Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

ANSGAR RÖSSIG

Verification of Neural Networks

ZIB Report 19-40 (Aug 2019)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Contents

1 Introduction

In this thesis we consider the problem of verifying linear properties of neural networks. Neural networks are employed succesfully for a broad range of tasks in application domains such as computer vision, speech recognition, or natural language processing (cf. Goodfellow et al. [27]). However, despite their success in many classification and prediction tasks, neural networks may return unexpected results for certain inputs. This is highly problematic with respect to the application of neural networks for safety-critical tasks, e.g. in autonomous driving. During the last few years, various approaches have been presented that aim to provide formal guarantees on the behaviour of neural networks. The use of such verification methods may be crucial to enable the secure and certified application of neural networks for safety-critical tasks. Moreover, based on first results of Szegedy et al. [50], awareness was raised that neural networks are prone to fail on so called adversarial examples. These are created by small perturbations of input samples, such that the changes are (almost) imperceptible to humans. However, these perturbations are often sufficient to make a neural network fail on the input sample. The existence of such adversarial examples can be ruled out by methods of neural network verification. In fact, a closely related line of research termed as robustness certification is focused explicitly on this topic.

The contribution of this thesis to the field of neural network verification is twofold. On the one hand, we aim to provide a good overview of the algorithmic approaches used for verification of neural networks with ReLU activation function. As detailed later, the ReLU function is piecewise linear and very commonly used as activation function in neural networks. We also present new theoretical results with respect to the approximation of ReLU neural networks. A strong focus is laid on methods for the approximation of neuron bounds, which are crucial for the performance of the known solving methods. Furthermore, we conduct extensive computational experiments on a combination of various sets of benchmark instances to compare different solvers. On the other hand, we also implement a solving model for verification of ReLU neural networks. It is based on the mixed integer programming (MIP) solver SCIP [25] and features a combination of several solving techniques. While some of these techniques are novel, many others have been proposed by various authors. However, to the best of our knowledge, we provide the first solver which allows a flexible combination of these different techniques. Especially, we report detailed runtime results to gain insights into the empirical performance of various algorithmic ideas. Of course, we also include our solving model in the computational study we perform. The results show advantages of integrating neural network verification with MIP solving and indicate that our approach is very competitive with others. Besides, it should be noted that our solving model is the only one which is able to solve the verification problem for instances which do not have independent bounds for each input neuron (see Remark 6).

The thesis is organized as follows. In the following sections of this introduction, we formally define the verification problem for ReLU neural networks and some other fundamental concepts. Chapter 2 covers previous research which is relevant in the context of neural network verification. The verification problem can be formulated as an MIP as we will see in Chapter 3. An extensive overview of methods for the approximation of ReLU neural networks, which serve for the computation of neuron bounds, is given in Chapter 4. In Chapter 5 we explain methods that serve to falsify incorrect properties of neural networks. Especially, we present a new primal heuristic which has this purpose. A generic branching algorithm for solving the verification problem can be found in Chapter 6. Additionally, we explain two specific branching rules and discuss how different solvers for the verification problem implement the generic algorithm. Details on our own implementation are presented in Chapter 7. In Chapter 8 we provide information on the origin and selection of the test instances that we use for our computational experiments. The corresponding results are reported in Chapter 9. Chapter 10 concludes the thesis with some final remarks. In the appendix, we report definitions of some newly created test cases, additional experimental results and provide a german summary of the thesis. The code of our implemented solving model and the benchmark instances, which are used for our computational experiments, can be found at `https://github.com/roessig/verify-nn`.

1.1 Preliminary notes

While the application of this thesis is clearly in the broad field of machine learning, this does not hold for the techniques used in itself. These originate mostly in the field of linear and discrete optimization. Hence we assume a certain familiarity of the reader with corresponding concepts. For the foundations of linear programming (LP, also for linear program) we refer to Bertsimas and Tsitsiklis [5]. A thorough introduction to discrete optimization can be found in Bertsimas and Weismantel [6]. Moreover, the theory of (convex) polyhedra is an essential basis for solving linear and discrete optimization problems. As we will consider some geometrical questions in Chapter 4, we recommend Grünbaum [29] for a solid overview of this topic.

Nevertheless, fundamental knowledge of machine learning and neural networks may be beneficial for the understanding of the motivation and results of this thesis. For a general introduction to machine learning we recommend Bishop [7], and Goodfellow et al. [27] for an extensive overview of deep learning and neural networks. Throughout this thesis, we will not consider the training process of neural networks (or other machine learning models). Subsequently, we regard neural networks as immutable and deterministic functions. We define the notion of a neural network based on the rigorous definition given in Bölcskei et al. [8]. For a textbook reference see e.g. Bishop [7], Section 5.1. In the definition and the rest of the thesis we use $[n]$ to denote the set $\{1, \dots, n\}$ for some $n \in \mathbb{N}$.

Definition 1 (Neural Network, cf. Bölcskei et al. [8], Definition 1.1)**.** *Let $L \in \mathbb{N}$, such that $L \geq 2$ is the number of layers and let $N_0, N_1, \dots, N_L \in \mathbb{N}$ be*

the numbers of neurons in each layer. That means, N_0 is the number of inputs to the neural networks and N_L the number of outputs. Let $A_l \in \mathbb{R}^{N_l \times N_{l-1}}$ for $l \in [L]$ be the weights of the network and $b_l \in \mathbb{R}^{N_l}$, $l \in [L]$ be the biases. The family $((A_l, b_l))_{l=1}^L$ together with a nonlinear activation function $\sigma : \mathbb{R} \to \mathbb{R}$ defines a neural network.

We will often regard a neural network as a function $F : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ which is defined by $F(x) := x_L$ for $x \in \mathbb{R}^{N_0}$, where x_L is obtained as follows:

$$\begin{aligned} x_0 &:= x \\ x_l &:= \sigma(A_l x_{l-1} + b_l) \qquad \forall l \in [L-1] \\ x_L &:= A_L x_{L-1} + b_L \end{aligned}$$

The activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is applied component-wise. We will sometimes refer to the family $((A_l, b_l))_{l=1}^L$ as the weights and biases corresponding to F. *The process of computing $x_1, \dots, x_L$ out of an input vector x_0 is called* forward propagation. *Sometimes the layers $1, \dots, L-1$ are called* hidden layers*, in contrast to the input layer 0 and output layer L.*

Remark 2. Throughout this thesis, we will strongly focus on the ReLU function as activation function which is defined by $\mathrm{ReLU}(x) := \max\{0, x\}$ for $x \in \mathbb{R}$. Indeed, common architectures of feedforward neural networks may also contain activation functions that do not map from $\mathbb{R}$ to $\mathbb{R}$. One example is the max-pooling function, which maps from $\mathbb{R}^n$ to $\mathbb{R}$ for some $n \in \mathbb{N}$, $n \geq 2$. In order to keep Definition 1 clear and concise, we restrict it to the case which is mostly relevant for this thesis.

1.2 Definition of the verification problem

In this section we give a formal definition of the verification problem for ReLU neural networks which forms the topic of this thesis. Additionally, we comment on some relevant properties of this problem and give a few further definitions.

Definition 3 (Verification Problem for ReLU Neural Networks)**.** *Assume that $\emptyset \neq X \subset \mathbb{R}^n$ is a polytope, and let $\emptyset \neq Y \subset \mathbb{R}^m$ be such that $Y = \bigcap_{i=1}^k Q_i$ or $Y = \bigcup_{i=1}^k Q_i$ where $k \in \mathbb{N}$ and $Q_i \subseteq \mathbb{R}^m$ is a halfspace for $i \in [k]$. Given a neural network $F : X \to \mathbb{R}^m$ with ReLU activation function, the verification problem consists in the decision whether $F(X) \subseteq Y$ holds. A triple (X, Y, F) will be called an* instance of the verification problem (for ReLU neural networks)*. Furthermore, if $F(X) \subseteq Y$, we say that the instance is* verifiable*, otherwise it is* refutable*.*

Depending on the model which is used to solve the problem, the halfspaces Q_i can either be open or closed. Though, either all of them must be closed or all of them must be open. Indeed, the use of floating point arithmetic by a solver for the verification problem makes this distinction rather unimportant, since numerical comparisons require the use of a certain threshold difference. Moreover, Katz et al. [35] show that the verification problem for ReLU neural networks is NP-complete. Hence we cannot expect that the problem can be solved efficiently in general. We also follow the naming concept of Katz et al.

[35] and refer to verifiable instances of the verification problem as *UNSAT* instances, and to refutable instances as *SAT* instances. This naming corresponds to the existence of a counterexample as defined in the following remark.

Remark 4. If an instance (X, Y, F) is refutable, i.e. $F(X) \not\subseteq Y$, we want to provide $x \in X$ such that $F(x) \notin Y$. We will refer to this $x \in X$ as a *counterexample* for the instance.

Remark 5. More complex properties can be investigated by spliting them into separate instances. For example, if $Y = \left(\bigcup_{i=1}^{k} Q_i\right) \cap \left(\bigcup_{j=1}^{l} P_j\right)$ for halfspaces Q_i and P_j and $k, l \in \mathbb{N}$, then $F(X) \subseteq Y$ holds if and only if $F(X) \subseteq (\bigcup_i^k Q_i)$ and $F(X) \subseteq (\bigcup_i^l P_i)$.

Remark 6. Considering an instance $\Pi = (X, Y, F)$ of the verification problem with $X \subset \mathbb{R}^n$, we will often assume the existence of bounds l_i, u_i for $i \in [n]$ such that $l_i \leq x_i \leq u_i$ for $x \in X$. Indeed, the requirements of Definition 3 justify this assumption. These bounds can be computed by solving one LP per bound. We set

$$l_i := \min_{x \in X} x_i \qquad \text{and} \qquad u_i := \max_{x \in X} x_i \qquad \text{for } i \in [n].$$

In fact, for all publicly available instances of the verification problem that we are aware of, the polytope X is actually a box which is directly given by the bounds l_i, u_i for $i \in [n]$. For these instances it is thus not necessary to solve any LP in order to obtain the bounds. However, in this thesis we also consider instances where X is not a box, cf. Section 8.2.

Remark 7. Assume that we are given an instance $\Pi = (X, Y, F)$ of the verification problem as introduced in Definition 3. Some solving models are not only limited to instances where the input polytope X is in fact a box. Also the choice of output constraints as represented by Y is more restricted for some solving models. These require that $Y = \bigcup_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$, where $k \in \mathbb{N}$ and $Q_i \subseteq \mathbb{R}^m$ is an open halfspace for $i \in [k]$. Indeed, this is the only of the cases which are regarded in Definition 3 where $\mathbb{R}^m \backslash Y$ is a polyhedron. Yet, it is possible to use such restricted solving models to solve an instance $\Pi = (X, Y, F)$ where $Y = \bigcap_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$ for open halfpaces $Q_i \subseteq \mathbb{R}^m$. To this end, it is necessary to split the corresponding instance into k instances (X, Q_i, F). Clearly, if $F(X) \subseteq Q_i$ for all $i \in [k]$, then it holds $F(X) \subseteq Y$ and Π is verifiable. On the other hand, if there is $x \in X$ and some $i \in [k]$ such that $F(x) \notin Q_i$, we know that Π is refutable since $F(X) \not\subseteq Y$. We will refer to such an instance Π as *conjunction instance*. On the other hand, an instance $\Pi = (X, Y, F)$ where $Y = \bigcup_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$ for open halfspaces $Q_i \subseteq \mathbb{R}^m$, will be called *disjunction instance*. We will also regard those instances as disjunction instances that fulfill $Y = Q$ for some open halfspace $Q \subseteq \mathbb{R}^m$. In fact, all instances that we consider in our computational experiments (see Chapter 8) are based on open halfspaces. Closed halfspaces are only mentioned in some cases to provide a comprehensive explanation.

1.3 Further definitions and notation

Throughout the thesis, we will consider subsets and elements of $\mathbb{R}^n$, where we assume that $n \in \mathbb{N}$, $n \geq 1$. In general, we assume $n \geq 1$ for $n \in \mathbb{N}$, too. Remind that we use $[n]$ to denote the set $\{1, \ldots, n\}$ for some $n \in \mathbb{N}$. Given a vector $x \in \mathbb{R}^n$, we will often refer to its components as x_i, $i \in [n]$, i.e. we assume $x = (x_1, \ldots, x_n)^T$. If other indices are present, we will also write $[x]_i$ instead of x_i. For example, e_i with $i \in [n]$ refers to the i-th unit vector in $\mathbb{R}^n$, so that $[e_i]_i = 1$ and $[e_i]_j = 0$ for $j \in [n] \backslash \{i\}$.

In accordance with Boyd and Vandenberghe [10], Section 2.2.4, we say that $P \subseteq \mathbb{R}^n$ is a *polyhedron* if it is the intersection of a finite family of closed halfspaces of $\mathbb{R}^n$. If P is also bounded, it is a *polytope*. Given $c \in \mathbb{R}^n$ and $\gamma \in \mathbb{R}$, $c^T x \leq \gamma$ is a *valid inequality* with respect to P if $c^T x \leq \gamma$ holds for all $x \in P$. Based on Grünbaum [29], Section 3.1, we note that a polytope is the convex hull of its extreme points which are called *vertices*. We use $\text{Vertices}(P)$ to refer to the set of vertices of polytope P. Any affine image of a polytope is again a polytope (cf. Grünbaum [29], Section 3.1). Especially, this holds for orthogonal projections which we shortly discuss in the following.

Given a linear subspace $V \subset \mathbb{R}^n$ of $\mathbb{R}^n$, the mapping $p_V : \mathbb{R}^n \to V$ is the *orthogonal projection* on V, if $(x - p_V(x))^T v = 0$ holds for all $x \in \mathbb{R}^n$ and all $v \in V$ (cf. Bosch [9], Section 7.2). In some cases we will consider an orthogonal projection p_V in $\mathbb{R}^n$ onto a subspace $V = \{x \in \mathbb{R}^n \mid \forall i \in I : x_i = 0\} \subset \mathbb{R}^n$ for an index set $\emptyset \neq I \subset [n]$. Without loss of generality, we assume $I = [m]$ for some $m \in \mathbb{N}$, $1 \leq m \leq n-1$. Then we see that V is isomorphic to $\mathbb{R}^m$ since $V = \{(x, 0)^T \in \mathbb{R}^n \mid x \in \mathbb{R}^m\}$. Given $A \subset \mathbb{R}^n$ and the resulting image $p_V(A) \subset \mathbb{R}^n$, we regard the natural embedding of this set in the space $\mathbb{R}^m$. We will refer to the embedding of $p_V(A)$ in $\mathbb{R}^m$ as the *embedded image* of the projection. It should be noted that this embedding is unique up to the ordering of the components. Assume that we have a polytope

$$P := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^{n+m} \;\middle|\; Ax + By \leq c \right\} \subset \mathbb{R}^{n+m}$$

for some $A \in \mathbb{R}^{k \times n}$, $B \in \mathbb{R}^{k \times m}$ and $c \in \mathbb{R}^k$. In this case, the embedded image of *projecting P onto the x variables* is the set $\{x \in \mathbb{R}^n \mid \exists y \in \mathbb{R}^m : Ax + By \leq c\}$.

Assume that we are given a set $V \subset \mathbb{R}^n$. According to Bosch [9], Section 1.4, we denote the linear span of this set as

$$\text{span}(V) := \left\{ \sum_{i=1}^{k} \lambda_i v_i \;\middle|\; k \in \mathbb{N}, \forall i \in [k] : \lambda_i \in \mathbb{R}, v_i \in V \right\}.$$

Furthermore, corresponding to Boyd and Vandenberghe [10], Section 2.1, the convex hull of V is given as

$$\text{conv}(V) := \left\{ \sum_{i=1}^{k} \lambda_i v_i \;\middle|\; k \in \mathbb{N}, \sum_{i=1}^{k} \lambda_i = 1,\ \forall i \in [k] : \lambda_i \geq 0, v_i \in V \right\},$$

and the affine hull of V as

$$\text{aff}(V) := \left\{ \sum_{i=1}^{k} \lambda_i v_i \;\middle|\; k \in \mathbb{N}, \sum_{i=1}^{k} \lambda_i = 1,\ \forall i \in [k] : \lambda_i \in \mathbb{R}, v_i \in V \right\}.$$

As in Grünbaum [29], Section 1.1, the dimension of the set V is defined by its affine hull, $\dim(V) := \dim(\operatorname{aff}(V))$. For that, it should be noted that $\operatorname{aff}(V)$ can be written as translation of a linear subspace $U \subseteq \mathbb{R}^n$. Hence, the dimension of $\operatorname{aff}(V)$ is given as $\dim(U)$ (see Grünbaum [29], Section 1.1). Especially, this is applicable for the case that V is a polyhedron.

In Chapter 4 we will present a semidefinite programming (SDP) relaxation of a ReLU neural network as suggested by Raghunathan et al. [43]. SDP is a special case of convex optimization, which is based on the cone of positive semidefinite square matrices. We refer to Boyd and Vandenberghe [10], Chapter 4 for an introduction to this topic.

As mentioned before, we define $\operatorname{ReLU}(x) := \max\{0, x\}$ for $x \in \mathbb{R}$. Moreover, for $x \in \mathbb{R}^n$, we let $\operatorname{ReLU}(x) := (\operatorname{ReLU}(x_1), \ldots, \operatorname{ReLU}(x_n))^T$ and for $X \subset \mathbb{R}^n$ we say that $\operatorname{ReLU}(X) := \{\operatorname{ReLU}(x) \mid x \in X\}$ is the ReLU image of X. Often, we will regard constraints of the form $y = \operatorname{ReLU}(x)$ for $x \in [l, u]$, $y \in \mathbb{R}$, that refer to a certain neuron with ReLU activation function. If the bounds $l, u \in \mathbb{R}$ with $l \leq u$ are such that either $l \geq 0$ or $u \leq 0$, we say that the corresponding neuron is *fixed in its phase*. That means, in these cases it holds $y = \operatorname{ReLU}(x) = x$ or $y = \operatorname{ReLU}(x) = 0$, respectively.

To describe the capacities of various algorithms for neural network verification and robustness certification, we borrow two terms from the field of mathematical logic. Grassmann and Tremblay [28] state that an *argument is* sound *if the premises together logically imply the conclusion*. Furthermore, they define that a system is *complete*, if it is *possible to derive every conclusion that logically follows from the premises*. These terms can also be used for the description of algorithms that solve decision problems. We say that such an algorithm is sound, if each result, which the algorithm computes, is correct. Clearly, this should hold for any reasonable algorithm. For that reason we assume the soundness of all algorithms which we consider, unless stated otherwise. An algorithm is complete, if it returns a result for every instance of the problem that it shall solve. Due to time and memory limits, an algorithm can rarely solve each problem instance computationally. However, for our presentation in Chapter 2, we are mainly interested in the question whether an algorithm is able to solve all instances from a theoretical point of view. Some of the algorithms we consider are not able to do so and therefore incomplete.

2 Related Work

Three slightly different problems are regarded in the literature which is relevant for the topic of this thesis. In fact, results are presented in the context of (i) neural network verification, (ii) output range analysis, or (iii) certified robustness. Definition 3 formalizes the verification problem for ReLU neural networks. While the exact definition is a result of the work on this thesis, the key properties of this problem are considered likewise in the literature [11, 12, 14, 17, 19, 35, 38, 41, 42, 51, 57, 56, 61, 62]. In view of an instance $\Pi = (X, Y, F)$ of "our" verification problem, these common properties can be summarized as follows.

A box, a polytope or a union of polytopes is defined as the feasible input domain X for the property which shall be verified. Then, linear properties are defined that we denote in terms of a set Y, such that Π is verifiable if and only if $F(X) \subseteq Y$. Complete algorithms are employed to solve this problem, i.e. if there exists $\tilde{x} \in X$ such that $F(\tilde{x}) \notin Y$, this will be reported. Dvijotham et al. [17] focus on the scalability to larger networks and therefore use an incomplete procedure. That means, sometimes they can prove that $F(X) \subseteq Y$, but in other cases their algorithm cannot compute any meaningful result. Of course, time and memory limits may render the other algorithms incomplete, too. Furthermore, the verification problem is not necessarily limited to neural networks with ReLU activations, i.e. other activation functions are sometimes considered, too. Cheng et al. [13] and Narodytska et al. [40] consider the verification problem on binarized neural networks, where weights and activations are constrained to assume one of the values 1 and -1. This allows to create more specific formulations of the problem that can be solved by SAT solvers. However, we do not regard this variant of the verification problem in detail. Indeed, the scope of this thesis is limited to the verification problem on standard neural networks with ReLU activations.

The idea of output range or reachability analysis is in principle to compute the output range $F(X)$ of a neural network F, given an input domain X. Since this is quite difficult, the relevant work of Dutta et al. [16] and Ruan et al. [44] is limited to computing the range $g(F(X))$, for some function $g : F(X) \to \mathbb{R}$. The function g should then give some insights into the output of the neural network F on input domain X. Clearly, this problem is very closely related to the verification problem.

Several authors [23, 43, 47, 48, 49, 52, 58, 59, 60, 63, 64] consider the problem of computing robustness guarantees for neural networks which are used for classification. Given an input sample x_0, the task of such a neural network F consists in computing $F(x_0) \in C$, where C is a finite set of classes. A frequent application is the classification of contents of an image such as handwritten digit recognition based on the MNIST data set [37]. Robustness means, that the classification of an input sample should remain the same when the input is changed by small perturbations. The computation of certified robustness bounds should rule out the existence of adversarial examples. As

mentioned before, adversarial examples are obtained by slightly perturbing an input sample which is classified correctly by the neural network in question. If the perturbations (although small with respect to some norm) suffice to provoke an incorrect classification of the obtained input sample, this is called an adversarial example. Indeed, this problem is a special case of neural network verification. In this context, the input domain $X \subset \mathbb{R}^n$ is typically chosen as $X := B_\varepsilon(x_0) = \{x \in \mathbb{R}^n \mid \|x - x_0\| \leq \varepsilon\}$ for some given input vector $x_0 \in \mathbb{R}^n$ and $\varepsilon > 0$. The input vector x_0 should be classified correctly by the neural network and the goal is to prove that this classification is robust to small, norm bounded perturbations. Various norms are considered in the literature, e.g. p-norms or the maximum norm, which can be easily represented by linear inequations. It is the goal to prove $F(X) = F(x_0)$ for some $\varepsilon > 0$. Clearly, binary search can be performed over the value of ε, in order to find the maximum value of ε such that $F(X) = F(x_0)$ holds. Except for Tjeng et al. [52], this problem is solved by incomplete algorithms. That means, an algorithm either returns a guarantee that $F(X) = F(x_0)$, or no result. In the latter case, it then remains unclear whether there is a counterexample $\tilde{x} \in X$, such that $F(\tilde{x}) \neq F(x_0)$. This happens, as approximations are used which can be sufficient to prove robustness properties, but failing on this task does not imply the presence of counterexamples.

Huang et al. [31] propose an algorithm that verifies neural networks for image classification with respect to manipulations of the input images. However, they assume the existence of "minimal manipulations" and hence the resulting algorithm is not sound. That means, robustness may be reported although counterexamples exist which cannot be obtained as a combination of such "minimal manipulations".

In the following sections we give an overview over the relevant work on neural network verification, output range analysis, and on robustness certification of neural networks. Owing to the topic of the thesis, we regard the literature on neural network verification in more detail.

2.1 Neural network verification

A first approach to verification of neural networks, presented by Pulina and Tacchella [41], goes back to the year 2010. They implement a tool NeVer for verification of neural networks based on the solver HySAT (see Fränzle and Herde [21]). HySAT integrates a SAT solver with a linear programming solving routine. However, the approach of Pulina and Tacchella [41], with an updated version in Pulina and Tacchella [42], remains restricted to very small neural networks of less than 30 neurons. In fact, the approach of Pulina and Tacchella [41] belongs to the field of satisfiability modulo theories (SMT). SMT generalizes the boolean satisfiability problem by replacing variables with predicates from various theories. One of such theories is the linear real arithmetic, which represents the standard model of the real numbers. Scheibler et al. [46] use the SMT solver iSAT3 [45] for an approach towards the verification of neural networks. iSAT3 [45] is a successor of the SMT solver HySAT [21]. Though, the solving model of Scheibler et al. [46] remains limited to very small neural networks, too.

Also the solver Reluplex for verification of neural networks is presented in this context, as laid out by Katz et al. [35]. However, it is able to solve instances which are significantly more difficult. In fact, Reluplex solves the verification problem using an extended version of the well known simplex algorithm. Katz et al. [35] include additional updating and pivoting rules, that allow the processing of ReLU constraints within the solving procedure. The open source LP solver GLPK [26] is used as a basis for that. Moreover, Katz et al. [35] introduce the most important set of benchmark instances for verification of neural networks which will be covered in detail in Sections 8.1 and 8.2. Ehlers [19] presents the solver Planet, which is also based on the LP solver GLPK [26], and on the SAT solver Minisat (see Eén and Sörensson [18]). However, the LP solver is used as it is to compute linear approximations of the neural networks, while the SAT solver is adapted to solve the verification problem as a satisfiability problem. Computational experiments of Bunel et al. [11, 12] show, that Planet is not competitive with Reluplex. Yet, Ehlers [19] introduces a linear approximation of ReLU constraints for Planet, which is still relevant for other solvers. Especially we use it in our solving model and hence present it as (4.4) in Section 4.1.

Dvijotham et al. [17] formulate the verification problem as a non-convex optimization problem (very similar to an MIP formulation), and consider a Langrangian relaxation of the problem. Using this relaxation, also properties on big neural networks can be proved. Though, the approach of Dvijotham et al. [17] is incomplete and therefore in general not able to decide whether counterexamples exist for an instance of the verification problem.

Xiang et al. [62] regard the propagation of an input polytope through a ReLU neural network, similar to our considerations in Section 4.4. Independently, they also state and prove that the ReLU image of a polytope is a union of polytopes (see Theorem 33). However, their work remains limited to theoretical considerations and the presentation of a numerical toy example. Especially, it remains unclear whether it is possible to implement the approach efficiently, as it tends to produce exponentially big numbers of polytopes. In Xiang et al. [61], a more practical approach is presented. It is based on discretizing the feasible input domain X to obtain single input vectors. Then the "maximum sensitivity" is computed for each of these vectors, which guarantees a maximum change of the neural network output if the perturbation of the input vector is bounded. If the discretization is fine-grained enough, this allows the computation of a good approximation of the output $F(X)$ of a neural network F, given the feasible input domain X. However, the experimental evaluation of Xiang et al. [61] remains limited to a toy example, i.e. a neural network with nine neurons in total.

