Masterarbeit im Fach Informatik
Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Proving Non-Existence of Imperceptible Adversarial Examples in Deep Neural Networks Using Symbolic Propagation With Error Bounds

26. August 2020

vorgelegt von:
Christopher Jan-Steffen Brix
Matrikelnummer 343982

Gutachter:
apl. Prof. Dr. T. Noll
Prof. Dr. B. Leibe

Betreuer:
apl. Prof. Dr. T. Noll

Erklärung

Brix, Christopher Jan-Steffen 343982
Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit~~/ Masterarbeit* mit dem Titel

Proving Non-Existence of Imperceptible Adversarial Examples in Deep Neural Networks Using Symbolic Propagation With Error Bounds

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 26. August 2020 _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 26. August 2020 _____

Ort. Datum Unterschrift

Acknowledgements

I would like to express my sincere gratitude to Prof. Thomas Noll, who not only allowed me to write this very interesting thesis but also supported me at every step with valuable feedback and guidance. Likewise, my thanks go to Prof. Joost-Pieter Katoen for letting me write this thesis at his chair and Prof. Bastian Leibe for agreeing to be the second reader and referee. I would like to thank Shiqi Wang, the author of Neurify, for all the helpful discussions and explanations. This thesis would not have been possible without the computational cluster at the chair of software modeling and verification.

I am forever grateful to my parents, Ulrich and Ulrike, as well as my brother, Robin, for their love, encouragement, and tremendous support throughout the duration of my studies.

Abstract

Neural networks are commonly used in safety-critical real-world applications. Unfortunately, the predicted output is often highly sensitive to small, and possibly imperceptible changes to the input data. Proving that either no such adversarial examples exist, or providing a concrete instance, is therefore crucial to ensure safe applications. As enumerating and testing all potential adversarial examples is computationally infeasible, verification techniques have been developed to provide mathematically sound proofs of their absence using overestimations of the network activations. This thesis proposes, implements, and evaluates an improved technique for computing tight upper and lower bounds of these node values, based on increased flexibility gained by computing both bounds independently of each other. Furthermore, additional speedups are gained by re-implementing parts of the original software „Neurify“. Combined, these adaptations reduce the necessary runtime by up to 94%, and allow a successful analysis for networks and inputs that were previously too complex. Additionally, bounds for max-pooling operations in convolutional networks are proposed and evaluated, directing potential future work. Finally, ablation studies for the chosen floating point precision and the selection algorithm for splitting of overestimated nodes are performed. The resulting software „Debona“ is open sourced to ensure widespread usability and to advance future research.

Contents

Chapter 1

Introduction

Enabled by faster hardware and major theoretical breakthroughs, neural networks have become increasingly popular for many real world applications, such as autonomous driving (Bojarski et al., 2016), healthcare (Esteva et al., 2019), or military use-cases (Wu et al., 2015). As many of those tasks require the solution to be highly accurate, the networks not only need to perform reasonably well for common inputs, but may also have to be *robust* in the face of manipulated data. Szegedy et al. (2013) show that networks can be fooled by *adversarial examples* that, for humans, are indistinguishable from normal input but still cause erroneous output.

To increase the robustness of the networks, it is possible to use *relaxation methods* during training (Dvijotham et al., 2018; Wong et al., 2018). Furthermore, trained networks can be analyzed by heuristic search techniques such as gradient descent (Carlini and Wagner, 2017; Szegedy et al., 2013), evolutionary algorithms (Nguyen et al., 2015), or saliency maps (Papernot et al., 2016) to determine their level of robustness before they are deployed publicly. However, the inability of such heuristics to find adversarial examples is not sufficient proof of their nonexistence, and may not satisfy the need for rigorous tests.

Formal verification techniques therefore attempt to provide mathematical proofs for the validity of defined safety properties. As a manual verification is not feasible, automated toolkits have been developed that can verify a given safety-property, such as the robustness of the network, without human interaction. To this end, they define the space of possible inputs, and propagate it through the network to determine the reachable output space. The output region can then be used to check whether the given safety property holds. Due to the non-linearity of activation functions used in neural networks, this propagation is not trivial, and, in fact, LP-complete (Katz et al., 2017). However, for sufficiently small networks or easy safety properties, a successful verification can be achieved.

As an alternative to an exact propagation of the reachable space using techniques such as *star sets* (Bak and Duggirala, 2017), *over-approximations* can be used to simplify the analyzed search regions. While each such relaxation risks the introduction of spurious adversarial examples, small over-approximations can minimize this risk while significantly speeding up the analysis. *Symbolic* representations of the

input space can be used to further minimize the introduced over-approximations. Wang et al. (2018a) propose one such technique, and implement it in the „Neurify“ toolkit.[1] There, they additionally utilize constraint refinement to split the search problem into two new branches, should the over-approximations introduce spurious adversarial examples. However, their linear relaxation of the activation functions used for their over-approximations is not optimal, as they assume the symbolic upper and lower bounds to be equal except for a scalar delta. As the activation function *ReLU* guarantees each node output to be non-negative, an independent definition of the upper and lower bounds may help to strengthen otherwise weak lower bounds, subsequently reducing the over-approximated output space and speeding up the verification process.

This master thesis provides the mathematical proofs for the proposed new over-approximation technique which is referred to as *zero bounding* as well as bounds for *max-pooling* operations commonly used in convolutional networks. Additionally, the toolkit Neurify has been optimized and amended with zero bounding. The resulting software is open-sourced as „Debona“[2] and provides speedups of up to 94% over Neurify, significantly increasing the rate of successful analyses. A subset of this thesis has previously been published in (Brix and Noll, 2020).

1.1 Outline

Chapter 2 provides a general introduction into artificial neural networks, describing the most popular different architectural building blocks and common applications. Chapter 3 describes Neurify, the tool for network verification that is extended in this thesis, gives proofs for the relevant mathematical techniques, and demonstrates the verification of a simple network. It also provides insights into the implementation specific choices and highlights possible shortcomings. Chapter 4 extends upon the previous work by defining an improved relaxation technique that allows to compute tighter bounds on the value of nodes, strengthening the verification power of the software. Analogue to Chapter 3, an exemplary verification process and implementation details are provided. Based upon those improvements, Chapter 5 details a number of experiments that test the previously introduced changes and shows their positive impact on the precision and efficiency. There, the effects are separated into implementation specific improvements and speedups due to the stronger bounds, allowing a detailed evaluation of their significance. Where applicable, ablation studies are used to further highlight the impact of individual program components. Finally, Chapter 6 summarizes the work and gives a short outlook into possible avenues for future research.

[1] `https://github.com/tcwangshiqi-columbia/Neurify`
[2] `https://github.com/ChristopherBrix/Debona`

Chapter 2

Artificial Neural Networks (ANNs)

This chapter introduces the basic theory behind neural networks as well as popular architectural designs and their application. Section 2.5 motivates the work done for this thesis, demonstrating the vulnerability of networks to adversarial attacks.

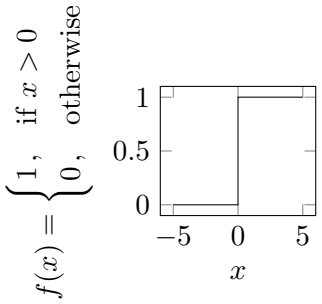
2.1 Neuron

Every neural network consists of a large number of *neurons* that are arranged according to the chosen network structure. As the smallest network unit, each neuron represents a single function, computing an output value for a number of inputs. How many and which inputs are used is defined individually for each neuron by the chosen *architecture* of the network. Concretely, each neuron first computes a weighted sum of all input values before applying an *activation function* and returning its result.

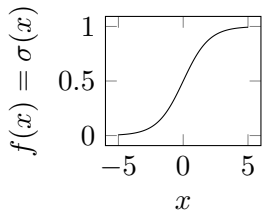
Rosenblatt (1957) proposes the *simple perceptron* as the first network with a single layer that could simulate the AND, OR, and NOT functions. Here, the activation function is defined by a threshold b, and returns a binary output

$$f(x) = \begin{cases} 1\,, & \text{if } wx > b \\ 0\,, & \text{otherwise} \end{cases} = \begin{cases} 1\,, & \text{if } wx - b > 0 \\ 0\,, & \text{otherwise} \end{cases} \tag{2.1}$$

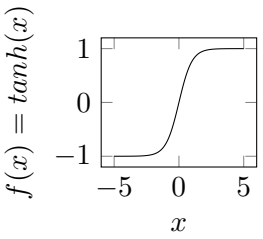
where wx represents the weighted sum of the input values x for weights w. This binary activation function is meant to mimic a real neuron as found in human or animal brains that start to fire if and only if a certain activation value is surpassed. A visualization of this activation function is given in Figure 2.1a. Because each such neuron depends on the weights w and the threshold b, both have to be tuned carefully to perform the desired computation. As modern networks employ millions of neurons, selecting the parameters by hand is not feasible. Instead, they are computed using gradient descent, a technique described in Section 2.4. However, the step function given in Equation 2.1 has a gradient of 0 for all inputs, preventing the parameter optimization. Thus, alternative functions with similar behavior but



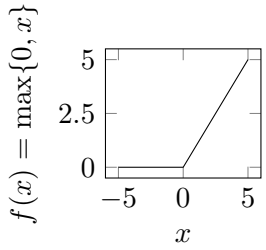
(a) A step function with $w = 1$ and $b = 0$



(b) The sigmoid function



(c) The tanh function



(d) The ReLU function

Figure 2.1: Different activation functions. Both the sigmoid and tanh function are similar to the step function but have non-zero gradients. Figure adapted from (Brix, 2018).

non-zero gradients have been proposed. Popular examples are both the *sigmoid* and *tanh* function. Their similarity with the step function is shown in Figure 2.1.

$$f(x) = \frac{1}{1 + \exp(-wx - b)} = \sigma(wx + b) \tag{2.2}$$

$$f(x) = \tanh(wx + b) \tag{2.3}$$

However, the activation function does not necessarily have to resemble the step function. Glorot et al. (2011) propose the *rectified linear unit* (ReLU, see Figure 2.1d) that gives an output of zero for negative inputs.

$$ReLU(x) = \max\{0, x\} \tag{2.4}$$

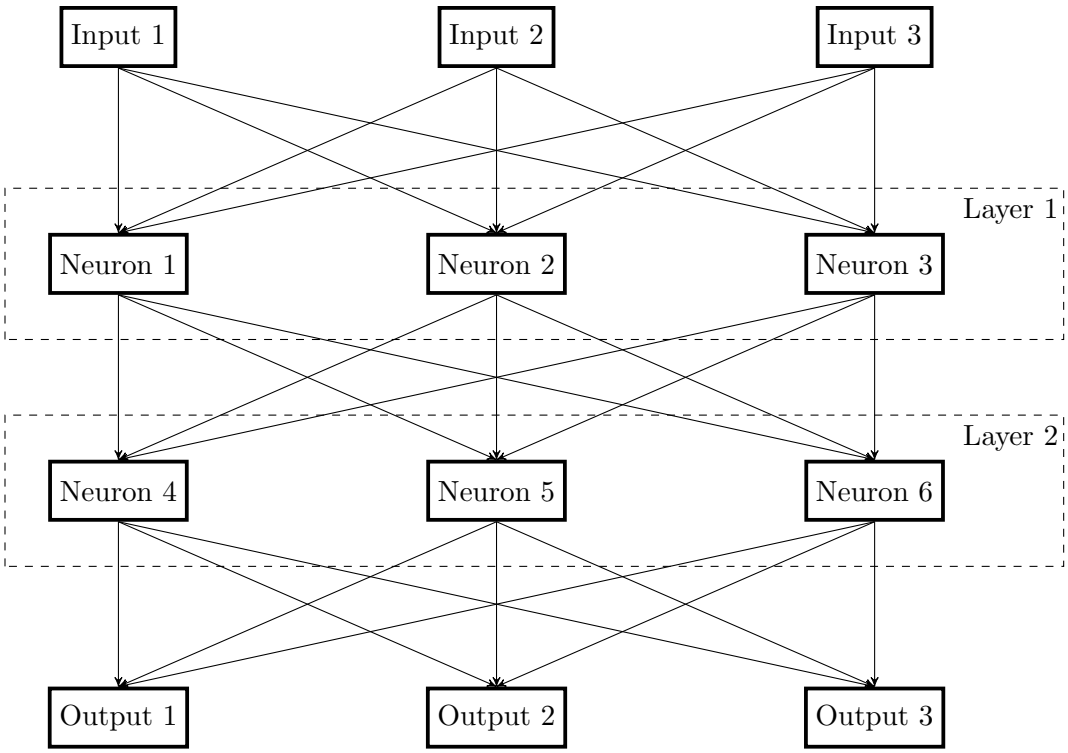


Figure 2.2: A simple feedforward network. Both layers have three neurons, the output of layer one is used as the input to layer two.

Networks using ReLU activation functions have been shown to sometimes outperform models using sigmoidal nonlinearities (Maas et al., 2013). Furthermore, their gradient can be computed efficiently, simplifying the optimization.

$$ReLU'(x) = \begin{cases} 0\,, & \text{if } x \leq 0 \\ 1\,, & \text{otherwise} \end{cases} \tag{2.5}$$

All networks evaluated in this thesis only use ReLU activation functions, as their piecewise linear behavior significantly simplifies their verification.

2.2 Feedforward Neural Networks (FFNNs)

The output of neurons can be used as the input for further nodes to generate a chain of computations. This technique, where neurons are ordered in layers and each layer receives input from the previous while passing on its output as the input for the next, is referred to as a *feedforward* neural network. Figure 2.2 visualizes this

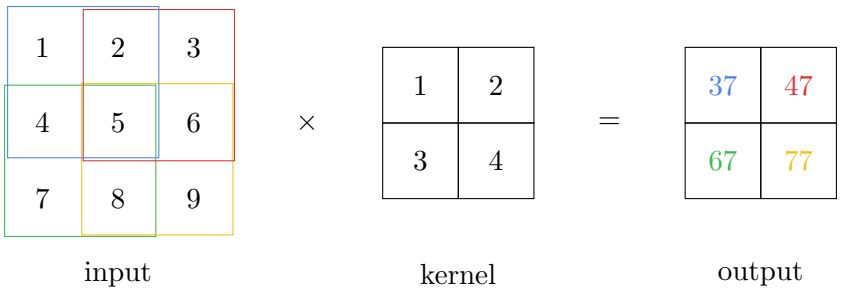


Figure 2.3: A simple convolutional layer. The 2×2 kernel is shifted with a stride of 1×1. Blue: $1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 5 = 37$, red: $1 \cdot 2 + 2 \cdot 3 + 3 \cdot 5 + 4 \cdot 6 = 47$, green: $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 7 + 4 \cdot 8 = 67$, yellow: $1 \cdot 5 + 2 \cdot 6 + 3 \cdot 8 + 4 \cdot 9 = 77$.

architecture. There, the layers are *fully conected*, i.e., each node value is computed as the weighted sum of all outputs from the previous layer.

