	Safe	Unsafe	Undec	time	speed-up
1	100	95	5	0	1.21s	95	5	0	2.02s	$\times$1.67
2	100	90	10	0	1.32s	90	9	1*	3.45s	$\times$2.61
5	100	73	27	0	13.88s	73	27	0	167.56s	$\times$12.07
10	100	24	76	0	21.36s	24	76	0	606.57s	$\times$28.40
15	100	8	92	0	72.38s	8	92	0	1977.83 s	$\times$27.32

The time meassurements are for datapoints where neither VeriNet nor Neurify timed-out
*) Neurify reported underflow for image 13, probably due to numerical precision.

The results for the 48-ReLU node network are as expected. The most straightforward cases with $\epsilon \in \{1, 2\}$ are almost all solved without branching by both algorithms. The speed-up for these experiments is mainly due to a somewhat lower initialisation overhead for our implementation. For the more difficult problems, we achieve a significant speed-up. Most of this is due to the local search of VeriNet finding all unsafe cases without branching; however, we also have a speed-up for all non-trivial safe cases. Neurify returned underflow for one image, determined to be unsafe by VeriNet. This is most likely due to floating-point precision and might have been avoided by using 64-bit floating-point numbers instead of 32-bit.

7.5.2 100 ReLU node network

		VeriNet				Neurify				
ϵ	Total	Safe	Unsafe	Undec	time	Safe	Unsafe	Undec	time	speed-up
1	100	97	3	0	1.33s	97	3	0	1.5s	$\times$1.13
2	100	93	7	0	1.91s	93	7	0	4.96s	$\times$2.60
5	100	78	22	0	243.33s	78	22	0	3490.83s	$\times$14.35
10	99*	25	71	3	340.11s	27	69	5	5096.92s	$\times$14.99
15	99*	4	89	6	2605.69s**	4	86	9	4303.44s	$\times$1.65**

The time meassurements are for datapoints where neither VeriNet nor Neurify timed-out
*) Neurify didn't terminate after time-out for at least one image. These images have not been included.
**) VeriNets runtime is dominated by one image running for 2589 seconds. The same image is solved in 1066.18s by Neurify.

The results for the 100-ReLU node network follow much the same pattern as the 48-ReLU node experiments. However, there is one interesting difference. For $\epsilon = 15$ we have a speed-up of only 1.65. VeriNet's runtime is dominated by a single data point, running for 2589 out of 2606 seconds. Neurify spends 1066.18 seconds on the same data point. This illustrates a weakness of using the average speed-up as a metric. Especially for the difficult problems, the runtime might be dominated by very few, or even one data point.

7.5.3 1024 ReLU node network

		VeriNet				Neurify				
ϵ	Total	Safe	Unsafe	Undec	time	Safe	Unsafe	Undec	time	speed-up
1	100	98	2	0	6.11s	98	2	0	11.75s	$\times$1.92
2	100	95	5	0	3.37s	95	5	0	5.34s	$\times$1.61
5	47*	26	6	15	27.03s	27	5	15	155.1s	$\times$5.74
10	49*	0	14	35	0.77s**	0	14	35	2.60s**	$\times$3.38**
15	49*	0	24	25	2.12s**	0	22	27	3.92s**	$\times$1.85**

The time meassurements are for datapoints where neither VeriNet nor Neurify timed-out
With epsilon 10 and 15 we had to reduce the number of threads for neurify to 7 and 4 respectively, to not run out of memory.
*) Neurify didn't terminate after time-out for at least one image. These images have not been included.
**) All cases were solved trivially in $<< 1$ second or at least one of the solvers timed-out.

The results for the largest MNIST fully-connected network are less interesting. For $\epsilon = \{1, 2, 10, 15\}$ all data points were either solved trivially in $<< 1$ second, or they timed out. The results with $\epsilon = 5$ are somewhat more interesting; however, the average time is still dominated by very few data points. A fully connected network of this size would probably see a relatively large gain from using l_1 or l_2 regularisation during training. This could also make verification easier since smaller weights would lead to tighter bounds and less overestimation in the linear relaxations.

7.6 Convolutional MNIST

Similar to the fully-connected experiments, we ran experiments with $l_\infty < \epsilon$ bounds on the input and $\epsilon \in \{1, 2, 5, 10, 15\}$. We used 50 images for each ϵ value. The results are presented in the table below.