Various authors consider MIP models for the verification problem, these are Tjeng and Tedrake [51], Lomuscio and Maganti [38], Fischetti and Jo [20] and Cheng et al. [14]. Indeed, it is rather straightforward to formulate the verification problem as an MIP, as we will see in Chapter 3. The performance of such MIP models is predominantly determined by the quality of the bounds which are computed for the ReLU neurons in the neural network. For that reason, the computation of such bounds is a major topic of this thesis and laid out in Chapter 4. The use of appropriate branching schemes is also important

for an MIP model of the verification problem, we will provide more details on this in Chapter 6. In fact, it is not necessary to solve the verification problem as an MIP if such approximation and branching methods are used. Bunel et al. [12] present such a branch-and-bound method without solving the verification problem as MIP directly. Moreover, they provide a good comparison of various methods for neural network verification. Besides their own approach, the empirical evaluation includes Reluplex [35], Planet [19], and an MIP model based on the suggestions of various authors [51, 38, 14]. While we also implement an MIP model to solve the verification problem, its functioning is more similar to the branch-and-bound method of Bunel et al. [12] than to the MIP model they use in their comparison. Besides, we consider various additional aspects, and therefore speed up the solving process significantly. For a computational comparison of other solvers with ours, we select Reluplex [35] and the branch-and-bound methods of Bunel et al. [12]. The other solvers regarded by Bunel et al. [12] are not competitive with these, as their experimental results show. Moreover, we regard the solvers ReluVal and Neurify as introduced by Wang et al. [57, 56]. In fact, ReluVal [57] was introduced first, and Neurify [56] is in principle an improved version. Although, these improvements are also merged into ReluVal, which is designed for different test instances than Neurify. The concept for both solvers is also a branch-and-bound scheme, that works with a frequent linear approximation of the regarded neural network. In contrast to the method of Bunel et al. [12], the approximation is not as good, but much faster to compute. Indeed, it comes without the need to solve a great number of linear programs but is based solely on matrix multiplication. Therefore, Wang et al. [56] report very short runtimes of their solvers. However, in their pure form, these solvers are limited to solving instances $\Pi = (X, Y, F)$ of the verification problem, where X is a box (cf. Remark 6). Still, the superior performance of ReluVal and Neurify on many instances is reason enough to include them in our computational experiments. Some more details on the solvers Reluplex [35], ReluVal/Neurify [56], and the branch-and-bound method of Bunel et al. [12] are explained in Section 6.3. We evaluate all of these computationally to compare them with our own solving model which combines an MIP formulation of the verification problem with specialized branching and approximation techniques. More details on the functioning and implementation can be found in Chapters 6 and 7. Based on previous experiments of Bunel et al. [12] and others, we can assume that our selection of solvers comprises the most competitive ones. However, some of the ideas that are considered for other solving routines are still relevant and therefore considered throughout the course of this thesis.

2.2 Output range analysis

The work of Dutta et al. [16] and Ruan et al. [44] is very closely related to the problem of neural network verification. Besides, Ruan et al. [44] explain how their approach can also be used for robustness certification. Both in Dutta et al. [16] and Ruan et al. [44], computing the range of output neurons (or the range of a function that statistically evaluates the network outputs) is the

main aspect. As in the case of neural network verification, it is crucial for this task to obtain good bounds of the neuron values in a neural network F given a feasible input domain X.

Dutta et al. [16] use an MIP model to formulate the problem and propose a local search heuristic. The heuristic uses a gradient ascent algorithm and is used to improve the bounds of the output range in consideration. After that, the MIP model is called to check whether this bound is tight or can be improved further. These steps are iterated, until no more improvement can be reached. We use the idea of this local search heuristic to implement a primal heuristic for neural network verification which we lay out in Chapter 5. Ruan et al. [44] use a nested optimization scheme, which is based on Lipschitz continuity of most neural networks, e.g. ReLU neural networks. An iterative algorithm is presented for the one-dimensional case, i.e. $X \subset \mathbb{R}$ and then extended for several dimensions.

Neither implementation is included in our computational experiments. The tool Sherlock of Dutta et al. [16] currently fails even on simple instances if a zero gradient appears during the local search procedure. Furthermore, if applied to neural network verification, it is unlikely to produce competitive results as it lacks promising bound computation procedures. However, it should be noted that Sherlock accepts polytopes in general and not only boxes as input domains. This is in contrast to the other solvers (except ours) which we regard for our computational experiments. On the other hand, the implementation of Ruan et al. [44] is very tailored towards their own test instances. Therefore, an execution of their method on different test instances would probably require to implement the algorithm from scratch.

2.3 Robustness certification

Tjeng et al. [52] present an MIP model for robustness certification of neural networks. For the computation of neuron bounds, they combine naive interval arithmetic and the linear approximation (4.4) of ReLU constraints. Among other bound computation approaches, we explain both concepts in Section 4.1. In fact, the model of Tjeng et al. [52] is very similar to MIP models for neural network verification (including ours), and therefore constitutes an algorithm which is sound and complete. As a matter of fact, prior work of Tjeng and Tedrake [51] was focused on the verification problem. However, several authors consider incomplete algorithms for certifying robustness of neural networks. Their general functioning is explained in the following paragraph.

Given a neural network F and a feasible input domain X, these algorithms compute an approximation $A \supseteq F(X)$. The set X contains all input vectors which are obtained by certain perturbations of a correctly classified input sample as explained in Section 1.3. As in the case of neural network verification, robustness is proved if $F(X) \subseteq Y$ holds for a suitable set Y. If the approximation A is good enough to prove the desired robustness property, i.e. $F(X) \subseteq A \subseteq Y$, the algorithm returns this as result. Though, it may be that the robustness property cannot be proved by means of approximation A, i.e. $A \not\subseteq Y$. In that case, no result is obtained as we do not know whether

$F(X) \subseteq Y$ holds or not. For that reason such an algorithm is incomplete. Various concepts of computing an approximation $A \supseteq F(X)$ can be found in the literature. We point out that all approximation techniques as presented in Section 4.1 can be used for this task. In the following, we give an overview of techniques which are used by various authors for the purpose of robustness certification.

Raghunathan et al. [43] propose an SDP relaxation of ReLU neural networks which can be used to approximate the output domain $F(X)$ of a neural network F on input domain X. We explain how this SDP relaxation can be obtained in Section 4.1. Wong and Kolter [59] use the linear approximation (4.4) of ReLU constraints to build a linear relaxation of the network constraints. Then they consider the corresponding dual linear program and use this to compute robustness guarantees. In subsequent work, Wong et al. [60] use the Fenchel conjugate function to obtain a dual relaxation which is applicable also for other types of neural network architectures. In two further lines of research, linear approximations of ReLU constraints are used to bound the network output in order to compute robustness guarantees. The work of Gehr et al. [23] and Singh et al. [47, 49] is based on numerical abstract domains which are used in static program analysis. Gehr et al. [23] build a system based on the library Apron (see Jeannet and Miné [32]) for numerical abstract domains. In this context, ReLU constraints are approximated by boxes, zonotopes, or polytopes. Singh et al. [47] use the same approximation as Wang et al. [56] for ReLU constraints, which we explain in Section 4.1 as (4.3). The same approximation is also used by Weng et al. [58], however the propagation of the bounds is performed by matrix multiplication rather than a static analyzer with abstract domains. Singh et al. [48] and Zhang et al. [63] both use the linear approximation of Ehlers [19] which we explain as (4.4) in Section 4.1, but use only one of the lower bound constraints (see Remark 19). Again, the propagation is either performed by a static analyzer for abstract domains (Singh et al. [48]) or via matrix multiplication (Zhang et al. [63]). Eventually, Singh et al. [49] refine the approaches of Singh et al. [47, 48] by computing better neuron bounds with the help of an MIP model. Another approach for robustness certification is presented in Weng et al. [58] and Zhang et al. [64]. Here, robustness is certified by computing bounds on the local Lipschitz constant of a neural network F at an input vector $x_0 \in X$. If the constant is bounded appropriately, the output domain of the neural network can be bounded likewise.

3 Neural Network Verification as MIP

It is straightforward to formulate the verification problem as a mixed integer program (MIP). This has been suggested by Cheng et al. [14] and by Dutta et al. [16] for output range analysis. Bunel et al. [12] and Tjeng et al. [52] present a slightly improved formulation, that we introduce in the following. Anderson et al. [3] give an ideal formulation for single ReLU neurons and show how it can be separated in linear time. This formulation is more complex and therefore discussed in Section 3.4.

In the formulation of Bunel et al. [12], each neuron is represented by one or two (continuous) variables. The value of a neuron before application of the ReLU function is given as a linear combination of the output values of the predecessor neurons in the network plus the bias. That means, this connection can be simply modelled by a linear equation in the MIP. We need two variables for neurons with ReLU activation function. Let variable x be the input value to the ReLU function and y be the output value. In this setting we will refer to x as the *ReLU input variable* and to y as *ReLU output variable*. We want to model $y = \max\{0, x\}$, which is represented using one additional binary variable d. Furthermore, we need that upper and lower bounds $l \leq x \leq u$ are known. Then we obtain the following constraints which are equivalent to $y = \max\{0, x\}$:

$$\begin{aligned} & y \geq x \\ & y \geq 0 \\ & y \leq x - (1-d)l \\ & y \leq d \cdot u \\ & d \in \{0, 1\} \\ & x \in [l, u],\ l < 0 < u \end{aligned} \tag{3.1}$$

Essentially, this is a classical big M formulation. Though, we use lower bound l and upper bound u separately, in order to improve the linear relaxation. Of course it is possible that we have $l \geq 0$ or $u \leq 0$ for the bounds. In these cases, we can omit the binary variable d and replace (3.1) as follows.

If $l \geq 0$, this implies $y = \max\{0, x\} = x$, i.e. (3.1) is replaced by $y = x$ for $x \in [l, u]$. If $u \leq 0$, we have $y = \max\{0, x\} = 0$ and thus we can set $y = 0$ for $x \in [l, u]$. These cases correspond to fixing the binary variable d to 1 or 0, respectively. Of course it is beneficial for the solving performance, if such fixings are applied as often as possible.

3.1 Formulation as feasibility problem

Let $\Pi = (X, Y, F)$ be a disjunction instance of the verification problem such that it holds $Y = \bigcup_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$ for certain open halfspaces Q_i, $i \in [k]$. Then it is straightforward to formulate an MIP which is feasible if and only if Π

is refutable. The instance Π of the verification problem is represented by the following constraints:

$$x \in X,\ y \in \mathbb{R}^m \backslash Y \text{ and } y = F(x) \tag{3.2}$$

This is an MIP, since $x \in X$ and $y \in \mathbb{R}^m \backslash\ Y$ can be represented by linear constraints. Indeed, X is a polytope and subsequently $x \in X$ can be used as a constraint for an MIP. Furthermore, it holds

$$\begin{aligned} & y \in \mathbb{R}^m \backslash Y \\ \Leftrightarrow \quad & y \in \mathbb{R}^m \backslash\ (\bigcup_{i=1}^{k} Q_i) \\ \Leftrightarrow \quad & y \in \bigcap_{i=1}^{k} (\mathbb{R}^m \backslash\ Q_i), \end{aligned}$$

which means that y is restricted to be an element of a polyhedron. This follows from the last line, which states that y must be contained in an intersection of closed halfspaces. $y = F(x)$ can also be expressed by linear constraints combined with integrality constraints for some auxiliary binary variables as discussed above. Now, if MIP (3.2) is feasible, there exists $x \in X$ such that $F(x) = y \notin Y$. This implies $F(X) \not\subseteq Y$ and hence Π is refutable. Otherwise, if MIP (3.2) is not feasible, that means that for all $x \in X$ it holds $F(x) = y \in Y$ and thus Π is verifiable.

Also for conjunction instances where $Y = \bigcap_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$ for open halfspaces Q_i, $i \in [k]$, we could express Π as an MIP which is checked for feasibility. Though, this requires the use of further integer variables, since the set $\mathbb{R}^m \backslash Y$ is not a polyhedron in this case. Indeed, the formulation as feasibility problem requires that the halfspaces Q_i which constitute the set Y must be open. Otherwise, constraints of the type $y \in \mathbb{R}^m \backslash Q_i$ cannot be included in a linear program.

We regard two other methods which allow to solve conjunction instances. One possibility is to split instance Π into k instances such that we have instances $\Pi_1 = (X, Q_1, F), \ldots, \Pi_k = (X, Q_k, F)$ and process each of these instances independently (cf. Remark 7). Then Π is verifiable if and only if Π_i is verifiable for each $i \in [k]$. This follows directly from the fact that

$$F(X) \subseteq Y = \bigcap_{i=1}^{k} Q_i \qquad \Leftrightarrow \qquad \forall i \in [k] : F(X) \subseteq Q_i.$$

Clearly, this splitting works indepently from the way that the instances $\Pi_1, \ldots, \Pi_k$ are solved. Bunel et al. [12] and Katz et al. [35] employ this splitting idea in their solving methods but do not use the here presented formulation as MIP. In spite of that, Bunel et al. [12] propose another method to deal with such instances. This is based on formulating the verification problem as an optimization problem which we lay out in the subsequent section.

3.2 Formulation as optimization problem

Bunel et al. [12] show how the verification problem can be converted into an optimization problem. In this setting, an instance $\Pi = (X, Y, F)$ is verifiable if the optimum value of the corresponding optimization problem is greater than zero and refutable if it is lower than zero. Indeed, this method supports both open and closed halfspaces Q_i which constitute the set Y as in Definition 3, although only one of both for each instance.

Assume that $Y = \bigcup_{i=1}^{k} Q_i \subseteq \mathbb{R}^m$ where Q_i, $i \in [k]$ are open halfspaces. This implies the existence of $q_i \in \mathbb{R}^m$ and $b_i \in \mathbb{R}$ for $i \in [k]$ such that we have halfspaces $Q_i = \{x \in \mathbb{R}^m \mid q_i^T x > b_i\}$. Then we see that

$$\begin{aligned} & y \in \bigcup_{i=1}^{k} Q_i \\ \Leftrightarrow \quad & \exists j \in [k] : \ y \in Q_j = \{x \in \mathbb{R}^m \mid q_j^T x > b_j\} \\ \Leftrightarrow \quad & \exists j \in [k] : \ q_j^T y - b_j > 0 \\ \Leftrightarrow \quad & \max_{i \in [k]} \left(q_i^T y - b_i\right) > 0. \end{aligned}$$

The same holds for closed halfspaces Q_i, $i \in [k]$, if all inequalities "$>$" are replaced by their counterparts "$\geq$". Analogously, with open halfspaces Q_i as before and $Y = \bigcap_{i=1}^{k} Q_i$ it holds

$$\begin{aligned} & y \in \bigcap_{i=1}^{k} Q_i \\ \Leftrightarrow \quad & \forall i \in [k] : \ y \in Q_i = \{x \in \mathbb{R}^m \mid q_i^T x > b_i\} \\ \Leftrightarrow \quad & \forall i \in [k] : \ q_i^T y - b_i > 0 \\ \Leftrightarrow \quad & \min_{i \in [k]} \left(q_i^T y - b_i\right) > 0. \end{aligned}$$

For the case $Y = \bigcup_{i=1}^{k} Q_i$ we consider the following MIP:

$$\begin{aligned} \text{minimize} \quad & t \\ \text{s.t.} \quad & x \in X \\ & y = F(x) \\ & z_i = q_i^T y - b_i && \forall i \in [k] \\ & t = \max\{z_1, \ldots, z_k\} \\ & y \in \mathbb{R}^m \\ & t, z_i \in \mathbb{R} && \forall i \in [k] \end{aligned} \tag{3.3}$$

Indeed, (3.3) is an MIP. Bunel et al. [12] and Tjeng et al. [52] show that the constraint $t = \max\{z_1, \ldots, z_k\}$ can be replaced by linear constraints using k

additional binary variables. The equivalent constraints are

$$\begin{aligned} & t \geq z_i && \forall i \in [k] \\ & t \leq z_i + (U - l_i)(1 - d_i) && \forall i \in [k] \\ & \sum_{i=1}^{k} d_i = 1 && \\ & d_i \in \{0, 1\} && \forall i \in [k] \end{aligned} \tag{3.4}$$

where U is an upper bound on all z_i, $i \in [k]$ and l_i is a lower bound for x_i, $i \in [k]$.

Lemma 8. *Constraints (3.4) are equivalent to* $t = \max\{z_1, \ldots, z_k\}$.

Proof. Let $j \in [k]$ such that $z_j = t := \max\{z_1, \ldots, z_k\}$. We set $d_j := 1$ and $d_i := 0$ for $i \in [k] \backslash \{j\}$. Then we have $\sum_{i=1}^{k} d_i = 1$ and for all $i \in [k]$ it holds $t \geq z_i$. Furthermore it holds $t \leq z_j$ and $t \leq U + z_i - l_i$ for $i \in [k] \backslash \{j\}$, since $t \leq U$ and $z_i \geq l_i$ for all $i \in [k]$.

Now assume that constraints (3.4) hold. Since we have $\sum_{i=1}^{k} d_i = 1$ and $d_i \in \{0, 1\}$ for $i \in [k]$, there is exactly one index $j \in [k]$ such that $d_j = 1$. We claim that $t = z_j = \max\{z_1, \ldots, z_k\}$. Since $t \geq z_i$ for all $i \in [k]$ we have $t \geq \max\{z_1, \ldots, z_k\}$. On the other hand, $t \leq z_j$, i.e. $t = z_j$, which implies $z_j = t \geq \max\{z_1, \ldots, z_k\}$ and hence the claim is shown. □

Theorem 9. *Instance* $\Pi = (X, Y, F)$, *where* $Y = \bigcup_{i=1}^{k} Q_i$ *for some open halfspaces* $Q_i = \{x \in \mathbb{R}^m \mid q_i^T x > b_i\}$, $q_i \in \mathbb{R}^m$ *and* $b_i \in \mathbb{R}$ *for* $i \in [k]$, *is verifiable if and only if the optimum value of (3.3) is greater than zero.*

Proof. Assume that Π is verifiable, i.e. $F(X) \subseteq Y = \bigcup_{i=1}^{k} Q_i$. Hence, for any $x \in X$ there exists $j \in [k]$ such that $y := f(x) \in Q_j$, i.e. $q_j^T y - b_j > 0$. It follows $t \geq z_j := q_j^T y - b_j > 0$ which implies the desired result since $x \in X$ was arbitrary. Remind that we regard optimum solutions of an MIP so it suffices to consider finitely many $x \in X$.

For the opposite direction, assume that the optimum value $\hat{t}$ of (3.3) fulfills $\hat{t} > 0$. Let $x \in X$ be arbitrary and $y = F(x)$. With $z_i = q_i^T y - b_i$ for $i \in [k]$ it holds $\max\{z_1, \ldots, z_k\} \geq \hat{t} > 0$ since $\hat{t}$ is optimal. In other words, there is $j \in [k]$ such that $q_j^T y - b_j = z_j > 0$ and thus $y \in Q_j \subseteq Y$. Since $x \in X$ was arbitrary, Π is verifiable. □

Remark 10. Theorem 9 can also be stated for the case $Y = \bigcap_{i=1}^{k} Q_i$ by replacing "max" with "min" in (3.3). The proof works analogously since $\min\{z_1, \ldots, z_k\} = -\max\{-z_1, \ldots, -z_k\}$. Furthermore, also closed halfspaces $Q_i = \{x \in \mathbb{R}^m \mid q_i^T x \geq b_i\}$, $i \in [k]$ can be handled. In this case, the optimum value of (3.3) must be greater than or equal to zero.

In practice, the optimum value $\hat{t}$ of (3.3) will usually be siginificantly greater than zero if an instance is indeed verifiable. Otherwise, we do not have a reliable proof that the instance is verifiable due to numerical inaccuracy. Indeed, Bunel et al. [12] mention that $\hat{t}$ can be regarded as a margin by which the neural network F is "safe" with respect to the properties that are

verified. Clearly, it is not necessary to actually compute $\hat{t}$ in order to solve the verification problem as Bunel et al. [12] point out. If the dual bound of (3.3) is greater than zero, the instance is verifiable, since the proof of Theorem 9 only relies on the fact that $\hat{t} > 0$. Therefore, the development of the dual bound throughout the solving process gives a good indication of the progress which is made. This is a clear advantage compared to the formulation as feasibility problem. For this reason and due to the simple handling of the case $Y = \bigcap_{i=1}^{k} Q_i$ we mainly use this formulation in our implementation. Nevertheless, we make some comparisons to the formulations presented in Sections 3.1 and 3.3.

On the other hand, if the primal bound of (3.3) is lower than zero, we know that the corresponding instance of the verification problem is refutable as $\hat{t} < 0$ is already implied. However, this case has less relevance since primal solutions are usually only found by specialized heuristics which we describe in Chapter 5.

3.3 Formulation as quadratic programming problem

In this section we present a formulation of the verification problem as quadratic program. As opposed to the MIP formulations which we presented in the sections before, this formulation does not require any integer or binary variables. The nonlinear behavior of the ReLU activations is modelled by the quadratic objective function. Let $\Pi = (X, Y, F)$ be an instance of the verification problem such that Π is a disjunction instance. We assume $Y = \bigcup_{i=1}^{k} Q_i$ for open halfspaces Q_i, $i \in [k]$ and $k \in \mathbb{N}$. Let $((A_l, b_l))_{l=1}^{L}$ be the weights and biases corresponding to F. Moreover, L is the number of layers in the neural network and $N_0, \dots, N_L$ are the numbers of neurons per layer as described in Definition 1. This implies $X \subseteq \mathbb{R}^{N_0}$ and $Y \subseteq \mathbb{R}^{N_L}$ and we can state the formulation as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{l=1}^{L-1} x_l^T (x_l - A_l x_{l-1} - b_l) & & \\
& x_l \geq A_l x_{l-1} + b_l & & \forall l \in [L-1] \\
& x_l \geq 0 & & \forall l \in [L-1] \\
& x_L = A_L x_{L-1} + b_L & & \\
& x_0 \in X & & \\
& x_L \in \mathbb{R}^{N_L} \setminus Y & & \\
& x_l \in \mathbb{R}^{N_l} & & \forall l \in [L-1]
\end{aligned}
\tag{3.5}
$$

Theorem 11. *Instance* Π *is refutable if and only if the quadratic program (3.5) is feasible and the optimum value is zero. Otherwise* Π *is verifiable.*

Proof. We first assume that Π is refutable so that we can find $x \in X$ with $F(x) \notin Y$. We set $x_0 := x$, $x_L := F(x) \in \mathbb{R}^{N_L} \setminus Y$ and for $l \in [L-1]$ we let $x_l := \mathrm{ReLU}(A_l x_{l-1} + b_l)$ which implies $x_l \geq 0$ and $x_l \geq A_l x_{l-1} + b_l$. Furthermore it is $x_L = A_L x_{L-1} + b_L$ and for each $l \in [l-1]$ we have for each $i \in [N_l]$ that either $[x_l]_i = 0$ or $[x_l]_i = [A_l x_{l-1} + b_l]_i$. Since $x_l \in \mathbb{R}^{N_l}$, this leads

to the conclusion that $x_l^T(x_l - A_l x_{l-1} - b_l) = 0$ for all $l \in [L-1]$. Hence, the quadratic program (3.5) is feasible and its optimum value is zero.

On the other hand, if (3.5) is feasible and the optimum value is zero, we know that there is $x_0 \in X$, such that $F(x) = x_L \notin Y$ which means that Π is refutable. Indeed, it holds $F(x) = x_L$ since for all $l \in [L-1]$ we have $x_l \geq 0$ and $x_l \geq A_l x_{l-1} + b_l$, i.e. $x_l^T(x_l - A_l x_{l-1} - b_l) \geq 0$ for all $l \in [L-1]$. Hence we know $x_l^T(x_l - A_l x_{l-1} - b_l) = 0$ for all $l \in [L-1]$ as the objective value of (3.5) is zero, and it follows that $[x_l]_i[(x_l - A_l x_{l-1} - b_l)]_i = 0$ for all $i \in [N_l]$ and $l \in [L-1]$. Subsequently it holds $[x_l]_i = \mathrm{ReLU}(A_l x_{l-1} - b_l)$ and thus we can conclude that $F(x) = x_L$. □

Remark 12. For $l \in [L-1]$ the constraints state $x_l \geq 0$ and $x_l \geq A_l x_{l-1} + b_l$ so that $\sum_{l=1}^{L-1} x_l^T(x_l - A_l x_{l-1} - b_l) \geq 0$ for any feasible solution of (3.5). Hence, if Π is verifiable, the quadratic program (3.5) is either infeasible, or feasible with an optimum value greater than zero.

The formulation of the verification problem as quadratic program has the drawback that it must be decided whether the objective value is zero or not, while it is always non-negative. Numerical inaccuracy may render this decision very problematic. It should be noted, that for this formulation the optimum value does not give a "safety" margin with respect to the properties that shall be verified, as opposed to the optimum value of (3.3).

To evaluate formulation 3.5 computationally, we try a plain implementation in SCIP [25]. While the variables x_l for $l \in [L-1]$ in (3.5) do not necessarily require bounds, we compute bounds using the naive approximation method as presented in Section 4.1. This could possibly improve the solving process, as the variable domains are somewhat restricted. However, within a time limit of one hour, SCIP is not able to solve any of the disjunction instances in our SAT and UNSAT evaluation sets which we describe in Section 8.4. Further experiments on some MNIST based instances as described in Section 8.3 show, that only very simple instances can be solved with this formulation. Detailed computational results can be found in Section B.3 of the appendix.

3.4 Ideal MIP formulations for ReLU constraints

This section highlights some recent results of Anderson et al. [3] which include ideal formulations for ReLU constraints. The work of Anderson et al. [3] also contains results on other activation functions such as leaky or clipped ReLU and the maximum of affine functions. Yet, we focus our presentation on the results regarding the ReLU function as in the rest of the thesis.

The notion of an ideal formulation for integer programs can e.g. be found in Bertsimas and Weismantel [6], Chapter 1. Here we cite the definition of Vielma [53], who is also an author of Anderson et al. [3]. For that, we use the following generic MIP formulation (without objective function) based on two

rational matrices A, B and a rational vector b of appropriate sizes:

$$\begin{aligned} &Ax + By \leq b \\ &x \in \mathbb{R}^n \\ &y \in \mathbb{Z}^m \end{aligned} \tag{MIP}$$

Definition 13 (Ideal Formulation, Vielma [53])**.** *Let A, B and b in (MIP) be rational and of appropriate sizes. For simplicity we assume that the LP relaxation of (MIP) has at least one extreme point or basic feasible solution. Then the MIP formulation (MIP) is* ideal *if and only if all the extreme points of its LP relaxation satisfy the integrality constraints $y \in \mathbb{Z}^m$.*

Clearly, it is desirable for an MIP formulation to be ideal, if the size of the formulation, i.e. the number of constraints, does not grow too big. An ideal formulation allows to solve the MIP by computing an extreme point solution for the corresponding LP relaxation (cf. Bertsimas and Weismantel [6], Chapter 1). The latter can be performed efficiently using the simplex algorithm. Nevertheless, it should be noted that the results of Anderson et al. [3] do refer only to single ReLU neurons and at most the layer before. Hence we do not have an ideal formulation of the whole network which implies that solving the verification problem cannot be reduced to solving an LP using the formulations laid out in this section.

In the context of this thesis, Proposition 3 is the most interesting result in Anderson et al. [3], which states an ideal MIP formulation that can replace (3.1). To state this formulation, we notice that the value of the ReLU input variable x in (3.1) is given as an affine combination of the neuron output values in the previous layer (which could also be the input layer). Hence there are $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ such that $x = w^T z + b$ where $z \in \mathbb{R}^n$ represents the neuron values of the previous layer. Clearly, $n \in \mathbb{N}$ is the number of neurons in that layer. Anderson et al. [3] assume that there are known bounds for z, although they only consider box constraints, i.e. $l_i \leq z_i \leq u_i$ for $i \in [n]$. For $i \in [n]$ they define

$$\tilde{l}_i = \begin{cases} l_i & \text{if } w_i \geq 0 \\ u_i & \text{if } w_i < 0 \end{cases} \qquad \text{and} \qquad \tilde{u}_i = \begin{cases} u_i & \text{if } w_i \geq 0 \\ l_i & \text{if } w_i < 0 \end{cases}$$

and $\operatorname{supp}(w) := \{i \in [n] \mid w_i \neq 0\}$. Using these definitions we can give an alternative to formulation (3.1).

Theorem 14 (Anderson et al. [3], Proposition 3)**.** *Assume that*

$$\min_{\forall i \in [n]:\ l_i \leq z_i \leq u_i} w^T z + b < 0 < \max_{\forall i \in [n]:\ l_i \leq z_i \leq u_i} w^T z + b.$$

The following is a valid and ideal formulation of $y = \operatorname{ReLU}(w^T z + b)$:

$$y \geq w^T z + b \tag{3.6a}$$

$$y \leq \sum_{i \in I} w_i (z_i - \tilde{l}_i (1 - d)) + \Big(b + \sum_{i \in [n] \setminus I} w_i \tilde{u}_i \Big) d \quad \forall I \subseteq \operatorname{supp}(w) \tag{3.6b}$$

$$l_i \leq z_i \leq u_i \quad \forall i \in [n] \tag{3.6c}$$

$$y \geq 0, \quad y \in \mathbb{R} \tag{3.6d}$$

$$d \in \{0, 1\} \tag{3.6e}$$

For the proof we refer to Anderson et al. [3]. If the assumption of the theorem is not met, the ReLU function does not need to be modelled but can be replaced by the zero or identity function, as pointed out by Anderson et al. [3]. This corresponds to our comment on the same situation regarding (3.1).