By using multiple layers, the expressiveness of the network is increased while reducing the number of necessary neurons. Hornik et al. (1989) and Cybenko (1989) show that a single *hidden* layer, ie., two layers of computation, are sufficient to approximate any continuous function. However, as this potentially requires an infeasible large amount of neurons, more layers are commonly used.

2.2.1 Convolutional Layers

Lecun and Bengio (1995) have proposed convolutions as alternatives to fully connected layers. Here, the network does not specify a weight for each individual input, but only for a small subset. This subset is selected by moving a window over the input according to the chosen *stride*. After each movement of the window, the *kernel* weights are reused to compute the weighted sum, generating one scalar value of the output. By repeating this process for each possible position of the window, the output is generated as the combination of all individual scalars. Figure 2.3 visualizes this computation.

Because each application of the kernel is independent of the other computations, convolutional layers can easily be parallelized. Furthermore, convolutions are robust to movements of the input. If a feature that can be detected by a kernel is moved in the input, the kernel is still being applied to it as the window moves over the output to the new feature position (Goodfellow et al., 2016). Those properties make convolutional layers very popular for many domains, such as image recognition or time series. Convolutional operations can be transformed into fully connected layers by extending the multiplication of the kernel with the selected window by additional multiplications of all remaining input values with zero.

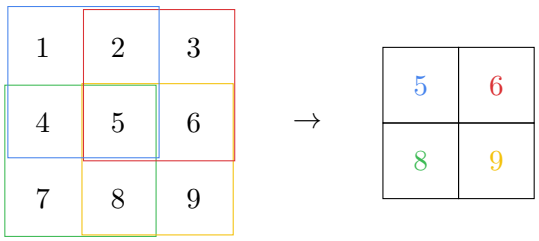


Figure 2.4: A max-pooling operation. The window of size 2×2 is moved over the input with stride 1×1. Blue: $\max\{1, 2, 4, 5\} = 5$, red: $\max\{2, 3, 5, 6\} = 6$, green: $\max\{4, 5, 7, 8\} = 8$, yellow: $\max\{5, 6, 8, 9\} = 9$.

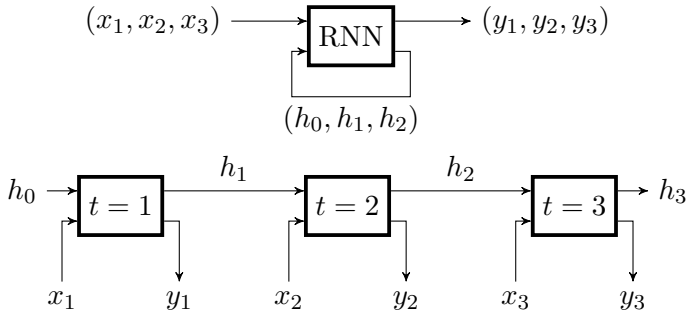


Figure 2.5: Unrolling of an RNN. In the unrolled version, all layers have the same activation function and weight matrix. Figure taken from (Brix, 2018).

Pooling Convolutional networks commonly use *pooling* layers to reduce the layer size and to potentionally introduce additional non-linearity. Similar to convolutions, a window is moved over the input in a specified stride. At each step, the chosen pooling operation is applied. One popular pooling technique is *max-pooling* (Zhou and Chellappa, 1988), visualized in Figure 2.4. Alternative pooling operations, such as average pooling, exist, but are not analyzed in this thesis.

2.3 Recurrent Neural Networks (RNNs)

Even though FFNNs are able to solve many tasks with high precision, they have inherent limitations on their applicability. As the number of inputs is defined by the architecture, it is impossible to process a sequence of arbitrary length as a whole. On the example of image recognition, individual frames of a video could be fed to an FFNN, but both previous and succeeding frames need to be ignored. Alternatively, the number of input nodes could be doubled, allowing to feed in two frames of a video. However, this approach does not scale to videos of arbitrary length, as

each frame count would require its own network. Furthermore, if the sequence of frames is split into individual images and processed separately, the network does not transfer knowledge gained from the first frame to the analysis of the second, as it is reset after each computation.

RNNs solve both issues by defining loops in the network that feed the output of later layers back to layers closer to the input. Therefore, the network can repeatedly process new inputs while keeping track of previous computations. Figure 2.5 visualizes an RNN and the corresponding *unrolled* version. At timestep 1, the network processes the input x_1 and a predefined initial memory value h_0. After computing the first output y_1 and updating the memory, the resulting h_1 and new data x_1 are used as the next input to the same RNN. This process is repeated until all inputs have been handled, generating an equal number of outputs. As the memory state is updated after each input, the knowledge gained from initial inputs can be used to determine the output for later inputs. Popular RNNs are long short-term memorys (LSTMs) (Hochreiter and Schmidhuber, 1997) and gated recurrent units (GRUs) (Cho et al., 2014). However, the verification of RNNs is out of the scope of this thesis.

2.4 Training

Once the network architecture has been defined, the weights and biases of all nodes are selected randomly. In order to tune them such that the network can perform its designated task, the model needs to be trained. In *supervised training* (Kotsiantis, 2007), a dataset with exemplary inputs and known correct target labels is used. By propagating the input through the network, the prediction can be computed. To determine the necessary weight updates that would improve this selection, the *loss* is computed as the difference between the predicted and the target output. Next, the gradients of all weights with respect to the loss can be determined, by propagating them backwards through the network. Given the gradient, it is then possible to decide whether the weights need to be increased or decreased. For positive gradients, a weight decrease would lead to a smaller loss and would therefore improve the prediction quality. For negative gradients, the weights need to be increased. This is visualized in Figure 2.6. The process of updating the gradients is repeated until external limits on the computation time or number of iterations have been reached, or the network has converged to a stable behavior.

2.4.1 Generalization

While the supervised training of neural networks requires labeled training data, the converged models are ultimately used to predict the output for previously unseen input. To measure whether a network is indeed able to *generalize* to examples

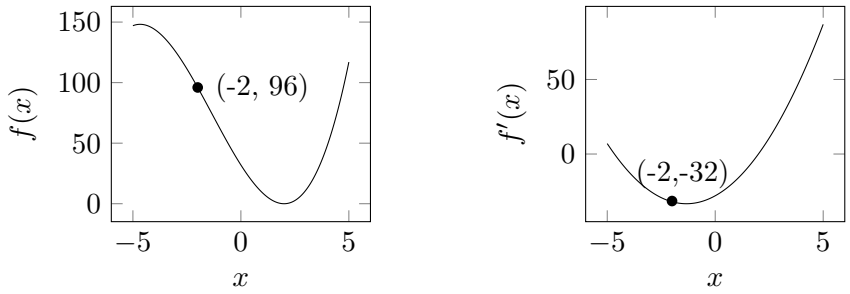


Figure 2.6: The gradient indicates the direction of the update. Because $f'(-2) < 0$, x has to be increased to get closer to the local optimum. Figure taken from (Brix, 2018).

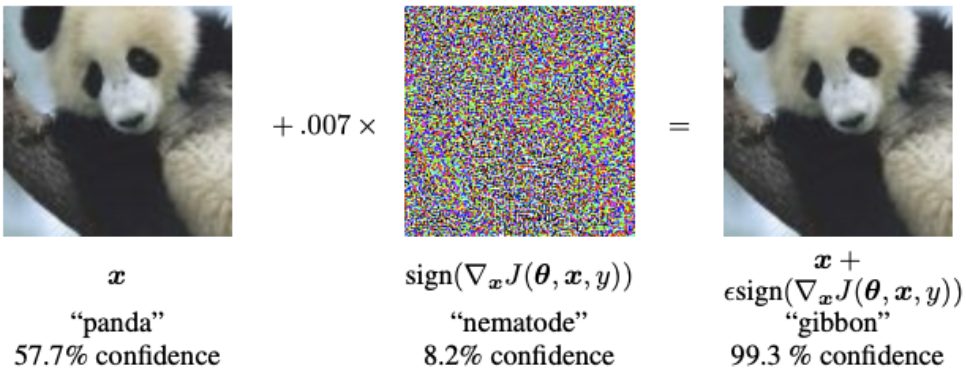


Figure 2.7: An adversarial example. By adding a small amount of noise, the image of a panda is classified as a gibbon. Figure taken from (Goodfellow et al., 2015).

outside of the training set, it is tested on a separate validation or test set that has known labels but was not used during training. Depending on the chosen network architecture and training procedure, the model is not guaranteed to generalize well. Instead of discovering underlying patterns that predict the correct output, the network may have *overfitted*, i.e., memorized the correct labels for the training set (Goodfellow et al., 2016). *Regularization* techniques can be used to overcome this problem by defining additional hard or soft constraints.

2.5 Adversarial Examples

As every non-trivial network with more than one possible output must have a decision boundary for the classification, it follows that every input can be modified in a way that changes the computed label. However, as long as those modifications are significant enough to make the manipulations obvious, the practical impact on real world applications is limited. Similarly, occasional misclassifications of inputs are to be expected for any non-trivial task.

In an *adversarial attack*, misclassifications are caused on purpose. Here, a given input that is correctly classified is modified slightly, causing the network to change its prediction to a wrong label. This process is visualized in Figure 2.7. The modified input that yields the incorrect prediction is referred to as an *adversarial example*. Because the modification is imperceptible to humans, those attacks pose a significant risk for safety critical applications, such as autonomous driving (Morgulis et al., 2019). To enforce the property of imperceptibility, the possible manipulations of the input are commonly bounded using L_p-norms (Lécuyer et al., 2019):

$$||\alpha||_p = \begin{cases} (\sum_{i=1}^{n} |\alpha_i|^p)^{\frac{1}{p}}, & \text{for } 1 < p \\ \max_{i=1}^{n} |\alpha_i|, & \text{otherwise} \end{cases} \quad \text{for } \alpha \in \mathbb{R}^n$$

If, for a network f, input x, a target label $f(x) = y$, a norm L_p, and an upper bound δ for the manipulation, it can be shown that $\arg\max f(\tilde{x}) = y \; \forall \tilde{x} : ||x - \tilde{x}||_p \leq \delta$, the network is referred to as *robust*.

Goodfellow et al. (2015) describe a technique using gradient search that uses the gradient to predict which modifications have the maximal impact on the prediction. As those computations require all network weights to be known, this is an example for a *white box* attack. For white box attacks, many different defenses have been proposed, but are commonly countered by more advanced attacks shortly after their publication (Athalye and Carlini, 2018). *Black box* attacks, where the specific network parameters are not known, are possible too (Szegedy et al., 2013). One possible attack is to submit a sequence of queries to the black box model until an adversarial example is found (Narodytska and Kasiviswanathan, 2017). Alternatively, it is possible to train a separate network on a different training set, and to compute the adversarial example on this second, distinct model. Liu et al. (2017) demonstrate that those adversarial examples transfer to the original network and that his technique can be successfully applied to commercial black box models.

For high-risk applications, it may be necessary to provide formal guarantees for the robustness. Lécuyer et al. (2019) propose a technique for proving a network to be robust with a chosen likelihood $\eta < 1$. To provide guarantees with 100% likelihood, alternative *verification* techniques need to be employed. This thesis

analyzes one such method, proposing and implementing significant improvements to its expressibility.

2.6 Notation

Commonly, the neurons of a layer are not analyzed individually, allowing for a simplified notation that refers to all neurons in a layer as a vector, and all weights between two layers as a matrix. Due to the need for detailed computations for each individual neuron, this is not possible in this thesis.

For a given neural network, let the size of the input layer be denoted by s_0, followed by n subsequent fully connected feedforward layers of size $s_1, \ldots, s_n$. As detailed in Section 2.2.1, convolutional layers can be converted to feedforward layers, so they are omitted for brevity. The node values $\hat{x}_1^{(0)}, \ldots, \hat{x}_{s_0}^{(0)}$ of the input nodes represent the network input. Node inputs in subsequent layers are computed as the weighted sum $x_i^{(l)} = \sum_{j=1}^{s_{l-1}} w_{i,j} \cdot \hat{x}_j^{(l-1)}$. For all intermediate layers, the node output $\hat{x}_i^{(l)}$ is computed by applying a non-linear activation function g: $\hat{x}_i^{(l)} = g(x_i^{(l)})$. Even though different activation functions exist, this work assumes only ReLU operations are applied, i.e., $\hat{x}_i^{(l)} = \max\{0, x_i^{(l)}\}$, as they are easy to compute, piecewise-linear, and commonly used. The propagation is stopped once the network output $x_i^{(n)}$ is computed. For notational simplicity, $x_i^{(l)}$ and $\hat{x}_i^{(l)}$ are referred to as x and $\hat{x}$ whenever the specific i and l are not important.

Eq^* is the higher order function returning the non-linear formula for a given node value x. Each node can be approximated by linear upper and lower bounds. These will be referred to as $Eq_{up}(x)$ and $Eq_{low}(x)$, respectively. Determining the bounds as a function as opposed to a simple interval reduces the overestimation error (see Section 3.1). For the range of valid inputs, specified by the concrete example and its maximal perturbation, both bounds have minimal and maximal values $\underline{Eq_{low}(x)}, \underline{Eq_{up}(x)}$ and $\overline{Eq_{low}(x)}, \overline{Eq_{up}(x)}$, respectively. Wherever a distinction of the upper and lower bounds is not necessary for the given argument, as it holds for both instances, they are referred to as $Eq(x)$. All listed equations can be determined for $\hat{x}$ as well.

Chapter 3

Neurify

A popular software for the verification of neural networks is Neurify[1], a tool written by Wang et al. (2018a). Due to its performance, readability, and permissive license, it was chosen as the basis for all evaluations and additional implementations in this work. In the following sections, both the underlying mathematical theory as well as implementation specific details are described, to provide the basis for the improvements detailed in Chapter 4.