With this network, Neurify starts splitting at the first fully connected layer, skipping both convolutional layers. Since the convolutional nodes are never split, Neurify is not complete for this network, and unable to solve several cases. Neurify reports ”underflow” when no solution can be found after splitting all nodes.

The speed-up for these experiments is not very interesting. For simple problems where the solution is found without branching, VeriNet finds the solution in ≈ 0.15 seconds, while Neurify solves it in ≈ 1 second. Since symbolic interval propagation is the most time-consuming part for large networks, much of the speed-up can be explained solely by our more efficient implementation.

The most interesting part of these experiments is the number of undecided cases. Since Neurify does not split in the convolutional layers, it can not solve the more difficult cases. All of Neurifies unsolved cases are due to underflow. VeriNet is complete and solves several non-trivial cases.

		VeriNet				Neurify				
ϵ	Total	Safe	Unsafe	Undec	time	Safe	Unsafe	Undec	time	speed-up
1	50	49	1	0	5.41s	49	1	0	30.45s	×5.63
2	50	49	1	0	5.96s	49	1	0	42.81s	×7.18
5	50	43	2	5	4.61s	37	2	11*	102.94s	×22.33
10	50	1	6	43	3.57s	1	4	45*	42.08s	×11.79
15	50	0	19	31	1.89s	0	12	38*	30.3s	×16

*) All of Neurifiy's undecided images reported underflow; none reached timeout.

7.7 Convolutional Cifar10

The Cifar10 network, as described in section 7.1, is, to the best of our knowledge, the largest network ever used in completed verification. However, we still managed to verify several non-trivial properties of the network, as shown in the table below. All unsafe properties were determined without the need for branching. The Cifar10 network was much more prone to adversarial examples than the MNIST networks. Already with a very-small epsilon of 0.05, we found unsafe datapoints.

For the Cifar10 experiments, our solver was memory-bound, compared to CPU-bound for all other experiments. We had to reduce the maximum number of processes to 6 to avoid out-of-memory exceptions. The reason for this is discussed in chapter 6, and these experiments indicate that improving memory performance might be essential in the future.

ϵ	Total	Safe	Unsafe	Undec
0.05	50	49	1	0
0.1	50	43	5	2
0.2	50	37	9	4
0.5	50	0	13	37
1	50	0	23	27

Results for the 55616- ReLU node Cifar10 classification network.

7.8 Sigmoid and Tanh

To the best of our knowledge, there are no other verification algorithms with an iterative refinement step for the Sigmoid and Tanh activation function. Instead, we compare against DeepPoly (Singh et al., 2019), a sound but incomplete algorithm. Since DeepPoly does not find counterexamples, we only compare the number of cases determined to be safe. We ran the experiments on the same network and images as used in the original paper. The networks are trained with PGD (Dong et al., 2018) adversarial training.