It is apparent that formulation (3.6) has a number of constraints which is exponential in $|\operatorname{supp}(w)| \leq n$, i.e. in the size of the previous layer in the network. Anderson et al. [3] present a separation routine which allows to separate constraints (3.6b) in time linear in n.

Theorem 15 (Anderson et al. [3], Proposition 4)**.** *Assume that we are given a vector $(\hat{z}, \hat{y}, \hat{d}) \in \mathbb{R}^n \times \mathbb{R}_{\geq 0} \times [0,1]$ where $l_i \leq \hat{z}_i \leq u_i$ for all $i \in [n]$, then we define the set*

$$\hat{I} := \Big\{ i \in \operatorname{supp}(w) \mid w_i \hat{x}_i < w_i \Big(\tilde{l}_i (1 - \hat{d}) + \tilde{u}_i \hat{d} \Big) \Big\}.$$

If

$$\hat{y} > b\hat{z} + \sum_{i \in \hat{I}} w_i \Big(\hat{x}_i - \tilde{l}_i (1 - \hat{d}) \Big) + \sum_{i \in [n] \setminus \hat{I}} w_i \tilde{u}_i \hat{d},$$

then the constraint corresponding to $\hat{I}$ in (3.6b) is the most violated in the family. Otherwise, none of the inequalities in (3.6b) is violated at $(\hat{z}, \hat{y}, \hat{d})$.

Again, for the proof we refer to Anderson et al. [3]. For the application of the ideal formulation in our solving model, we follow the suggestion of Anderson et al. [3] to use formulation (3.1) as a basis and strengthen the formulation by cuts which are generated using Theorems 14 and 15. Indeed, this task is the purpose of a separator in SCIP, as constraints (3.6) are not necessary for the correctness of the model, but may help to strengthen the LP relaxation of our formulation based on (3.1).

The proofs of Theorem 14 and Theorem 15 in Anderson et al. [3] are obtained as special cases within a more general setting which treats the ideal formulation of the maximum of two affine functions. Furthermore, Anderson et al. [3] also present convex relaxations of ReLU constraints and tight formulations for the maximum of more than two affine functions. Beyond that, Anderson et al. [3] discuss the fact that formulation (3.6) is ideal for a single neuron, but not for a whole (deep) neural network that encompasses several layers of many ReLU neurons. In this context, Anderson et al. [3] present extended versions of Theorem 14 and Theorem 15, for which they assume that each z_i, $i \in [n]$ is the output of a ReLU neuron in the previous layer. By incorporating the binary variables (corresponding to d in (3.6) or (3.1)) from that layer into the formulation of (3.6b), it is possible to strengthen this family of constraints. Of course, this is only one step towards a better LP relaxation for the whole network, since it is still limited to the consideration of single neurons and their immediate predecessor neurons. In constrast to the separation routine in Theorem 15, this extended version is not evaluated computationally in Anderson et al. [3]. Therefore and also due to the higher complexity of the extended version, we refrain from implementing it in our model.

4 Approximations of ReLU Neural Networks

Solving the problem of neural network verification requires to model constraints of the form $y = \max\{0, x\}$ for all ReLU input variables x and corresponding ReLU output variables y of each layer. In order to obtain a good model which allows to solve instances of relevant size, it is crucial to obtain tight bounds $l,\ u$ on the value of x before the application of the ReLU function. Hence, we present different possibilities to compute these bounds and discuss advantages and drawbacks. A main focus is put on the linear approximation of these constraints for a whole layer at once. The idea of regarding a whole layer of ReLU neurons at once can also be found in Anderson et al. [3]. Though, the considerations of Anderson et al. [3] are limited to the extended versions of Theorems 14 and 15 as discussed in Section 3.4. Also Bunel et al. [12] mention the idea to approximate several ReLU neurons at once rather than each neuron on its own, but do not investigate this topic further. Moreover, we present an SDP relaxation for ReLU constraints as proposed by Raghunathan et al. [43], which acts on a whole layer of ReLU neurons. However, we mainly focus on linear approximations of whole layers of ReLU constraints, and present new contributions to this topic. Eventually, we also provide details on the geometrical aspects of the ReLU image which arises from the application of the ReLU function to a polytope. One of our results has been shown independently by Xiang et al. [62], who also consider ReLU images of polytopes.

The structure of this chapter, which constitutes a major contribution of this thesis, is as follows: In the first part, we describe approximations for ReLU neural networks as they can be found in the literature. Then, in Section 4.2, we define the notion of a *ReLU approximation* which allows for a detailed and mathematically well based comparison of different approximation techniques. Especially, we show that the approximation proposed by Ehlers [19] is best possible in a certain sense. Section 4.3 deals with the efficient implementation of an LP based approximation method. In Section 4.4 we lay the theoretical foundation which allows to investigate how the image under the ReLU function can be computed for whole layers. Then we show in Section 4.5 that the approximation of Ehlers [19] can be improved by making a joint approximation of the ReLU function for several neurons and not just one. In Section 4.7 we present a new practical method for an improved approximation which is based on the aforementioned theoretical results. Before that, we use Section 4.6 to complete our investigations with some negative results on the possibilities for approximating ReLU constraints.

Given an instance $\Pi = (X, Y, F)$ of the verification problem with $X \subset \mathbb{R}^n$, we will assume the availabilty of input bounds $l_i,\ u_i$ with $i \in [n]$ for the components of X throughout this chapter. See Remark 6 for an explanation of the existence of these bounds. Not all methods presented here require

to actually compute bounds l_i, u_i explicitly, since the definition of X as an intersection of halfspaces may be sufficient.

All approximation methods that we present are executed layer by layer. Based on the input bounds, we compute bounds for the neurons in the following layer. This process is iterated until the last layer is reached, i.e. the output layer. Depending on the instance and the bound computation approach, it may be possible to prove that an instance $\Pi = (X, Y, F)$ of the verification problem is verifiable using only these bounds for the output layer. Assume that we have a set A which approximates the neural network output $F(X)$, i.e. $F(X) \subseteq A$. In case that $A \subseteq Y$, we have thus shown that $F(X) \subseteq Y$, which means that Π is verifiable. Theoretically, it could be shown that Π is refutable using only approxmiation A of $F(X)$. If $A \cap Y = \emptyset$, we know that $F(X) \cap Y = \emptyset$ and hence there is $x \in X$ such that $f(x) \notin Y$, i.e. Π is refutable. Though, if Y is such that it defines a reasonable property to verify, this case will most likely not occur.

4.1 Basic approximation methods for bound computations in neural networks

The most simple approach is to use naive interval arithmetic, which is regarded by Ehlers [19] and Wang et al. [57]. Assume we want to compute bounds for a neuron x (before application of any activation function). Let

$$x = \sum_{i=1}^{n} \alpha_i z_i \ + b,$$

where $z_i, i \in [n]$ are the output values of the neurons which are predecessors of x in the neural network, $\alpha_i \in \mathbb{R}$ for $i \in [n]$ are the weights and $b \in \mathbb{R}$ is the bias. Given bounds $[l_i, u_i]$ for all z_i, $i \in [n]$, we compute bounds $[l, u]$ for x as follows:

$$l = \sum_{\substack{i\in[n]\\ \alpha_i>0}} \alpha_i l_i + \sum_{\substack{i\in[n]\\ \alpha_i<0}} \alpha_i u_i + b \qquad \text{and} \qquad u = \sum_{\substack{i\in[n]\\ \alpha_i>0}} \alpha_i u_i + \sum_{\substack{i\in[n]\\ \alpha_i<0}} \alpha_i l_i + b$$

Assume that we have a set of feasible values $\{z_1, \ldots, z_n\}$ with $z_i \in [l_i, u_i]$ for the predecessor neurons of x. Then we have

$$x = \sum_{\substack{i\in[n]\\ \alpha_i>0}} \alpha_i z_i + \sum_{\substack{i\in[n]\\ \alpha_i<0}} \alpha_i z_i + b \geq \sum_{\substack{i\in[n]\\ \alpha_i>0}} \alpha_i l_i + \sum_{\substack{i\in[n]\\ \alpha_i<0}} \alpha_i u_i + b = l.$$

Analogously, we can show that u is a feasible upper bound for x.

The ReLU activation function, which enforces $y = \max\{0, x\}$, can be simply modelled by setting $[\max\{0, l\}, \max\{0, u\}]$ as bounds for y. If the corresponding ReLU neuron cannot already be fixed in its phase, we obtain bounds $[0, u]$ for the ReLU output variable y. Figure 4.1 provides a visual representation of this approximation, to which we will refer as *naive approximation*. If there is no activation function at the respective neuron, we can just keep the bounds $[l, u]$ as computed above. After all bounds for one layer

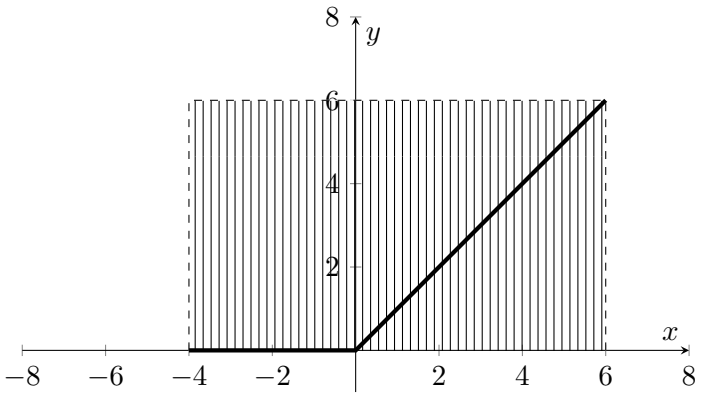

Figure 4.1: Naive approximation of ReLU function in one dimension. Here we have lower bound -4 and upper bound 6 for the ReLU input variable x. The feasible domain of the ReLU output variable y is given by the solid black line for the actual ReLU function and by the shaded area for the naive approximation.

have been computed, the bounds of the next layer are computed based on the previous bounds.

This simple approach mainly suffers from the fact, that it assumes the independency of all predecessor neurons when computing a new bound. Usually not all neurons of a layer can be at their upper bound or lower bound, respectively, at the same time. Therefore, the bounds computed with this method are so bad, that they only serve to solve tiny instances.

Wang et al. [57] use symbolic interval arithmetic to keep track on some of the neuron dependencies in order to compute better bounds. The idea is to keep a symbolic equation, based on the input values of the network, for each neuron. Although such symbolic equations can be constructed for all neurons of a ReLU neural network, each ReLU activation introduces a non-linearity in the equation. Therefore, these equations can only be handled efficiently, as long as no ReLU activation has appeared so far. Of course, any neural network of interest to us contains several ReLU activations. To explain the concept though, we assume a neural network without activation functions, i.e. in terms of Definition 1, we let σ be the identity function on $\mathbb{R}$. We have layers $x_0, \dots, x_L$, where $x_l \in \mathbb{R}^{N_l}$ and $N_l \in \mathbb{N}$ is the number of neurons in layer l, $l = 0, \dots, L$, and $L \in \mathbb{N}$ is the number of layers. That means, x_0 is the input layer and x_L is the output layer. Then for $l \in [L]$ it holds $x_l = A_l x_{l-1} + b_l$ for weight matrices $A_l \in \mathbb{R}^{N_l \times N_{l-1}}$ and bias vectors $b_l \in \mathbb{R}^{N_l}$. Now we can express each layer in terms of the input layer x_0, i.e. it holds for $l \in [L]$:

$$x_l = A_l(A_{l-1}(\dots(A_2(A_1 x_0 + b_1) + b_2)\dots) + b_{l-1}) + b_l \tag{4.1}$$

Because no activation function is employed, this is a plain linear equation of the input vector x_0. This is the symbolic equation we would have for layer x_l without any activation function. Indeed, we could integrate the ReLU function in the equation. But then the equation would be no longer linear, and, especially in deeper layers, very difficult to evaluate. For that reason, Wang et al. [57] propose to convert the linear equation into numerical lower

and upper bounds when a ReLU activation occurs, which cannot be fixed in its phase. First we see how to obtain explicit neuron bounds from the symbolic equation (4.1). Assume we want to bound the j-th neuron in layer $l \in [L]$, i.e. $j \in [N_l]$. Essentially, (4.1) can be written as $x_l = \tilde{A}_l x_0 + \tilde{b}_l$ by summarizing all weights and biases in $\tilde{A}_l \in \mathbb{R}^{N_l \times N_0}$ and $\tilde{b}_l \in \mathbb{R}^{N_l}$. We consider the j-th row of this equation which reads $[x_l]_j = [\tilde{A}_l]_j x_0 + [\tilde{b}_l]_j$. Let $(a_1, \ldots, a_{N_0}) := [\tilde{A}_l]_j$ and let $[l_0, u_0], \ldots, [l_{N_0}, u_{N_0}]$ be the bounds of the input neurons $[x_0]_1, \ldots, [x_0]_{N_0}$. Then we can bound $[x_l]_j$ like this:

$$\sum_{\substack{k \in [N_0] \\ a_k > 0}} a_k l_k + \sum_{\substack{k \in [N_0] \\ a_k < 0}} a_k u_k + [\tilde{b}_l]_j \leq [x_l]_j \leq \sum_{\substack{k \in [N_0] \\ a_k > 0}} a_k u_k + \sum_{\substack{k \in [N_0] \\ a_k < 0}} a_k l_k + [\tilde{b}_l]_j \quad (4.2)$$

The bounds follow directly from the symbolic equation and can be verified analogously as we showed for the naive interval arithmetic. Now we consider the integration of the ReLU activations. Wang et al. [57] propose to convert the symbolic equation into concrete numeric bounds if linearity is broken due to a ReLU activation. For that, let eq be the symbolic expression for a ReLU input variable. We compute the numeric bounds that are implied by eq as in (4.2), based on the feasible input domain for the instance of the verification problem. We denote the numeric lower and upper bounds (before application of the ReLU function) as l_{in} and u_{in}, respectively. Accordingly, l_{out} and u_{out} denote the bounds of the variable after application of the ReLU function. As long as $l_{in} \geq 0$, the symbolic equation remains valid after the ReLU application, i.e. it can be kept as if there was no ReLU activation. If $l_{in} < 0 < u_{in}$, which is the usual case, we cannot keep the linear symbolic equation. Wang et al. [57] set $l_{out} = 0$ in this case, which of course implies the loss of dependency information for following bound computations. Furthermore, also the upper bound u_{out} must be set to its current numeric value u_{in}, since the linear equation eq, relating the neuron value to the network inputs, is broken due to the ReLU application. This corresponds to the approximation which we described for the case of naive interval arithmetic as depicted in Figure 4.1. If $u_{in} \leq 0$, it can be set $l_{out} = u_{out} = 0$ which fixes the neuron value to 0. Note that in this case we do not loose dependency information, as there is no possibility that this ReLU output variable will have any value different from 0. Table 4.1 shows a comparison of naive and symbolic interval arithmetic for the neural network depicted in Figure 4.2. It indicates, that this symbolic approach can only provide better bounds if at least some of the ReLU activations can be fixed positively, i.e. $l_{in} \geq 0$. Otherwise, the symbolic interval arithmetic uses the same bounds as the naive method and computes new bounds in the same way. Only in the case of positively fixed ReLU neurons the symbolic equation is maintained and can lead to better bounds. Negatively fixed ReLU neurons behave the same way in the symbolic and the naive approach.

To overcome this drawback, Wang et al. [56] improve the method by introducing a different approximation for the case $l_{in} < 0 < u_{in}$. The main idea is to maintain the symbolic dependencies also in this case. Though, we cannot keep the linear equation for the value of such a neuron, but instead we introduce symbolic equations which provide a lower and upper bound for the neuron value. These symbolic bounds can then be propagated through the network

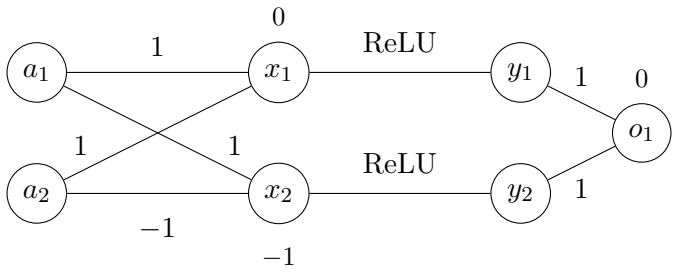

Figure 4.2: Neural network with input neurons $a_1 \in [-3, 2]$ and $a_2 \in [-5, 4]$. For our example we try to find an upper bound for the output $o_1 = y_1 + y_2$ of the network. We will use this network as a running example throughout the chapter to illustrate the various bound computation techniques.

Method	x_1	x_2	y_1	y_2	o_1
Naive IA	$[-8, 6]$	$[-8, 6]$	$[0, 6]$	$[0, 6]$	$[0, 12]$
Symbolic IA [57]	$a_1 + a_2$	$a_1 - a_2 - 1$	$[0, 6]$	$[0, 6]$	$[0, 12]$

Table 4.1: Bounds for the network in Figure 4.2 using interval arithmetic (IA). In this network, the symbolic IA of Wang et al. [57] shows no improvement over the naive IA. This method only produces better bounds, if at least some of the ReLU activations can be fixed positively.

and have the advantage that the dependency information partially remains.

First, we present the approximation that Wang et al. [56] propose for the ReLU constraints instead of the naive approximation which is employed in the naive and symbolic interval arithmetic. Assume we want to approximate the constraint $y = \max\{0, x\}$ for $x \in [l, u]$ where $l < 0 < u$. Then, according to Wang et al. [56], it holds

$$y \geq \frac{ux}{u-l} \qquad \text{and} \qquad y \leq \frac{u(x-l)}{u-l} \tag{4.3}$$

which we prove in the following theorem. A visual representation of these constraints is given in Figure 4.3.

Theorem 16. *Given $l < 0 < u$, $x \in [l, u]$ and $y = \max\{0, x\}$, inequalities (4.3) hold.*

Proof. $y = \max\{0, x\}$ implies that $y \geq 0$ and $y \geq x$. Therefore it holds

$$\begin{aligned} & y \geq \frac{ux}{u-l} \\ \Leftrightarrow \quad & uy - yl \geq ux \\ \Leftrightarrow \quad & u(y-x) \geq yl \end{aligned}$$

which proves correctness of the lower bound since $u(y-x) \geq 0 \geq yl$. For the upper bound we consider two cases. If $x < 0$ we have $y = 0 \leq \frac{u(x-l)}{u-l}$, since

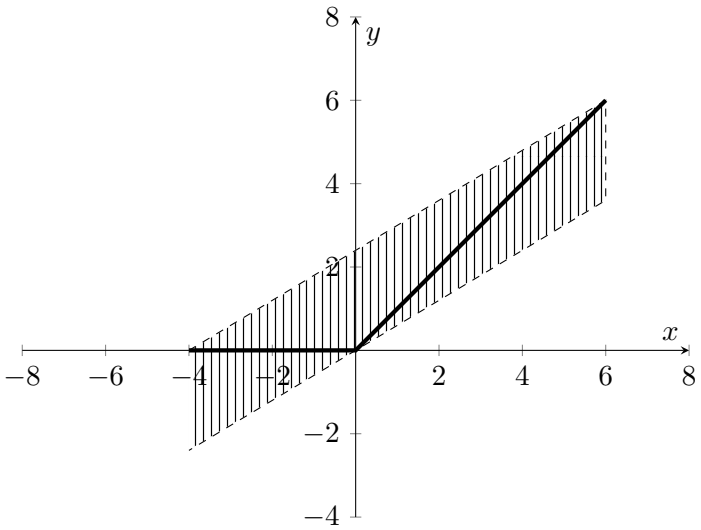

Figure 4.3: Approximation of ReLU function in one dimension as proposed by Wang et al. [56]. Here we have lower bound -4 and upper bound 6 for the ReLU input variable x. The feasible domain of the ReLU output variable y is given by the solid black line for the actual ReLU function and by the shaded area for the approximation.

$u > 0,\ u - l > 0$ and $x - l \geq 0$. If $x \geq 0$, we have $y = x$ and see:

$$\begin{aligned} & y = x \leq \frac{u(x-l)}{u-l} \\ \Leftrightarrow \quad & (u-l)x \leq u(x-l) \\ \Leftrightarrow \quad & -lx \leq -lu \\ \Leftrightarrow \quad & x \leq u \end{aligned}$$

The last step holds due to the fact that $l < 0$. Therefore, approximation (4.3) gives a feasible over-approximation of the equation $y = \max\{0, x\}$. □

Now we describe how Wang et al. [56] apply the approximation (4.3) to the symbolic equations. In general, each neuron has a symbolic lower bound eq_{low} and a symbolic upper bound eq_{up}. As long as the neuron depends only linearly on the input neurons, eq_{low} and eq_{up} coincide and give the exact value of the neuron (depending on the inputs of the network). This corresponds to the state that we still have the symbolic equation for the neuron in the first approach of Wang et al. [57]. If we have the situation that $eq := eq_{low} = eq_{up}$ describes the exact neuron value, and the numeric bounds of this neuron fulfill $l_{in} < 0 < u_{in}$ before the ReLU application, the approximation (4.3) is applied to the symbolic equation eq. Thus we obtain symbolic lower and upper bounds:

$$eq_{low} = \frac{u_{in}}{u_{in} - l_{in}} eq \qquad \text{and} \qquad eq_{up} = \frac{u_{in}}{u_{in} - l_{in}} (eq - l_{in})$$

If $l_{in} \geq 0$ the ReLU function is fixed in its positive phase, thus the symbolic equations $eq = eq_{low} = eq_{up}$ can be kept without change. If $u_{in} \leq 0$, the symbolic equations $eq = eq_{low} = eq_{up}$ can be fixed to 0.

If ReLU approximations have already occured, such that $eq_{low} \neq eq_{up}$, we do not have a symbolic equation for the neuron value anymore. Instead, the approximations are applied to the symbolic upper and lower bounds. To this end, Wang et al. [56] compute numeric lower and upper bounds l_{low}, u_{low} and l_{up}, u_{up} for eq_{low} and eq_{up}, respectively. It holds $l_{low} \leq l_{up}$, $u_{low} \leq u_{up}$ and $l_{low} \leq u_{up}$. Indeed, eq_{low} and eq_{up} are such that $eq_{low}(a) \leq eq_{up}(a)$, which denotes the evaluation of the symbolic equations for the input vector $a \in \mathbb{R}^{L_0}$. Of course, a needs to meet all input constraints. If $l_{low} \geq 0$ the ReLU function is already fixed in its positive phase. In this case, eq_{low} and eq_{up} can be kept without any change. If $u_{up} \leq 0$, both eq_{low} and eq_{up} can be set to 0. Otherwise we have $l_{low} < 0 < u_{up}$ and the symbolic bounds are updated like this[1]:

$$eq_{low} = \begin{cases} 0, & u_{low} \leq 0 \\ \frac{u_{low}}{u_{low} - l_{low}} eq_{low}, & u_{low} > 0 \end{cases}$$

$$eq_{up} = \begin{cases} eq_{up}, & l_{up} \geq 0 \\ \frac{u_{up}}{u_{up} - l_{up}} (eq_{up} - l_{up}), & l_{up} < 0 \end{cases}$$

Sometimes it is possible to fix either the ReLU phase of the upper symbolic bound or the ReLU phase of the lower symbolic bound. This is reached by distinguishing the different cases as above. Applying this scheme, the upper and lower symbolic bounds are propagated through the network. For any neuron, numeric bounds can be computed on basis of the feasible input domain of the instance of the verification problem. In (4.2) this computation is denoted explicitly. Each symbolic bound allows the computation of a lower and an upper numeric bound. The lower numeric bound of the lower symbolic bound and the upper numeric bound of the upper symbolic bound, i.e. l_{low} and u_{up}, are then the neuron bounds which we obtain.

Neuron	Symbolic bounds	Numeric bounds	Naive IA
x_1	$[a_1 + a_2, a_1 + a_2]$	$[-8, 6]$	$[-8, 6]$
x_2	$[a_1 - a_2 - 1, a_1 - a_2 - 1]$	$[-8, 6]$	$[-8, 6]$
y_1	$[\frac{3}{7}(a_1 + a_2),\ \frac{3}{7}(a_1 + a_2 + 8)]$	$[\frac{-24}{7}, 6]$	$[0, 6]$
y_2	$[\frac{3}{7}(a_1 - a_2 - 1),\ \frac{3}{7}(a_1 - a_2 + 7)]$	$[\frac{-24}{7}, 6]$	$[0, 6]$
o_1	$[\frac{3}{7}(2a_1 - 1),\ \frac{3}{7}(2a_1 + 15)]$	$[-6,\ \frac{57}{7}]$	$[0, 12]$

Table 4.2: The symbolic bounds and the concrete numerical values for our example network in Figure 4.2 computed as proposed by Wang et al. [56]. The symbolic relaxation allows also negative values for the neurons y_1 and y_2 which are known to be non-negative. Though, the upper bound of $\frac{57}{7}$ for o_1 is better than the bound 12 which is computed using symbolic or naive interval arithmetic (see column "Naive IA" and cf. Table 4.1).

Comparing Figures 4.1 and 4.3 we see, that the approximation (4.3) suggested by Wang et al. [56] can lead to superior results compared to the

[1] In Appendix B of Wang et al. [56], u_{low} and l_{up} are exchanged by mistake in the corresponding formulas, as confirmed by the main author.

naive approximation. This can be also seen in Table 4.2 which shows the resulting bounds for our example network depicted in Figure 4.2. While the ReLU function maps only to non-negative values, and also the naive approximation restricts its range to non-negative values, approximaton (4.3) allows also negative values after the ReLU application. Therefore, it is possible to find networks, for which the approximation (4.3) actually performs worse than the naive approximation.

Theorem 17. *The approximation of Wang et al. [56] can be worse than naive interval arithmetic.*

Proof. We use the neural network depicted in Figure 4.4 as an example to show that the bounds computed with the method of Wang et al. [56] can be worse than the bounds computed with naive interval arithmetic. We obtain the following bounds:

Variable	Symbolic Bound (Wang et al. [56])	Bound of Naive IA
x_1	$[a_1, a_1]$	$[-2, 4]$
x_2	$[a_2, a_2]$	$[-3, 6]$
y_1	$[\frac{2}{3}a_1, \frac{2}{3}a_1 + \frac{4}{3}]$	$[0, 4]$
y_2	$[\frac{2}{3}a_2, \frac{2}{3}a_2 + 2]$	$[0, 6]$
o_1	$[\frac{2}{3}(a_1 + a_2) + 5, \frac{2}{3}(a_1 + a_2) + 8\frac{1}{3}]$	$[5, 15]$

The symbolic bounds for o_1 can be evaluated numerically by substituting the lower bounds of a_1 and a_2 for the symbolic lower bound of o_1, and the upper bounds of a_1 and a_2 for the symbolic upper bound. Like this we obtain bounds $[\frac{5}{3}, 15]$ for o_1. Obviously the lower bound is inferior to the lower bound computed by naive interval arithmetic, while the upper bounds are identical. □

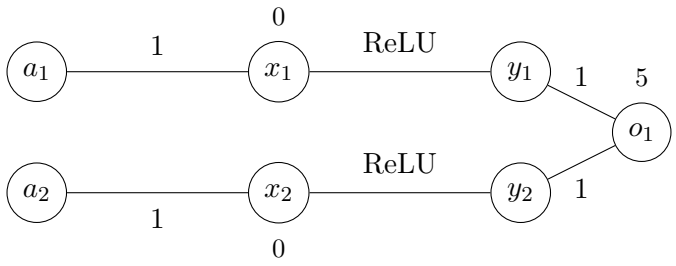

Figure 4.4: Neural network with input neurons $a_1 \in [-2, 4]$ and $a_2 \in [-3, 6]$. The numbers denote the weights or biases, respectively.