3.1 Underlying Theory

In order to provide strict mathematical proofs for absence of adversarial examples, all steps taken by Neurify need to be based on sound mathematic arguments. The relevant theories are described in this section. All techniques are also described in (Wang et al., 2018a), except for the subsection about dependency tracking, which is omitted in the original paper.

3.1.1 Definition of the Input Intervals

As described in Section 2.5, adversarial examples are defined as inputs that are close to a given data point in the input space, but produce widely different output predictions. Depending on the chosen setting, the region of the input space from which the adversarial example may be chosen varies. For the simplest case of an L_∞ bounded space, the possible values of the adversarial example can be defined dimension-wise: Given a concrete input value x, the bounds are $[x - L_\infty, x + L_\infty]$. Depending on the nature of the chosen task, additional restrictions may apply. For the example of image recognition, the pixel values used as inputs are bounded by $[0, 255]$, resulting in bounds for the adversarial example of $[\max\{0, x - L_\infty\}, \min\{255, x + L_\infty\}]$ Alternative bounds, such as L_1 or L_2 (Goodfellow et al., 2015) can be over-approximated by L_∞ and are not the main focus of Neurify or this thesis.

[1] `https://github.com/tcwangshiqi-columbia/Neurify`

3.1.2 Forward Propagation of Intervals

After defining the bounds in the input space, the transformations as defined by the trained network need to be applied. The simplest technique to propagate the intervals from one layer to the next is to apply *interval propagation* (Moore et al., 2009). By using the elementary rules for additions and multiplications, the intervals can be computed layer by layer, until the output is reached.

However, this approach weakens the bounds with each step, and misses opportunities to determine stronger constraints on the respective spaces of each layer. Therefore, the final output is not necessarily as tightly bounded as possible, reducing the effectiveness of the analysis. By using *symbolic propagation*, more information about the underlying dependencies can be retained, and stronger bounds can be computed. This is demonstrated in Equations 3.1 and 3.2.

$$a \in [0,1] \Rightarrow a - 0.5a \overset{\text{interval}}{\in} [0,1] - 0.5 \cdot [0,1] = [0,1] - [0,0.5] = [-0.5,1] \tag{3.1}$$

$$a \in [0,1] \Rightarrow a - 0.5a \overset{\text{symbolic}}{=} 0.5a \in [0,0.5] \tag{3.2}$$

In addition to the linear transformations applied by each layer, the networks analyzed in this work employ ReLU functions as nonlinear activations. Their influence on the node values needs to be reflected in the symbolic representation by relaxing them if necessary. To this end, Wang et al. (2018a) identify three regions the bound $Eq(x)$ of the node x may fall into:

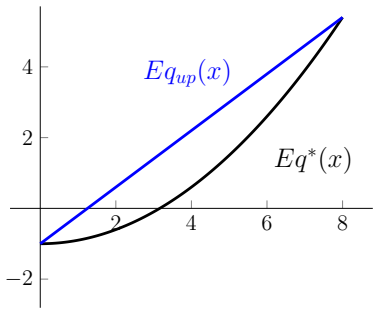
1. $0 \leq \underline{Eq_{low}(x)} \leq \overline{Eq_{up}(x)}$: If the lowest value taken by the lower bound is already non-negative, the ReLU operation has no effect on it. Therefore, input and output are equal and $Eq(\hat{x}) = Eq(x)$.

2. $\underline{Eq_{low}(x)} \leq \overline{Eq_{up}(x)} \leq 0$: If the largest value the upper bound may take is non-positive, the ReLU operation guarantees that $Eq(\hat{x}) = 0$.

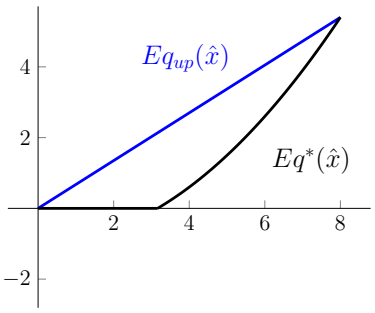
3. $\underline{Eq_{low}(x)} \leq 0 \leq \overline{Eq_{up}(x)}$: If the boundary indicates that the node may become both negative and positive, there is a non-linear dependence between the input and output of the node. Thus, the output bounds need to be relaxed. Wang et al. (2018a) refer to these nodes as *overestimated*.

For overestimated nodes, they propose to use *symbolic linear relaxation* to find new bounds for the ReLU output that are valid, but still tight. They prove that the relaxations

$$\begin{aligned} \overline{Eq_{up}}(\hat{x}) &= Relax(\max\{0, Eq_{up}(x)\}) \\ &= \frac{\overline{Eq_{up}(x)}}{\overline{Eq_{up}(x)} - \underline{Eq_{up}(x)}}(Eq_{up}(x) - \underline{Eq_{up}(x)}) \end{aligned} \tag{3.3}$$

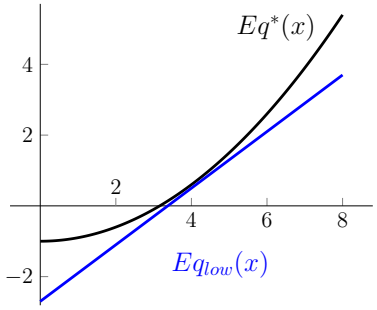


(a) The node input may be negative

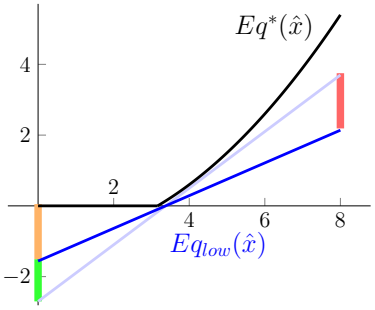


(b) Relaxation ensures the positivity of the upper bound

Figure 3.1: Relaxation of the upper bound is mandatory.



(a) The node input may be negative



(b) Green: improvement of $\underline{Eq_{low}(\hat{x})}$; red: worsening of $\overline{Eq_{low}(\hat{x})}$; orange: equality to the red bar defines $Eq_{low}(\hat{x})$

Figure 3.2: Relaxation of the lower bound is optional and a trade off.

$$\begin{aligned} Eq_{low}(\hat{x}) &= Relax(\max\{0, Eq_{low}(x)\}) \\ &= \frac{\overline{Eq_{low}(x)}}{\overline{Eq_{low}(x)} - \underline{Eq_{low}(x)}} Eq_{low}(x) \end{aligned} \tag{3.4}$$

minimize the maximal distance between $\max\{0, Eq(x)\}$ and $Eq(\hat{x})$. This relaxation is visualized in Figures 3.1 and 3.2. However, as shown in Section 4.1.1, it is not optimal for approximating $Eq^*(\hat{x})$.

Given that Equations 3.3 and 3.4 are structured very similarly, Wang et al. (2018a) make the additional assumption that both bounds are separated only by some scalar δ, i.e.:

$$Eq_{up}(x) = g(x) + \delta^x_{up} \tag{3.5}$$
$$Eq_{low}(x) = g(x) + \delta^x_{low} \tag{3.6}$$

As this implies that the upper and lower bounds cannot be adapted individually, Equations 3.3 and 3.4 need to be simplified by overestimating $\underline{Eq_{up}(x)} \geq \underline{Eq_{low}(x)}$ and $\overline{Eq_{low}(x)} \leq \overline{Eq_{up}(x)}$.

$$\begin{aligned} Eq_{up}(\hat{x}) &= Relax(\max\{0, Eq_{up}(x)\}) \\ &= \frac{\overline{Eq_{up}(x)}}{\overline{Eq_{up}(x)} - \underline{Eq_{low}(x)}}(Eq_{up}(x) - \underline{Eq_{low}(x)}) \end{aligned} \tag{3.7}$$
$$\begin{aligned} Eq_{low}(\hat{x}) &= Relax(\max\{0, Eq_{low}(x)\}) \\ &= \frac{\overline{Eq_{up}(x)}}{\overline{Eq_{up}(x)} - \underline{Eq_{low}(x)}} Eq_{low}(x) \end{aligned} \tag{3.8}$$

This ensures that both bounds are scaled by the same amount, yielding

$$Eq_{up}(\hat{x}) = g(\hat{x}) + \delta^{\hat{x}}_{up} = \frac{\overline{Eq_{up}(x)}}{\overline{Eq_{up}(x)} - \underline{Eq_{low}(x)}}(g(x) + \delta^x_{up} - \underline{Eq_{low}(x)}) \tag{3.9}$$
$$Eq_{low}(\hat{x}) = g(\hat{x}) + \delta^{\hat{x}}_{low} = \frac{\overline{Eq_{up}(x)}}{\overline{Eq_{up}(x)} - \underline{Eq_{low}(x)}}(g(x) + \delta^x_{low}) \tag{3.10}$$

However, the applied overestimation implies that the bounds are weakened. By decoupling the upper and lower bounds, it becomes theoretically possible to determine better estimations of $\underline{Eq_{up}(x)}$ to tighten the upper bound as well as to improve the lower bound by a stronger relaxation logic. Both techniques are described in Sections 4.1.1 and 4.1.2.

Optimization by Tracking Dependencies

At each node, both the upper and lower bounds are computed symbolically. Given the symbolic values at the previous layer, the new upper (lower) bound is determined as the weighted sum of the upper (lower) bound of all previous nodes that are

assigned a positive weight plus the weighted sum of all the lower (upper) bounds of all previous nodes that are assigned a negative weight.

$$Eq_{up}(x_j^{(l+1)}) = \sum_{i\in[1,\dots,s_l],w_{j,i}>0} w_{j,i} \cdot Eq_{up}(\hat{x}_i^{(l)}) + \sum_{i\in[1,\dots,s_l],w_{j,i}<0} w_{j,i} \cdot Eq_{low}(\hat{x}_i^{(l)}) \tag{3.11}$$

However, in a fully connected network, nodes that are located in layers more than one before the current layer have many possible paths connecting them to the current node x. Whenever one such path influences x positively, while another one decreases the value of x, the effective change of x cancels out. In order to provide strong bounds, this effect has to be taken into account as shown by the following example. For $l_i(x) \leq f_i(x) \leq u_i(x)\ \forall i \in [1,3]$, $f_2(x) = -\frac{1}{2}f_1(x)$, and $f_3(x) = f_1(x) + f_2(x)$, the bounds of f_3 could be computed as

$$u_3(x) = u_1(x) + u_2(x) = u_1(x) - \frac{1}{2}l_1(x) \tag{3.12}$$

$$l_3(x) = l_1(x) + l_2(x) \quad = l_1(x) - \frac{1}{2}u_1(x) \tag{3.13}$$

even though $f_3(x) = f_1(x) + f_2(x) = f_1(x) - \frac{1}{2}f_1(x) = \frac{1}{2}f_1(x)$ and therefore

$$u_3(x) = \frac{1}{2}u_1(x) \leq u_1(x) - \frac{1}{2}l_1(x) \tag{3.14}$$

$$l_3(x) = \frac{1}{2}l_1(x) \ \geq l_1(x) - \frac{1}{2}u_1(x) \tag{3.15}$$

Thus, it is important to track different paths that lead to the same node to first simplify the underlying equation as much as possible, before determining the new upper and lower bounds. This can be achieved by tracing the dependencies of each node backwards from the current layer to the input layer and simplifying at each step. Even though this is not described in (Wang et al., 2018a), it is implemented in Neurify.

3.1.3 Interpretation of Output Intervals

The last layer of a neural network commonly uses a softmax (Goodfellow et al., 2016) operation to shift the probability mass towards the most likely prediction.

$$\forall x \in \mathbb{R}^K : \sigma(x)_j = \frac{e^{x_j}}{\sum_{i=i}^{K} e^{x_i}} \tag{3.16}$$

However, as this function keeps the order of labels with respect to their probability intact, it does not influence the discovery of adversarial examples. Therefore, it can be omitted from the analysis.

3.1.4 Search using an LP Solver

Once concrete intervals for the output layer have been computed, they can be compared to check for the possible existence of adversarial examples. Given a target label y and a different label $\tilde{y} \neq y$, with respective output probability intervals of $I_y = [a, b]$ and $I_{\tilde{y}} = [\tilde{a}, \tilde{b}]$, $\tilde{b} \geq a$ indicates that there may exist an adversarial input that is classified as $\tilde{y}$ with probability $\tilde{b}$, while label y is assigned the lower probability a. Importantly, it does not guarantee the existence, as $I_{\tilde{y}}$ might have been overestimated, artificially increasing the value of $\tilde{b}$.

If the intervals indicate the possible existence, an LP solver can be utilized to find a concrete possible adversarial example. To this end, a linear problem with N variables can be used, where N equals the number of input neurons. By bounding them with the corresponding input bounds that result from the concrete input example and the given L_∞ value, the LP is restricted to acceptable inputs. Additional constraints (see Section 3.1.5) may be added to further refine the search. Finally, the constraint $\tilde{b} \geq a$ ensures that the possible solution is chosen such that it may be a valid adversarial example. Afterwards, the LP solver may be used as a black box to generate either a concrete example, or prove that no solution to the problem exists. If no example can be found, this mathematically ensures that no adversarial example may exist, and the search can be stopped. If a possible input is found, it needs to be analyzed by the network to check whether the predicted output does differ from the label of the original input. Due to the overestimations and relaxations during the forward propagation of bounds, the LP does not necessarily restrict the search space to valid adversarial examples. Should the result be confirmed to be an adversarial example, the search can be stopped. Otherwise, the search has to be continued.

3.1.5 Splitting

Whenever the LP solver returns results that are not valid adversarial examples, the search has to be continued. To this end, additional constraints need to be added to guide the LP solver either towards real adversarial examples, or to enable it to prove their non-existence. As no further information about the network exists, those restraints need to be added by performing a case distinction. Specifically, any overestimated node as defined in Section 3.1.2 may be *split* into the region where its input is negative and the region where it is positive. This distinction can be added to the LP to limit the search space. In addition, this split implies that the node

is no longer overestimated and does not need to be relaxed anymore. Therefore, the bounds of nodes in following layers may be tightened, further simplifying the search. Wang et al. (2018b) show that the reunion of both resulting output intervals is at least as tight than the original, motivating this approach. The main caveat of splitting is that each split potentially doubles the number of searches that need to be performed. Should a real adversarial example be found along one of the resulting branches, the remaining search tree can be skipped. However, should the search prove one of the branches to be robust against adversarial attacks, the complete other branch needs to be tested as well.