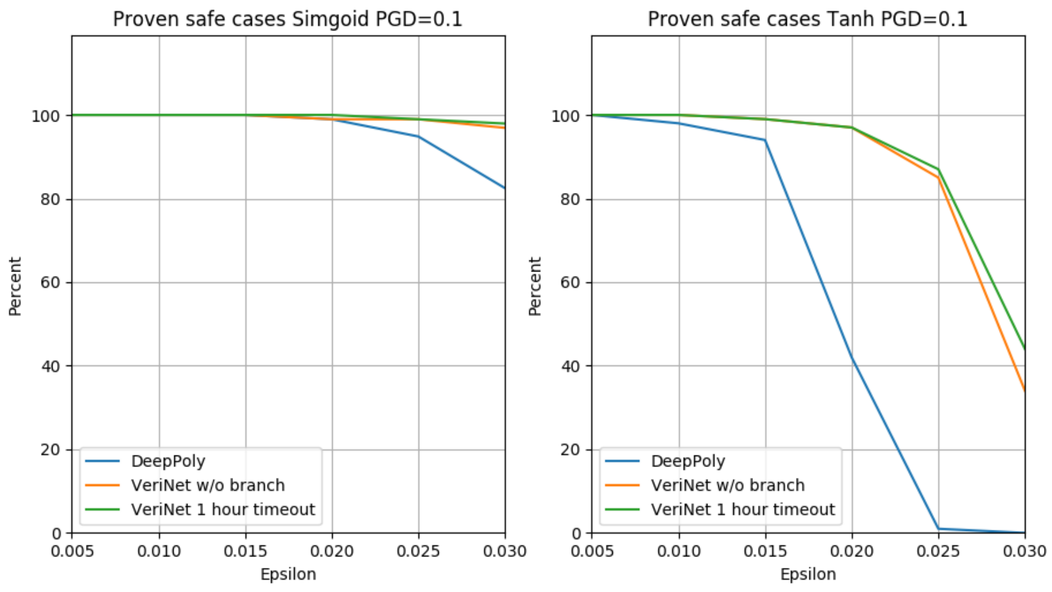


Figure 7.4: Cases verified as safe by VeriNet and DeepPoly

VeriNet determines a significantly larger amount of safe-cases, even without branching. For cases where our algorithm did not have to branch, it used at most 0.28 seconds, and usually 0.17-0.18 seconds. DeepPoly used more than 8 seconds for all cases. However, DeepPoly is sound with respect to floating-point arithmetic, and as we discussed in section 5.2.2, our algorithm is not.

Many of the extra safe-cases we found compared to DeepPoly can probably be explained by our improved linear relaxations from section 3.5.1. However, the floating-point soundness of DeepPoly also factors in here, and it is difficult to say how much.

We also compared our improved relaxations with the naïve relaxations introduced in (Singh et al., 2018), as explained in section 2.6.4. We ran VeriNet without branching for both relaxations and plotted the results in figure 7.5. VeriNet manages to verify significantly more cases with our improved relaxation. For $\epsilon = 0.03$ the naïve relaxation can only verify 23 out of 97 cases, while the improved relaxation verifies 95 cases. Moreover, the improved version is also only marginally slower, usually using 0.17-0.18 seconds for each case, while the naïve method uses 0.16-0.17 seconds.

7.9 Marabou

During our project, a new verification framework, Marabou (Katz et al., 2019) was published. All of the experiments in their paper are done on networks with low-dimensional input. Marabou achieved significantly better results than Planet (Ehlers,

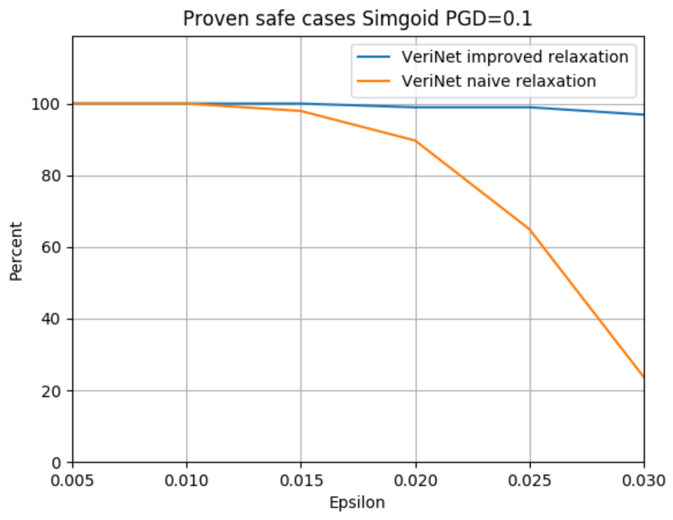


Figure 7.5: Cases verified by VeriNet with the naïve and improved Sigmoid relaxations

2017) and Reluplex (Katz(B) et al., 2017), and comparable results to ReluVal (Wang et al., 2018b). However, they did not test against Neurify, so we decided to do some experiments against Marabou on the MNIST networks. Marabou has two distinct modes, a standard single-threaded mode, and a parallel divide-and-conquer mode, as explained in section 2.6.2. We ran it with both modes and the results are plotted in figure 7.6. Marabou only supports linear constraints on the output, and as discussed in section 2.2 the general robustness verification problem can not be encoded as linear constraints. We solved this by running the algorithm once for each output. VeriNet and Neurify implement this verification problem natively; however, in the background both algorithms employ a similar process.

The current version of Marabou does not seem to scale well with the robustness verification problem and high-dimensional input networks, and Marabou timed out for several images with $\epsilon = 15$ in the standard mode and $\epsilon = 5$ in the divide- and-conquer mode. Since Neurify was clearly faster than Marabou for these networks, we decided not to continue the experiments with Marabou.

The current implementation seems to focus on low-dimensional inputs, and the framework stands out as a very well-designed foundation for future improvements. Optimizations for high-dimensional input networks might significantly improve these results in the near future.