We will now consider another bound computation approach that enables a tighter approximation than the previously discussed approaches. Moreover, we will show that it is best possible in a certain sense, which we define in the following section. This linear approximation of the ReLU constraints was first

proposed by Ehlers [19]. Given $x \in [l, u]$, where $l < 0 < u$, and $y = \max\{0, x\}$ the following holds:

$$\begin{aligned} y &\geq 0 \\ y &\geq x \\ y &\leq \frac{u(x-l)}{u-l} \end{aligned} \tag{4.4}$$

We graphically depict this approximation in Figure 4.5 and show that in fact it coincides with the linear relaxation of the MIP formulation (3.1) for ReLU constraints. The following theorem is also mentioned in Anderson et al. [3] as Proposition 17, while the proof can be found in similar form in Bunel et al. [12], Appendix C.

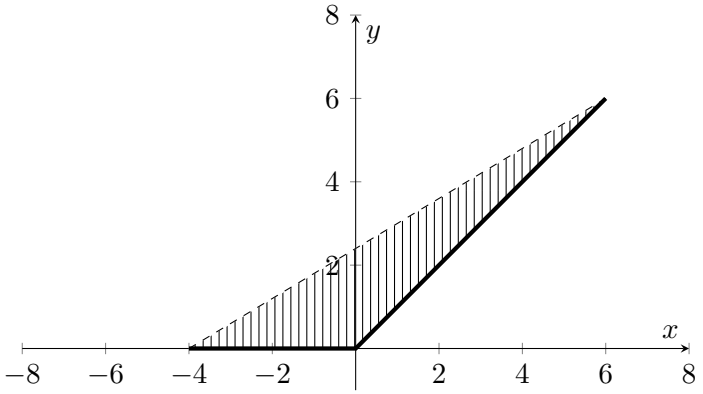

Figure 4.5: Approximation of ReLU function in one dimension as proposed by Ehlers [19]. Here we have lower bound -4 and upper bound 6 for the ReLU input variable x. The feasible domain of the ReLU output variable y is given by the solid black line for the actual ReLU function and by the shaded area for the convex approximation (4.4).

Theorem 18 (cf. Bunel et al. [12] and Anderson et al. [3])**.** *Given $l < 0 < u$, $x \in [l, u]$ and $y = \max\{0, x\}$, inequalities (4.4) hold. Moreover, the feasible set defined by these inequalities is identical to the embedded image which is obtained by projecting the linear relaxation of the MIP formulation (3.1) onto the variables x and y.*

Proof. First, we see that for $l \leq x \leq u$ and $y = \max\{0, x\}$ the pair (x, y) fulfills the approximation constraints. $y \geq 0$ and $y \geq x$ trivially hold and the inequality for the upper bound was already proved for Theorem 16. Now consider the LP relaxation of (3.1). Again, we have the constraints $y \geq x$ and $y \geq 0$. In addition, we obtain the following constraints:

$$\begin{aligned} y &\leq x - (1-d)l \\ y &\leq d \cdot u \\ d &\in [0, 1] \end{aligned}$$

Both constraints give an upper bound on the value of y. We consider the right hand sides as two functions $f, g : [0,1] \to \mathbb{R}$ of the variable d, i.e.

$$f(d) := x - (1-d)l = ld + x - l \qquad \text{and} \qquad g(d) := ud.$$

Since there are no other constraints on d in the model, we can write the upper bound of y equivalently as

$$y \leq \max_{d \in [0,1]} \min\{f(d), g(d)\}$$

For fixed x, both functions are affine, f has negative slope l and g has positive slope u. Moreover, $f(0) = x - l \geq 0 = g(0)$ and $f(1) = x \leq u = g(1)$. That means, due to the intermediate value theorem, there is $d \in [0,1]$ such that $f(d) = g(d)$. In this point, the maximum of $\min\{f(d), g(d)\}$ is attained, since both f and g are monotonic. We compute

$$\begin{aligned} & f(\hat{d}) = g(\hat{d}) \\ \Leftrightarrow \quad & l\hat{d} + x - l = u\hat{d} \\ \Leftrightarrow \quad & \hat{d} = \frac{x-l}{u-l} \end{aligned}$$

and thus the upper bound of y is given as $g(\hat{d}) = u\frac{x-l}{u-l}$. This is exactly the upper bound enforced by the approximation (4.4). □

Remark 19. Of course, the linear approximation (4.4) remains valid, if either the constraint $y \geq 0$ or the constraint $y \geq x$ is removed. This is regarded by Singh et al. [48] and Zhang et al. [63] for the application to robustness certification. While leaving out one of the constraints clearly weakens the approximation, it provides the advantage that there is only one inequation which acts as a lower bound. Surely, this is not useful if bounds are computed using linear programming. Though, it enables the use of matrix multiplication (cf. Zhang et al. [63]) or static analyzers with abstract domains (cf. Singh et al. [48]) for the propagation of the inequations.

In the context of robustness certification, Raghunathan et al. [43] propose an SDP relaxation for ReLU neural networks. Especially, this relaxation acts simultaneously on all neurons of a layer. First, Raghunathan et al. [43] show that the ReLU constraint $y = \max\{0, x\}$ for $x, y \in \mathbb{R}$ can be replaced by three linear or quadratic constraints: (i) $y(y - x) = 0$, (ii) $y \geq x$, and (iii) $y \geq 0$.

This allows to formulate the verification problem as a quadratically constrained quadratic program (QCQP), which can be relaxed to a semidefinite program (SDP). Therefore, we denote the ReLU constraint for a whole layer containing $n \in \mathbb{N}$ neurons as $y = \text{ReLU}(x)$ where x, $y \in \mathbb{R}^n$. This constraint can be replaced by the following linear and quadratic constraints, where $\odot$ denotes the elementwise multiplication of two vectors:

$$\begin{aligned} y &\geq 0 \\ y &\geq x \\ y \odot y &= x \odot y \\ x, y &\in \mathbb{R}^n \end{aligned} \tag{4.5}$$

In order to relax (4.5) to an SDP formulation, Raghunathan et al. [43] define

$$v := \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} \quad \text{and} \quad P := vv^T = \begin{bmatrix} 1 & x^T & y^T \\ x & xx^T & xy^T \\ y & yx^T & yy^T \end{bmatrix},$$

so that an SDP relaxation of (4.5) is given by:

$$\begin{aligned} y &\geq 0 \\ y &\geq x \\ \operatorname{diag}(yy^T) &= \operatorname{diag}(xy^T) \\ P_{1,1} &= 1 \\ P &\succcurlyeq 0 \end{aligned} \tag{4.6}$$

The constraint $P \succcurlyeq 0$ expresses that the matrix P must be positive semidefinite. We refer to Raghunathan et al. [43] for a more thorough analysis of this relaxation and a comparison to the linear approximation (4.4). Furthermore, Raghunathan et al. [43] explicitly denote the relaxation for neural networks with several ReLU layers and provide some hints on the inclusion of additional neuron bounds in the relaxation. In contrast to that, the rest of this chapter is restricted to results regarding linear approximations of ReLU constraints.

4.2 Comparison of linear ReLU approximations

In general, one ReLU layer contains several neurons, and we are interested to compute an approximation of the output range of the layer. This approximated output range can then be regarded as input to the next layer. As we want to reach a quick propagation of the output ranges through the layers, it is important that the approximated output range is a polytope. This allows to compute neuron bounds quickly using linear programming. We notice that in approximation (4.4) each ReLU output variable y only depends on the corresponding ReLU input variable x, but not on any other neurons in the same layer. In the following, we develop a theoretical framework to analyse different linear approximations, so that we can compare them to this approximation (4.4).

Definition 20 (ReLU approximation)**.** *Let $n, m \in \mathbb{N}$, $A, B \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, and $P \subset \mathbb{R}^n$ be a polytope. We say that*

$$Q := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in P \times \mathbb{R}^n \mid Ax + By \leq c \right\} \subset \mathbb{R}^{2n}$$

is a ReLU approximation (of P) if it holds that $(P \times ReLU(P)) \subseteq Q$. Q is called an independent ReLU approximation*, if for all $j \in [m]$, there exists $i \in [n]$ such that $A_{jl} = B_{jl} = 0$ for all $l \in [n] \setminus \{i\}$. A polytope P is called* ReLU proper*, if for all $i \in [n]$ it holds*

$$\min_{x \in P} x_i < 0 < \max_{x \in P} x_i.$$

Remark 21. For any ReLU approximation Q of $P \subset \mathbb{R}^n$, it is certain that $[P \times \operatorname{conv}(\operatorname{ReLU}(P))] \subseteq Q$. In addition, we point out that in an independent ReLU approximation, each row of A or B has at most one nonzero coefficient. Regarding the same row index in A and B, this coefficient must also have the same column index in both A and B. The consideration of ReLU proper polytopes simplifies the formulation of statements, as fixed ReLU neurons are not regarded.

The naive approximation applied to a ReLU proper polytope gives a box $[0, u_1] \times \ldots \times [0, u_n]$, where u_i is the upper bound for the corresponding variable. We see that this is an *independent ReLU approximation*. Let $A = 0 \in \mathbb{R}^{2n \times n}$ and for each $i \in [n]$, we add two rows to matrix B and vector c to enforce $0 \leq y_i \leq u_i$ for $i \in [n]$, i.e. $m = 2n$ for the m in Definition 20. These rows are $e_i^T \; y \leq u_i$ and $-e_i^T \; y \leq 0$, where e_i is the i-th unit vector in $\mathbb{R}^n$. Hence, we have exactly one non-zero coefficient in each row of B and only zero coefficients in A, so that the property holds. In passing we notice that the approximation (4.3) proposed by Wang et al. [56] is an independent ReLU approximation, too.

As our discussion of various approximation methods shows, the quality of these differs significantly. Now we use our defintion of a ReLU approximation for a more thorough investigation of the possibilities to approximate ReLU constraints. Within the restrictions of the definition, we would like to find matrices A, B and c for a ReLU proper polytope P, such that Q is as small as possible (with respect to inclusion). First, we will restrict our analysis to independent ReLU approximations and claim: the approximation (4.4) proposed by Ehlers [19] is best possible among all independent ReLU approximations of a ReLU proper polytope. We define this approximation formally as a ReLU approximation in order to state the result in Theorem 26.

Definition 22. *Let $P \subset \mathbb{R}^n$ be a ReLU proper polytope. The ReLU approximation of P corresponding to approximation (4.4) will be denoted as Q_E. In detail, for $i \in [n]$, we set*

$$A^{(i)} = \begin{bmatrix} 0 \\ e_i^T \\ \frac{u_i}{l_i - u_i} e_i^T \end{bmatrix}, \; B^{(i)} = \begin{bmatrix} -e_i^T \\ -e_i^T \\ e_i^T \end{bmatrix} \text{ and } c^{(i)} = \begin{bmatrix} 0 \\ 0 \\ \frac{u_i l_i}{l_i - u_i} \end{bmatrix}.$$

For that, we use $l_i := \min_{x \in P} \; x_i$ and $u_i := \max_{x \in P} \; x_i$ and eventually define

$$A_E = \begin{bmatrix} A^{(1)} \\ \vdots \\ A^{(n)} \end{bmatrix}, \; B_E = \begin{bmatrix} B^{(1)} \\ \vdots \\ B^{(n)} \end{bmatrix} \text{ and } c_E = \begin{bmatrix} c^{(1)} \\ \vdots \\ c^{(n)} \end{bmatrix}.$$

Thus we obtain

$$Q_E := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in P \times \mathbb{R}^n \mid A_E x + B_E y \leq c_E \right\} \subset \mathbb{R}^{2n}.$$

Remark 23. Indeed, Q_E is an independent ReLU approximation. All rows of A and B are either 0 or a multiple of a transposed unit vector $e_i^T \in \mathbb{R}^n$. If the latter is the case, $i \in [n]$ is the same both in A and B when regarding the same row indices of A and B.

Definition 24. *Let $S \subset \mathbb{R}^{2n}$. Then, for $i \in [n]$, we denote the embedded image (in $\mathbb{R}^2$) of the orthogonal projection of S on the subspace* $\operatorname{span}\{e_i, e_{i+n}\}$ *as $S\big|_i$.*

Lemma 25. *Let $P \subset \mathbb{R}^n$ be a ReLU proper polytope and $Q \subset \mathbb{R}^{2n}$ an independent ReLU approximation of P. Then it holds for all $i \in [n]$ and all $\hat{x} \in P$:*

$$Q\big|_i \cap (\{\hat{x}_i\} \times \mathbb{R}) = \big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i \tag{4.7}$$

Furthermore, there exist $\alpha_i \leq \beta_i$, $\alpha_i \in \mathbb{R} \cup \{-\infty\}$ and $\beta_i \in \mathbb{R} \cup \{+\infty\}$ for $i \in [n]$ such that

$$\{\hat{x}\} \times [\alpha_1, \beta_1] \times \ldots \times [\alpha_n, \beta_n] = Q \cap (\{\hat{x}\} \times \mathbb{R}^n). \tag{4.8}$$

Proof. As in Definition 20, we use

$$Q = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in P \times \mathbb{R}^n \mid Ax + By \leq c \right\} \subset \mathbb{R}^{2n}.$$

For the first part we see

$$\begin{aligned}
&Q\Big|_i \cap (\{\hat{x}_i\} \times \mathbb{R}) \\
&= \left\{ \begin{pmatrix} x_i \\ y_i \end{pmatrix} \;\middle|\; x \in P,\ y \in \mathbb{R}^n,\ Ax + By \leq c \right\} \cap (\{\hat{x}_i\} \times \mathbb{R}) && (4.9)\\
&= \left\{ \begin{pmatrix} \hat{x}_i \\ y_i \end{pmatrix} \;\middle|\; x \in P : x_i = \hat{x}_i,\ y \in \mathbb{R}^n,\ Ax + By \leq c \right\} && (4.10)\\
&= \left\{ \begin{pmatrix} \hat{x}_i \\ y_i \end{pmatrix} \;\middle|\; x \in P, x_i = \hat{x}_i,\ y \in \mathbb{R}^n,\ A^{(i)}x + B^{(i)}y \leq c^{(i)} \right\} && (4.11)\\
&= \left\{ \begin{pmatrix} \hat{x}_i \\ y_i \end{pmatrix} \;\middle|\; y \in \mathbb{R}^n,\ A^{(i)}\hat{x} + B^{(i)}y \leq c^{(i)} \right\} && (4.12)\\
&= \left\{ \begin{pmatrix} \hat{x}_i \\ y_i \end{pmatrix} \;\middle|\; y \in \mathbb{R}^n,\ A\hat{x} + By \leq c \right\} && (4.13)\\
&= \left[\left\{ \begin{pmatrix} x \\ y \end{pmatrix} \;\middle|\; x \in P, y \in \mathbb{R}^n,\ Ax + By \leq c \right\} \cap (\{\hat{x}\} \times \mathbb{R}^n) \right]\Big|_i && (4.14)\\
&= \big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i && (4.15)
\end{aligned}$$

where (4.9) holds due to the defintion of Q and the orthogonal projection, and (4.10) is rewritten. (4.11) holds, since the set is determined only by the values of $\hat{x}_i$ and y_i. The values of x_j and y_j for $j \in [n],\ j \neq i$ are not relevant for the set and therefore we do not need any constraints on these. Hence we can restrict the inequality in the set to the submatrices which are relevant for x_i and y_i. In (4.12) it suffices to consider $\hat{x}$ because all columns except the i-th column of $A^{(i)}$ are zero. We can now switch back to the original inequality in (4.13), as this affects only x_j and y_j for $j \in [n],\ j \neq i$. Then we can write the set as an intersection (4.14) and apply the definition of Q (4.15).

For the second part of the lemma, we stress that for $(x, y)^T \in Q$ each y_k, $k \in [n]$ is independent of the values of all other y_l, $l \neq k$. This is an immediate consequence of the definition of an *independent* ReLU approximation. Because $\hat{x} \in P$ is fixed, we can find a lower and upper bound for each y_k, $k \in [n]$ which define the feasible range. Of course, these bounds can be infinite. Thus, we obtain (4.8). □

Theorem 26. *Let $P \subset \mathbb{R}^n$ be a ReLU proper polytope and Q_E be the approximation of P as in Definition 22. For any independent ReLU approximation Q of P it holds $Q_E \subseteq Q$.*

Proof. Let $i \in [n]$ be fixed and Q be an independent ReLU approximation of P. We define the set

$$C_i := \Big\{(x_i, y_i) \mid x \in P,\ y_i = \max\{0, x_i\}\Big\}$$

and let $Q\big|_i$ be the embedded image (in $\mathbb{R}^2$) of the projection of Q on the variables x_i and y_i. We claim that it holds $\operatorname{conv}(C_i) \subseteq Q\big|_i$. By definition of Q we have $C_i \subseteq Q\big|_i$ and $Q\big|_i$ is a polyhedron, hence convex and therefore the claim holds. In the first part of this proof, we show that $\operatorname{conv}(C_i) = Q_E\big|_i$. To this end, we set

$$\tilde{Q}_E := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in P \times \mathbb{R}^n \mid A^{(i)}x + B^{(i)}y \leq c^{(i)} \right\} \subset \mathbb{R}^{2n}$$

where $A^{(i)}, B^{(i)}$ and $c^{(i)}$ are defined according to Definition 22 and for our fixed index i. Obviously, we have $Q_E \subseteq \tilde{Q}_E$ which implies $Q_E\big|_i \subseteq \tilde{Q}_E\big|_i$. We will show that $\tilde{Q}_E\big|_i = \operatorname{conv}(C_i)$, which allows us to conclude that

$$\operatorname{conv}(C_i) \subseteq Q_E\big|_i \subseteq \tilde{Q}_E\big|_i = \operatorname{conv}(C_i).$$

Let $l_i := \min_{x \in P}\ x_i$ and $u_i := \max_{x \in P}\ x_i$. Then we find

$$C_i = \big\{(x_i, \max\{0, x_i\}) \bigm|\ x_i \in [l_i, u_i]\big\},$$

which is a direct consequence of the convexity of P. Because P is ReLU proper we have $l_i < 0 < u_i$. Thus we can write

$$C_i = \big\{(x_i, 0) \bigm|\ x_i \in [l_i, 0]\big\} \ \cup\ \big\{(x_i, x_i) \bigm|\ x_i \in [0, u_i]\big\}.$$

Hence, C_i is the union of two one-dimensional polytopes and $\operatorname{conv}(C_i)$ has three vertices at coordinates $(l_i, 0), (0, 0)$ and (u_i, u_i). These are exactly the vertices of the polytopes that constitute C_i. Definition 22 establishes $A^{(i)}, B^{(i)}$ and $c^{(i)}$ such that they imply the following constraints for $(x, y)^T \in \tilde{Q}_E$:

$$\begin{aligned} y_i &\geq 0 \\ y_i &\geq x_i \\ y_i &\leq \frac{u_i(x_i - l_i)}{u_i - l_i} \end{aligned}$$

The segments between the vertices of $\operatorname{conv}(C_i)$ induce the following lines:

$$\begin{array}{c|c} (l_i, 0), (0, 0) & y_i = 0 \\ (0, 0), (u_i, u_i) & y_i = x_i \\ (u_i, u_i), (l_i, 0) & y_i = \frac{u_i(x_i - l_i)}{u_i - l_i} \end{array}$$

Taking the respective third vertex into account, we obtain exactly the constraints of $\tilde{Q}_E$ and hence show that $\tilde{Q}_E\big|_i = \operatorname{conv}(C_i)$. Since

$$\operatorname{conv}(C_i) \subseteq Q_E\big|_i \subseteq \tilde{Q}_E\big|_i = \operatorname{conv}(C_i),$$

we find that $\operatorname{conv}(C_i) = Q_E\big|_i$. Now, we will combine this result with Lemma 25 in order to prove the theorem.

To this end, we fix an arbitrary $\hat{x} \in P$. Combining the last result with (4.7) and applying it to Q_E, we obtain:

$$\operatorname{conv}(C_i) \cap (\{\hat{x}_i\} \times \mathbb{R}) = \big[Q_E \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i.$$

Using $\operatorname{conv}(C_i) \subseteq Q\Big|_i$, which we showed in the beginning of the proof, and (4.7) applied to Q, gives

$$\operatorname{conv}(C_i) \cap (\{\hat{x}_i\} \times \mathbb{R}) \subseteq Q\Big|_i \cap (\{\hat{x}_i\} \times \mathbb{R}) = \big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i.$$

Thus we obtain

$$\big[Q_E \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i = \operatorname{conv}(C_i) \cap (\{\hat{x}_i\} \times \mathbb{R}) \subseteq \big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i. \quad (4.16)$$

As a consequence of (4.8) in Lemma 25 we have

$$\big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big]\Big|_i = \{\hat{x}_i\} \times [\alpha_i, \beta_i]$$

and

$$Q \cap (\{\hat{x}\} \times \mathbb{R}^n) = \{\hat{x}\} \times [\alpha_1, \beta_1] \times \ldots \times [\alpha_n, \beta_n]$$

is entirely determined by the projections for all $i \in [n]$. The same holds for Q_E instead of Q. Since we fixed $i \in [n]$ arbitrarily in the beginning, with (4.16) we are now able to conclude that

$$\big[Q_E \cap (\{\hat{x}\} \times \mathbb{R}^n)\big] \subseteq \big[Q \cap (\{\hat{x}\} \times \mathbb{R}^n)\big].$$

Now fix an arbitrary $(\check{x}, \check{y}) \in Q_E$. Since $\hat{x} \in P$ was also arbitrary, we have

$$(\check{x}, \check{y}) \in \big[Q_E \cap (\{\check{x}\} \times \mathbb{R}^n)\big] \subseteq \big[Q \cap (\{\check{x}\} \times \mathbb{R}^n)\big]$$

which implies $(\check{x}, \check{y}) \in Q$. This proves $Q_E \subseteq Q$. □

In Sections 4.4 and 4.5 we discuss possibilities for approximations that are stronger than (4.4). Therefore we consider ReLU approximations which are not *independent*. Finally, we use this theoretical basis for the development of a novel approximation technique which is laid out in Section 4.7.

Nevertheless, the linear approximation (4.4) can be used to compute tight bounds for the neuron values in a neural network by solving linear programs. This method is proposed by Ehlers [19] and extensively used by Bunel et al. [11]. We give an outline of this approach in Section 4.3. In addition, we explain techniques that can be used to speed up the computation of the linear programs and make this method more effective and efficient.

4.3 Efficient optimization based bound tightening for neural network verification

If we build the MIP model using some preliminary lower and upper bounds for each ReLU neuron, we can use the LP relaxation of the model to approximate the output values of the neural networks. As shown in Theorem 18, this LP relaxation coincides with the approximation (4.4) proposed by Ehlers [19]. While optimizing this LP only with respect to the network outputs is fast, it does usually not suffice to prove relevant properties which are defined for the network. Yet, this linear approximation of the network can be used to tighten the lower and upper bounds for all neurons. Ehlers [19] and Bunel et al. [11] apply this bound tightening technique, which we now describe. For each ReLU input variable x we compute an optimal solution of the LP relaxation for the objective functions x and $-x$. The optimum objective values hence give the new bounds for x in the neural network. In accordance with Gleixner et al. [24], we call this technique optimization based bound tightening (OBBT).

After the bound update, it is crucial to improve the MIP formulation (3.1) or the corresponding linear relaxation (4.4), respectively. This allows to compute significantly tighter bounds in the next layer. Of course, the best case is that the lower bound is greater than 0, or the upper bound smaller than 0. In these cases, formulation (3.1) or (4.4) can actually be replaced by

$$\begin{cases} y = x & \text{if the lower bound of } x \text{ is positive, or} \\ y = 0 & \text{if the upper bound of } x \text{ is negative.} \end{cases}$$

Otherwise, formulation (3.1) or (4.4), respectively, can be strengthened by adding new constraints with the tighter values for l and u as bounds. Clearly, the bounds should be computed layer by layer, starting with the first and ending with the last layer. As the bounds of a neuron depend only on the bounds of the predecessor neurons, this order allows to incorporate stronger relaxations of the ReLU constraints for the neurons that follow behind. Indeed, it is possible to build the approximation of the whole network only during the process. That means, each variable is added separately to the model. If it is a ReLU input variable, the bounds of this variable are optimized and then the ReLU output variable is added using the optimized bounds for the ReLU constraint. This course of action has the advantage that the size of the LP is always as small as possible, as the variables which are not necessary are not yet included in the model.

The major drawback of this LP based approach is its very high computational cost, as it requires to solve a number of LPs which is up to twice the number of neurons in the network. We regard the ideas of Gleixner et al. [24], who implemented an OBBT propagator in SCIP [25]. This propagator tries to improve the bounds of a variable v by maximizing or minimizing the LP relaxation with objective function v. Although this propagator cannot be applied directly to our problem, as it is focused on nonlinear problems, we investigate which parts are most useful for verification of neural networks. Gleixner et al. [24] treat two main topics: First, they show how to generate and propagate Langrangian variable bounds (LVBs), and second, they propopse methods for

the acceleration of OBBT.

LVBs are valid inequalities with respect to the LP relaxations of mixed integer nonlinear problems (MINLP), which also includes LP relaxations of MIPs. Gleixner et al. [24] state that *LVBs can be viewed as a one-row relaxation of the given MINLP*, that provide *a local approximation of the effect of OBBT*. They are obtained as a by-product of the LP solutions which are computed during the execution of OBBT. For the following theorem, which introduces LVBs, we use the definition of Gleixner et al. [24] for an MINLP.

Definition 27 (MINLP, Gleixner et al. [24])**.**

$$\begin{aligned} \min \quad & c^T x & & \\ & g_k(x) \leq 0 & \forall k \in [p] & \\ & l_i \leq x_i \leq u_i & \forall i \in [n] & \\ & x \in \mathbb{R}^n & & \\ & x_i \in \mathbb{Z} & \forall i \in \Lambda & \end{aligned} \tag{4.17}$$

is called a mixed integer nonlinear program (MINLP). We have $c \in \mathbb{R}^n$ as objective function vector, $l_i \in \mathbb{R} \cup \{-\infty\}$ and $u_i \in \mathbb{R} \cup \{+\infty\}$ for $i \in [n]$ as lower and upper bounds, and $\Lambda \subseteq [n]$ is the index set of integral variables. The constraint functions $g_k : \{x \mid \forall i \in [n] : l_i \leq x_i \leq u_i\} \to \mathbb{R}$, with $k \in [p]$ for some $p \in \mathbb{N}$, may be non-convex.