Selection of Nodes to Split

As previously explained, the splitting of overestimated nodes may speed up the analysis by reducing the search space. For the selection of which specific overestimated node should be splitted next, Wang et al. (2018b) propose to compute the gradient of the network output with respect to the node intervals. Because those gradients indicate the influence of each node on the output, splitting the node with the largest absolute gradient value promises to maximize the impact on the remaining analysis.

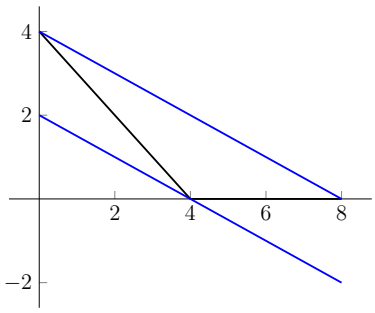
Because the gradients are influenced by the ReLU activation function which sets the gradient to zero if the input was non-positive, the backward propagation of the gradient is non-linear as well. Therefore, overestimated nodes require the gradient to be overestimated by a relaxation. While overestimations weaken the forwards propagated bounds as they move towards the output, the backwards propagated bounds of the gradient become weaker for layers closer to the input, reducing the chance of correctly predicting which node has the highest gradient. Alternative selection techniques are presented and analyzed in Section 5.4, showing that the impact of the selection mechanism is significant.

3.1.6 Bounds May Be Arbitrarily Weak

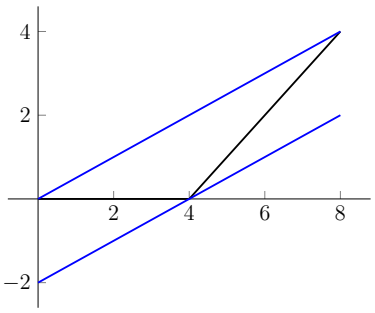
In all figures in this thesis, bounds are depicted to be tangential to the respective target functions. However, this is not always given, and bounds may be arbitrarily weak. This is visualized in Figure 3.3.

3.2 Example

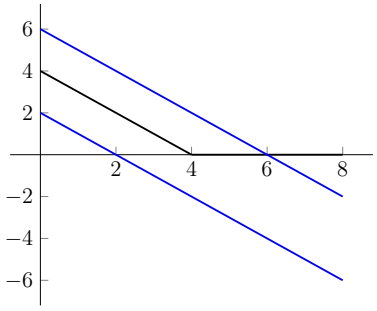
This section demonstrates the techniques explained above using a simple example network. Section 4.2 analyzes the same setting to demonstrate the improvements by better bounds.



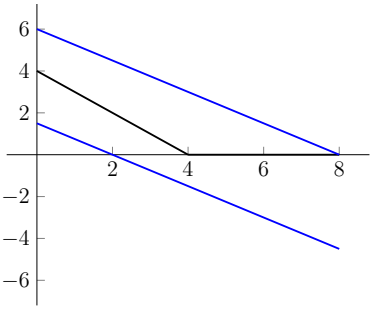
(a) Bounds for node $\hat{x}_1$



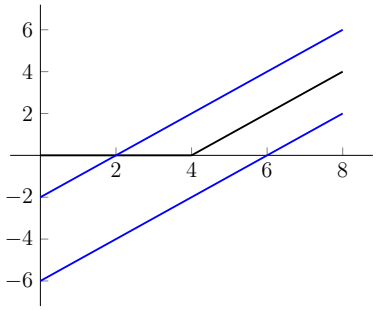
(b) Bounds for node $\hat{x}_2$



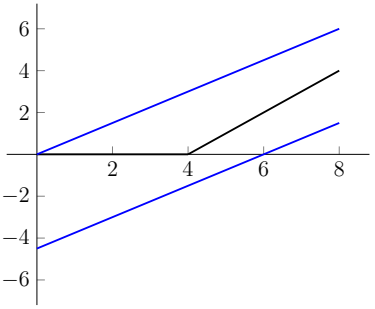
(c) Bounds for node $x_3 = \hat{x}_1 - \hat{x}_2$



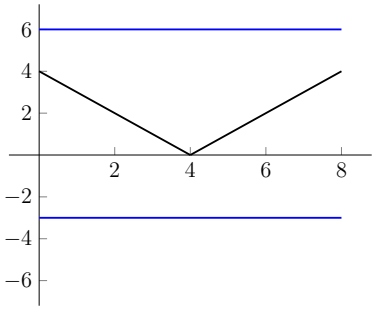
(d) Bounds for node $\hat{x}_3 = \max\{0, x_3\}$



(e) Bounds for node $x_4 = \hat{x}_2 - \hat{x}_1$



(f) Bounds for node $\hat{x}_4 = \max\{0, x_4\}$



(g) Bounds for node $x_5 = \hat{x}_3 + \hat{x}_4$

Figure 3.3: Bounds may become arbitrarily weak. The black lines depict the real function, blue lines visualize the upper and lower bounds.

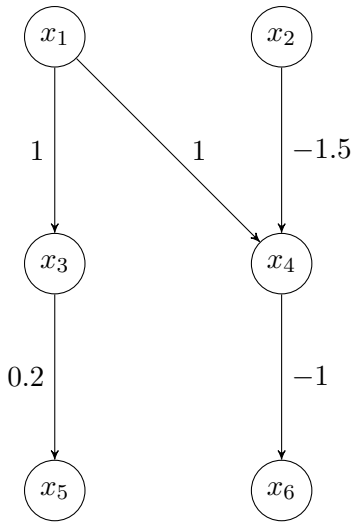


Figure 3.4: Simple network used for the analysis. All biases and omitted weights are zero.

The network depicted in Figure 3.4 visualizes a very simple network with only one hidden layer and two neurons in each layer. For a concrete input of $\hat{x}_1 = \hat{x}_2 = 1.5$, the output can be computed as follows.

$$x_3 = \hat{x}_1 = 1.5 \tag{3.17}$$
$$\hat{x}_3 = \max\{0, x_3\} = \max\{0, 1.5\} = 1.5 \tag{3.18}$$
$$x_4 = \hat{x}_1 - 1.5\hat{x}_2 = 1.5 - 1.5 \cdot 1.5 = -0.75 \tag{3.19}$$
$$\hat{x}_4 = \max\{0, x_4\} = \max\{0, -0.75\} = 0 \tag{3.20}$$
$$x_5 = 0.2\hat{x}_3 = 0.2 \cdot 1.5 = 0.3 \tag{3.21}$$
$$x_6 = -\hat{x}_4 = 0 \tag{3.22}$$

As $x_5 = 0.3 > 0 = x_6$, x_5 is the target label and an adversarial example would have to satisfy $x_5 < x_6$. By defining $L_\infty = 0.5$, the search space for possible attacks becomes $1 \leq \hat{x}_1 \leq 2 \land 1 \leq \hat{x}_2 \leq 2$.

For the initial nodes x_1 and x_2, the upper and lower symbolic bounds are trivially given as $Eq_{low}(\hat{x}_1) = Eq_{up}(\hat{x}_1) = \hat{x}_1$ and $Eq_{low}(\hat{x}_2) = Eq_{up}(\hat{x}_2) = \hat{x}_2$. The bounds

of all subsequent layers need to be computed by propagating those initial bounds forward. For the first layer, this gives

$$\begin{aligned} Eq_{up}(x_3) &= Eq_{up}(\hat{x}_1) \\ &= \hat{x}_1 && (3.23) \\ \Rightarrow \underline{Eq_{up}(x_3)} &= 1 \wedge \overline{Eq_{up}(x_3)} = 2 && (3.24) \\ Eq_{low}(x_3) &= Eq_{low}(\hat{x}_1) \\ &= \hat{x}_1 && (3.25) \\ \Rightarrow \underline{Eq_{low}(x_3)} &= 1 \wedge \overline{Eq_{low}(x_3)} = 2 && (3.26) \\ Eq_{up}(x_4) &= Eq_{up}(\hat{x}_1) - 1.5 \cdot Eq_{low}(\hat{x}_2) \\ &= \hat{x}_1 - 1.5\hat{x}_2 && (3.27) \\ \Rightarrow \underline{Eq_{up}(x_4)} &= -2 \wedge \overline{Eq_{up}(x_4)} = 0.5 && (3.28) \\ Eq_{low}(x_4) &= Eq_{low}(\hat{x}_1) - 1.5 \cdot Eq_{up}(\hat{x}_2) \\ &= \hat{x}_1 - 1.5\hat{x}_2 && (3.29) \\ \Rightarrow \underline{Eq_{low}(x_4)} &= -2 \wedge \overline{Eq_{low}(x_4)} = 0.5 && (3.30) \end{aligned}$$

To compute the symbolic bounds for the output of both nodes, they have to be classified according to the three categories defined in Section 3.1.2. Node x_3 is known to be always positive (category 1), so

$$\begin{aligned} Eq_{up}(\hat{x}_3) &= Eq_{up}(x_3) && (3.31) \\ Eq_{low}(\hat{x}_3) &= Eq_{low}(x_3) && (3.32) \end{aligned}$$

However, node x_4 may have both negative and positive input (category 3), so it needs to be overestimated. According to the Equations 3.7 and 3.8, this results in the bounds

$$\begin{aligned} Eq_{up}(\hat{x}_4) &= \frac{\overline{Eq_{up}(x_4)}}{\overline{Eq_{up}(x_4)} - \underline{Eq_{low}(x_4)}}(Eq_{up}(x_4) - \underline{Eq_{low}(x_4)}) \\ &= \frac{0.5}{0.5 - (-2)}(Eq_{up}(x_4) - (-2)) \\ &= 0.2Eq_{up}(x_4) + 0.4 && (3.33) \end{aligned}$$

$$\begin{aligned} Eq_{low}(\hat{x}_4) &= \frac{\overline{Eq_{up}(x_4)}}{\overline{Eq_{up}(x_4)} - \underline{Eq_{low}(x_4)}} Eq_{low}(x_4) \\ &= \frac{0.5}{0.5 - (-2)} Eq_{low}(x_4) \\ &= 0.2 Eq_{low}(x_4) \end{aligned} \tag{3.34}$$

Finally, the bounds for the output nodes can be determined as

$$\begin{aligned} Eq_{up}(x_5) &= 0.2 Eq_{up}(\hat{x}_3) \\ &= 0.2 Eq_{up}(x_3) && (3.35) \\ Eq_{low}(x_5) &= 0.2 Eq_{low}(\hat{x}_3) \\ &= 0.2 Eq_{low}(x_3) && (3.36) \\ Eq_{up}(x_6) &= -Eq_{low}(\hat{x}_4) \\ &= -0.2 Eq_{low}(x_4) && (3.37) \\ Eq_{low}(x_6) &= -Eq_{up}(\hat{x}_4) \\ &= -(0.2 Eq_{up}(x_4) + 0.4) \\ &= -0.2 Eq_{up}(x_4) - 0.4 && (3.38) \end{aligned}$$

Based on these bounds, it is possible to define the LP problem that needs to be solved in order to check for possible adversarial examples. As no splits of overestimated nodes have previously been performed, the problem consists solely of the definition of the input space and a check for an output that satisfies $Eq_{up}(x_6) > Eq_{low}(x_5)$.

$$\begin{aligned} LP &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (Eq_{up}(x_6) > Eq_{low}(x_5)) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-0.2 Eq_{low}(x_4) > 0.2 Eq_{low}(x_3)) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-0.2(\hat{x}_1 - 1.5\hat{x}_2) > 0.2\hat{x}_1) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-0.4\hat{x}_1 + 0.3\hat{x}_2 > 0) \end{aligned} \tag{3.39}$$

The LP solver may return the variable assignments $\hat{x}_1 = 1 \wedge \hat{x}_2 = 2$. Unfortunately, a check of those concrete inputs proves that this is a spurious example and not a successful adversarial attack, as $x_6 > x_5$ is not satisfied.

$$x_3 = \hat{x}_1 = 1 \tag{3.40}$$
$$\hat{x}_3 = \max\{0, x_3\} = \max\{0, 1\} = 1 \tag{3.41}$$
$$x_4 = \hat{x}_1 - 1.5\hat{x}_2 = 1 - 1.5 \cdot 2 = -2 \tag{3.42}$$
$$\hat{x}_4 = \max\{0, x_4\} = \max\{0, -2\} = 0 \tag{3.43}$$
$$x_5 = 0.2\hat{x}_3 = 0.2 \cdot 1 = 0.2 \tag{3.44}$$
$$x_6 = -\hat{x}_4 = 0 \tag{3.45}$$

To continue the analysis, an overestimated node has to be split in order to tighten the bounds and restrict the LP problem. Usually, the node would be selected based upon the gradients. As node x_4 is the only overestimated node in this example, the computation of the gradients is omitted for brevity. The Equations 3.24 to 3.32 are not impacted by the split and remain the same. For the case $Eq_{up}(x_4) < 0$, node x_4 is in category 2 and therefore

$$Eq_{up}(\hat{x}_4) = 0 \tag{3.46}$$
$$Eq_{low}(\hat{x}_4) = 0 \tag{3.47}$$

resulting in

$$\begin{aligned} Eq_{up}(x_5) &= 0.2Eq_{up}(\hat{x}_3) \\ &= 0.2Eq_{up}(x_3) \end{aligned} \tag{3.48}$$
$$\begin{aligned} Eq_{low}(x_5) &= 0.2Eq_{low}(\hat{x}_3) \\ &= 0.2Eq_{low}(x_3) \end{aligned} \tag{3.49}$$
$$\begin{aligned} Eq_{up}(x_6) &= -Eq_{low}(\hat{x}_4) \\ &= 0 \end{aligned} \tag{3.50}$$
$$\begin{aligned} Eq_{low}(x_6) &= -Eq_{up}(\hat{x}_4) \\ &= 0 \end{aligned} \tag{3.51}$$

leading to the LP problem

$$\begin{aligned} LP &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (Eq_{up}(x_6) > Eq_{low}(x_5)) \wedge (Eq_{up}(x_4) < 0) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (0 > 0.2Eq_{low}(x_3)) \wedge (\hat{x}_1 - 1.5\hat{x}_2 < 0) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-0.2\hat{x}_1 > 0) \wedge (\hat{x}_1 - 1.5\hat{x}_2 < 0) \end{aligned} \tag{3.52}$$

which is unsatisfiable. Notably, this LP problem not only uses the tightened bounds, but also contains the assumption $Eq_{up}(x_4) < 0$ which is introduced by the performed split.