7.10 Possible experimental weaknesses

We have done our best to design experiments reflecting the true performance of our solver, and resulting in fair comparisons. However, we have to keep in mind that this

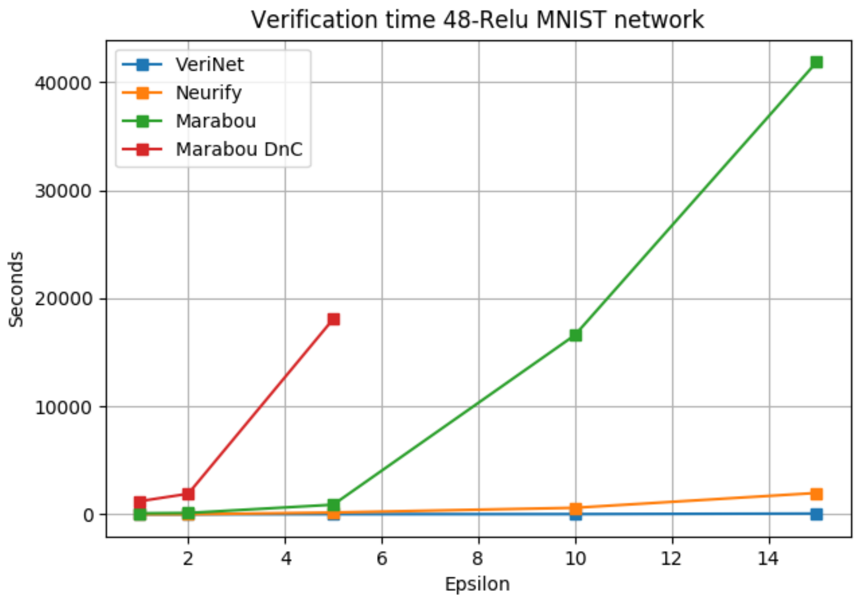


Figure 7.6: Comparing the time spent verifying 100 images on the 48-ReLU node MNIST network for VeriNet, Neurify and Marabou

field of research is new, and there doesn't seem to be a consensus on what kind of experiments should be used for benchmarking. For our experiments, several factors should be taken into account.

First of all, neural networks come in a wide variety of architectures. There is no guarantee that verification algorithms doing well on classification networks also perform well on other networks. Unfortunately, the long runtime for the experiments limits the number of networks we can use.

Our experiments also indicate that a vast majority of properties are either solved trivially in $<< 1$ second or time-out. Using average speed-up as a metric in comparisons can lead to a few data points dominating the timing results. Significantly increasing the number of data points to avoid this problem is unrealistic due to the extended runtime.

Furthermore, many neural networks used in verification experiments are designed and trained without modern techniques such as batch-normalisation, dropout, and skip-connections. Experimental results for these networks do not necessarily transfer well to neural networks used in practice. This simple network design seems to be used for two reasons. First, complete verification research of neural networks is still in its beginning stages. Implementing support for modern techniques does currently not seem to be prioritised in this research. Moreover, comparing different algorithms is simpler when most networks are trained with similar techniques. However, we still believe this is a challenge and have tried to improve on this by introducing the Cifar10 network trained with more realistic techniques.

7.11 Summary

Our experiments show that VeriNet can verify more cases within an hour time-out than the present state-of-the-art toolkit, Neurify (Wang et al., 2018a). For images verified by at least one of the two toolkits, we achieve a speed-up around an order of magnitude for non-trivial safe cases and three orders of magnitude for the unsafe cases. Our implementation also generalises better to convolutional networks due to the novel adaptive splitting strategy.

Furthermore, we have been able to do meaningful verification on a large convolutional network trained on the Cifar-10 dataset. This network was trained using modern regularisation techniques and has significantly more nodes than the largest network used for complete verification before this project.

Finally, our experiments on networks using the Sigmoid and Tanh activation functions show that our algorithm generalises well to networks using non-linear activation functions.

Chapter 8

Conclusion

In this project, we have designed a complete verification algorithm for neural networks with high-dimensional inputs and implemented it in a Python library, VeriNet. Our algorithm extends the symbolic interval propagation approach of (Wang et al., 2018b,a) with several novel contributions. The experimental results show a speed-up of about an order of magnitude for most difficult verification problems, and at least 3-orders of magnitude for unsafe cases, compared to the current state-of-the-art verification tool. We have also proven that VeriNet can do meaningful verification with significantly larger and more complex networks than previously verified networks.