Theorem 28 (Gleixner et al. [24], Theorem 1)**.** *Suppose we have a linear relaxation of MINLP (4.17) which has the feasibility set*

$$S := \left\{x \in \mathbb{R}^n \mid Ax \geq b,\ \forall i \in [n] : l_i \leq x_i \leq u_i\right\}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $U \in \mathbb{R}$ is a valid upper bound on the optimum objective value of (4.17). Furthermore, suppose that we have an optimal primal-dual solution $(\tilde{x}, \tilde{\lambda}, \tilde{\mu})$ to

$$\min\left\{x_k \mid Ax \geq b, c^T x \leq U,\ \forall i \in [n] : l_i \leq x_i \leq u_i\right\}$$

where $\tilde{\lambda} \in \mathbb{R}^m_{\geq 0}$ is the vector of dual multipliers for $Ax \geq b$ and $\tilde{\mu} \leq 0$ is the dual mutliplier of the objective cut-off constraint. Let $\tilde{r} = e_k - A^T\tilde{\lambda} - c\tilde{\mu}$ be the vector of reduced costs, where e_k is the k-th unit vector. Then

$$x_k \geq \sum_{j=1}^{n} \tilde{r}_j x_j + \tilde{\mu} U + \tilde{\lambda}^T b \tag{4.18}$$

is a valid inequality for all $x \in S \cap \left\{x \in \mathbb{R}^n \mid c^T x \leq U\right\}$, which is tight at $\tilde{x}$. If $(\tilde{x}, \tilde{\lambda}, \tilde{\mu})$ with $\tilde{\lambda} \in \mathbb{R}^m_{\leq 0}$ and $\tilde{\mu} \geq 0$ is optimal for

$$\max\left\{x_k \mid Ax \geq b, c^T x \leq U,\ \forall i \in [n] : l_i \leq x_i \leq u_i\right\}$$

then the same holds for

$$x_k \leq \sum_{j=1}^{n} \tilde{r}_j x_j + \tilde{\mu} U + \tilde{\lambda}^T b. \tag{4.19}$$

Valid inequalities of type (4.18) and (4.19) are referred to as LVBs. The proof of Theorem 28 is based on basic LP duality and can be found in Gleixner et al. [24]. We also refer to Gleixner et al. [24] for more details on questions regarding the existence and usefulness of LVBs.

Remark 29. If we model the neural network verification problem as optimization problem as described in Section 3.2, we are only interested whether there exists a solution with objective value smaller than or equal to zero or not. Hence, we can safely cut off all solutions with an objective value greater than some $\varepsilon > 0$. Either the regarded instance Π is refutable and there exists an optimum solution of the corresponding MIP with negative (or non-positive, cf. Remark 10) objective value, which is hence smaller than ε. Otherwise Π is verifiable such that the corresponding MIP has optimum objective value $\gamma \geq 0$. If $\gamma \leq \varepsilon$, we have a non-negative optimum solution as it is the case without application of LVBs. On the other hand, if $\gamma > \varepsilon$ the MIP becomes infeasible and we also know that the corresponding instance of the verification problem is verifiable. For our experiments we set $\varepsilon := 0.1$ to have a sufficiently big margin to zero in order to prevent erroneous results. If the verification problem is formulated as feasibility problem, we cannot use this approach for the generation of LVBs as there is no objective function.

The advantage of LVBs is that they can be propagated efficiently through a branch-and-bound tree, while the frequent application of OBBT requires a great computational effort for each branch-and-bound node that is processed. In fact, LVBs are redundant inequalities and thus it is not benefical to add them to the LP relaxation as Gleixner et al. [24] (Remark 3) point out. Nevertheless, LVBs *identify bounds that are already implied by the relaxation and by making them explicit* (cf. Gleixner et al. [24], Remark 3) they can be propagated to subsequently tighten other bounds. This is possible, since the tightening of variable bounds for some x_j in (4.18) or (4.19) tightens the bound for the corresponding x_k on the left hand side. We refer to Gleixner et al. [24] for more details on the question in which order this propagation can be performed reasonably. Of course, the propagation of LVBs cannot tighten the variable bounds as much as the repeated application of OBBT. Yet, we see in our experiments that for some instances the creation of LVBs is able to speed up the solving process significantly. We refer to Section 9.2 for an overview of the experiments. For detailed computational results see the configurations marked by "genv" in Section C.3 of the appendix.

Gleixner et al. [24] regard three aspects which shall serve to accelerate the application of OBBT. The first aspect is the selection of variables for which to apply OBBT. While Gleixner et al. [24] give some arguments for which variables OBBT should be applied or not, the answer to this question is rather clear in our case. It is only useful to apply OBBT to those neuron variables that follow a ReLU layer. Moreover, we also apply OBBT to the objective variable, if the problem is solved as an optimization problems as described in Section 3.2. This is useful, because the objective variable depends non-linearly on the variables which model the properties to verify. This non-linearity is modelled with the help of various binary variables, as laid out in Section 3.2.

That means, in general OBBT is applied to all ReLU input variables,

except the ones in the first ReLU layer, which depend linearly on the input neurons. It is also applied to variables which are part of a layer without ReLU activations if that layer follows a ReLU layer. It is not useful to apply OBBT to ReLU output variables, since the bounds for these variables can immediately be obtained by applying the ReLU function to the bounds of the corresponding ReLU input variables. Furthermore, for neuron variables which follow a layer without activation function, the variable bounds are simply an affine combination of the bounds in the previous layer. Hence, these bounds can be easily computed and there is no need to apply OBBT. Summing up, we have a subset of variables for which it is useful to apply OBBT, while for the rest of the variables applying OBBT cannot reach any improvements.

Of course, it can be considered not to apply OBBT to all variables where it can be useful. It could be advantageous not to optimize bounds for neurons in rear layers which typically have loose bounds. These optimizations take more time, since the corresponding LPs contain more variables and constraints. Also, they have less impact on other neurons than optimizations in the front layers of a network. Therefore, we consider to skip OBBT if the bounds of a variable are too loose. More exactly, we consider the difference of upper bound and lower bound before application of OBBT. If this value exceeds a certain threshold, we do not execute OBBT. On the one hand, this mostly affects neurons in rear layers, so that these bounds do not have much impact on further bound computations. On the other hand, we are mostly interested in bounds that switch their sign from positive to negative or vice versa. If a bound is quite large before OBBT is applied, it is very likely that applying OBBT will not make the bound switch its sign. Therefore we try if we can save computation time by skipping OBBT for bounds with (too) large absolute values. However, this does not seem to be the case as the performance of the configuration "no_heur_base_200" in Table 9.6 shows.

The second aspect is filtering bounds which can hardly be improved by executing OBBT. Assume that y is the value of a variable, which is a candidate for the application of OBBT, in a feasible solution of the LP relaxation. Moreover, let $l \leq y \leq u$ be the bounds which are currently known for this variable. If then $y - l \leq \varepsilon$ or $u - y \leq \varepsilon$ for some $\varepsilon > 0$, *OBBT can strengthen the corresponding bound by at most* ε, as Gleixner et al. [24] point out. The exclusion of such variables, whose LP solution values are already close to their bounds, is called bound filtering. One can compute optimal solutions of the LP relaxation using certain objective functions to hopefully obtain such solutions with variable values close to their bounds. This is called *aggressive bound filtering* by Gleixner et al. [24]. More exactly, Gleixner et al. [24] propose to use positive and negative coefficients in the objective function to push variables towards their upper or lower bound, respectively. If a variable's bound can be filtered, the corresponding objective coefficient is set to zero (or could be set to a coefficient of opposite sign to try and filter the other bound).

Though, for two reasons we do not make use of bound filtering in our solving model for verification of neural networks. On the one hand, initial experiments showed, that usually almost all bounds can be improved significantly by OBBT. Subsequently, also aggressive bound filtering as suggested by Gleixner et al. [24] cannot provide solutions with variable values close to the

previously known bounds. This happens, because the actual variable bounds, computed by OBBT, are usually clearly tighter than the bounds which can be obtained by any simple method, which even includes the symbolic bound computation of Wang et al. [56]. On the other hand, it is very beneficial to keep the LP relaxation small by including only those variables which are relevant for the value of the current neuron variable, as described earlier in this subsection. This implies, that normally we do not know the solution value of a variable in the LP relaxation, for which OBBT has not been applied. Before OBBT is applied to this variable, it is not included in the LP relaxation. Therefore, it seems reasonable not to employ bound filtering for our purpose.

The third aspect is the order of the variables for which OBBT is executed. Roughly, this is clear in our case, since variables should be processed layerwise. The approximation of ReLU layers in the front of the network impacts the bounds which can be computed for rear layers. Hence, OBBT should be executed first for the first applicable layer, then for the second until the last layer is reached. Yet, within the layers, it might be beneficial to use an order of variables which reduces the number of simplex iterations that are needed to solve the LPs which arise during the application of OBBT. Gleixner et al. [24] propose three different strategies to order the variables for which OBBT is applied. As the computation of good bounds is crucial for verification of neural networks, and OBBT performs very well for this task, we do not establish a time limit for OBBT or a limit on the number of simplex iterations as opposed to Gleixner et al. [24]. Therefore, we are only interested to reduce the number of simplex iterations which are necessary for the computation of all bounds. Subsequently, we are never forced to skip OBBT for some variables due to an iteration limit which is reached.

Gleixner et al. [24] suggest to use a greedy ordering, i.e. OBBT is applied first to the variable, whose value in a current solution of the LP relaxation is closest to one of its previously computed bounds. The idea of this approach is that it requires less simplex iterations to reach optimality if there is not much difference between the variable's relaxation value and the known bound of the variable. After OBBT is executed, the next variable is chosen based on the LP solution that was found by OBBT and the differences to the variable bounds. In this stratey, maximization and minimization are mixed arbitrarily, i.e. if a lower bound is closest to a variable value, we minimize, otherwise it is maximized. On the contrary, the strategy called *min-max* by Gleixner et al. [24] first solves all minimization and then all maximization LPs. *This is motivated by the conjecture that the solutions to the minimization LPs are close to each other and that processing them sequentially may (indirectly) exploit the simplex algorithm's warm starting capabilities; likewise for the solutions to the maximization LPs* (Gleixner et al. [24]). The reverse greedy strategy which is also considered by Gleixner et al. [24] is not useful in our setting. In this strategy variables are chosen first for optimization, that have a large distance to their bounds. The goal is to find *"important" reductions, which are often those with a large ratio between the original and the reduced domain* (Gleixner et al. [24]), before the iteration limit is reached. Yet, the solutions are likely to be more scattered across the feasible region of the LP relaxation which implies the necessity for more simplex iterations. Since we do not impose an iteration

limit, this strategy does not seem to be useful for our application.

We compare these orders with our standard order, which processes all neurons of each layer sequentially in a fixed order that arises from the definition of the neural network. Then, each ReLU input variable is first minimized and then maximized. However, our experiments do not show a significant reduction of the runtime if the greedy ordering or the min-max strategy is used. In some cases the number of solving nodes can be reduced, as certain nodes can be cut off earlier if one of these orders is used. The computational results can be found in Section B.1 of the appendix.

Eventually, we consider another approach for bound computations in neural networks that is also a form of OBBT. Instead of using the LP relaxation to compute bounds, it is also possible to employ the exact MIP model and compute bounds for the neurons with OBBT. This might sound absurd in the first place, since computing bounds of the output neurons of a neural network using a MIP model is the topic of the entire thesis. Indeed, applying OBBT based on MIPs is a technique which is worth to be considered by an empirical evaluation. Computing the best possible neuron bounds using the MIP fomulation instead of the LP relaxation leads to strongly improved bounds. Although not all MIPs are solved to optimality, clear improvements of the corresponding bounds can be reached within a time limit of few seconds per MIP. These improvements render it possible to solve also relevant instances without specialized branching rules for neural network verification.

Clearly, OBBT could also be applied to the SDP relaxation of Raghunathan et al. [43] that we presented in Section 4.1. This should provide better bounds than OBBT on the LP relaxation, while solving the SDPs should be significantly faster than solving the corresponding MIPs. However, we do not implement this approach in the context of this thesis. In future work, it would be interesting to use the SCIP plugin SCIP-SDP (see Gally et al. [22]) to implement an SDP relaxation based model for neural network verification. For the rest of the thesis, the abbreviation OBBT refers to OBBT on the LP relaxation if not stated differently.

4.4 Polyhedral aspects of ReLU function in higher dimensions

In general we regard neural network layers that feature ReLU activations for all neurons of the layer. Hence, we investigate in more detail how the ReLU function behaves in higher dimensions, i.e. if the ReLU function is applied componentwise to layers with several neurons. The following definition allows us to describe the image of the ReLU function in higher dimensions.

Definition 30 (Orthant)**.** *An orthant in $\mathbb{R}^n$ is a set*

$$\Omega(\epsilon_1, \dots, \epsilon_n) := \{x \in \mathbb{R}^n \mid \forall i \in [n] : \epsilon_i x_i \geq 0\}$$

where $\epsilon_i \in \{-1, 1\}$ for $i \in [n]$. Furthermore, we say that

$$\Theta := \{\Omega(\epsilon_1, \dots, \epsilon_n) \mid \forall i \in [n] : \epsilon_i \in \{-1, 1\}\}$$

is the orthant system in $\mathbb{R}^n$.

An orthant is called quadrant if $n = 2$, or octant if $n = 3$, respectively. Given $n \in \mathbb{N}$, there are 2^n different combinations of values for the ϵ_i, hence there are 2^n different orthants.

Remark 31. An orthant is a polyhedron, which can be seen directly from the definition. Moreover, if Θ is the orthant system in $\mathbb{R}^n$, then for each $x \in \mathbb{R}^n$ there is $\Omega \in \Theta$ such that $x \in \Omega$. This is easily seen, since for $x = (x_1, \ldots, x_n)$ we can set $\epsilon_i = \operatorname{sgn}(x_i)$ for $i \in [n]$. Hence $x \in \Omega(\epsilon_1, \ldots, \epsilon_n)$, because for $i \in [n]$ it holds $\epsilon_i x_i \geq 0$.

Lemma 32. *Let $P \subset \mathbb{R}^n$ be a polytope, and $\Omega := \Omega(\epsilon_1, \ldots, \epsilon_n)$ be an orthant. We define $S := \{i \in [n] \mid \epsilon_i = -1\}$. The image of the ReLU function over the domain $P \cap \Omega$ is $\operatorname{ReLU}(P \cap \Omega)$, which is given as the image (in $\mathbb{R}^n$) of the orthogonal projection of P to the subspace $V = \{x \in \mathbb{R}^n \mid \forall i \in S : x_i = 0\}$. Especially, $\operatorname{ReLU}(P \cap \Omega)$ is a polytope of dimension at most $n - |S|$.*

Proof. We define $c \in \mathbb{R}^n$ such that for $i \in [n]$, $c_i = 0$ if $i \in S$ and $c_i = 1$ otherwise. Now let C be the diagonal matrix with c on the diagonal. We claim that C represents the orthogonal projection on V. Let $x \in \mathbb{R}^n$ be arbitrary, then we find $Cx \in V$. Moreover, for $v \in V$, we have $v_i = 0$ for all $i \in S$, and $[Cx]_i = x_i$ for all $i \in [n] \setminus S$. Hence, $v^T(Cx - x) = 0$ for all $v \in V$ and the claim is shown.

Regard $x \in P \cap \Omega$ and let $r := \operatorname{ReLU}(x)$. Then it holds $r_i = 0$ for all $i \in S$ and $r_i = x_i$ for all $i \in [n] \setminus S$. That means, $r = Cx$. Hence, the orthogonal projection C is identical to the application of the ReLU function on the domain $P \cap \Omega$. Especially, this means that $\operatorname{ReLU}(P \cap \Omega) = \{Cx \mid x \in P \cap \Omega\}$ is a polytope, as it is the image of a polytope under an affine map.

Eventually, V can be written as $V = \{Cx \mid x \in \mathbb{R}^n\}$. As C has rank $n - |S|$, this implies $\dim(V) = n - |S|$. Since $\operatorname{ReLU}(P \cap \Omega) \subseteq V$, it follows $\dim(\operatorname{ReLU}(P \cap \Omega)) \leq n - |S|$. □

Theorem 33 (cf. Xiang et al. [62], Corollary 1)**.** *For a polytope $P \subset \mathbb{R}^n$, $\operatorname{ReLU}(P)$ is a finite union of polytopes.*

Proof. Let Θ be the orthant system in $\mathbb{R}^n$ and note that $|\Theta| = 2^n$. Moreover, Remark 31 implies that $P \subseteq \bigcup_{\Omega \in \Theta} \Omega = \mathbb{R}^n$. Thus, applying Lemma 32 to $P \cap \Omega$ for all $\Omega \in \Theta$ immediately proves the theorem. □

Hence we see that the convex hull of the union of polytopes in Theorem 33 is the best possible convex approximation of the ReLU activations of a layer. This is clearly true, since Theorem 33 describes the exact feasible set after the application of the ReLU function. The best convex approximation of this set is its convex hull, which follows directly from the definition of the convex hull.

4.5 Computation and comparison of the ReLU image and its approximation

We investigate a simple example to show how the approximation (4.4) of Ehlers [19] differs from the best possible convex approximation. First we show how to compute the convex hull of the ReLU image in higher dimensions.

Theorem 34. *Let $P \subset \mathbb{R}^n$ be a polytope and Θ be the orthant system of $\mathbb{R}^n$. We let*

$$V_\Omega := \text{Vertices}(\Omega \cap P) \text{ for } \Omega \in \Theta \quad \text{and} \quad V := \bigcup_{\Omega \in \Theta} V_\Omega.$$

Then it holds $\text{conv}(\text{ReLU}(P)) = \text{conv}(\text{ReLU}(V))$.

Proof. The definition of V implies $V \subseteq P$ and hence it holds the inclusion $\text{conv}(\text{ReLU}(V)) \subseteq \text{conv}(\text{ReLU}(P))$. For the opposite direction, we show $\text{ReLU}(P) \subseteq \text{conv}(\text{ReLU}(V))$ which implies the desired inclusion due to the definition of the convex hull.

Let $x \in P$ and choose $\Omega \in \Theta$ such that $x \in \Omega$, which is possible as laid out in Remark 31. Then we have $x \in \Omega \cap P$ which is a polytope since P is a polytope and Ω is a polyhedron. Subsequently, we can also use the convex hull representation of this polytope which implies $x \in \text{conv}(V_\Omega) = \Omega \cap P$. Hence there exist $\lambda_j \geq 0$, $j \in [m]$ for some $m \in \mathbb{N}$ such that $\sum_{j=1}^m \lambda_j = 1$ and $x = \sum_{j=1}^m \lambda_j v_j$ for certain $v_1, \dots, v_m \in V_\Omega$. In the following we see that

$$\text{ReLU}(x) = \sum_{j=1}^m \lambda_j \,\text{ReLU}(v_j). \tag{$*$}$$

For $i \in [n]$ we have $x_i = \sum_{j=1}^m \lambda_j [v_j]_i$ and we distinguish three cases. If $x_i > 0$, it is

$$\text{ReLU}(x_i) = x_i = \sum_{j=1}^m \lambda_j [v_j]_i = \sum_{j=1}^m \lambda_j \,\text{ReLU}([v_j]_i).$$

On the other hand, if $x_i < 0$, we have

$$\text{ReLU}(x_i) = 0 = \sum_{j=1}^m \lambda_j \cdot 0 = \sum_{j=1}^m \lambda_j \,\text{ReLU}([v_j]_i).$$

In both cases, the last equation holds because $x, v_1, \dots, v_m \in \Omega$ and Ω is an orthant. That means $x_i > 0$ implies $[v_j]_i \geq 0$ for all $j \in [m]$, and $x_i < 0$ implies $[v_j]_i \leq 0$ for all $j \in [m]$.

If $x_i = 0$, which is the third case, it is either $\lambda_j [v_j]_i \geq 0$ for all $j \in [m]$ or $\lambda_j [v_j]_i \leq 0$ for all $j \in [m]$. This holds, since $\lambda_j \geq 0$ for all $j \in [m]$, $v_1, \dots, v_m \in \Omega$ and Ω is an orthant. Subsequently, with $0 = x_i = \sum_{j=1}^m \lambda_j [v_j]_i$ we have for all $j \in [m]$, that either $\lambda_j = 0$ or $[v_j]_i = 0$. In this case it holds that

$$\text{ReLU}(x_i) = 0 = \sum_{\substack{j \in [m] \\ \lambda_j = 0}} \lambda_j [v_j]_i + \sum_{\substack{j \in [m] \\ [v_j]_i = 0}} \lambda_j [v_j]_i = \sum_{j=1}^m \lambda_j \,\text{ReLU}([v_j]_i).$$

As the ReLU function acts componentwise on x, we can apply these three cases for $i = 1, \dots, n$ to prove $(*)$. Thus we find

$$\text{ReLU}(x) \in \text{conv}(\text{ReLU}(V_\Omega)) \subseteq \text{conv}(\text{ReLU}(V)).$$

Since $x \in P$ was arbitrary, we have shown that $\text{ReLU}(P) \subseteq \text{conv}(\text{ReLU}(V))$. □

Now we prove how to compute the linear approximation (4.4) of the ReLU function for several variables. To this end we use the ReLU approximation Q_E of Definition 22 and consider its projection $Q_E\big|_y$ to the y variables, which represent the output of the ReLU layer. It should be noted that the following theorem and its proof are rather technical, and serve for the correct computation of examples in the rest of this chapter.

Theorem 35. *Let $P \subset \mathbb{R}^n$ be a ReLU proper polytope and Q_E be the ReLU approximation of P as in Definition 22. Furthermore, let Θ be the orthant system of $\mathbb{R}^n$. We let*

$$V_\Omega := \text{Vertices}(\Omega \cap P) \text{ for } \Omega \in \Theta \quad \text{and} \quad V := \bigcup_{\Omega \in \Theta} V_\Omega.$$

In addition, let $l_i := \min_{x \in P} x_i$ and $u_i := \max_{x \in P} x_i$ and for $a \in [l_i, u_i]$, $i \in [n]$, set

$$l^{(i)}(a) := \begin{cases} a, & a > 0 \\ 0, & a \leq 0 \end{cases} \qquad \text{and} \qquad u^{(i)}(a) := (a - l_i)\frac{u_i}{u_i - l_i}.$$

We define a mapping $f : P \to \mathbb{P}(\mathbb{R}^n)$ such that for $x \in P$

$$f(x) = \Big\{(z_1, \ldots, z_n) \in \mathbb{R}^n \;\Big|\; \forall i \in [n] : z_i \in \{l^{(i)}(x_i), u^{(i)}(x_i)\}\Big\}.$$

Moreover, for $W \subseteq P$ we define

$$f(W) := \bigcup_{w \in W} f(w).$$

Then it holds $Q_E\big|_y = \text{conv}(f(V))$.

Proof. By definition of V we have $V \subseteq P$ so that $f(V)$ is well defined. First we see that for all $v \in V$ it holds $f(v) \subseteq Q_E\big|_y$. This implies $\text{conv}(f(V)) \subseteq Q_E\big|_y$ since $Q_E\big|_y$ is a polytope and hence convex. Therefore, let $v \in V \subseteq P$ be arbitrary and $z \in f(v)$, i.e. $z_i \in \{l^{(i)}(v_i), u^{(i)}(v_i)\}$ for $i \in [n]$. We need to show that $z \in Q_E\big|_y$ which we do by showing that $(v, z) \in Q_E$. In view of Definition 22 it holds that $(v, z) \in Q_E \Leftrightarrow A_E v + B_E z \leq c_E$ which is equivalent to

$$\begin{aligned} -z_i &\leq 0 && \Leftrightarrow z_i \geq 0 \\ v_i - z_i &\leq 0 && \Leftrightarrow z_i \geq v_i \\ \frac{u_i}{l_i - u_i} v_i + z_i &\leq \frac{u_i l_i}{l_i - u_i} && \Leftrightarrow z_i \leq \frac{u_i(v_i - l_i)}{u_i - l_i} \end{aligned}$$

for all $i \in [n]$. We fix an $i \in [n]$ and show that the three inequalities hold. For better readability, we will write l and u instead of $l^{(i)}$ and $u^{(i)}$ in the following.

Clearly, $z_i \geq 0$ holds, since $l(v_i) \geq 0$ by definition of l and

$$u(v_i) = (v_i - l_i)\frac{u_i}{u_i - l_i} \geq 0.$$

The latter follows from $u_i \geq 0$ (since P is ReLU proper), $u_i - l_i \geq 0$ and $v_i - l_i \geq 0$. Next, we show $z_i \geq v_i$.

While $l(v_i) \geq v_i$ is clear, for $u(v_i)$ we see

$$\begin{aligned} & u(v_i) = (v_i - l_i)\frac{u_i}{u_i - l_i} \geq v_i \\ \Leftrightarrow \quad & -l_i u_i \geq -v_i l_i \\ \Leftrightarrow \quad & u_i \geq v_i \end{aligned}$$

The last line clearly holds and is equivalent to the line before, since P is ReLU proper and thus $-l_i > 0$. We are left to show the last inequality

$$z_i \leq \frac{u_i(v_i - l_i)}{u_i - l_i} = u(v_i).$$

Assume $z_i = l(v_i)$, i.e. $z_i = 0$ or $z_i = v_i$ and we have already shown these inequalities above. Otherwise, if $z_i = u(v_i)$, the inequality holds by definition of u. Hence, we see that $z \in Q_E\big|_y$ and we can conclude that $f(v) \subseteq Q_E\big|_y$ since $z \in f(v)$ was chosen arbitrarily. For the second part of the proof, we we need to show the opposite inclusion $Q_E\big|_y \subseteq \operatorname{conv}(f(V))$.

Let $(x, y) \in Q_E$ be arbitrary. Definition 22 and the ReLU approximation (4.4) imply $y_i \geq 0$, $y_i \geq x_i$ and $y_i \leq (x_i - l_i)\frac{u_i}{u_i - l_i}$, i.e. $l(x_i) \leq y_i \leq u(x_i)$ for $i \in [n]$. First, we regard the case $l(x_i) = u(x_i)$, i.e. $l(x_i) = y_i = u(x_i)$. This implies either $u(x_i) = 0$ or $u(x_i) = x_i$, which is the case if $x_i = l_i$ or $x_i = u_i$, respectively, and then we define $c_i := \frac{1}{2}$, so that

$$(1 - c_i)l(x_i) + c_i u(x_i) = \frac{1}{2}l(x_i) + \frac{1}{2}u(x_i) = y_i.$$

Otherwise it holds $l(x_i) < u(x_i)$ and we define

$$c_i := \frac{y_i - l(x_i)}{u(x_i) - l(x_i)} \in [0, 1].$$

This definition implies

$$\begin{aligned} & (1 - c_i)l(x_i) + c_i u(x_i) \\ = \; & l(x_i)(1 - \frac{y_i - l(x_i)}{u(x_i) - l(x_i)}) + u(x_i)\frac{y_i - l(x_i)}{u(x_i) - l(x_i)} \\ = \; & \frac{l(x_i)(u(x_i) - l(x_i)) - l(x_i)y_i + l(x_i)^2 + u(x_i)(y_i - l(x_i))}{u(x_i) - l(x_i)} \\ = \; & \frac{y_i(u(x_i) - l(x_i))}{u(x_i) - l(x_i)} = y_i. \end{aligned}$$

As in the proof of Theorem 34, there is $\Omega \in \Theta$ such that $x \in \Omega \cap P$. Analogously we find $m \in \mathbb{N}$, $v_1, \ldots, v_m \in V_\Omega$, and $\lambda_1, \ldots, \lambda_m \geq 0$ with $\sum_{j=1}^m \lambda_j = 1$ such that $x = \sum_{j=1}^m \lambda_j v_j$. We see that

$$\begin{aligned} y_i &= (1 - c_i)l(x_i) + c_i u(x_i) \\ &= (1 - c_i)l\big(\sum_{j=1}^m \lambda_j [v_j]_i\big) + c_i u\big(\sum_{j=1}^m \lambda_j [v_j]_i\big) \\ &= \sum_{j=1}^m \lambda_j \Big[(1 - c_i)l([v_j]_i) + c_i u([v_j]_i)\Big]. \end{aligned} \tag{4.20}$$

The last equation holds, since u is linear and l is also linear, if restricted to one of the domains $\mathbb{R}_{\geq 0}$ or $\mathbb{R}_{\leq 0}$. Since $v_1, \ldots, v_m \in \Omega$ and Ω is an orthant, and $\lambda_1, \ldots, \lambda_m \geq 0$, all summands are either non-positive or non-negative.