For $Eq_{low}(x_4) > 0$, node x_4 is in category 1 and therefore

$$Eq_{up}(\hat{x}_4) = Eq_{up}(x_4) \tag{3.53}$$
$$Eq_{low}(\hat{x}_4) = Eq_{low}(x_4) \tag{3.54}$$

resulting in

$$\begin{aligned} Eq_{up}(x_5) &= 0.2Eq_{up}(\hat{x}_3) \\ &= 0.2Eq_{up}(x_3) && (3.55) \\ Eq_{low}(x_5) &= 0.2Eq_{low}(\hat{x}_3) \\ &= 0.2Eq_{low}(x_3) && (3.56) \\ Eq_{up}(x_6) &= -Eq_{low}(\hat{x}_4) \\ &= -Eq_{low}(x_4) && (3.57) \\ Eq_{low}(x_6) &= -Eq_{up}(\hat{x}_4) \\ &= -Eq_{up}(x_4) && (3.58) \end{aligned}$$

leading to the LP problem

$$\begin{aligned} LP &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (Eq_{up}(x_6) > Eq_{low}(x_5)) \wedge (Eq_{low}(x_4) > 0) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-Eq_{low}(x_4) > 0.2Eq_{low}(x_3)) \wedge (Eq_{low}(x_4) > 0) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-(\hat{x}_1 - 1.5\hat{x}_2) > \hat{x}_1) \wedge (\hat{x}_1 - 1.5\hat{x}_2 > 0) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (-2\hat{x}_1 + 1.5\hat{x}_2 > 0) \wedge (\hat{x}_1 - 1.5\hat{x}_2 > 0) \end{aligned} \tag{3.59}$$

which is unsatisfiable as well.

After both branches have been explored and were shown not to contain any adversarial examples, the analysis has successfully proven the network to be robust for the given input and L_∞ value.

3.3 Implementation and Bugs

Due to the complexity of the verification, a manual proof of the robustness of networks is not feasible, and the verification has to be performed automatically. To this end, Wang et al. (2018a) publish Neurify, which applies the aforementioned techniques without the need for human interaction. Even though the main focus of both the general research and this thesis lies on the mathematical theory, the

efficiency and correctness of the implementation are paramount for its applicability. This section highlights some of the implementation specific choices done for Neurify as well as bugs in the original code that decrease the performance.

3.3.1 Parallelization

As detailed in Section 3.1.5, Neurify may split the search space into two branches if additional restraints are needed to guide the LP solver. The resulting two branches do not depend on each other, apart from the fact that a valid adversarial example in one of both branches eliminates the need to evaluate the other. Therefore, a parallel search along both branches significantly increases the overall evaluation speed. Neurify realizes this potential by using parallel threads up to a configurable threshold that can be tuned to the available architecture. Once the predefined number of threads is reached, further parallelism is avoided to prevent overloading the operating system with the management of an increasing number of threads that compete for the same resources. At this point, the search is continued in a depth-first manner.

Due to a bug, Neurify may significantly decrease the defined upper limit of parallel threads. This can cause the analysis to use less than the available computational power, increasing the overall runtime. Chapter 5 compares the original Neurify code with a cleaned up version that avoids this bug and other bottlenecks.

3.3.2 Optimization by Tracking Dependencies

As shown in Section 3.1.2, it is important to track all paths along the network to a given node in order to simplify the equations before upper and lower bounds are computed. In Neurify, this can be done efficiently during a single forward propagation, by keeping track of the δ assigned to each node. Instead of accumulating all deltas in the bias term of each node, Neurify stores the impact of each delta separately. Therefore, when propagating the bounds forward, a node that was assigned a delta during the relaxation and that influences a later node both positively and negatively can be detected easily. With a memory consumption of $\mathcal{O}(\sum_{l=1}^{n} s_l)$ and an operational overhead of $\mathcal{O}((\sum_{l=1}^{n} s_l)^2)$, this can be done very efficiently, especially given the small constant factor associated with both memory consumption and operational cost.

3.3.3 LP Solver

After all bounds have been propagated to the output layer, Neurify relies on an LP solver to search for possible adversarial examples. As the search problem can be transformed into a regular LP problem, no specific requirements on the solver exist. Therefore, it can be used as a black box and chosen at will. Neurify utilizes

lp_solve (Berkelaar et al., 2004), which provides a permissive open-source license. It is well documented, even though (Gearhartq et al., 2013) report that it is not the fastest solver available. Replacing lp_solve with alternative solvers should be possible, but is not within the scope of this work.

3.3.4 Selection of Nodes to Split

For feedforward networks (see Section 2.2), Neurify utilizes the gradients of each node with respect to the output interval to select the next overestimated node for splitting (see Section 3.1.5). However, for convolutional layers (see Section 2.2.1), the gradients are not computed and are assumed to be zero. Neither Wang et al. (2018a) nor the code of Neurify provide an explanation for this omission. It implies that as soon as the backpropagation reaches the first convolutional layer, all gradients are lost, even if further feedforward layers follow. Therefore, the selection of nodes in the initial layers of the network is performed unguided. Here, Neurify splits the overestimated nodes in the order in which they were defined for the network. Also, Neurify may fail to correctly compute the gradients if intermediate layers have more or less nodes than the second to last network layer.

3.3.5 Effects of Splitting

When overestimated nodes are split, the search space is reduced and overestimations become less likely. However, it is also possible that nodes that were previously not overestimated need to be relaxed after a predecessor node was split. For the network depicted in Figure 3.5, input bounds of $0 \leq \hat{x}_1 \leq 20$ lead to the following first analysis.

$$
\begin{aligned}
Eq_{up}(x_2) &= Eq_{up}(x_1) - 10 \\
&= \hat{x}_1 - 10 && (3.60)\\
\Rightarrow \underline{Eq_{up}(x_2)} &= -10 \wedge \overline{Eq_{up}(x_2)} = 10 && (3.61)\\
Eq_{low}(x_2) &= Eq_{low}(x_1) - 10 \\
&= \hat{x}_1 - 10 && (3.62)\\
\Rightarrow \underline{Eq_{low}(x_2)} &= -10 \wedge \overline{Eq_{low}(x_2)} = 10 && (3.63)\\
Eq_{up}(\hat{x}_2) &= 0.5Eq_{up}(x_2) + 5 \\
&= 0.5\hat{x}_1 && (3.64)\\
Eq_{low}(\hat{x}_2) &= 0.5Eq_{low}(x_2) \\
&= 0.5\hat{x}_1 - 5 && (3.65)\\
Eq_{up}(x_3) &= Eq_{up}(\hat{x}_2) + 5 \\
&= 0.5\hat{x}_1 + 5 && (3.66)\\
\Rightarrow \underline{Eq_{up}(x_3)} &= 5 \wedge \overline{Eq_{up}(x_3)} = 15 && (3.67)\\
Eq_{low}(x_3) &= Eq_{low}(\hat{x}_2) + 5 \\
&= 0.5\hat{x}_1 && (3.68)\\
\Rightarrow \underline{Eq_{up}(x_3)} &= 0 \wedge \overline{Eq_{up}(x_3)} = 10 && (3.69)
\end{aligned}
$$

Thus, node x_3 does not need to be overestimated. However, after performing a split for x_2, the search branch that explores the case $\underline{Eq_{low}(x_2)} \geq 0$ results in

$$
\begin{aligned}
Eq_{up}(\hat{x}_2) &= Eq_{up}(x_2) \\
&= \hat{x}_1 - 10 && (3.70)\\
Eq_{low}(\hat{x}_2) &= Eq_{low}(x_2) \\
&= \hat{x}_1 - 10 && (3.71)\\
Eq_{up}(x_3) &= Eq_{up}(\hat{x}_2) + 5 \\
&= \hat{x}_1 - 5 && (3.72)\\
\Rightarrow \underline{Eq_{up}(x_3)} &= -5 \wedge \overline{Eq_{up}(x_3)} = 15 && (3.73)\\
Eq_{low}(x_3) &= Eq_{low}(\hat{x}_2) + 5 \\
&= \hat{x}_1 - 5 && (3.74)\\
\Rightarrow \underline{Eq_{up}(x_3)} &= -5 \wedge \overline{Eq_{up}(x_3)} = 15 && (3.75)
\end{aligned}
$$

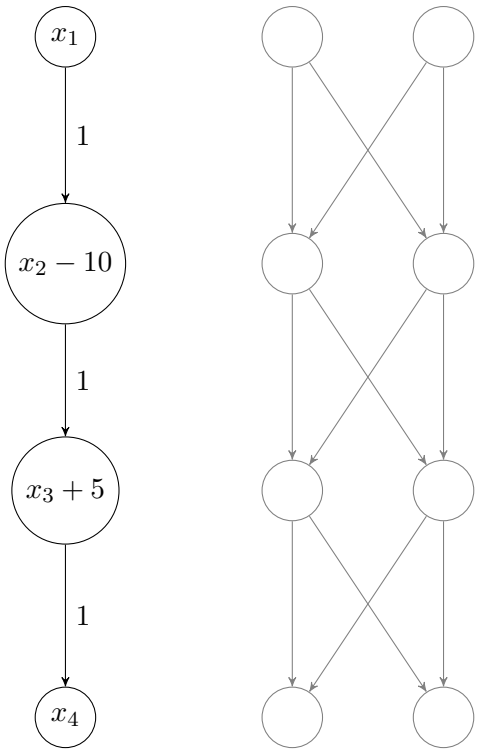


Figure 3.5: Splitting may create new overestimated nodes. Nodes to the right are irrelevant for the example.

and therefore x_3 becomes overestimated. Neurify fails to detect overestimated nodes that are created throughout the analysis, leading to potential edge cases where it misses the existence of adversarial examples and the analysis is therefore not sound.

3.4 Alternative Toolkits

Neurify is not the only available toolkit that tries to verify neural network safety properties. Although this thesis focuses on comparisons with Neurify, the following enumeration lists a number of popular alternatives.

„nnenum“[2] (Bak, 2020) and „NNV“[3] (Xiang et al., 2018; Tran et al., 2019b,a, 2020a,b) use star sets (Bak and Duggirala, 2017) to propagate the input space through the network. This can be done either exactly, or using over-approximations. For the exact case, they also allow to compute the complete counter input set, i.e.,

[2] `https://github.com/stanleybak/nnenum`
[3] `https://github.com/verivital/nnv`

all potential adversarial examples. „VeriNet“[4] (Henriksen and Lomuscio, 2020) is similar to Neurify, but extends the verification to networks utilizing sigmoid and tanh activation functions. „Oval“[5] (Bunel et al., 2017) uses a Branch-and-Bound framework (Bunel et al., 2020) and provides support for GPU based computations. „MIPVerify“[6] (Tjeng et al., 2019) transforms the verification task into a mixed-integer linear programming problem, solvable by third-party toolkits. „ERAN“[7] (Singh et al., 2018, 2019a,b,c) uses abstract interpretation and extends the analysis to sigmoid, tanh and max-pooling operations. A comparison of the aforementioned toolkits has been performed in the VNN competition.[8] However, as all contestants use different hardware, a fair comparison of the results is not possible.

[4] `https://vas.doc.ic.ac.uk/software/neural/`
[5] `https://github.com/oval-group/PLNN-verification`
[6] `https://github.com/vtjeng/MIPVerify.jl`
[7] `https://github.com/eth-sri/eran`
[8] `https://sites.google.com/view/vnn20/vnncomp`

Chapter 4

Debona

The previous chapter explained the theory and implementation for the software Neurify. Improving both the theoretical arguments as well as the implementation specific details is the main topic of this thesis. The following sections provide both the mathematical proofs for tighter bounds during the forward propagation and possible linear relaxations for max-pooling operations as well as a short description of major aspects of the implementation. Furthermore, the example from Section 3.2 is reused to demonstrate the difference between both approaches. The key ideas presented in these sections are the basis of the improved software version, referred to as „Debona“ and made publicly available.[1]

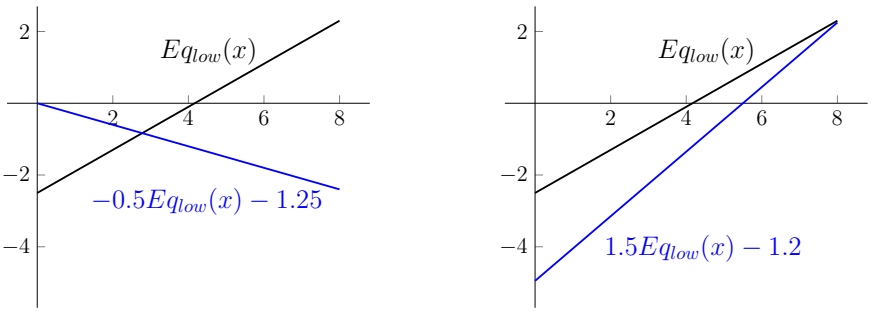
4.1 Underlying Theory

This section details the mathematical arguments that form the basis of Debona by providing tighter bounds on the network nodes.

4.1.1 Zero Bounding

As described in Section 3.1.2, the upper bound must be relaxed to account for the ReLU operation. Wang et al. (2018a) prove that Equation 3.4 minimizes the maximal distance between $\max\{0, Eq_{low}(x)\}$ and $Eq_{low}(\hat{x})$. However, the lower bound $Eq_{low}(\hat{x})$ can be improved by choosing it such that

[1] `https://github.com/ChristopherBrix/Debona`



(a) A negative m always weakens the lower bound.

(b) $m > 1$ requires n to be negative, weakening the lower bound.