8.1 Improvements introduced by our algorithm

Our project extends previous algorithms by adding several novel contributions, most importantly:

- We have shown that the local search step results in a speed-up of several thousand times for finding counterexample over the current state-of-the-art implementation, Neurify (Wang et al., 2018a). For all unsafe cases, except 1, our algorithm found a counterexample without branching, while Neurify timed-out for several of those data points after one hour.
- The adaptive splitting strategy makes our algorithm more general than Neurify in the sense that we can handle any supported network architecture without manually choosing which layer we want to start splitting. With our splitting strategy, VeriNet is also complete for all ReLU networks, while Neurify (Wang et al., 2018a) is only complete when we choose to split from the first layer.
- We have added support for the Sigmoid and Tanh activation functions and batch-normalisation layers. The same approach can also be extended to other activation functions and layers, as long as we can define a linear relaxation for those functions.
- Finally, we tested our algorithm on a realistically trained and designed network used for classification on the Cifar10 network with 55616 ReLU nodes. We

managed to verify several interesting properties of this network, within the one-hour timeout. To the best of our knowledge, the largest network used for experiments with complete verification algorithms so far was in (Wang et al., 2018a), where they used a convolutional network with 10 276 ReLU nodes.

8.2 Future work

Even though we have made much progress during this project, we believe there is still room for significant improvements. We have listed a few of the most interesting areas for future research below. This is a long list, but in no way exhaustive, so we expect to see some exciting research over the coming years.

- As we discussed in chapter 6, for the largest networks, our algorithm is currently memory-bound not CPU-bound. The symbolic interval propagation is the main bottleneck for memory consumption. Our current implementation stores a significant amount of intermediate calculations for efficient recalculations after splitting. Much of this information could probably be discarded, without significant impact on computational performance. One example is that for larger cone-shaped networks, our algorithm performs most splits in later layers. So, the intermediate calculations for the first layers could be discarded since we rarely split in those layers. This comes at the cost of recalculating more symbolic bounds if we were to split in those layers.

- Currently, our algorithm only has comparable good results on networks with high-dimensional input. An interesting topic for future research is to attempt to extend this algorithm to work efficiently with low-dimensional input. As discussed in section 4.2, the most promising way to go is to also split on the input domain. The main challenge of this is to design a good splitting heuristic able to compare the value of splitting an input node and an activation node.

- Our adaptive splitting strategy is not exclusively better than Neurify's (Wang et al., 2018a) hierarchical. For some cases in the MNIST fully-connected experiments, Neurify had a speed-up over VeriNet. While we expected that this might happen, we did not expect to see it happen only for the most challenging cases. Further research into precisely what triggers this behaviour could lead to better splitting heuristics.

- We have extended our algorithm to support new activation functions and batch-normalisation layers. However, there are still several modern techniques, such as Dropout, skip-connections and RNN architectures that are not supported by our current implementation, which could be a goal for future implementations.

- Both VeriNet and Neurify use an LP-solver as part of the algorithm. Other complete verification algorithms, such as Reluplex (Katz(B) et al., 2017) and Marabou (Katz et al., 2019) have designed LP-solvers with support for ReLU activation functions. It would be interesting to look at the possibility of combining the advantages of the different approaches by using one of those solvers, instead of standard LP-solvers.

- As discussed in 5.2.2, we have little knowledge about the effect of rounding errors on symbolic interval propagation. Both our algorithm and Neurify have the option to perform outward rounding when calculating concrete bounds, which guarantees that this operation does not result in invalid bounds. However, several other operations have not been accounted for, including repeated matrix multiplications, which theoretically can lead to significant rounding errors. Future research should look into the necessary conditions to avoid invalid bounds due to rounding errors in practical applications.

- Our local search via adversarial attacks has significantly improved the performance of our algorithm for finding counterexamples. However, we have not looked at how our algorithm compares to more specialised adversarial attack algorithms. It might be possible to further increase the performance of the local search by including more advanced adversarial attacks.

- Finally, our algorithm is still limited in the types of verification problems we can handle. Neurify supports a broader range of verification problems, including l_1, brightness, and contrast based constraints. We believe that we can use many of the techniques from Neurify to add support for these constraints in VeriNet.