After regarding the components y_i of y we will now see how y can be written as a convex combination of points in $f(V)$. For $z_k^{(j)} \in f(v_j)$ we define coefficients $c_k^{(j)}$. The definition of f implies $|f(v_j)| = 2^n$, that means we have indices $k \in [2^n]$. For better readability, we will assume $j \in [m]$ as fixed and remove it partly from the notation. We define $c_k^{(j)} = \prod_{i=1}^n \gamma_i^{(k)}$ with

$$\gamma_i^{(k)} = \begin{cases} 1 - c_i, & \text{if } [z_k^{(j)}]_i = l([v_j]_i) \\ c_i, & \text{if } [z_k^{(j)}]_i = u([v_j]_i) \end{cases}.$$

We claim that

$$y = \sum_{j=1}^{m} \sum_{k=1}^{2^n} \lambda_j c_k^{(j)} z_k^{(j)} \quad \text{and} \quad \sum_{j=1}^{m} \sum_{k=1}^{2^n} \lambda_j c_k^{(j)} = 1 \tag{4.21}$$

which means that $y \in \operatorname{conv}(f(V))$ since $z_k^{(j)} \in f(V)$ and $\lambda_j c_k^{(j)} \geq 0$, $k \in [2^n]$, is clear. As $y \in Q_E\big|_y$ was chosen arbitrarily, we are left to show (4.21) to complete the proof of the theorem.

First we fix an order of $z_k^{(j)}$, $k \in [2^n]$. This order is lexicographic, i.e. the $z_k^{(j)}$ are ordered depending on the value of $[z_k^{(j)}]_1$ to $[z_k^{(j)}]_n$. For each $i \in [n]$, we first have all $z_k^{(j)}$ such that $[z_k^{(j)}]_i = l([v_j]_i)$ and then all $z_k^{(j)}$ such that $[z_k^{(j)}]_i = u([v_j]_i)$. We execute this order, starting from $i = 1$ and ending at $i = n$. Of course, the $c_k^{(j)}$ are ordered correspondingly, such that they are ordered likewise lexicographicly by the factors $\gamma_i^{(k)}$. Here, first come those $c_k^{(j)}$ that start with factors $\gamma_i^{(k)} = 1 - c_i$ and afterwards those with $\gamma_i^{(k)} = c_i$. In fact, each $z_k^{(j)}$ can be represented by a binary number $b = b_1 b_2 \ldots b_n$ such that our order corresponds to the order of the natural numbers, i.e. $1, 2, \ldots, n$. The binary numbers are written with leading zeros and defined by

$$b_i := \begin{cases} 0, & \text{if } [z_k^{(j)}]_i = l([v_j]_i) \\ 1, & \text{if } [z_k^{(j)}]_i = u([v_j]_i) \end{cases} \quad \text{for } i \in [n].$$

Using this order, it holds for $l = 0, \ldots, n-2$:

$$\begin{aligned} &\sum_{k=1}^{2^{n-l}} \prod_{i=l+1}^{n} \gamma_i^{(k)} \\ &= \sum_{k=1}^{2^{n-l-1}} \prod_{i=l+1}^{n} \gamma_i^{(k)} + \sum_{k=2^{n-l-1}+1}^{2^{n-l}} \prod_{i=l+1}^{n} \gamma_i^{(k)} \\ &= \sum_{k=1}^{2^{n-l-1}} (1 - c_{l+1}) \prod_{i=l+2}^{n} \gamma_i^{(k)} + \sum_{k=2^{n-l-1}+1}^{2^{n-l}} c_{l+1} \prod_{i=l+2}^{n} \gamma_i^{(k)} \\ &= (1 - c_{l+1}) \sum_{k=1}^{2^{n-l-1}} \prod_{i=l+2}^{n} \gamma_i^{(k)} + c_{l+1} \sum_{k=2^{n-l-1}+1}^{2^{n-l}} \prod_{i=l+2}^{n} \gamma_i^{(k)} \end{aligned}$$

$$= \sum_{k=1}^{2^{n-l-1}} \prod_{i=l+2}^{n} \gamma_i^{(k)}$$

The last equation holds due to the fact that $c_{l+1} \in [0,1]$ and that

$$\sum_{k=1}^{2^{n-l-1}} \prod_{i=l+2}^{n} \gamma_i^{(k)} = \sum_{k=2^{n-l-1}+1}^{2^{n-l}} \prod_{i=l+2}^{n} \gamma_i^{(k)},$$

which is a consequence of our fixed order. Indeed, in the product, index i runs from $l+2$ to n and it holds $\gamma_i^{(k)} = \gamma_i^{(k+2^{n-l-1})}$ for $k \in [2^{n-l-1}]$.

By using all the equalities from $l = 0$ to $l = n-2$ we conclude that

$$\sum_{k=1}^{2^n} c_k^{(j)} = \sum_{k=1}^{2^n} \prod_{i=1}^{n} \gamma_i^{(k)} = \sum_{k=1}^{2} \gamma_n^{(k)} = (1-c_n) + c_n = 1.$$

Subsequently we see the second part of (4.21), i.e.

$$\sum_{j=1}^{m} \sum_{k=1}^{2^n} \lambda_j c_k^{(j)} = \sum_{j=1}^{m} \lambda_j \sum_{k=1}^{2^n} c_k^{(j)} = \sum_{j=1}^{m} \lambda_j = 1.$$

To prove the first part of (4.21), it is left to show for $i \in [n]$ that

$$(1-c_i) l([v_j]_i) + c_i u([v_j]_i) = \sum_{k=1}^{2^n} c_k^{(j)} [z_k^{(j)}]_i, \tag{4.22}$$

which can be combined with (4.20) to obtain (4.21). First we regard the case $i = 1$. According to our order it holds

$$\begin{aligned}
\sum_{k=1}^{2^n} c_k^{(j)} [z_k^{(j)}]_1 &= \sum_{k=1}^{2^{n-1}} c_k^{(j)} [z_k^{(j)}]_1 + \sum_{k=2^{n-1}+1}^{2^n} c_k^{(j)} [z_k^{(j)}]_1 \\
&= \sum_{k=1}^{2^{n-1}} c_k^{(j)} l([v_j]_1) + \sum_{k=2^{n-1}+1}^{2^n} c_k^{(j)} u([v_j]_1) \\
&= l([v_j]_1)(1-c_1) \sum_{k=1}^{2^{n-1}} \prod_{d=2}^{n} \gamma_d^{(k)} + u([v_j]_1) c_1 \sum_{k=2^{n-1}+1}^{2^n} \prod_{d=2}^{n} \gamma_d^{(k)}
\end{aligned}$$

Moreover, our order implies that

$$\alpha := \sum_{k=1}^{2^{n-1}} \prod_{d=2}^{n} \gamma_d^{(k)} = \sum_{k=2^{n-1}+1}^{2^n} \prod_{d=2}^{n} \gamma_d^{(k)},$$

so that we find

$$\sum_{k=1}^{2^n} c_k^{(j)} [z_k^{(j)}]_1 = l([v_j]_1)(1-c_1)\alpha + u([v_j]_1) c_1 \alpha. \tag{4.23}$$

Using $c_1 \in [0,1]$, we see that

$$\begin{aligned}\alpha &= (1-c_1)\alpha + c_1\alpha \\ &= (1-c_1)\sum_{k=1}^{2^{n-1}}\prod_{d=2}^{n}\gamma_d^{(k)} + c_1\sum_{k=2^{n-1}+1}^{2^n}\prod_{d=2}^{n}\gamma_d^{(k)} \\ &= \sum_{k=1}^{2^{n-1}}\prod_{d=1}^{n}\gamma_d^{(k)} + \sum_{k=2^{n-1}+1}^{2^n}\prod_{d=1}^{n}\gamma_d^{(k)} \\ &= \sum_{k=1}^{2^n}\prod_{d=1}^{n}\gamma_d^{(k)} = 1.\end{aligned}$$

Now we can substitute α by 1 in (4.23) and obtain (4.22). If the fixed order is changed accordingly, the argument serves to prove (4.22) analogously for any index $i \in [n] \setminus \{1\}$. More exactly, we must start to order $z_k^{(j)}$ and $c_k^{(j)}$ by component $[z_k^{(j)}]_i$ rather than component $[z_k^{(j)}]_1$. After that, the order can be performed as before running from the first to the last component, except i, which is used in the first place. In fact, there is no need to actually change the order. The sum on the right hand side of (4.22) only has to be splitted such that the factors $(1-c_i)$ and c_i can be extracted separately. Although, this is difficult to denote since 2^i sums are required to separate all $c_k^{(j)}$ containing $1-c_i$ or c_i, respectively. However, this splitting can be performed for all components $i \in [n]$ of $z_k^{(j)}$, which establishes the correctness of (4.22) for all $i \in [n]$. Finally, we combine (4.22) with (4.20) which results in

$$y_i = \sum_{j=1}^{m}\lambda_j\sum_{k=1}^{2^n}c_k^{(j)}[z_k^{(j)}]_i.$$

Thus we have shown (4.21) componentwise which concludes the proof. □

Now we apply this theorem to compute the approximation (4.4) of Ehlers [19] for our example neural network as depicted in Figure 4.2. Our goal is to find a good upper bound for the output o_1 of the network. For example, this would be useful to prove or disprove a property like $o_1 \leq c$ for some constant $c \in \mathbb{R}$. We assume $a_1 \in [-3,2]$ and $a_2 \in [-5,4]$ as feasible domains for the input values of the network. The weights and biases of the network imply that $x_1 = a_1 + a_2$ and $x_2 = a_1 - a_2 - 1$. Figure 4.6 shows the feasible input polytope of the ReLU layer and the corresponding ReLU image. The same ReLU image can be seen in Figure 4.7, replenished with a depiction of its convex hull, approximation (4.4) and the naive approximation.

Figure 4.7 clearly shows that even for only two variables the convex hull of the ReLU image is strictly smaller compared to approximation (4.4). The naive approximation using only the upper bounds of the variables is even worse, as can also be seen in Figure 4.7. Indeed, the difference between the naive "box" approximation and approximation (4.4) makes a big difference when it comes to the capacity of solving the verification problem for instances of relevant size. Especially, the quality of the bounds in previous layers heavily

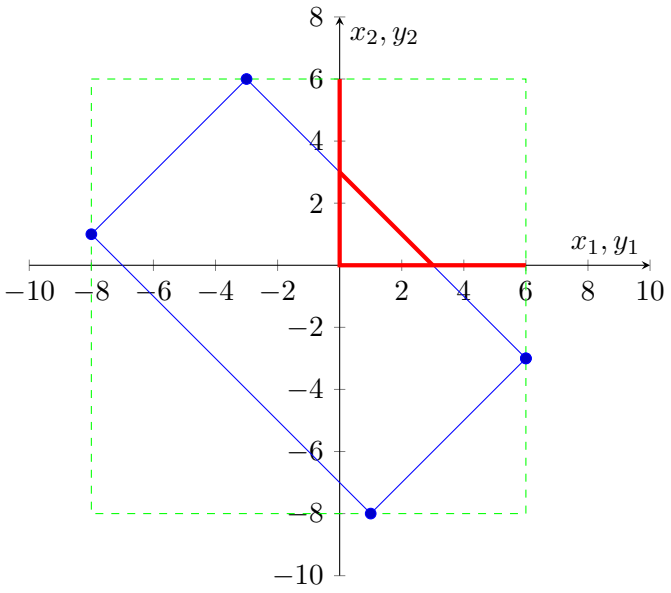

Figure 4.6: Feasible set before and after application of the ReLU function in the example neural network shown in Figure 4.2. The blue polytope is the set of feasible inputs to the ReLU layer, while the set which is enclosed by red lines shows the corresponding ReLU image. Moreover, the green dashed lines indicate the maximum and minimum values for both ReLU input variables x_1 and x_2.

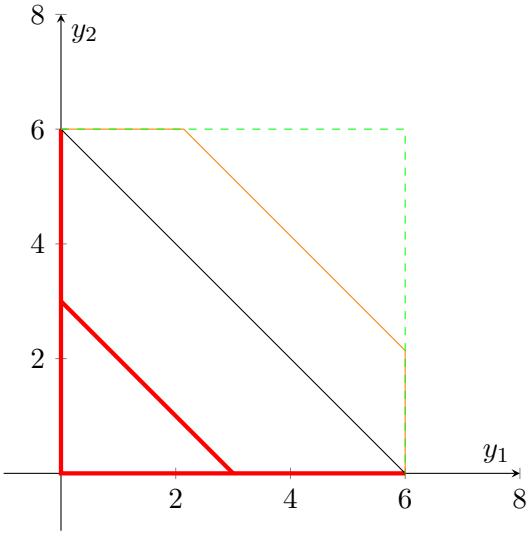

Figure 4.7: ReLU image and different convex approximations of it for the neural network shown in Figure 4.2. The red lines enclose the ReLU image and the black line (with the coordinate axes) indicates the convex hull of this ReLU image. Approximation (4.4) of the ReLU image is depicted by the orange segments and the coordinate axes, while the naive approximation is the area between the green lines and the coordinate axes.

impacts the quality of the bounds in layers behind. Hence it seems appealing to find an improved approximation of the ReLU image closer to the convex hull, which is the best possible convex approximation.

4.6 Negative approximation results

Theorem 34 shows how the convex hull of the ReLU image of a polytope can be computed explicitly. Though, this approach requires the enumeration of exponentially many vertices. The same convex hull can also be computed based on Theorem 33, which describes the ReLU image as a union of polytopes. Balas [4] shows that the convex hull of a union of polytopes can easily be stated explicitly. Moreover, Jones et al. [33] present a projection algorithm that could be applicable for the computation of the polytope-orthant intersections as implied by Lemma 32. Unfortunately, the number of polytopes in the union grows exponentially with the number of neurons in the corresponding layer. In general, for each orthant there can be one polytope in the union, i.e. the union contains up to 2^n polytopes where n is the number of neurons in the layer. Therefore a direct application of this method is practically impossible, unless the number of polytopes in the union can be significantly reduced.

If the dimension of the input is low, e.g. ≤ 10, the computations could be tractable. The affine transformations between the layers of a neural network do not increase this dimension, also if they map to a higher dimensional space. Though, we prove that the ReLU image of a low dimensional polytope in a high dimensional space may in general have the full dimension of the space. Hence, the low input dimension can probably not be maintained when the polytopes are propagated through the layers.

Theorem 36. *Let $n \in \mathbb{N}$. Then there is a polytope $P \subset \mathbb{R}^n$ of dimension one, such that the ReLU image* $\mathrm{ReLU}(P)$ *has dimension n. Especially, the convex hull of* $\mathrm{ReLU}(P)$ *also has dimension n.*

Proof. We define

$$P := \left\{ \lambda \begin{bmatrix} -1 \\ -2 \\ \vdots \\ -n \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \;\middle|\; \lambda \in [0, 2] \right\}$$

and show that it proves the theorem. Obviously P has dimension one. We choose $n + 1$ vectors in P and show that they form a set of $n + 1$ affinely independent vectors after applying the ReLU function to each of them. This implies that $\mathrm{ReLU}(P)$ has dimension n and its convex hull, too. We choose the following $n + 1$ vectors, which are contained in P:

λ	1	$\frac{1}{2}$	$\frac{1}{3}$	$\cdots$	$\frac{1}{n-1}$	$\frac{1}{n}$	0
	$\begin{bmatrix} 0 \\ -1 \\ \vdots \\ 2-n \\ 1-n \end{bmatrix}$	$\begin{bmatrix} \frac{1}{2} \\ 0 \\ \vdots \\ \frac{3-n}{2} \\ \frac{2-n}{2} \end{bmatrix}$	$\begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \\ \vdots \\ \frac{4-n}{3} \\ \frac{3-n}{3} \end{bmatrix}$	$\cdots$	$\begin{bmatrix} \frac{n-2}{n-1} \\ \frac{n-3}{n-1} \\ \vdots \\ 0 \\ -\frac{1}{n-1} \end{bmatrix}$	$\begin{bmatrix} \frac{n-1}{n} \\ \frac{n-2}{n} \\ \vdots \\ \frac{1}{n} \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}$

Now we apply the ReLU function to all these vectors and obtain the following set of vectors:

λ	1	$\frac{1}{2}$	$\frac{1}{3}$	$\cdots$	$\frac{1}{n-1}$	$\frac{1}{n}$	0
	$\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} \frac{1}{2} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\cdots$	$\begin{bmatrix} \frac{n-2}{n-1} \\ \frac{n-3}{n-1} \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} \frac{n-1}{n} \\ \frac{n-2}{n} \\ \vdots \\ \frac{1}{n} \\ 0 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}$

A set of $n+1$ vectors is affinely independent, if we subtract one vector from all the others and the remaining n vectors are linearly independent (cf. Bertsimas and Weismantel [6], Proposition A.1). We see that $0 \in \mathrm{ReLU}(P)$ is one of our vectors, so we can subtract it from the others without any computation. The other vectors form an upper triangular matrix with strictly positive diagonal. Hence this matrix is invertible and the n column vectors are linearly independent. Subsequently, the $n+1$ vectors (including 0 again) are affinely independent, which we wanted to show. □

Our discussion shows that in general it is apparently difficult or impossible to compute the convex hull of the ReLU image $\mathrm{ReLU}(P)$ for a polytope P. On the other hand, we will now see that the convex hull of $\mathrm{ReLU}(P)$ can still give very bad bounds when it comes to approximating the value of neurons in a network. Indeed, we can give neither a multiplicative nor an additive approximation guarantee for the quality of the bounds in a neural network which are computed by a convex approximation. The approximation of $\mathrm{ReLU}(P)$ by $\mathrm{conv}(\mathrm{ReLU}(P))$ for only one layer is suffient for this negative result.

Consider a neural network $F : \mathbb{R}^n \to \mathbb{R}$ with ReLU activation function as in Definition 1. The restriction to a neural network with output dimension one only serves to present the result more concisely. Clearly, it can also be

applied to neural networks with higher output dimension. Let $X \subset \mathbb{R}^n$ be a polytope of feasible input values for this network. We can bound the output of the neural network by $U \in \mathbb{R}$, such that it holds $F(x) \leq U$ for all $x \in X$, i.e. $F(X) \subseteq]-\infty, U]$. Assume that the bound U is tight, i.e. for all $\epsilon > 0$ there exists $x \in X$ with $F(x) \geq U - \epsilon$. The exact output range $F(X)$ can be computed by propagating the input polytope X through the network. In fact, Theorem 33 shows that the ReLU image of a polytope is a union of polytopes to which we could again apply the ReLU function. However, this is computationally infeasible as the number of polytopes grows exponentially. To alleviate this effect, we assume that after the application of the ReLU function, the resulting union of polytopes is replaced by its convex hull. If this convex hull is then propagated further through the network, we do not obtain the exact output range $F(X)$, but an approximation $A \supseteq F(X)$ of it. The union of polytopes can be replaced by its convex hull after each ReLU application or only after some of the layers. In the latter case, we have to deal with more polytopes but on the other hand the approximation is better. Eventually, $\tilde{U} := \sup A$ is an upper bound for the output range of the neural network on the input polytope X. We will now show that we cannot bound the value of $\tilde{U}$ compared to the tight bound value U. For that it suffices that for at least one layer the ReLU image is approximated by its convex hull, before it is propagated further through the network. Under this assumption and based on the previous explanations, we state the following theorem.

Theorem 37. *A multiplicative or additive approximation guarantee for $\tilde{U}$, based on the value of U, cannot be given.*

Proof. We shall see that the neural network which is depicted in Figure 4.8 shows this result. This network has input and output dimension one, and especially it has constant output $0 \in \mathbb{R}$ for any input $a_1 \in \mathbb{R}$. Subsequently, we have a tight upper bound $U = 0$. Because the input dimension is one, any polytope $X \subset \mathbb{R}$ of feasible input values for this network is indeed an interval. Any convex approximation of the first hidden layer produces bounds in $\Omega(d)$ for the output neuron, where d is the length of the input interval. Hence, we can neither give a mutliplicative nor an additive approximation guarantee for the output of the network.

First we see that the network has constant output $0 \in \mathbb{R}$ by regarding input values $\lambda \geq 0$ and $\lambda < 0$ separately. The following table shows all neuron values as they appear within the forward propagation of input value $\lambda \in \mathbb{R}$.

a_1	x_1	x_2	y_1	y_2	x_3	x_4	y_3	y_4	o_1
$\lambda \geq 0$	λ	λ	$-\lambda$	0	$-\lambda$	0	0	0	0
$\lambda < 0$	λ	0	$-\lambda$	$-\lambda$	$-\lambda$	$-\lambda$	$-\lambda$	$-\lambda$	0

Now we consider the feasible output set of the first ReLU layer, i.e. the values of variables x_2 and y_2 for $\lambda \in \mathbb{R}$, and denote this set as R. Then we have

$$R = \{(\lambda, 0) \mid \lambda \geq 0\} \cup \{(0, -\lambda) \mid \lambda < 0\} = \{(a, b) \in \mathbb{R}^2_{\geq 0} \mid a = 0 \vee b = 0\}.$$

We assume that $\lambda \in [-c, c]$ for some $c > 0$, which implies that $(0, c)$ and $(c, 0)$ are contained in R. Subsequently, we find that $(\frac{c}{2}, \frac{c}{2})$, which is a convex combination of these two vectors, is an element of $\operatorname{conv}(R)$. A forward propagation of $x_2 = \frac{c}{2}$ and $y_2 = \frac{c}{2}$ through the network leads to the following result:

x_2	y_2	x_3	y_3	x_4	y_4	o_1
$\frac{c}{2}$	$\frac{c}{2}$	0	$\frac{c}{2}$	0	$\frac{c}{2}$	$\frac{c}{2}$

We see that we have an output value of $\frac{c}{2}$ which is in $\Omega(2c)$ as we claimed in the beginning. Especially, $\frac{c}{2}$ cannot be bounded by any multiplicative or additive guarantee which is based on the constant output value 0 of the network. Moreover, it should be noticed that for the second layer we did not use an approximation but computed the exact ReLU image. □

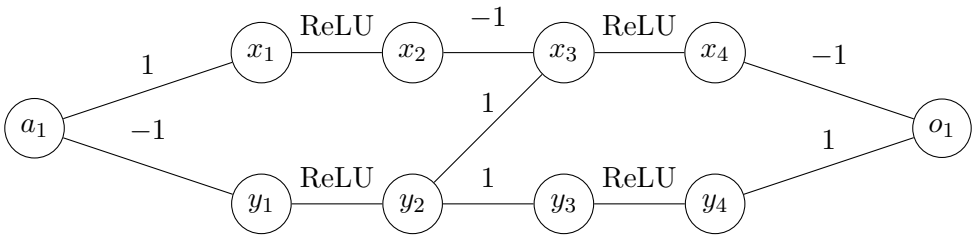

Figure 4.8: Neural network with constant output 0. All biases are 0, weights as denoted.

In spite of the theorem which we just presented, convex approximations of ReLU constraints are highly useful. Especially this holds true for the approximation (4.4) of Ehlers [19], which is applied succesfully by Bunel et al. [12] and in our solving model (see Chapter 9 for computational results). We use the following section to describe a method which improves this approximation.

4.7 Optimization based relaxation tightening for two variables

In this section, we propose an efficient method which can strengthen the convex ReLU approximation (4.4) by considering at least pairs of two neurons jointly. In contrast to the approximation (4.4), this ReLU approximation is not independent (cf. Definition 20). Because computing the projection of a polytope on two of its variables remains a problem, we circumvent this issue by solving a linear program which is sufficient for our purpose. We reconsider the example from before, i.e. the neural network in Figure 4.2. The depiction in Figure 4.7 shows, that in this situation we could actually add one inequality and would improve the approximation to be exactly the convex hull of the ReLU image. This inequality is induced by the connecting segment between the vertices of the convex hull that maximize y_1 or y_2, respectively. Of course, we cannot make this inequality tighter, since otherwise feasible points of the ReLU image would be cut off. Though, the segment between the vertices that maximize y_1 or y_2, respectively, does not always induce a valid inequality as we show in the following example. Figure 4.9 shows a polytope of feasible x_1, x_2 values and the corresponding ReLU image of feasible values for y_1 and y_2, such that $y_1 = \max\{0, x_1\}$ and $y_2 = \max\{0, x_2\}$. The polytope is two dimensional, but can also be considered as embedded image of a higher dimensional polytope

which is projected onto its variables x_1 and x_2. These two variables correspond to two neurons in one layer of a ReLU neural network. The dimension of the original polytope is then the number of all neurons in that layer. It should be noted that we use these projections to $\mathbb{R}^2$ only for the visualization of our method. The goal of our method is to obtain a tighter approximation without computing projections of higher dimensional polytopes. In Figure 4.9, the segment between the vertices that maximize y_1 or y_2, respectively, does not induce a valid inequality with respect to the ReLU image.

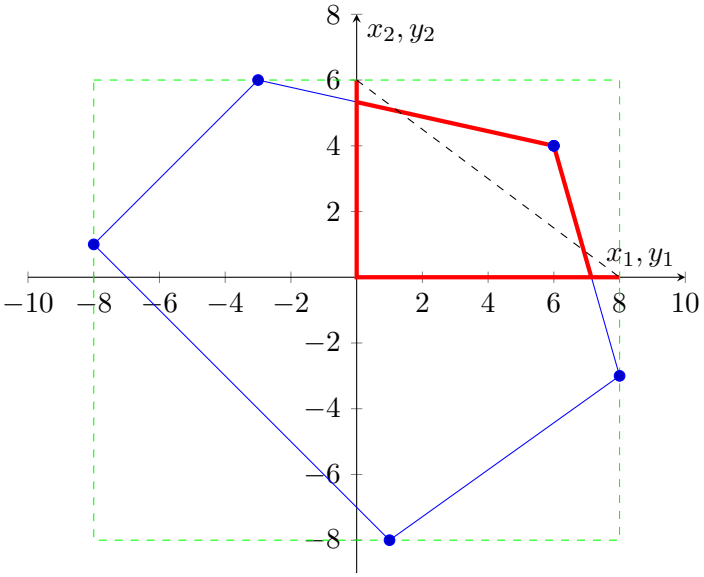

Figure 4.9: Feasible set before (blue) and after ReLU application (red) for a different input polytope than in Figure 4.7. Clearly, the connecting segment between the vertices that maximize y_1 or y_2, respectively, does not induce a valid inequality for the convex hull of the ReLU image.

Now the idea is to add an inequality to the model which partly cuts off the polytope resulting from the approximation (4.4), but leaves the ReLU image intact. The cut is parallel to the segment between the vertices that maximize y_1 or y_2, respectively. Depending on the situation, these vertices will either meet the inequality with equality or not. Figure 4.10 depicts this inequality and shows that adding this constraint considerably improves the approximation of the convex hull, compared to approximation (4.4). In the following we describe how this constraint can be computed. A linear approximation of the ReLU neural network in question serves as a basis. Naturally, we can use the LP relaxation for this purpose if the verification problem is formulated as an MIP. Remind that this is in fact identical to the linear approximation (4.4) of the network (see Theorem 18), if the basic MIP formulation (3.1) is used.

Assume we want to tighten the approximation for the ReLU output variables y_1 and y_2 which correspond to ReLU input variables x_1 and x_2. All of these variables are contained in the LP relaxation of the neural network. In the final solution it must hold $y_1 = \max\{0, x_1\}$ and $y_2 = \max\{0, x_2\}$ due to the ReLU constraints. Let $\hat{a}$ and $\hat{b}$ be the optimum solutions when maximizing x_1 or x_2, respectively, in the current LP relaxation. Then we write

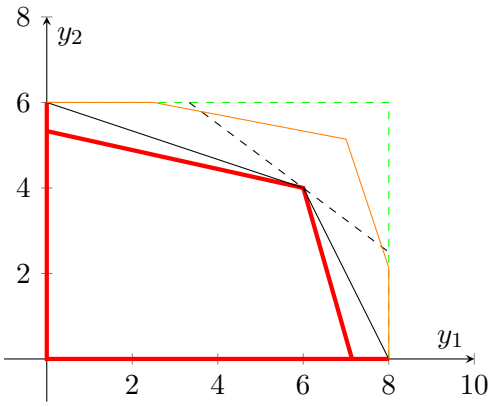

Figure 4.10: Here we see the ReLU image of the polytope depicted in Figure 4.9 colored in red, its convex hull in black, approximation (4.4) in orange, the inequality which we want to introduce as a black dashed line and the constraints of the naive approximation as a green dashed line. All sets are limited by the coordinate axes.