Figure 4.1: To maximize the relaxed lower bound, $n = 0$ and $0 \leq m \leq 1$.

$$Eq_{low}(\hat{x}) = \arg\min_{Eq_{low}(\hat{x})} \{(\underline{\max\{0, Eq_{low}(x)\}} - \underline{Eq_{low}(\hat{x})}) + (\overline{\max\{0, Eq_{low}(x)\}} - \overline{Eq_{low}(\hat{x})})\} \tag{4.1}$$

$$= \arg\min_{Eq_{low}(\hat{x})} \{0 - \underline{Eq_{low}(\hat{x})} + \overline{Eq_{low}(x)} - \overline{Eq_{low}(\hat{x})}\} \tag{4.2}$$

$$= \arg\min_{Eq_{low}(\hat{x})} \{-\underline{Eq_{low}(\hat{x})} - \overline{Eq_{low}(\hat{x})}\} \tag{4.3}$$

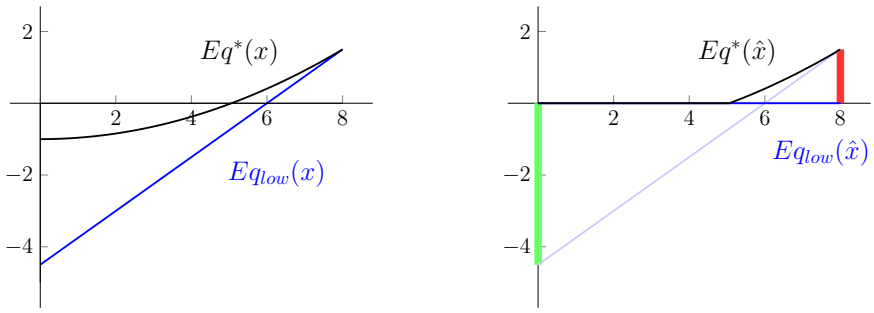
where the last transformation is valid as $\overline{Eq_{low}(x)}$ is constant with respect to $Eq_{low}(\hat{x})$. Notably, this represents the sum of the maximum error on both the positive and the negative regime of the bound, whereas Wang et al. (2018a) minimize the maximum of both errors. By minimizing the sum, the bound estimation is able to perform a trade off between optimizing both errors, reducing the overall overestimation.

Because the lower bound must be a linear equation, $Eq_{low}(\hat{x}) = m \cdot Eq_{low}(x) + n$. Therefore

$$Eq_{low}(\hat{x}) = \arg\min_{m \cdot Eq_{low}(x)+n} \{-\underline{m \cdot Eq_{low}(x) + n} - \overline{m \cdot Eq_{low}(x) + n}\} \tag{4.4}$$

$$= \arg\max_{m \cdot Eq_{low}(x)+n} \{\underline{m \cdot Eq_{low}(x) + n} + \overline{m \cdot Eq_{low}(x) + n}\} \tag{4.5}$$

In the positive region of the bound, $Eq_{low}(\hat{x})$ must not provide stronger estimates than $Eq_{low}(x)$, i.e., $Eq_{low}(\hat{x})\big|_{Eq_{low}(x)\geq 0} \leq Eq_{low}(x)\big|_{Eq_{low}(x)\geq 0}$, and both $\underline{Eq_{low}(\hat{x})}$ and $\overline{Eq_{low}(\hat{x})}$ are to be maximized. Therefore, $n = 0$ and $0 \leq m \leq 1$. This



(a) $\overline{Eq_{low}(x)} < -\underline{Eq_{low}(x)}$

(b) Setting the bound to zero reduces the sum of both errors.

Figure 4.2: Zero bounding decreases the overall error.

argument is demonstrated in Figure 4.1. A positive n is impossible, a negative m always weakens the bounds, and to allow $m > 1$, n would need to be negative, again weakening the overall bound. It follows

$$Eq_{low}(\hat{x}) = \underset{m \cdot Eq_{low}(x)}{\arg\max} \{\underline{m \cdot Eq_{low}(x)} + \overline{m \cdot Eq_{low}(x)}\} \tag{4.6}$$

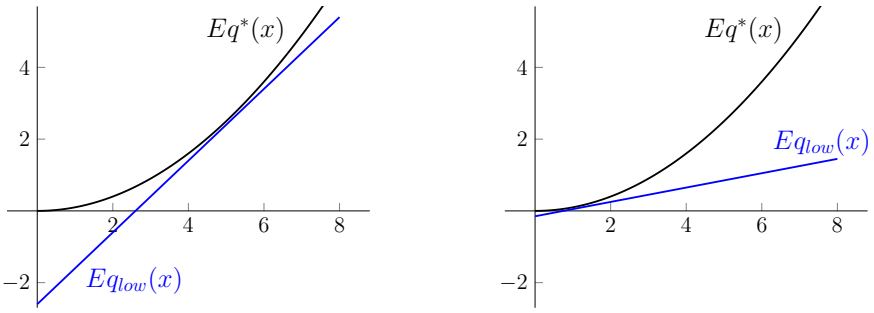
$$= \underset{m \cdot Eq_{low}(x)}{\arg\max} \{m \cdot (\underline{Eq_{low}(x)} + \overline{Eq_{low}(x)})\} \tag{4.7}$$

$$= \begin{cases} Eq_{low}(x), & \text{if } \underline{Eq_{low}(x)} + \overline{Eq_{low}(x)} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

Thus, Debona keeps the lower bound unchanged unless $\underline{Eq_{low}(x)} + \overline{Eq_{low}(x)} < 0$, in which case it is replaced with a constant boundary of zero. This process is referred to as *zero bounding*. The tightened lower bound positively impacts both the positive and negative bounds of subsequent layers. A visualization of zero bounding is shown in Figure 4.2.

4.1.2 Over-Approximation of Upper Bounds

As described in Section 3.1, the upper bound needs to be relaxed if it could be negative. To this end, $\underline{Eq_{up}(x)}$ and $\overline{Eq_{up}(x)}$ have to be computed. For $\underline{Eq_{up}(x)}$, $\underline{Eq_{low}(x)}$ as used by Neurify is a valid over-approximation. However, many possible lower bounds $Eq_{low}(x)$ exist, and it is possible to choose different approximations for different purposes. Specifically, it may be beneficial to trade a weaker overall bound for an increased minimum, if only the minimum is used in the respective formula. Therefore, the $\underline{Eq_{up}(x)}$ in Equation 3.7 could be increased by selecting an appropriate bound. A visualization of this trade off is shown in Figure 4.3.



(a) The lower bound is strong on average, but has a low minimum.

(b) By weakening the bound, the minimum is increased.

Figure 4.3: Maximizing the minimal lower bound.

Empirically, however, the increased computational overhead does not justify the improved boundary accuracy, and therefore, Debona does not use this optimization technique.

4.1.3 Max-Pooling

Convolutional layers are commonly used in combination with pooling operations (see Section 2.2.1). Due to the non-linear nature, the linear upper and lower bounds need to be relaxed when they are propagated through a max-pooling layer. The following approximation can be derived:

$$\max\{Eq_{up}(x_j^{(l)}), \ldots, Eq_{up}(x_{j+k}^{(l)})\} \tag{4.9}$$

$$= \max\{ \sum_{\substack{i\in[1,s_{l-1}],\\ w_{j,i}>0}} w_{j,i} \cdot Eq_{up}(\hat{x}_i^{(l-1)}) + \sum_{\substack{i\in[1,s_{l-1}],\\ w_{j,i}<0}} w_{j,i} \cdot \tilde{Eq}_{low}(\hat{x}_i^{(l-1)}), \ldots, \tag{4.10}$$

$$\sum_{\substack{i\in[1,s_{l-1}],\\ w_{j+k,i}>0}} w_{j+k,i} \cdot Eq_{up}(\hat{x}_i^{(l-1)}) + \sum_{\substack{i\in[1,s_{l-1}],\\ w_{j+k,i}<0}} w_{j+k,i} \cdot \tilde{Eq}_{low}(\hat{x}_i^{(l-1)})\}$$

$$\leq \max\{ \sum_{\substack{i\in[1,s_{l-1}],\\ w_{j,i}>0}} w_{j,i} \cdot Eq_{up}(\hat{x}_i^{(l-1)}), \ldots, \sum_{\substack{i\in[1,s_{l-1}],\\ w_{j+k,i}>0}} w_{j+k,i} \cdot Eq_{up}(\hat{x}_i^{(l-1)})\} \tag{4.11}$$

$$\leq \sum_{i\in[1,s_{l-1}]} \max\{w_{j+r,i} \mid r \in [0,k], w_{j+r,i} > 0\} \cdot Eq_{up}(\hat{x}_i^{(l-1)}) \tag{4.12}$$

$$= \sum_{i\in[1,s_{l-1}]} \max\{w_{j,i}, \ldots, w_{j+k,i}, 0\} \cdot Eq_{up}(\hat{x}_i^{(l-1)}) \tag{4.13}$$

$$\min\{Eq_{low}(x_j^{(l)}), \ldots, Eq_{low}(x_{j+k}^{(l)})\} \tag{4.14}$$

$$\overset{\text{analogous}}{\geq} \sum_{i\in[1,s_i]} \min\{w_{j,i}, \ldots, w_{j+k,i}, 0\} \cdot Eq_{up}(\hat{x}_i^{(l-1)}) \tag{4.15}$$

where $\tilde{Eq}_{low}(\hat{x}_i^{(l-1)}) = \max\{0, Eq_{low}(\hat{x}_i^{(l-1)})\}$ is a valid tightened lower bound if layer $l-1$ uses ReLUs as the activation function.

Preliminary experiments showed that these bounds are too weak, and introduce large uncertainty even if the input bounds are exact. If $x_1 = x_2 = 1$ is known,

$$x_3 = x_1 = 1 \tag{4.16}$$

$$x_4 = -x_1 + x_2 = -1 + 1 = 0 \tag{4.17}$$

$$x_5 = \max\{x_3, x_4\} = \max\{1, 0\} = 1 \tag{4.18}$$

However, the over-approximation gives

$$x_5 = \max\{x_3, x_4\} \tag{4.19}$$

$$= \max\{x_1, -x_1 + x_2\} \tag{4.20}$$

$$\leq \max\{1, -1\}x_1 + \max\{1\}x_2 \tag{4.21}$$

$$= x_1 + x_2 \tag{4.22}$$

$$= 2 \tag{4.23}$$

To further improve the bounds, in addition to the above overestimation, the maximal upper and lower bounds of each input node are taken into account. A node is known to never be selected by the max-pooling operation if its upper bound is smaller than the lower bound of another input node. Thus, this node can be removed from the analysis and does not need to be part of the maximum operation in Equation 4.13. The above example can therefore be improved to

$$x_5 = \max\{x_3, x_4\} \tag{4.24}$$

$$= \max\{x_3\} \tag{4.25}$$

$$= x_3 \tag{4.26}$$

$$= x_1 \tag{4.27}$$

$$= 1 \tag{4.28}$$

where Equation 4.25 utilizes $\underline{x_3} \geq \overline{x_4}$.

4.2 Example

In Section 3.2, the simple example network from Figure 3.4 is used to demonstrate the techniques used during the analysis. Here, the same network is analyzed again, to show how the tightened bounds help to speed up the analysis.

For the concrete inputs of $x_1 = x_2 = 1.5$, $x_5 = 0.3 > 0 = x_6$ was computed (see Equations 3.17 to 3.22). For $L_\infty = 0.5$, the input space is given as $1 \leq \hat{x}_1 \leq 2 \wedge 1 \leq \hat{x}_2 \leq 2$. Equations 3.24 to 3.32 remain the same

$$Eq_{up}(x_3) = \hat{x}_1 \tag{4.29}$$
$$\Rightarrow \underline{Eq_{up}(x_3)} = 1 \wedge \overline{Eq_{up}(x_3)} = 2 \tag{4.30}$$
$$Eq_{low}(x_3) = \hat{x}_1 \tag{4.31}$$
$$\Rightarrow \underline{Eq_{low}(x_3)} = 1 \wedge \overline{Eq_{low}(x_3)} = 2 \tag{4.32}$$
$$Eq_{up}(x_4) = \hat{x}_1 - 1.5\hat{x}_2 \tag{4.33}$$
$$\Rightarrow \underline{Eq_{up}(x_4)} = -2 \wedge \overline{Eq_{up}(x_4)} = 0.5 \tag{4.34}$$
$$Eq_{low}(x_4) = \hat{x}_1 - 1.5\hat{x}_2 \tag{4.35}$$
$$\Rightarrow \underline{Eq_{low}(x_4)} = -2 \wedge \overline{Eq_{low}(x_4)} = 0.5 \tag{4.36}$$
$$Eq_{up}(\hat{x}_3) = Eq_{up}(x_3) \tag{4.37}$$
$$Eq_{low}(\hat{x}_3) = Eq_{low}(x_3) \tag{4.38}$$

For node x_4, $\underline{Eq_{low}(x_4)} + \overline{Eq_{low}(x_4)} = -2 + 0.5 = -1.5 < 0$, so the lower bound is set to zero.

$$\begin{aligned} Eq_{up}(\hat{x}_4) &= \frac{\overline{Eq_{up}(x_4)}}{\overline{Eq_{up}(x_4)} - \underline{Eq_{low}(x_4)}}(Eq_{up}(x_4) - \underline{Eq_{low}(x_4)}) \\ &= \frac{0.5}{0.5 - (-2)}(Eq_{up}(x_4) - (-2)) \\ &= 0.2Eq_{up}(x_4) + 0.4 \end{aligned} \tag{4.39}$$
$$Eq_{low}(\hat{x}_4) = 0 \tag{4.40}$$

Therefore, the bounds for the output layer are

$$\begin{aligned} Eq_{up}(x_5) &= 0.2Eq_{up}(\hat{x}_3) \\ &= 0.2Eq_{up}(x_3) && (4.41) \\ Eq_{low}(x_5) &= 0.2Eq_{low}(\hat{x}_3) \\ &= 0.2Eq_{low}(x_3) && (4.42) \\ Eq_{up}(x_6) &= -Eq_{low}(\hat{x}_4) \\ &= 0 && (4.43) \\ Eq_{low}(x_6) &= -Eq_{up}(\hat{x}_4) \\ &= -(0.2Eq_{up}(x_4) + 0.4) \\ &= -0.2Eq_{up}(x_4) - 0.4 && (4.44) \end{aligned}$$

The search for a variable assignment with $Eq_{up}(x_6) > Eq_{low}(x_5)$ yields the LP problem

$$\begin{aligned} LP &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (Eq_{up}(x_6) > Eq_{low}(x_5)) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (0 > 0.2Eq_{low}(x_3)) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (0 > 0.2\hat{x}_1) \\ &= (1 \leq \hat{x}_1 \leq 2) \wedge (1 \leq \hat{x}_2 \leq 2) \wedge (0 > \hat{x}_1) && (4.45) \end{aligned}$$

which is unsatisfiable. At this point, the analysis can be stopped, and node x_4 does not need to be split. This reduces the overall runtime from three full evaluations in Neurify to one evaluation in Debona.