8.3 Summary

The main goal of this project was to design an algorithm able to do verification on larger networks, and a broader range of network architectures, than current state-of-the-art algorithms. We feel that we achieved this goal in an excellent way. However, the long list of potential research topics we discovered while working on this project shows that the work in this field has barely started. We believe that research along the paths listed in the previous section, and others, will lead to massive improvements in formal verification of neural networks over the coming years.

Bibliography

Akintunde, M., Lomuscio, A., Maganti, L., and Pirovano, E. (2018). Reachability analysis for neural agent-environment systems. In *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*. pages 1

Akintunde, M. E., Kevorchian, A., Lomuscio, A., and Pirovano, E. (2019). Verification of rnn-based neural agent-environment systems. In *Proceedings of the 33th AAAI Conference on Artificial Intelligence (AAAI19). Honolulu, HI, USA. AAAI Press. To appear*. pages 1

Chinneck, J. W. and Dravnieks, E. W. (1991). Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing, 3(2):157–168*. pages 22

Dong, Y., Liao, F., Pang, T., Su, H., Zhu, J., Hu, X., and Li, J. (2018). Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition,* pages 9185–9193. pages 67

Ehlers, R. (2017). Formal verification of piece-wise linear feed-forward neural networks. *In Deepak D'Souza and K. Narayan Kumar, editors, Automated Technology for Verification and Analysis,* Springer International Publishing. ISBN 978-3-319-68167-2:269–286. pages 1, 2, 6, 8, 9, 21, 22, 26, 45, 68, 84

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. (2018). Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP),* pages 3–18. pages 1, 17

Gurobi Optimization, I. (2019). Gurobi optimizer reference manual. http://www.gurobi.com. pages 48, 50

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition,* pages 770–778. pages 4

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*. pages 2, 5

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. (2019).

The marabou framework for verification and analysis of deep neural networks. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 443–452, Cham. Springer International Publishing. pages 1, 2, 6, 21, 22, 45, 54, 68, 73, 84

Katz(B), G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An efficient smt solver for verifying deep neural networks. *In Proc. CAV17, 97–117*. pages 1, 2, 6, 21, 22, 23, 45, 69, 73

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. pages 52, 61

Kochenderfer, M. J., Holland, J. E., and Chryssanthacopoulos, J. P. (2012). Next-generation airborne collision avoidance system. *Technical report, Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States*. pages 1, 45

Krizhevsky, A., Nair, V., and Hinton, G. (2014). The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55. pages 61

LECUN, Y. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*. pages 60

Numba (2019). Numpy. http://numba.pydata.org. pages 48

Numpy (2019). Numpy. http://www.numpy.org. pages 48

OpenBLAS (2019). Openblas. https://www.openblas.net. pages 2, 48

PyTorch (2019). Pytorch. https://pytorch.org. pages 48

Singh, G., Gehr, T., Mirman, M., Püschel, M., and Vechev, M. (2018). Fast and effective robustness certification. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 10802–10813. Curran Associates, Inc. pages 1, 17, 23, 24, 25, 32, 68

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019). An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30. pages 1, 17, 25, 32, 54, 60, 62, 67

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019). Boosting robustness certification of neural networks. In *2019 ICLR Seventh International Conference on Learning Representations*. pages 25

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2013). Intriguing properties of neural networks. *CoRR*, abs/1312.6199. pages 1, 18

Tjeng, V., Xiao, K., and Tedrake, R. (2018). Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*. pages 1, 2, 6, 23

Wang, S., Pei, K., Justin, W., Yang, J., and Jana, S. (2019a). Neurify. https://github.com/tcwangshiqi-columbia/Neurify. pages 21

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018a). Efficient formal safety analysis of neural networks. In *NeurIPS*. pages 1, 2, 3, 6, 8, 9, 11, 12, 13, 14, 21, 24, 29, 30, 31, 32, 43, 44, 45, 49, 52, 54, 60, 62, 63, 71, 72, 73, 84

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018b). Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 1599–1614, Berkeley, CA, USA. USENIX Association. pages 1, 7, 10, 14, 20, 44, 45, 69, 72

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2019b). Reluval. https://github.com/tcwangshiqi-columbia/ReluVal. pages 21

Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. (2018). Efficient neural network robustness certification with general activation functions. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 4939–4948. Curran Associates, Inc. pages 1, 32, 34

Appendices

Appendix A

User-guide VeriNet

A.1 Installation

VeriNet is dependent on several libraries, and we have used pipenv to manage most of our dependencies.