$\hat{a}_1$ and $\hat{a}_2$ for the values of the variables x_1 and x_2 in the solution $\hat{a}$. Analogously we write $\hat{b}_1$ and $\hat{b}_2$ for the corresponding variable values in solution $\hat{b}$. It should be noted that these LP solutions are computed during the execution of OBBT, and can therefore be obtained at no additional cost. Obviously it holds $\hat{a}_1 \geq \hat{b}_1$ and $\hat{b}_2 \geq \hat{a}_2$ due to the choice of objective functions. Now we define $a_1 := \max\{0, \hat{a}_1\}$ and analogously a_2, b_1 and b_2. For a visualization of these points see Figures 4.11, 4.12, 4.13 and 4.14, which show different situations that may occur. We compute new objective coefficients as $c_1 := b_2 - a_2$ and $c_2 := a_1 - b_1$, i.e. $c_1,\ c_2 \geq 0$. The latter holds due to the fact that $\alpha \geq \beta$ implies $\max\{0, \alpha\} \geq \max\{0, \beta\}$ for $\alpha, \beta \in \mathbb{R}$. Again, we solve an LP using the current relaxation and maximize the objective function $c_1 x_1 + c_2 x_2$. We denote the optimum objective value as γ and compute $\delta := c_1 a_1 + c_2 a_2$. It can be noticed that

$$\delta = c_1 a_1 + c_2 a_2 = (b_2 - a_2)a_1 + (a_1 - b_1)a_2 = b_2 a_1 - b_1 a_2,$$

hence it holds

$$\delta = b_2 a_1 - b_1 a_2 = (b_2 - a_2)b_1 + (a_1 - b1)b_2 = c_1 b_1 + c_2 b_2.$$

Indeed, we determined the objective to be orthogonal to the segment between the vertices (a_1, a_2) and (b_1, b_2). After this computation we can strengthen the LP relaxation by adding the constraint

$$c_1 y_1 + c_2 y_2 \leq \max\{\gamma, \delta\}. \tag{4.24}$$

Theorem 38. *Constraint (4.24) is a valid inequality with respect to the ReLU image corresponding to y_1 and y_2. That means, constraint (4.24) can strengthen the LP relaxation of our MIP for the verification problem but cannot cut off any feasible solution.*

Proof. We remind that it holds $y_1 = \max\{0, x_1\}$, $y_2 = \max\{0, x_2\}$ due to the ReLU constraints, and $a_1, a_2, b_1, b_2, c_1, c_2 \geq 0$, hence $\delta \geq 0$. That means, if $(y_1, y_2) = (0, 0)$ we have $c_1 y_1 + c_2 y_2 = 0 \leq \delta$. If $(y_1, y_2) = (x_1, 0)$ it holds $x_1 \leq a_1$ and hence $c_1 y_1 + c_2 y_2 = c_1 x_1 + 0 \leq c_1 a_1 + c_2 a_2 = \delta$. On the other hand, the case $(y_1, y_2) = (0, x_2)$ implies $x_2 \leq b_2$ and subsequently we see $c_1 y_1 + c_2 y_2 = 0 + c_2 x_2 \leq c_1 b_1 + c_2 b_2 = \delta$. Otherwise it holds $(y_1, y_2) = (x_1, x_2)$ which implies $c_1 y_1 + c_2 y_2 \leq \gamma$ and we can conclude the proof. □

Remark 39. Approximation (4.4) can be improved by adding constraints of type (4.24) to the LP relaxation of the model. According to Theorem 26, the improved approximation cannot be independent anymore. Indeed, if an inequality of type (4.24) is effective, i.e. it actually strengthens approximation (4.4), then c_1, $c_2 > 0$ in (4.24). Otherwise (4.24) reads $0 \leq 0$, $y_1 \leq a_1$ or $y_2 \leq b_2$ where the two latter hold by definition of a_1 and b_2. In view of Definition 20, c_1, $c_2 > 0$ implies that matrix B has at least one row which features two non-zero coefficients. In fact, there will be one such row for each pair of neurons for which an inequality of type (4.24) is added. Hence, by definition the corresponding ReLU approximation is not independent.

Although we have to solve only one LP per pair of neurons, applying this method to all possible pairs of neurons would lead to an immense computational cost. Therefore, we select only some pairs of neurons for which it is likely to significantly strengthen the LP relaxation by adding the new inequality to our model. It can be noticed that the tightness of approximation (4.4) depends strongly on the bounds $l < 0 < u$ for the corresponding neuron. On the other hand, if l and u are fixed, the output $y = \max\{0, x\}$ is approximated better or worse for different values $l \leq x \leq u$ of the ReLU input variable x. Indeed, the approximation is exact if $x = l$ or $x = u$ and the error is maximal for $x = 0$ as we will see in the following. Theorem 35 shows that these errors directly translate into the size of the polytope which results from applying approximation (4.4) to the ReLU constraints. Therefore, we want to find pairs of neurons, such that for both neurons the ReLU image is approximated rather badly. In these cases we can expect the most substantial improvements of the linear relaxation by our method, such that the cost of solving an additional LP is hopefully outweighed by the improvements which are reached due to the better LP relaxation.

We use Figures 4.11, 4.12, 4.13 and 4.14 to provide an overview of different situations where the error of approximation (4.4) is either low or high. All figures show a projection on variables x_1 and x_2 of the polytope before the application of the ReLU function in blue. Red lines indicate the ReLU image of this polytope, projected on variables y_1 and y_2, which are the corresponding ReLU output variables. Black solid lines mark the shape of approximation (4.4) which can be obtained by applying Theorem 35. A black dashed line shows the cut which we introduce with our method and the green dashed lines indicate the lower and upper bounds that can be found by OBBT for the (blue) input polytope. Besides, the intersection of the non-negative quadrant with the box which is limited by the green dashed lines, is the naive approximation of the ReLU image.

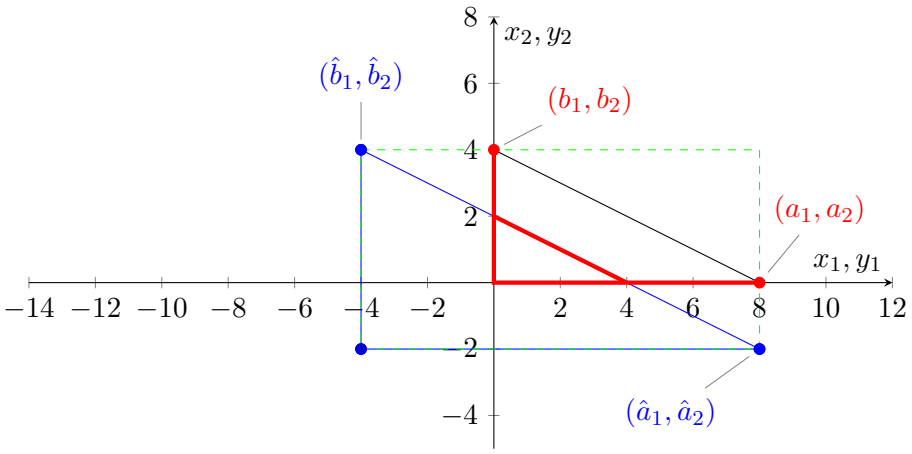

Figure 4.11: In this case approximation (4.4) is tight, i.e. it coincides with the convex hull of the ReLU image. Hence, it is not useful to apply our method in this case, as no improvement can be reached. Regarding inequality (4.24), it holds $\delta = \max\{\gamma, \delta\}$.

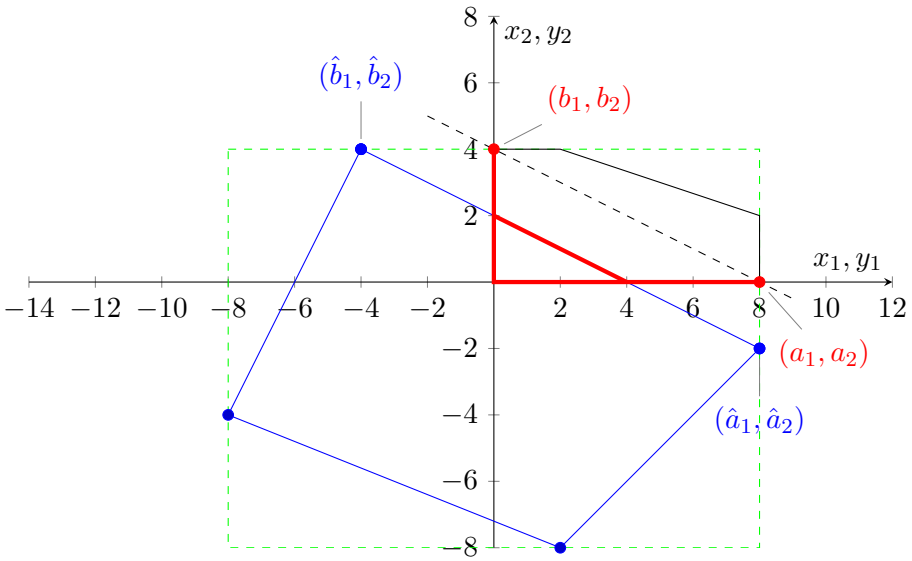

Figure 4.12: Here we see that a significant error is introduced by approximation (4.4), shown by the black lines. Adding inequality (4.24), where $\delta = \max\{\gamma, \delta\}$, as a cut reduces this error, as indicated by the black dashed line.

We regard the quality of ReLU approximations in two ways. On the one hand, we compute the maximum approximation error that may occur, if we know bounds $l < 0 < u$ for a neuron with ReLU activation. On the other hand, we can also include information about the value of the ReLU input variable x. In that case, we can compute the maximum approximation error given $l < 0 < u$ and given the value of x with $l \leq x \leq u$.

It is especially interesting, whether a (large) approximation error occurs when approximating $\text{ReLU}((\hat{a_1}, \hat{a_2}))$ and $\text{ReLU}((\hat{b_1}, \hat{b_2}))$, i.e. the ReLU

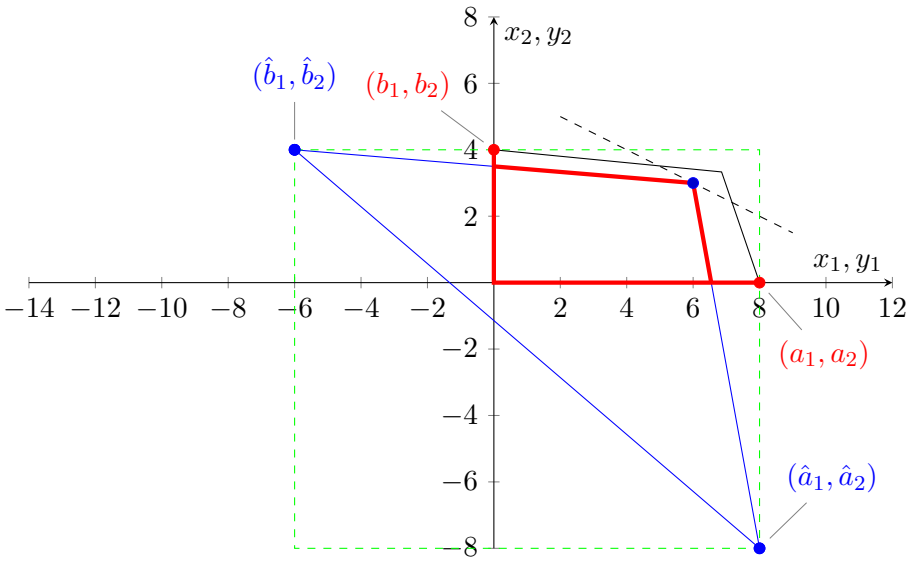

Figure 4.13: Here and in Figure 4.14 the maximum in inequality (4.24) is assumed at γ. Although approximation (4.4) is exact for the points $(\hat{a}_1, \hat{a}_2)$ and $(\hat{b}_1, \hat{b}_2)$, it does not reach the convex hull. However, adding inequality (4.24) as a cut does not seem to be very effective.

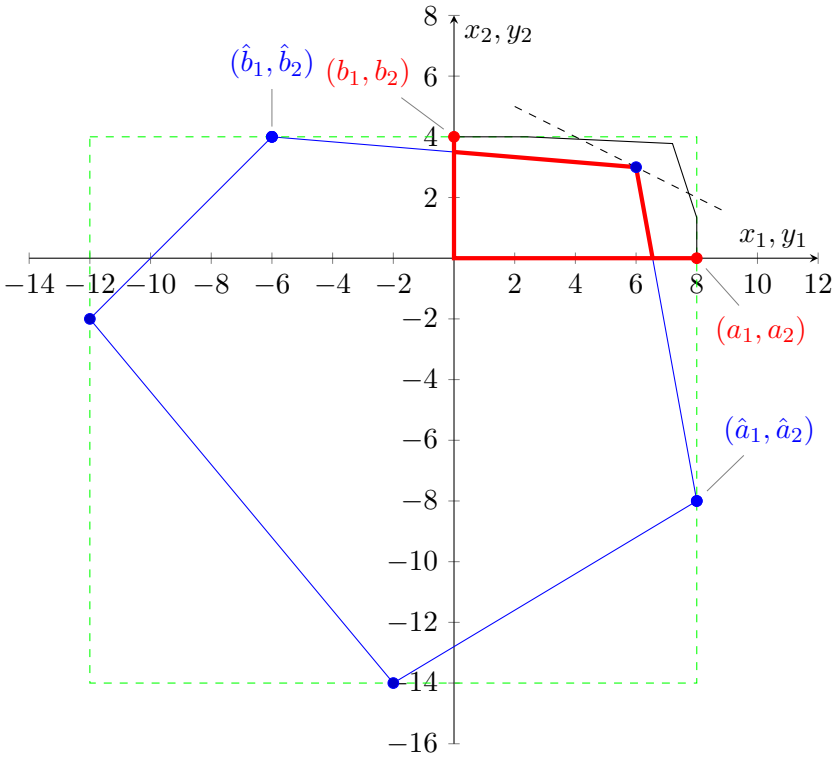

Figure 4.14: This situation is similar to the one in Figure 4.13, though here approximation (4.4) is not exact for $(\hat{a}_1, \hat{a}_2)$ and $(\hat{b}_1, \hat{b}_2)$. Adding inequality (4.24) as a cut seems to to be more effective in this case.

images of the OBBT solutions as laid out before Theorem 38. If the maximum is assumed at δ in constraint (4.24), this constraint is met with equality by (a_1, a_2) and (b_1, b_2), i.e. the exact values of $\mathrm{ReLU}((\hat{a_1}, \hat{a_2}))$ and $\mathrm{ReLU}((\hat{b_1}, \hat{b_2}))$. Subsequently, we can be sure to cut off some parts of the ReLU image approximation polytope, if the ReLU images of $(\hat{a_1}, \hat{a_2})$ and $(\hat{b_1}, \hat{b_2})$ are not approximated exactly and the maximum in (4.24) is attained at δ. This situation is depicted in Figure 4.12. Therefore, it seems reasonable to compute the error which can occur in approximation of $\mathrm{ReLU}((\hat{a_1}, \hat{a_2}))$ and $\mathrm{ReLU}((\hat{b_1}, \hat{b_2}))$. Since approximation (4.4) is exact if $x = l$ or $x = u$, it follows that $\mathrm{ReLU}(\hat{a_1})$ and $\mathrm{ReLU}(\hat{b_2})$ are approximated exactly, in contrast to $\mathrm{ReLU}(\hat{a_2})$ and $\mathrm{ReLU}(\hat{b_1})$. We will now see, how big the error in approximating $\mathrm{ReLU}(\hat{a_2})$ and $\mathrm{ReLU}(\hat{b_1})$ may become.

Analogously to the definitions in Theorem 35, we let $l_i := \min_{x \in P} x_i$ and $u_i := \max_{x \in P} x_i$ where P is the polytope corresponding to the current LP relaxation. Then, for $a \in [l_i, u_i]$, $i \in \{1, 2\}$, we set (as in Theorem 35)

$$l^{(i)}(a) := \begin{cases} a, & a > 0 \\ 0, & a \leq 0 \end{cases} \qquad \text{and} \qquad u^{(i)}(a) := (a - l_i)\frac{u_i}{u_i - l_i}.$$

Assume that $\bar{a_2}$ is a feasible approximation of $\mathrm{ReLU}(\hat{a_2})$ according to approximation (4.4) and $\bar{b_1}$ for $\mathrm{ReLU}(\hat{b_1})$. Then it holds $l^{(2)}(\hat{a_2}) \leq \bar{a_2} \leq u^{(2)}(\hat{a_2})$ and $l^{(1)}(\hat{b_1}) \leq \bar{b_1} \leq u^{(1)}(\hat{b_1})$. Hence, the approximation error is $u^{(2)}(\hat{a_2}) - l^{(2)}(\hat{a_2})$ or $u^{(1)}(\hat{b_1}) - l^{(1)}(\hat{b_1})$, respectively. To see this, it suffices to notice that $l^{(i)}(a) = \mathrm{ReLU}(a)$ for $i \in \{1, 2\}$ and $a \in [l_i, u_i]$, while the upper bounds $u^{(2)}(\hat{a_2})$ or $u^{(1)}(\hat{b_1})$ can be attained by the approximated values $\bar{a_2}$ or $\bar{b_1}$, respectively. An actual numeric value can be obtained by differentiating two cases. If $\hat{a_2} > 0$ we have

$$u^{(2)}(\hat{a_2}) - l^{(2)}(\hat{a_2}) = (\hat{a_2} - l_2)\frac{u_2}{u_2 - l_2} - \hat{a_2} \tag{4.25a}$$

and for $\hat{a_2} \leq 0$ it holds

$$u^{(2)}(\hat{a_2}) - l^{(2)}(\hat{a_2}) = (\hat{a_2} - l_2)\frac{u_2}{u_2 - l_2}. \tag{4.25b}$$

For $\hat{b_1}$ the values can be obtained analogously. The computation of these values is based on the concrete values of $\hat{a_2}$ and $\hat{b_1}$ with $l_2 \leq \hat{a_2} \leq u_2$ and $l_1 \leq \hat{b_1} \leq u_1$, respectively. In fact, $\hat{a_2}$ and $\hat{b_1}$ are the values of two ReLU input variables in a layer (in different LP solutions $\hat{a}$ and $\hat{b}$).

Now, we compute the maximum approximation error for known bounds $l < 0 < u$ but unknown value of the ReLU input variable x, which fulfills $l \leq x \leq u$. To this end, we differentiate whether x has a non-positive or non-negative value. Assume $x \leq 0$, which implies that $y = \mathrm{ReLU}(x) = 0$ while the approximation only imposes $0 \leq y \leq \frac{u(x-l)}{u-l}$. Thus the maximum error in this case is

$$\max_{x \in [l,0]} \frac{u(x-l)}{u-l} = \max_{x \in [l,0]} x\frac{u}{u-l} - \frac{lu}{u-l} = -\frac{lu}{u-l}.$$

The last equation holds, because $\frac{u}{u-l} > 0$ and thus the maximum is assumed for $x = 0$. Similarly, for $x \geq 0$ we see that $y = \mathrm{ReLU}(x) = x$ is the desired

equation, while the approximation gives $x \leq y \leq \frac{u(x-l)}{u-l}$. Here the maximum error is

$$\max_{x\in[0,u]} \frac{u(x-l)}{u-l} - x = \max_{x\in[0,u]} x(\frac{u}{u-l} - 1) - \frac{lu}{u-l} = -\frac{lu}{u-l}.$$

Analogously, the maximum is assumed for $x = 0$, since $\frac{u}{u-l} < 1$ and $x \geq 0$. Hence the maximum error is always $e_{\max} = -\frac{lu}{u-l}$ and occurs for $x = 0$.

Now assume that $c := u-l$ is fixed and we will see which values of $l < 0 < u$ maximize the maximum approximation error. To this end we notice that $u = c + l$ and define

$$f(l) := -\frac{lu}{u-l} = -\frac{l(c+l)}{c} = -\frac{l^2}{c} - l.$$

We can use the first and second derivative of f to show that it has a maximum at $\hat{l} = -\frac{c}{2}$. That means, the error is maximized for $u = \frac{c}{2}$ and $l = -\frac{c}{2}$. In other words, if the difference between l and u is fixed, the biggest approximation error may occur if $l < 0 < u$ are such that $|l| = |u|$. Furthermore, we see that

$$e_{\max} = -\frac{lu}{u-l} = \frac{\frac{c^2}{4}}{c} = \frac{c}{4},$$

i.e. the maximum approximation error grows linearly in the difference of u and l. This computation substantiates the intuition (cf. Figure 4.5) that the approximation (4.4) is rather good if one of the bound values is close to zero.

Based on this error analysis, we use the following concept for the selection of variable pairs in a ReLU layer. First, we compute the maximum approximation error $-\frac{lu}{u-l}$ for all neurons of the layer and sort the neurons in descending order of this error value. Then we consider the first $k \in \mathbb{N}$ neurons in this order for the application of our method. We fix n_1 as one of these neurons, for that we need to find another neuron n_2 from the same layer to combine it with. As before, we let x_1 and x_2 be the ReLU input variables corresponding to n_1 and n_2. Likewise, $\hat{a}$ and $\hat{b}$ denote the LP solutions when maximizing x_1 or x_2, respectively, and $\hat{a}_1$, $\hat{a}_2$, $\hat{b}_1$, $\hat{b}_2$ are the values of x_1 and x_2 in these LP solutions. Our idea is to select such pairs (n_1, n_2) that we maximize the sum of the maximum possible errors in approximation of $\mathrm{ReLU}(\hat{a_2})$ and $\mathrm{ReLU}(\hat{b_1})$, i.e. $u^{(2)}(\hat{a_2}) - l^{(2)}(\hat{a_2}) + u^{(1)}(\hat{b_1}) - l^{(1)}(\hat{b_1})$. This value can be computed as described by (4.25). In fact, for all combinations of n_1 with a neuron n_2 in the same layer, we compute this value. Then we choose those pairs (n_1, n_2) for which the value is among the $l \in \mathbb{N}$ highest of these values. For all of these pairs, we apply our optimization based relaxation tightening. We will refer to the method described in this section as OBBT2 and sometimes also specify values for k and l.

Unfortunately, it seems that our selection rule for the neuron pairs, based on the error analysis, is not successful. In our computational experiments, it does not perform better than the selection of neuron pairs based on a fixed order, which does not regard the quality of the approximations. Corresponding computational results can be found in Section 9.2.

5 Primal Heuristics

For the problem of neural network verification the use of primal heuristics lies in the quick falsification of incorrect properties. Surprisingly, even a trivial heuristic, which only performs random sampling within the set of feasible inputs, can often find counterexamples to incorrect properties quickly. On the other hand, finding such counterexamples can take an immense computation time if one only relies on the standard functionalities of an MIP solver like SCIP. In addition, primal solutions can be used to tighten neuron bounds in the rear parts of the neural network at hand. In this case, a good primal bound for the objective value of the MIP formulation as optimization problem (cf. Section 3.2) can be propagated backwards through the network and shrink the feasible domain of some neurons.

5.1 Random sampling heuristic

The idea of the random sampling heuristic is plain and simple: Given an instance $\Pi = (X, Y, F)$ of the verification problem, we randomly pick $x \in X$ and check whether $F(x) \in Y$. In case that $F(x) \notin Y$, we know that Π is refutable. Moreover, using the MIP formulation as optimization problem, the input vector x is also useful if it leads to a decrease of the primal bound, since this may help to tighten neuron bounds as noted above. Clearly, the primal bound only depends on the network outputs $F(x)$ and hence on $x \in X$.

In general it is not trivial to obtain $x \in X$, if $X \subset \mathbb{R}^n$ is an arbitrary polytope. However, as mentioned in Remark 6, many of the instances we regard feature a polytope X which is actually a box. In this case, we simply pick $x_i \in [l_i, u_i]$ uniformly at random for $i \in [n]$, where l_i, u_i are the bounds of X for each component. Otherwise, if X is not a box, we solve an LP to obtain $x \in X$ using a random objective function. We set $x := \arg\min\{c^T y \mid y \in X\}$ where $c \in [-1, 1]^n$ is drawn uniformly at random. It should be noticed, that for our instances as described in Section 8.2 it is $n = 5$, such that solving these LPs does not impose a big effort. For the input layer of the network we set $x_0 := x$, and then forward propagation is used to compute the values of all neurons in the network. After that, the values of the output neurons are checked for feasibility with respect to the linear properties that should be verified, i.e. it is inspected whether $F(x) \in Y$. If the verification problem is modelled as an optimization problem as described in Section 3.2, we compute the value of the objective variable t. If this value is below zero, the instance is shown to be refutable. Moreover, the optimization based approach allows to rank the solutions that are generated by random sampling. For $t > 0$, the corresponding input $\tilde{x}_0$ is "closer" to being a counterexample if it is smaller. This does not mean, that there exists an input vector close to $\tilde{x}_0$ regarding any norm, which is actually a counterexample. However, we suspect that there is a counterexample which activates most of the ReLU neurons in the same phases. We will therefore use the idea of minimizing t in the more elaborated

heuristic which we propose.

Bunel et al. [11] quite successfully apply random sampling to reduce the runtime on instances that admit counterexamples. However, they only use the simple sampling method, assuming that X is a box. The verification approach of Wang et al. [56] explicitly assumes that X is a box. In fact, they only check whether setting $x_i := \frac{u_i - l_i}{2}$ to obtain an input vector $x = (x_1, \ldots, x_n)^T \in X$ forms a counterexample for instance Π. The bounds l_i and u_i for x_i change according to a branching scheme though, so that throughout the branch-and-bound process various input vectors are checked.

5.2 LP based heuristic

We propose another heuristic that can be used in addition to the random sampling heuristic. It is based on the local search proposed by Dutta et al. [16], which is used for output range analysis of ReLU neural networks. Though, we omit the use of gradient information and fit the heuristic more naturally into the framework of MIP solving. The main idea is to fix all neurons in one of their phases, such that the optimization variant of neural network verification consists only in solving a linear program. More exactly, the heuristic works as follows.

Assume that we have an instance $\Pi = (X, Y, F)$ of the verification problem. We start with a feasible input $x_0 \in X$ for the neural network and use forward propagation to compute the values of all neurons in the network when applied to input vector x_0. Then, for each ReLU neuron, we fix the binary variable d in (3.1) to zero or one, corresponding to the phase of the neuron that is determined by propagating x_0 through the network. Furthermore, the binary variables in the formulation of the maximum function for objective variable t (3.4) are also fixed, such that $t = \max\{z_1, \ldots, z_k\}$. With all binary variables fixed, the MIP as described in Section 3.2 becomes an LP.

Dutta et al. [16] define the notion of a *locally active region*, which corresponds roughly to the feasible set of such an LP. In fact, the only difference is that we incorporate the output constraints as modelled by the z_i variables in (3.4). These are not regarded by Dutta et al. [16], as they focus on output range analysis without verification of special properties.

Our LP is minimized with respect to variable t as objective function. After the first minimization LP has been solved, we choose a ReLU input variable $\bar{x}$ (corresponding to one ReLU neuron) of value zero if possible. For this variable, we switch the fixed value of the corresponding binary variable $\bar{d}$ from zero to one or vice versa. Then we optimize again and obtain a new input vector $\hat{x}_0 \in X$ for the neural network. Clearly, this is given by the values of the corresponding variables in the LP. After that, we switch the fixing of another binary variable, whose corresponding ReLU input variable has value 0 in the solution. This process is iterated until we find a feasible counterexample, i.e. the optimal value of the LP is smaller than zero, or we reach a predefined iteration limit. In case that none of the ReLU input variables is equal to zero, we have to abort the procedure. Although, this case does not occur too often. Dutta et al. [16] similarly solve an LP which is defined on the locally active

region, though the objective function is defined by the gradient of the neural network at the current input vector.