4.3 Implementation

Similar to Section 3.3, this sections describes some of the implementation specific changes that go beyond simple fixed bugs in Neurify.

4.3.1 Optimization by Tracking Dependencies

Due to the decoupling of the upper and lower bounds, the information necessary to store the impact of an overestimation is increased. Instead of a single delta value, the modifications to all coefficients need to be stored. This is not feasible with the approach by Neurify, as it poses a significant overhead for the computation of node values in deeper layers. To still provide the possibility of tracking dependencies, the symbolic dependencies are tracked backwards from the current node to the input nodes, instead of propagating them forward. This allows to simplify the dependencies at each layer level and achieves the same effect as Neurify

does while enabling the separation of upper and lower bounds. Notably, the repeated backtracking increases the computational cost from $\mathcal{O}((\sum_{l=1}^{n} s_l)^2)$ steps to $\mathcal{O}((\sum_{l=1}^{n} s_l) \cdot (\max_{i \in [0,n]} s_i)^n)$. However, as all currently analyzable networks are relatively shallow, the additional cost is negligible. To further reduce the impact of this change, the symbolic upper and lower bounds of each node are cached to ensure that they are only computed once, even if their respective values are used multiple times.

4.3.2 Picking Splits

For convolutional layers, the gradient is computed by first transforming them into feedforward layers, to avoid the need for an explicit implementation. Determining the correct gradients for max-pooling layers is more complicated, as it is not clear which of the input nodes is chosen for the output. Currently, gradients for max-pooling layers are not computed, setting them to zero, leaving a more reasonable implementation as future work. For feedforward layers, the bug in Neurify is fixed, allowing the successful gradient computation for arbitrarily shaped layers.

4.3.3 Unification of Code

After computing the concrete output values to determine the target label, Neurify performs a one shot approximation of the analysis. Here, the search is aborted once the output intervals are computed, and no splits are performed. This is used to provide some information to the user, such as the number of overestimated nodes. However, this implementation causes a duplication of much of the code relevant for the analysis, as one version performs only the one shot search while the other version of the code tracks necessary splits and additional information that is needed for the full analysis. By unifying both code regions, the development overhead for Debona is significantly decreased. The code responsible for the user output has been moved and all other code that performs the one shot approximation has been deleted. This also fixes a rare edge case in Neurify where the analysis may fail if not a single node is overestimated.

4.3.4 Further Improvements

For some LPs, the solver may take a very long time trying to find a suitable solution. Often, adding small additional constraints alleviates this problem. Therefore, Debona aborts the LP solver if it takes more than 30 seconds to return a result. In these cases, the analysis proceeds with splitting the next node. Furthermore, Debona defines the search space using bounds on the input variables, instead of using constraints that define the upper and lower bounds. For 784 input variables, this removes 1,568 constraints from the LP problem, leading to an additional speedup.

Chapter 5

Experimental Evaluation

This chapter describes both the setup of the experiments that were performed to test the different improvements as well as their respective results.

5.1 Setup

The differences between Neurify and Debona can be split into two categories. Purely implementation specific changes and improvements that decrease the computational cost by reducing the runtime or memory consumption are summarized in the software version *Debona 1.0*. Additionally, *Debona 1.1* includes the gains realized by the tighter bounds as well as the logic for max-pooling operations.

To test the different software versions on a variety of network architectures, eight trained networks are used. *ff2x24*, *ff2x50*, *ff2x512*, *ff3x24*, *ff3x50*, and *ff5x24* are fully connected feedforward networks with the specified number and sizes of hidden layers (i.e., a network structure of e.g. $784 \times 24 \times 24 \times 10$ for ff2x24). For convolutions, the networks *conv2x2* and *conv2x2Pool* are used. conv2x2 consists of two convolutional layers with a 2x2 window, a stride of two and 16 and 32 output channels, respectively. conv2x2Pool uses only one convolutional layer with window size 2x2, stride two and 16 output channels, followed by a max-pooling operation with the same configuration. To compute their final output, both networks have two additional fully connected feedforward layers of size 100 and 10, respectively. To facilitate a comparison with Wang et al. (2018a), the networks ff2x24, ff2x50, ff2x512, and conv2x2 are exactly the same as those used in their evaluations. However, they use single precision floating point algorithmic in their experiments, while, unless otherwise noted, the experiments evaluated here use double precision floating point algorithmic. The networks ff3x24, ff3x50, ff5x24, and conv2x2Pool are trained on the MNIST corpus (LeCun et al., 2010) for six epochs, using the Adam optimizer (Kingma and Ba, 2015) with a learning rate of 0.001. MNIST consists of black and white images depicting handwritten digits using 28×28 pixels. Together, these networks demonstrate the applicability of the analysis on deeper architectures. Importantly, the training of the networks was not optimized either for quality or for robustness, and both aspects could possibly be improved by more carefully chosen

training techniques. However, the aim of this thesis is solely to demonstrate the effectiveness of the analyses, so the networks themselves are not as important for the evaluation.

Unless otherwise noted, all analyses are run on a machine with eight cores and 16 GB RAM. If the analysis has not terminated after one hour (i.e., no adversarial example has been found, but the network has also not been proven to be robust), it is aborted. For each experiment, the networks are tested against 1,000 test images from the MNIST dataset for a maximal input perturbation of $L_\infty = 10$ for all fully connected feed forward networks, $L_\infty = 5$ for conv2x2 and $L_\infty = 1$ for conv2x2Pool. The results listed in the following tables report the average wall-clock time spent on a single image as well as the number of analyses that have been performed. Because the computation is aborted after one hour, including analyses that timed out would distort the average. Similarly, comparing the runtime of a successful analysis for a faster implementation with the timed out analysis of a slower software version is not reasonable, so the reported averages are computed over the common *subset* of the 1,000 test images that could be successfully analyzed by Neurify, Debona 1.0 and Debona 1.1. The averages for *all* successful analyses are given as well, to provide an estimate for the necessary cost associated with the respective results. Furthermore, each experiment is performed twice, to account for hardware or parallelism related fluctuations in the runtime. For each analyzed image, the fastest execution of both experiments is used to compute the overall average, reducing the variance of the measurements.

5.2 Fully Connected Feedforward Networks

For the six different fully connected networks that are analyzed, Table 5.1 summarizes the results. Across all experiments, Debona performs significantly better than the original Neurify software. The implementation specific optimizations done in Debona 1.0 alone lead to a speedup of 17 to 74%, demonstrating the importance of a highly efficient implementation. By further decreasing speed and memory related bottlenecks, maximizing the parallelization and upgrading the underlying hardware, additional improvements should be possible. Debona 1.1 implements the tightened boundary computation by zero bounding described in Section 4.1.1. The increased accuracy reduces the search space and limits the number of splits that need to be performed until a network can be proven to be robust. This further decreases the necessary runtime, leading to an additional improvement of 18 to 87% and a total speedup of 53 to 94%. For all networks except ff2x24 and ff3x24, the average runtime over all successful analyses increases from Neurify to Debona 1.1. This is to be expected, as the faster and tighter approximation allows the verification of networks that result in a timeout in Neurify and that are therefore

omitted in the average over the subset of analyzable inputs. For ff2x24 and ff3x24, the failed analyses are due to bugs in Neurify, which were fixed in Debona and do not cause a slowdown.

Notably, Neurify wrongly returns „no adv. ex.“ for two out of the 6,000 performed analyses, even though an adversarial example provably exists. For the networks ff2x512 and ff3x50, Debona 1.1 fails to find some of the adversarial examples that Neurify was able to detect, causing the respective analyses to result in a timeout. Due to the modified node boundaries, both the order in which nodes are split as well as the results computed by the LP solver change. Even though this effect is able to explain why previously detectable adversarial examples are now missed, it does not provide a reason for why previously undetected adversarial examples are not discovered at the same rate. Therefore, future work should look into possible explanations for this behavior as well as applicable counter measures. However, Debona is able to verify significantly more provably robust networks, so overall the number of unverified inputs is decreased for all experiments.

Due to the parallelism and additional hardware fluctuations, the exact runtime of the individual experiments is not deterministic. Table 5.2 lists the variance of the results. For each setting, it is computed as

$$\frac{1}{|Term|}\sum_{p=1}^{|Term|}\frac{1}{2}\left(\left(Exp_1^{(p)}-\frac{Exp_1^{(p)}+Exp_2^{(p)}}{2}\right)^2+\left(Exp_2^{(p)}-\frac{Exp_1^{(p)}+Exp_2^{(p)}}{2}\right)^2\right) \tag{5.1}$$

where $Term$ is the set of analyses that terminate without a timeout for both executions of the analysis in all three software versions, and $Exp_i^{(p)}$ refers to the i-th execution of the analysis for input p. Notably, $Term$ is a subset of the analyses taken into account for the averages in Table 5.1, as the previous averages require only one of both executions to terminate successfully. The results indicate a general trend of reduced runtime variance for the improved software versions. However, most reported values are heavily influenced by individual experiments that have a significantly higher variance than all other analyses in the same category, indicating that on rare occasions the used cluster nodes may experience slowdowns caused by external factors. To limit the influence of those fluctuations on the comparisons, each experiment should be run repeatedly to report the average or best case. This justifies the additional computational cost of running each analysis twice that was invested in this thesis.

Table 5.1: Performance of analyses for fully connected networks. The subsets analyzable by all networks have sizes 992, 871, 236, 974, 774, and 921, respectively.

Netw.	Tool	Time [s]		Sub-analyses		Analysis result		
		subset	all	subset	all	adv.	non-adv.	un-det.
ff2x24	Neurify	4.3	4.3	140	140	770	222	8
	Deb. 1.0	1.1 (−74%)	1.3	124 (−11%)	150	773	227	0
	Deb. 1.1	0.9 (−79%)	1.0	95 (−32%)	116	773	227	0
ff2x50	Neurify	83.8	83.1	2,833	2,811	663	215	123
	Deb. 1.0	31.1 (−63%)	113.0	2,728 (−4%)	10,065	677	250	73
	Deb. 1.1	13.1 (−84%)	94.2	1,198 (−58%)	8,572	675	264	61
ff2x512	Neurify	6.9	6.2	135	121	252	12	736
	Deb. 1.0	3.1 (−55%)	24.7	126 (−7%)	1,125	252	17	731
	Deb. 1.1	0.4 (−94%)	67.7	2 (−99%)	2,998	224	59	717
ff3x24	Neurify	16.5	16.5	457	457	603	371	26
	Deb. 1.0	5.0 (−70%)	5.6	396 (−13%)	446	608	391	1
	Deb. 1.1	3.8 (−77%)	4.2	295 (−35%)	327	608	391	1
ff3x50	Neurify	134.5	133.7	3,990	3,964	561	218	221
	Deb. 1.0	75.0 (−44%)	194.4	3,894 (−2%)	10,216	562	268	170
	Deb. 1.1	29.6 (−78%)	159.9	1,630 (−59%)	9,019	559	301	140
ff5x24	Neurify	91.1	94.5	2,455	2,563	677	246	77
	Deb. 1.0	75.6 (−17%)	120.7	1,984 (−19%)	3,124	690	260	50
	Deb. 1.1	42.8 (−53%)	102.3	1,124 (−54%)	2,788	691	270	39

5.3 Convolutional Networks

Table 5.3 presents the results for the analyses of the convolutional networks conv2x2 and conv2x2Pool. As Neurify does not support max-pooling layers, conv2x2Pool is only evaluated for Debona 1.1. For the convolutional network without max-pooling operations, Debona 1.1 increases the number of successful robustness proofs by 191, eliminating two thirds of the previously undetermined cases. The increase of runtime by 86% can be explained with the additional preprocessing that is done to transform the convolutions into fully connected layers. However, for the previously analyzable inputs the absolute increase amounts to only 2.4 seconds, justifying this technique. Notably, the average number of analyses over all successful verifications

Table 5.2: Variance of the runtime.

Network	Tool	Time variance [s^2]	Sub-analyses variance
ff2x24	Neurify	11	1,259
	Debona 1.0	0	1,030
	Debona 1.1	2	2,983
ff2x50	Neurify	3,394	1,855,582
	Debona 1.0	506	665,614
	Debona 1.1	40	50,339
ff2x512	Neurify	12	0
	Debona 1.0	1	0
	Debona 1.1	0	0
ff3x24	Neurify	41	9,797
	Debona 1.0	73	4,222
	Debona 1.1	5	3,633
ff3x50	Neurify	3,548	14,528
	Debona 1.0	1,216	2,281,422
	Debona 1.1	56	42,538
ff5x24	Neurify	3,042	16,795
	Debona 1.0	821	47,428
	Debona 1.1	223	69,911

is 1 for Neurify and 6 for Debona 1.1. Therefore, Neurify is not able to proof any robustness if the initial analysis is not already sufficiently tight, i.e., splitting was not powerful enough to limit the search space to a manageable size. Debona however may occasionally perform splits that simplify the verification enough to prove the robustness or to generate adversarial examples.

For conv2x2Pool, the results show that while Debona is able to verify some of the inputs, it fails for 846 settings. Also, for all successful runs, only a single sub-analysis is performed, indicating that splitting does not help to verify more complex settings. Notably, the chosen L_∞ value of 1 is much smaller than that used for pure convolutions or fully connected layers. Future work should further strengthen the upper and lower bounds, to enable stronger robustness proofs for larger perturbations.

Table 5.3: Performance of analyses for convolutional networks. The subset analyzable by all setups for conv2x2 has size 708.