Installing pipenv and dependencies:

```
$ sudo apt-get install python-pip
$ pip install pipenv
$ cd <your_verinet_path>/VeriNet/src
$ pipenv install
```

If the last command fails, try: "$ pipenv install torch==1.1.0" and rerun: "$ pipenv install"

Two dependencies cannot be installed through pipenv. First of all, we are using the Gurobi LP-solver. Gurobi requires a license; however, they have free licenses for academic use. Secondly, compiling Numpy from source with OpenBLAS might significantly improve the performance.

Installing Gurobi

1. Go to https://www.gurobi.com, download gurobi and get the license.
2. Follow the install instructions from http://abelsiqueira.github.io/blog/installing-gurobi-7-on-linux/
3. Activate pipenv by changing directory to your VeriNet/src folder and typing:
   $ pipenv shell
4. Find your python directory by running:
   $which python
5. Change directory to your Gurobi installation and run:
   $ <your_python_directory> setup.py install

Compilling Numpy with OpenBLAS

Numpy should use the OpenBLAS backend for our project. Using other backends might work but might also create problems with our multi-processing implementation. Even if your operating system has a version of OpenBLAS pre-installed, we recommend compiling Numpy from source with an updated OpenBLAS version. We have noticed a speed-up on important Numpy functionality of an order of magnitude in some systems. Instructions for compiling Numpy with OpenBLAS can be found at:

https://hunseblog.wordpress.com/2014/09/15/installing-numpy-and-openblas/

We have created a small script that can be used as an indication of the Numpy speed. The script can be found in ”VeriNet/src/scripts/test_numpy.py”. Our test-values on a Ubuntu 18.04 system with a AMD Ryzen 3700x processor are:

```
dotted two (1000, 1000) matrices in 45.1 ms
dotted two (4000) vectors in 2.40 us
SVD of (2000, 1000) matrix in 0.668 s
Eigendecomp of (1500, 1500) matrix in 2.340 s
```

Adding VeriNet to Pythonpath

Make sure that VeriNet is visible to python; this can be done using the command:

```
$ export PYTHONPATH=”$PYTHONPATH:<your_verinet_path>”
```

This line can also be added to ~/.bashrc.

A.1.1 Environment variables

To make sure that our library runs as expected, we need to set some environment variables. First of all, pythons multiprocessing doesn’t work well Cuda. To make sure that no library activates Cuda, we hide all Cuda devices. Secondly, our program performs best with one OpenBLAS thread; this is also controlled by an environment variable.

Both variables are set automatically when using our pipenv environment, and can be found in ”VeriNet/src/.env”. If you are not using pipenv, you need to set them manually with:

```
$ export OMP_NUM_THREADS=1
$ export CUDA_DEVICE_ORDER = ”PCI_BUS_ID”
$ export CUDA_VISIBLE_DEVICES = ””
```

A.2 Usage

A.2.1 Running the program

Example-runs for the arbitrary, targeted and bounded verification problems using nnet files can be found in the VeriNet/examples/examples.py file. The file also contains a simple example of how to manually define a neural network instead of using the nnet files. Scripts for all of our benchmarking runs can be found in VeriNet/src/scripts/.

Several hyper-parameters can be specified when initialising VeriNet and calling the verify method. More information can be found in the doc-strings.

A.2.2 Pytorch models

VeriNet uses Pytorch to represent neural networks. Networks should be created by sub-classing the VeriNetNN model found in src.neural_networks.verinet_nn. Each layer of the network is represented by a torch.nn.Sequential object. Each sequential object should contain a layer, and optionally an activation function. The sequential objects are stored as a list in the class parameter "self.layers". It is important not to overwrite VeriNetNN's forward method.

Currently supported activation functions are: torch.nn.Sigmoid, torch.nn.Tanh, and torch.nn.Relu. Supported layers are torch.nn.modules.linear.Linear, torch.nn.Conv2d, and torch.nn.BatchNorm2d.

An example of this can be found in src/neural_networks/cifar10.py.

A.2.3 nnet models

The ".nnet" is a human-readable format for storing neural networks and has been used by several verification algorithms before us. It was first introduced for the ACAS Xu network, https://github.com/sisl/NNet. We do also support the nnet format; however, we had to make some minor modifications since the original version does not support convolutional or batch-normalisation layers. The documentation for our nnet format can be found in the readme in /src/data_loader.