Netw.	Tool	Time [s]		Sub-analyses		Analysis result		
		subset	all	subset	all	adv.	non-adv.	un-det.
con2x2	Neurify	2.8	2.8	1	1	37	673	290
	Deb. 1.0	5.9 (+111%)	6.2	2 (+100%)	2	37	674	289
	Deb. 1.1	5.2 (+86%)	7.3	1 (±0%)	6	37	865	98
conv2x2Pool	Deb. 1.1	3.8	3.8	1	1	7	147	846

5.4 Ablation Study for the Node Selection

To determine the importance of the order in which overestimated nodes are selected for splitting, an ablation study is performed. Table 5.4 lists the results for the original splitting technique as well as two alternative approaches. Usually, the gradient of the output with respect to the node *intervals* is computed. As this requires the gradients to be approximated (see Section 3.1.5), the maximum of the resulting intervals is used to determine which node should be split first. A possible simplification is to use the *concrete* gradients computed for the unmodified input that is used for the analysis. This eliminates the need to recompute the gradients at each step, but as more and more nodes are split, the flow of the gradients changes, and the initial concrete gradients may become less useful. Alternatively, the gradients can be ignored completely, and the nodes can be split according to the order they were defined in the network *architecture*. As evident by the results, the technique for node selection has a significant influence on the overall evaluation speed. For ff3x50, the simplified heuristics result in a slowdown of 36 and 50%, reducing the number of analyzable inputs. The evaluation of conv2x2 is not affected as much by the chosen technique, because almost no splits are performed.

5.5 Floating Point Precision

All previous experiments use double floating point precision for their computations. By reducing the accuracy to single precision floating point numbers, additional speedups can be realized at the cost of less precise computations. Due to rounding errors, the existence of adversarial examples may be missed, causing the network to be declared robust erroneously. Table 5.5 lists the comparison between both setups. The robustness proofs for ff3x50 are sped up significantly, reducing the necessary runtime by 50%. This speedup allows to perform more sub-analyses

Table 5.4: Ablation study for the node selection. The subsets analyzable by all setups have sizes 833 and 899, respectively.

Netw.	Prec.	Time [s]		Sub-analyses		Analysis result		
		subset	all	subset	all	adv.	non-adv.	un-det.
ff3x50	Int.	97.9	159.9	5,518	9,019	559	301	140
	Con.	132.8 (+36%)	177.7	7,487 (+36%)	10,097	556	293	151
	Arch.	147.9 (+51%)	171.1	8,269 (+50%)	9,603	554	288	158
conv2x2	Int.	5.2	7.3	1	6	37	865	98
	Con.	4.8 (−8%)	5.8	1 (±0%)	4	37	865	98
	Arch.	5.4 (+4%)	5.4	1 (±0%)	1	37	862	101

Table 5.5: Speed impact of the floating point precision. The subsets analyzable by all setups have sizes 850 and 902, respectively.

Netw.	Prec.	Time [s]		Sub-analyses		Analysis result		
		subset	all	subset	all	adv.	non-adv.	un-det.
ff3x50	Double	136.4	159.9	7,676	9,019	559	301	140
	Single	68.0 (−50%)	151.8	7,606 (−1%)	17,046	562	317	121
conv2x2	Double	7.3	7.3	6	6	37	865	98
	Single	4.0 (−45%)	4.0	6 (±0%)	6	37	865	98

in the same time, allowing to verify 19 additional inputs. For the convolutional network conv2x2, the absolute difference is small, even though the runtime is sped up by 45%. This is caused by the fact that all successful executions terminate after very few sub-analyses, minimizing the potential for improvement. Networks that cannot be proven robust using double precision floating point algorithmic do not benefit from the switch to single precision as they still result in a timeout.

5.6 Comparison With Other Toolkits

The main focus of this thesis is the improvement over Neurify. However, as listed in Section 3.4, there exist a number of alternative toolkits that attempt to verify networks. VNN-COMP[1] is a competition that tries to compare the different soft-

[1] `https://sites.google.com/view/vnn20/vnncomp`

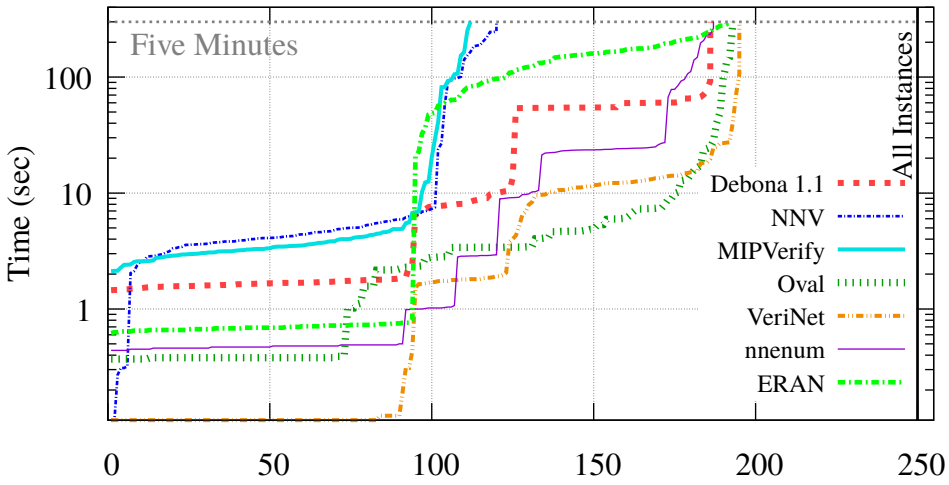


Figure 5.1: Performance comparison with other toolkits. All contestants used different hardware, only nnenum and Debona are comparable.

ware implementations on a common set of tasks to identify their strengths and weaknesses. As every competitor is able to use the hardware of their choice, a fair comparison of the results is not possible. However, it may still provide a first overview, guiding future work. Figure 5.1 shows a visualization of the results over three of the four tasks for convolutional networks from COLT (Mislav Balunovic, 2020). They consist of one network trained on MNIST and two networks trained on CIFAR10 (Krizhevsky, 2009). All three networks are optimized for robustness, allowing for a significantly larger L_∞ norm of 25.5 for MNIST and 2 and 8 for CIFAR10, respectively. For the listed results, Debona was executed on an Amazon EC2 m4.10xlarge cloud instance with a 40-core 2.4GHz processor and 160GB RAM, which is the same architecture used for nnenum. All analyses use a timeout of five minutes and single precision floating point algorithmic. As evident in the visualization, Debona outperforms three of the six alternative toolkits. For nnenum, which is the only possible fair comparison due to the identical hardware, Debona can validate a similar number of inputs but usually requires more time to do so. As nnenum utilizes star sets (Bak and Duggirala, 2017) to improve the approximation, it may be beneficial to also look into possible integrations of star sets into Debona for future work. On the fourth test, which uses a network trained for MNIST with $L_\infty = 76.5$, Debona results in a timeout for 24 of the 99 inputs.

Chapter 6

Conclusion and Outlook

In conclusion, this thesis has shown that the decoupling of upper and lower bounds allows for zero bounding, which provides significant improvements to the tightness of the over-approximation. By limiting the search space, the verification can be sped up, allowing for the analysis of inputs that previously resulted in a timeout. For fully connected layers, Debona 1.1, which supports zero bounding and removes implementation specific bottlenecks, is able to analyze inputs up to 94% faster than Neurify. For convolutions, Debona takes 2.4 seconds longer for easy inputs, due to additional preprocessing. However, it is able to analyze two thirds of the inputs that previously resulted in a timeout, providing a significant improvement. Furthermore, this thesis provides over-estimations for the max-pooling operation commonly used in convolutional networks. However, the performed experiments show that in most cases, the resulting bounds are not tight enough to enable the successful analysis. By performing an ablation study for the selected floating point precision, it has been shown that the limitation to single precision allows for additional significant speedups. Future work should look into techniques such as outward rounding, in order to guarantee sound analysis results even for rounded operations caused by a reduced floating point precision. The comparison of different selection algorithms for the splitting of nodes showed a significant slowdown for less accurate techniques. Therefore, more sophisticated heuristics should be developed and analyzed to speed up the detection of adversarial examples. The comparison of Debona with other verification toolkits showed that it outperforms three of the six analyzed softwares, performing similar to the three other implementations. As Debona fails to return results within the timeout for most of the inputs in the fourth test, future work should perform a more detailed comparison of the techniques used in the different toolkits, to combine them in a single software. Furthermore, future work may try to verify robustness as well as other more general safety properties for more complex network architectures.

List of Figures

List of Tables

Glossary

FFNN feedforward neural network. 5

GRU gated recurrent unit. 5

LSTM long short-term memory. 5

RNN recurrent neural network. 5

Bibliography

Anish Athalye and Nicholas Carlini. On the robustness of the CVPR 2018 white-box adversarial example defenses. *CoRR*, abs/1804.03286, 2018. URL `http://arxiv.org/abs/1804.03286`.

Stanley Bak. Execution-guided overapproximation (ego) for improving scalability of neural network verification, 2020.

Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 401–420. Springer, 2017. doi: 10.1007/978-3-319-63387-9_20. URL `https://doi.org/10.1007/978-3-319-63387-9_20`.

Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. URL `http://lpsolve.sourceforge.net/5.5/`.

Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. URL `http://arxiv.org/abs/1604.07316`.

Christopher Brix. *Extension of the attention mechanism in neural machine translation.* Bachelor's thesis, RWTH Aachen University, March 2018.

Christopher Brix and Thomas Noll. Debona: Decoupled boundary network analysis for tighter bounds and faster adversarial robustness proofs, 2020.

Rudy Bunel, Ilker Turkaslan, Philip H.S. Torr, Pushmeet Kohli, and M. Pawan Kumar. Piecewise linear neural networks verification: A comparative study. *arxiv:1711.00455*, 2017.

Rudy Bunel, Jingyue Lu, Ilker Turkaslan, Pushmeet Kohli, Philip H.S. Torr, and M. Pawan Kumar. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017. doi: 10.1109/SP.2017.49.

Kyunghyun Cho, Bart van Merriënboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Bengio Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1179`.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL `https://doi.org/10.1007/BF02551274`.

Krishnamurthy Dvijotham, Sven Gowal, Robert Stanforth, Relja Arandjelovic, Brendan O'Donoghue, Jonathan Uesato, and Pushmeet Kohli. Training verified learners with learned verifiers. *CoRR*, abs/1805.10265, 2018. URL `http://arxiv.org/abs/1805.10265`.

Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. A guide to deep learning in healthcare. *Nature medicine*, 25(1): 24–29, 2019. doi: 10.1038/s41591-018-0316-z.

Jared Lee Gearhartq, Kristin Lynn Adair, Justin David. Durfee, Katherine A. Jones, Nathaniel Martin, and Richard Joseph Detry. Comparison of open-source linear programming solvers. 10 2013. doi: 10.2172/1104761.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL `http://proceedings.mlr.press/v15/glorot11a.html`.

Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015. URL `http://arxiv.org/abs/1412.6572`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Patrick Henriksen and Alessio Lomuscio. Efficient neural network verification via adaptive refinement and adversarial search. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI20)*, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification*, pages 97–117. Springer, 2017. doi: 10.1007/978-3-319-63387-9_5.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations (ICLR 2015)*, 2015. URL http://arxiv.org/abs/1412.6980.

Sotiris B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies*, page 3–24, NLD, 2007. IOS Press. ISBN 9781586037802.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Yann Lecun and Y. Bengio. Convolutional networks for images, speech, and time-series. 01 1995.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits, 2010. URL http://yann.lecun.com/exdb/mnist/.

Mathias Lécuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 656–672. IEEE, 2019. doi: 10.1109/SP.2019.00044. URL https://doi.org/10.1109/SP.2019.00044.

Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=Sys6GJqxl`.

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

Martin Vechev Mislav Balunovic. Adversarial training and provable defenses: Bridging the gap. In *Proc. International Conference on Learning Representations (ICLR)*, 2020.

Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, USA, 2009. ISBN 0898716691.

Nir Morgulis, Alexander Kreines, Shachar Mendelowitz, and Yuval Weisglass. Fooling a real car with adversarial traffic signs. *CoRR*, abs/1907.00374, 2019. URL `http://arxiv.org/abs/1907.00374`.

Nina Narodytska and Shiva Kasiviswanathan. Simple black-box adversarial attacks on deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1310–1318, 2017.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436. IEEE, 2015. doi: 10.1109/CVPR.2015.7298640.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016. doi: 10.1109/EuroSP.2016.36.

Frank Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton (Project Para)*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL `https://books.google.de/books?id=P_XGPgAACAAJ`.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10802–

10813. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification.pdf.

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. In *Advances in Neural Information Processing Systems 32*, pages 15098–15109. Curran Associates, Inc., 2019a. URL http://papers.nips.cc/paper/9646-beyond-the-single-neuron-convex-barrier-for-neural-network-\certification.pdf.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL): 41:1–41:30, 2019b. ISSN 2475-1421.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *Proc. International Conference on Learning Representations (ICLR)*, 2019c.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199v4, 2013. URL http://arxiv.org/abs/1312.6199v4.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *ICLR*, 2019.

Hoang-Dung Tran, Patrick Musau, Diego Manzanas Lopez, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-based reachability analysis for deep neural networks. In *23rd International Symposium on Formal Methods (FM'19)*. Springer International Publishing, October 2019a.

Hoang-Dung Tran, Patrick Musau, Diego Manzanas Lopez, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Parallelizable reachability analysis algorithms for feed-forward neural networks. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering (FormaliSE'19)*, FormaliSE '19, pages 31–40, Piscataway, NJ, USA, May 2019b. IEEE Press. doi: 10.1109/FormaliSE.2019.00012.

Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. Verification of deep convolutional neural networks using imagestars. In *32nd International Conference on Computer-Aided Verification (CAV)*. Springer, July 2020a.

Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. NNV:

The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *32nd International Conference on Computer-Aided Verification (CAV)*, July 2020b.

Shiqi Wang, Kexin Pei, Whitehouse Justin, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6369–6379, Red Hook, NY, USA, 2018a. Curran Associates Inc. URL `http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks.pdf`.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium*, pages 1599—1614. USENIX Association, 2018b. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi`.

Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J. Zico Kolter. Scaling provable adversarial defenses. In *Advances in Neural Information Processing Systems 31*, pages 8400–8409. Curran Associates, Inc., 2018. URL `http://papers.nips.cc/paper/8060-scaling-provable-adversarial-defenses.pdf`.

Hui Wu, Hui Zhang, Jinfang Zhang, and Fanjiang Xu. Typical target detection in satellite images based on convolutional neural networks. In *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 2956–2961. IEEE, 2015. doi: 10.1109/SMC.2015.514.

Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5777–5783, 2018.

Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *IEEE 1988 International Conference on Neural Networks*, pages 71–78 vol.2, 1988.