The NNET class from src.data_loader.nnet is used to handle all functionality for the nnet format. The class defines five public methods:

```
init_nnet_from_file(path: str)
init_nnet_from_verinet_nn(model: VeriNetNN,
                          input_shape: np.array,
                          min_values: np.array,
                          max_values: np.array,
                          input_mean: np.array,
                           input_range: np.array)
from_nnet_to_verinet_nn()
write_nnet_to_file(filepath: str)
normalize_input(x: np.array)
```

The first method is used to load a NNET model from a file. The second method is used to create a NNET model from a Pytorch model. The third converts the NNET model to a Pytorch model, and the fourth saves the NNET model to a file. Notice that when creating a NNET model from a Pytorch model, we have to specify several parameters. These parameters are meant to be used for normalisation, and our last method uses these parameters to normalise a given input. More details can be found in the methods doc-strings.

A.2.4 The Verification objective

Before running VeriNet we have to define the verification objective. We have implemented support for three different verification tasks, each represented by a class. The classes can be found in src.algorithm.verification_objective.

```
ArbitraryAdversarialObjective(correct_class: int,
                              input_bounds: np.array,
                              output_bounds: np.array=None,
                              output_size: int=None
                              )
TargetAdversarialObjective(correct_class: int,
                           target_class: int,
                           input_bounds: np.array,
                           output_bounds: np.array=None,
                           output_size: int=None
                           )
BoundedOutputObjective(input_bounds: np.array,
                       output_bounds: np.array=None,
                       output_size: int = None
                       )
```

The first, ArbitraryAdversarialObjective, is used for classification problems checking that a input within some given concrete bounds is never misclassified. The second, TargetAdversarialObjective, is the same; however we only check that the output for the given target class is never larger than the correct class. Finally, BoundedOutputObjective, checks that the output stays within given concrete bounds. The given input-bounds should be in the same shape as the input of the neural network, with one added axis at the end of size 2. Index 0 of the last axis should contain the lower bounds, index 1 should contain the upper bounds. More details can be found in the classes doc-strings.

New verification objectives can be added by sub-classing the abstract class VerificationObjective in src.algorithm.verification_objective.

A.2.5 Activation functions

The activation functions are defined in src.algorithm.activation_function_abstractions. New activation functions can be added by sub-classing ActivationFunctionAbstraction, defining all abstract methods (most notably the linear relaxation) and updating the activation_map dictionary returned by
ActivationFunctionAbstraction.get_activation_mapping_dict().

A.2.6 Benchmarking runs

We have made python scripts to run all the benchmarks from this report. The scripts can be found in /src/scripts.

Appendix B

Independent Study Option (ISO)

Prior to this project, I did a literature review of complete verification algorithms for neural networks as an ISO. We cover some of the same algorithms in the background section; however, the presentation significantly differs as we have tailored it towards our application. For full disclosure, I'll still give a detailed account of the individual chapters that contain some thematic overlap here.

Section 2.1 and 2.2 provide some background information on neural networks and the verification problem, which were also relevant to the ISO. However, there is no overlap in the presentation. In the ISO, I used a completely different graph-based notation for neural networks and did not cover convolutional layers, Sigmoid, Tanh and batch normalisation. The definition of the general verification problem in 2.2 differs in that we require concrete lower and upper bounds for the input, and linear constraints for the output while the ISO handles a more general verification problem. The robustness verification problems are also new to this report.

The first paragraph in section 2.3 informally explains the difference between complete and sound algorithms and a similar distinction was made in the ISO. Section 2.4 provides a brief recap of traditional solvers, which we also did in the ISO.

The ISO briefly covered both linear relaxations presented in section 2.5.1 together with their associated algorithms (Ehlers, 2017; Wang et al., 2018a). We have extended on this by adding a general definition for linear relaxations. We also explain in more detail why relaxations are used, compared to the ISO only focusing on how they are used in their respective algorithms.

The naïve and symbolic linear relaxations from section 2.5.2 are also both mentioned in the ISO. However, my coverage here is significantly more detailed, we provide all necessary formulas for propagation, and we added examples and illustrations. The error-based symbolic interval propagation was not mentioned at all in the ISO.

The complete algorithms from section 2.6.1 to 2.6.3, except (Katz et al., 2019), were covered in detail in the ISO. The presentation here is much more concise and targeted more towards our use.

The rest of the report does not have any similarities with the ISO.