It is easy to see that switching the fixings of the binary variables as described, can only reduce the objective value of the optimum LP solution. If the ReLU input variable x is zero, the corresponding output variable y must also be zero. This holds both for $d = 1$ and $d = 0$ in the set of constraints (3.1). Hence, if x is zero, the LP solution remains valid after switching the value of d. The optimum solution of the changed LP can thus not be worse, i.e. the objective value is equal to or lower than the previous one. In fact, often the objective value cannot be improved after switching the phase of a ReLU variable. Yet this is not a major problem, as the simplex algorithm can often prove optimality of the LP immediately in this case. Hence, there is not much time spent on such cases.

In the following we describe, how we combine our LP based heuristic with the random sampling heuristic. First we use the random sampling heuristic, as described in the first part of this chapter, to find an input vector $x_0 \in X$. The random sampling process and forward propagation are very fast, and therefore we try many (e.g. 1000) random inputs to find an input $x_0 \in X$. Out of all sampled input vectors, we select $x_0 \in X$ such that it corresponds to the lowest value of objective variable t. The hope is, that x_0 can be converted into an actual counterexample by computing a new input vector $\hat{x}_0$. This is given by the optimum LP solution after some ReLU phase switches as described. Instance Π is shown to be indeed refutable, if the value of t is below zero in this optimum LP solution. As mentioned in the beginning of this chapter, the heuristically found primal solution may also be useful, if the value of t is still positive. If it is low enough, it may strengthen the neuron bounds in the rear parts of the network. This happens as the value of t is in fact an upper bound of $\max\{z_1, \ldots, z_k\}$ in (3.4).

Of course, both heuristics can be applied several times throughout the solving process and not only in the beginning. Indeed, this is especially useful in combination with the input domain branching scheme which we describe in Section 6.1. In this case, the domain, from which random inputs are sampled, becomes smaller over time due to the branching. Applied in a branch-and-bound scheme, this helps to focus the random sampling process on smaller domains in which it could be possible to find a counterexample. We make use of this strategy in our experimental evaluation.

The LP based heuristic cannot necessarily find an input vector $\hat{x}_0 \in X$, which corresponds to a lower value of t than the start vector $x_0 \in X$, which was obtained by random sampling. Yet, our experimental evaluation shows that this does happen in many cases. In fact, the mean runtime on our evaluation set of SAT instances, as described in Section 8.4, drops from 330.1 to 71.7 seconds if our LP based heuristic is employed. Detailed results are provided in Table B.3 in Section B.2 of the appendix. On the other hand, the mean runtime on our evaluation set of UNSAT instances (cf. Section 8.4) increases only slightly from 915.3 to 943.7 seconds due to the application of our LP based heuristic. Table B.4 shows detailed results for this experiment.

6 Branching Methods for Neural Network Verification

The verification problem can be solved with a generic branch-and-bound approach as described by Bunel et al. [12]. An introduction into the concept of branch-and-bound can be found e.g. in Chapter 11 of Bertsimas and Weismantel [6]. Indeed, it is not necessary to formulate and solve the problem as an MIP, as it is sufficient to compute approximations of the neural network using one of the methods we presented in Chapter 4. However, the branching rules which we present for neural network verification, can be integrated into the MIP solving process if a suitable MIP solver like SCIP [25] is used. First we present a generic algorithm, that solves the verification problem with branching. In contrast to the generic branch-and-bound algorithm presented in Bunel et al. [12], we do not require that the verification problem is solved as an optimization problem. For the algorithm, remind that SAT refers to a refutable instance, while UNSAT refers to a verifiable instance.

```
Function is_verifiable(Π)
    r ← approximate(Π)
    if r = UNSAT then
        return True
    else if r = SAT then
        return False
    else
        Π_1, Π_2 ← split(Π)
        if is_verifiable(Π_1) = False then
            return False
        end
        if is_verifiable(Π_2) = False then
            return False
        end
        return True
    end
```

The verification problem can be solved by the recursive function `is_verifiable`. Applied to an instance $\Pi = (X, Y, F)$, the function returns "True", if the instance is verifiable, and "False" otherwise. In the latter case, a valid counterexample should also be provided, which we omit here for brevity. The function `is_verifiable` calls another function `approximate`, that computes neuron bounds using one of the approximation methods presented in Chapter 4. Then it checks, whether the approximation suffices to conclude that the instance is verifiable (UNSAT) or refutable (SAT). We mentioned how this can be done in the beginning of Chapter 4. In these cases, the function `is_verifiable` returns the corresponding result. Otherwise,

we split the current instance into two sub-instances for which the function `is_verifiable` is called again. In this chapter, we present two possibilities how the `split` function can perform the branching. If the verification problem is solved as an MIP (cf. Chapter 3), the behaviour of the function `is_verifiable` can be integrated into the branch-and-bound procedure of the MIP solver. Basically, a single execution of the function `is_verifiable` corresponds to the processing of one node in the branch-and-bound tree of the MIP solver. This processing comprises the following steps. First, tighter neuron bounds are computed (by the function `approximate`), using an approximation method as presented in Chapter 4. In SCIP, it is suitable to implement a propagator for this task. Then the "UNSAT" case in function `is_verifiable` corresponds to a node which can be cut off. If the "SAT" case occurs, a counterexample for the verification problem is found, and the computation can be terminated. Otherwise, branching can be used to create child nodes which can be approximated better to finally solve the problem. There are slight differences between the different MIP formulations of the verification problem as introduced in Sections 3.1 and 3.2.

Using the formulation as satisfiability problem, the "UNSAT" case in the function `is_verifiable` corresponds to an infeasible node. If a feasible solution is found at the current node, this corresponds to the "SAT" case. Similarly, if no feasible solution is found, and infeasibility cannot be proved, new child nodes are created by branching. On the other hand, if the formulation as optimization problem is used, the "UNSAT" case correponds to a dual (lower) bound which is greater than zero. Accordingly, the "SAT" case corresponds to a primal (upper) bound lower than zero. Otherwise, branching is applied to create child nodes. Our course of action differs slightly from the standard branch-and-bound approach for MIP solving, since we do not need to compute the actual optimum value of the MIP. Therefore, we can cut off a node whenever the dual bound is greater than zero. If an optimum solution should be computed, a node may only be cut off if the dual bound of the node is greater or equal to the global primal bound. However, if we find a primal solution with negative objective value, we have found a counterexample for the instance and can immediately stop the solving process. Before that and in general for all verifiable instances, the primal bound is always positive. Therefore we can cut off solving nodes earlier, if we only demand that the dual bound is greater than zero. In this case, it is certain that no counterexample can be found at this node, which would imply an objective value smaller than zero. It is possible that we cut off the optimal solution of a verifiable instance if we cut off a node where the dual bound is greater than zero, but smaller than the global primal bound. But this is not a problem with respect to the question whether the instance is verifiable or refutable. When all nodes are solved or cut off, and the value of the dual bound is positive, we know that the instance is verifiable. We refer to Section 3.2, especially Remark 10, for some details on the (non-)necessity of strictly positive or negative bounds.

In principle, we can use standard routines of an MIP solver for the whole computation, after we have computed initial neuron bounds using one of the approximation methods as laid out in Chapter 4. The initial bounds are necessary for both formulations of the verification problem as an MIP model.

We can improve the branch-and-bound procedure by cutting nodes off earlier, as described above. However, the initially computed bounds are often quite loose. Therefore, many relevant instances of the verification problem cannot be solved if an approximation of the network is computed only once. Clearly, this depends on the quality of the approximation. If OBBT is applied directly to the MIP model rather than the LP relaxation, as described in the end of Section 4.3, it may often be sufficient to compute this approximation just once in the beginning. If the neuron bounds are good enough, the MIP model can be solved by standard techniques of an MIP solver. Sometimes, even solving the linear relaxation may be sufficient to show whether the instance is verifiable or refutable. However, the application of OBBT on the MIP model is very time consuming and hence not the best choice in many cases. Corresponding computational results can be found in Table 9.6 in Section 9.2. Therefore it is often necessary to compute a better approximation after branching. In this chapter we present two branching rules specific to neural network verification. Using these branching rules, it is possible to compute tighter neuron bounds for the resulting sub-instances after branching. A well chosen combination of approximation methods and branching rules is crucial for solving relevant instances of the verification problem. This can be supported by using other MIP solving techniques such as cutting plane generation, domain propagation, and conflict analysis.

6.1 Input domain branching

Bunel et al. [12] propose a branching rule which splits the set of feasible input vectors for an instance of the verification problem. The same rule is also used by Wang et al. [57] for their solver Reluval and called "iterative interval refinement". Given an instance $\Pi = (X, Y, F)$ of the verification problem, the design of the branching rule is based on the assumption that X is a box. However, the branching rule can also be applied if X is not a box, although it might be less efficient in this case. We use our general assumption from Remark 6, that expresses the existence of bounds $l_i \leq x_i \leq u_i$ for all $x \in X \subset \mathbb{R}^n$ and $i \in [n]$. In case that X is a box, it holds $X = [l_1, u_1] \times \ldots \times [l_n, u_n]$ by definition of X. Otherwise, we have $X \subseteq [l_1, u_1] \times \ldots \times [l_n, u_n]$. Bunel et al. [12] propose to select $j \in [n]$ and split the domain of variable x_j to subdomains $[l_j, \frac{u_j - l_j}{2}]$ and $[\frac{u_j - l_j}{2}, u_j]$. The domains of all other variables x_i, $i \in [n] \backslash \{j\}$ are left unchanged and thus we obtain two instances $\Pi_{j1} = (X_{j1}, Y, F)$ and $\Pi_{j2} = (X_{j2}, Y, F)$ where

$$X_{j1} = X \cap \Big([l_1, u_1] \times \ldots \times [l_j, \tfrac{u_j - l_j}{2}] \times \ldots \times [l_n, u_n]\Big) \qquad \text{and}$$
$$X_{j2} = X \cap \Big([l_1, u_1] \times \ldots \times [\tfrac{u_j - l_j}{2}, u_j] \times \ldots \times [l_n, u_n]\Big).$$

If X is not a box, it may happen that $X_{j1} = \emptyset$ and $X_{j2} = X$ or vice versa. That means, we do not perform a sensible branching in this case. However, the computational experiments with our self defined instances, where X is not a box, indicate that this branching rule may still be useful for such instances. Corresponding results are reported in Sections 9.2 and 9.3.

Assuming $X_{j1}, X_{j2} \neq \emptyset$, we have thus two instances Π_{j1} and Π_{j2} of the verification problem with smaller input polytopes than Π. We know that Π is verifiable, if $F(X) \subseteq Y$. This holds, if both $F(X_{j1}) \subseteq Y$ and $F(X_{j2}) \subseteq Y$, i.e. if Π_{j1} and Π_{j2} are verifiable. However, using one of the approximation methods that we presented in Chapter 4, tighter neuron bounds can be computed for the instances Π_{j1} and Π_{j2}. Possibly, these tighter bounds suffice to conlude that these instances are verifiable as laid out in the beginning of Chapter 4. Otherwise, we can branch on these instances again. This can be continued, until the applied approximation method is able to prove verifiability for all instances at leaves of the branch-and-bound tree. Clearly, verifiability can also be proven by other means of an MIP solver, if the problem is formulated and solved as an MIP.

The selection of the branching variable is very important for the performance of the branching rule, as noted already by Bunel et al. [12]. The first proposal for a selection rule can be found in Bunel et al. [11], where the selection follows a fixed scheme. For that, an order of the input neurons, which correspond to the possible branching variables, is fixed. Let $N_0 \in \mathbb{N}$ be the number of input neurons, which we index from 0 to $N_0 - 1$. For a solving node at depth d of the branch-and-bound tree, the branching variable is determined as the $(d \mod N_0)$-th variable in the fixed order. We will refer to this selection rule with the name "standard". An advantage of this branching rule is that the chosen branching variables are evenly distributed over all input neurons. On the other hand, this rule does not use any information about the importance of the input neurons for the output of the neural network. Especially with respect to the properties that shall be verified, it can be more efficient to branch the domains of a few variables more often, and not evenly the domains of all variables.

Bunel et al. [12] implement a selection rule which is based on work of Wong and Kolter [59]. The idea is similar to the idea of strong branching for MIP solving. Wong and Kolter [59] propose a method to compute an approximation $A \supseteq F(X)$ of the network output for an instance $\Pi = (X, Y, F)$ of the verification problem. In fact, they use approximation (4.4) as introduced by Ehlers [19]. With that they obtain an LP which is a linear relaxation of the verification problem, formulated as optimization problem. Then they consider the dual of this LP, which they solve to compute a lower bound of the primal. According to Bunel et al. [12], this gives only a loose bound, but can be perfomed very fast. Hence, Bunel et al. [12] try all input neurons as branching variables, and choose the one which results in the best lower bound, according to the approximation method of Wong and Kolter [59].

For our implementation we mainly use a selection rule "gradient" which is quite similar to the one used in Wang et al. [57] and works as follows. Given an instance $\Pi = (X, Y, F)$, it is possible to extend the neural network represented by F to a neural network $\tilde{F}$. This network $\tilde{F}$ encodes also the properties which shall be verified. It has output dimension one, and for a fixed input $x \in X$, the output is the same as the value of the objective variable t in the MIP formulation as optimization problem (3.3). We use a max-pooling layer to model the computation of the maximum in (3.3) in the neural network $\tilde{F}$ and refer to Bunel et al. [12] for more details on the construction. As in

Wang et al. [57], we use gradient information for the selection of the branching variable. $X \subset \mathbb{R}^n$ is the feasible input domain of the instance Π, and we use the input bounds $l_i \leq x_i \leq u_i$ for $i \in [n]$ and $x \in X$. We compute the gradient of $\tilde{F}$ at the input vectors $x_1 = (l_1, \ldots, l_n)$, $x_2 = (\frac{u_1-l_1}{2}, \ldots, \frac{u_n-l_n}{2})$, and $x_3 = (u_1, \ldots, u_n)$. We let

$$g := \nabla\tilde{F}(x_1) + \nabla\tilde{F}(x_2) + \nabla\tilde{F}(x_3) \in \mathbb{R}^n$$

and for $i \in [n]$ we compute $z_i := |g_i| \cdot (u_i - l_i)$. Then we choose the branching variable $j \in [n]$ such that $z_j = \max\{z_1, \ldots, z_n\}$. The intuition is that verifiability of the instance depends mainly on the values of input neurons with a high (averaged) absolute gradient value. Like Wang et al. [57], we also include the size of the current domain into the decision for the branching variable. If the domain of a variable is very large, then branching on this domain may be more effective than branching on a variable with a small domain, although the corresponding absolute gradient value might be bigger. We use the absolute value of the gradient as we want to measure the impact of input changes on the network verifiability, but not in a special direction. See Table 9.6 in Section 9.2 for runtime results of our solving model in various configurations. In the corresponding computational experiments we also evaluate the performance of the domain branching rule that we explained here.

6.2 Branching on ReLU nodes

It is a natural possibility to perform branching over the two phases of a ReLU neuron. If the verification problem is formulated as an MIP, this corresponds exactly to branching over the binary variables which correspond to the ReLU neurons. The concept of this branching rule is already used by Katz et al. [35], Ehlers [19], and Cheng et al. [14]. Wang et al. [56] also apply it for their solver Neurify and call it "directed constraint refinement".

Given a ReLU input variable x and the corresponding output variable y, we can branch the constraint $y = \max\{0, x\}$ into two cases: either $x \leq 0$, $y = 0$, or $x > 0$, $y = x$. Regarding fomulation (3.1) of a ReLU constraint, which uses a binary variable d along with some linear constraints, these cases correspond to $d = 0$ or $d = 1$, respectively. Since our solving model is based on the formulation of the verification problem as an MIP, we can simply perform branching over the binary variables. Hence, the only question is how to select the branching variables.

In Ehlers [19] this decision is made by the underlying SAT solver and not specific to the problem of neural network verification. Katz et al. [35] adapt the simplex algorithm to create their solver Reluplex, which always maintains an assignment to all variables, which may violate some constraints, though. If several intents to resolve a conflict by pivoting of variables are unsuccessful, branching is performed. This procedure does not yield a selection rule which could be used within a MIP solver. Cheng et al. [14] propose to branch first on variables that are located in the front of the neural network, since these branching decisions immediately influence the ReLU phases in rear layers. Wang et al. [56] prioritize variables that belong to ReLU neurons which have a large gradient with respect to the outputs of the network.

We use the two latter ideas for the implementation of the branching rule in our model. Of course we do not branch on ReLU neurons whose phases are already fixed. In fact, if the phase of a ReLU neuron can be fixed, we do so by fixing the corresponding binary variable to zero or one, respectively. Using the idea of Cheng et al. [14], we implement a selection rule "standard". For that, we order all ReLU neurons layer by layer, starting at the input layer. Then, we consider all those binary variables which belong to ReLU neurons that are not yet fixed and branch on one after the other. Moreover, we implement a selection rule "gradient" that selects a branching variable out of the binary variables which are not fixed. We compute the gradients of the weights on all incoming edges of this neuron with respect to the outputs of the extended network $\tilde{F}$. This is the same extended network $\tilde{F}$ as introduced in Section 6.1 for the input domain branching rule. We compute the gradients at some of the input vectors which are generated during the execution of the primal heuristic as described in Chapter 5. Then we take the absolute value of the sum over the gradients at the incoming edges and select the variable with the highest value. However, our experiments show that this strategy does not work good, as it leads to the selection of many variables in rear layers. Therefore, the selection rule "standard", which selects branching variables from the front layers, performs better in our experiments. Computational results with respect to the performance of these selection rules can be found in Table 9.6 along with corresponding explanations in Section 9.2.

6.3 Realization in different solvers

In this section, we explain how the solving methods of Bunel et al. [12], Wang et al. [56], and Katz et al. [35] can be viewed as implementations of the generic algorithm which we presented in the beginning of this chapter. We will provide details on our own solving model in Chapter 7.

Bunel et al. [12] formulate the verification problem as an optimization problem analogously to the MIP formulation we presented in Section 3.2. Yet, the problem is not solved directly as an MIP. Bunel et al. [12] build an LP relaxation of the problem for which they use the linear approximation (4.4) as proposed by Ehlers [19]. Based on this LP relaxation, OBBT can be applied as described in Section 4.3. Bunel et al. [12] use the basic version of OBBT without regarding any of the additional ideas we laid out in Section 4.3 based on the work of Gleixner et al. [24]. Gurobi is employed as LP solver. As long as the lower bound, obtained by solving the linear relaxation, is not positive, input domain branching is applied. The selection of the branching variable is based on the procedure of Wong and Kolter [59] for robustness certification as laid out in Section 6.1. Additionally, random sampling over the input domain, as we described in Chapter 5, is used to find counterexamples. At each node of the branch-and-bound tree, Bunel et al. [12] randomly select 1024 vectors from the current input domain. Forward propagation is applied to check whether any of these vectors is in fact a feasible counterexample for the instance. The algorithm terminates, if either a counterexample is found by the random sampling heuristic, or the global lower bound of the problem

can be proved to be positive. Although Bunel et al. [12] solve the verification problem as an optimization problem which allows the processing of conjunction instances, this is not implemented by Bunel et al. [12]. Subsequently, conjunction instances must be splitted as described in Remark 7 to be solved by the implementation of Bunel et al. [12].

Wang et al. [56, 57] regard the verification problem as a satisfiability problem. They use symbolic upper and lower bounds for all neurons, combined with approximation (4.3). The functioning of their algorithm is similar to the approach of Bunel et al. [12]. They compute an approximation of the output domain of a neural network using their bound computation approach, which we described in detail in Section 4.1. As this is usually not sufficient to prove that an instance is verifiable, branching on ReLU nodes (see Section 6.2) is used to solve the problem in the solver Neurify [56]. For the solver ReluVal [57], which is designed to be used on the ACAS data set (cf. Section 8.1), input domain branching is used. This is based on the fact, that the ACAS Xu neural networks have only five inputs, which makes input domain branching quite efficient. It should be noted, that the current version of ReluVal also uses the bound computation approach with symbolic equations as presented in Wang et al. [56]. In both cases, gradient information is used for the selection of the branching variable. That means, gradients are computed for the interval bounds of each ReLU neuron (for ReLU branching) or each input neuron (for input domain branching). The larger of both gradients is multiplied with the interval width. Branching is then performed at the neuron for which the maximum value was computed. In order to find counterexamples, a technique similar to the random sampling heuristic described in Chapter 5 is used. However, at each branch-and-bound node, it is checked for only one vector whether it forms a counterexample for the instance. Therefore, finding counterexamples may take relatively long. Wang et al. [56] also implement an optional "adversary check mode" in ReluVal and Neurify. If this mode is used, the approximation of the neural network is not refined anymore at all nodes that have a depth of 20 or higher in the branch-and-bound tree. Instead, it is only checked at these nodes, whether a counterexample is found. To obtain a candidate vector for a counterexample, each component is set to the middle of the corresponding interval in the input domain. The input domain is split as in the case of input domain branching (cf. Section 6.1), such that different candidate vectors are obtained. ReluVal and Neurify are only able to work on instances $\Pi = (X, Y, F)$ of the verification problem where X is a box. Due to the bound computation approach, it is not possible to solve an instance Π where X is not a box. This is in constrast to the approaches of Bunel et al. [12] and Katz et al. [35]. In principle, X could be any polytope for these algorithms, but this is not implemented in the solvers. On the other hand, ReluVal and Neurify are able to process conjunction instances directly, whereas these need to be split as described in Remark 7 for the other solvers.

The solver Reluplex of Katz et al. [35] is based on an extended version of the simplex algorithm, which was introduced by Dantzig [15] for linear programming. Although, it still can be seen as an implementation of the generic branching algorithm we presented in the beginning of the chapter. In the algorithm of Katz et al. [35], each variable is assigned to some value

that may possibly violate variable bounds or ReLU constraints. Variables are either in the so called basis or non-basic. Basic variables can be represented as linear combinations of non-basic variables. These equations form the simplex tableau. Pivoting operations can be used so that a non-basic variables enters the basis and one of the basic variables leaves the basis. As in MIP formulations of the verification problem, each ReLU neuron is represented by two variables x and y. In any feasible solution, it must then hold $y = \max\{0, x\}$ for all ReLU neurons. As long as there is a pair (x, y) of such variables with $y \neq \max\{0, x\}$, there are two options in the Reluplex procedure. Either the value of one of the variables is changed or a split is introduced. Introducing a split corresponds to branching on a ReLU neuron. If either of the variables x and y is non-basic, its assigned value is updated so that $y = \max\{0, x\}$ holds. This implies that the assigned values of basic variables are also changed. However, this may result in broken ReLU constraints of other neurons or violated variable bounds. In case that both x and y are in the basis, one of these variables may be pivoted out of the basis, so that its value can be updated. A split is introduced for a ReLU pair (x, y), if the number of updates to the variables x and y exceeds a certain threshold. So the introduction of a new split is the last possible measure to resolve a broken ReLU equation. This shall prevent the creation of too many sub-instances that need to be solved. Furthermore, variable bounds can be tightened throughout the execution of the algorithm. Since basic variables are represented by a linear combination of non-basic variables in the simplex tableau, this connenction can be used to improve the bounds of basic variables. In some cases, this may help to fix the phase of ReLU pairs, which accelerates the solving process. If a feasible solution is found that fulfills all ReLU constraints and respects all variable bounds, the algorithm terminates. In the generic algorithm, this corresponds to the "SAT" case. There is no special heuristic for finding counterexamples. Otherwise, if the value of a basic variables violates its bounds and the variable cannot be pivoted out of the basis, the corresponding (sub-)instance is verifiable. This corresponds to the "UNSAT" case in the generic algorithm. Pivoting, updates of variable values, and bound tightenings can be seen as part of the `approximate` function in the generic algorithm. The splitting of ReLU pairs corresponds to the function `split`. Katz et al. [35] take the following strategy for the execution of the algorithm. First, violations of variable bounds are fixed, and after that violations of ReLU constraints are resolved. As the fixing of violated ReLU pairs may introduce new bound violations, this process is iterated. If a ReLU pair has already been updated or pivoted five times, this pair is splitted, i.e. branching is applied. For each variable that enters the basis, bound tightening is performed. Additional bound tightenings are performed after a fixed number of pivoting operations. As we mentioned before, Reluplex is only able to work with conjunction instances if these are splitted into disjunction instances as laid out in Remark 7.

7 Implementation Details

In this chapter we describe the implementation of our solving model for neural network verification. It is based on the academic and non-commercial MIP and MINLP solver SCIP [2]. In fact, SCIP is designed as a solver for *constraint integer programming* (CIP), which encompasses both MIP and MINLP. However, our use of SCIP is limited to its MIP capabilities. We access SCIP via its corresponding Python interface called PySCIPopt [39]. Our code is mostly written in Python 3.6, while some extensions are made to PySCIPopt using the Cython programming language. The implementation is publicly available at `https://github.com/roessig/verify-nn`. In the first section of this chapter, we give a short outline of the functioning of SCIP and its interplay with PySCIPopt. We limit our presentation to those components that are relevant for our solving model. After that, we provide some details regarding the implementation of our solving model in this framework. In the last part of this chapter, we explain the available settings of our solving model.

7.1 Structure of SCIP and PySCIPopt

The solver SCIP is embedded in the SCIP Optimization Suite [25]. Especially, this suite includes the LP solver SoPlex, which is the default LP solver used by SCIP. However, SCIP provides interfaces to other LP solvers, too. We use SCIP 6.0.1 and combine it with CPLEX 12.8.0.0 as LP solver. In many cases, most of the solving time in our model is spent on LP solves. Therefore, preliminary experiments showed significant reductions of solving times when CPLEX 12.8.0.0 was used instead of SoPlex in the current version 4.0.1.

SCIP uses a branch-and-bound approach to solve MIPs. This is supported by the computation of LP relaxations and the introduction of cutting planes. Several plugin types are defined that allow to include problem specific algorithms in the MIP solving process. For our solving model, we implement branching rules, a primal heuristic, domain propagators, an event handler, and a separator. In fact, all of these plugin are implemented via the Python interface PySCIPopt. As some of our methods need access to the public API of SCIP, these are directly implemented into PySCIPopt.

7.2 Implementation of the components

Our solving model is designed such that it allows a flexible combination of various techniques, which we described during the course of this thesis. Its structure is based on the generic algorithm that we presented in Chapter 6. Bound computation approaches, branching rules and further techniques can be combined in several ways. Moreover, a variety of settings allows to tune the behaviour of the various components.

The general approach can be laid out as follows. Based on the basic MIP

formulation of ReLU constraints (3.1), we model the verification problem as an MIP. Various options exist, how the upper and lower bounds used in (3.1) are computed. The verification problem is either formulated as a feasibility problem (see Section 3.1) or as an optimization problem (see Section 3.2). Major parts of our implementation are focused on the formulation as optimization problem, though.

We implement two branching rules, corresponding to the two specific branching rules for neural network verification as presented in Chapter 6. To these we will refer as (input) domain branching and ReLU branching. In the case of ReLU branching, we call the function `branchVar` in PySCIPopt for the binary variable d in formulation (3.1) corresponding to the selected ReLU neuron. This function performs branching on integer variables, so that SCIP automatically creates the corresponding nodes in the branch-and-bound tree. Furthermore, we explicitly tighten the bounds of the variables x and y in (3.1) at the child nodes corresponding to the branching. In the domain branching rule, we use the function `branchVarVal` in PySCIPopt to bisect the selected input interval. Of course, standard MIP branching can also be used instead of or in combination with the specialized branching rules. Depending on the instance which is solved, the default branching rules of SCIP may even outperform ReLU and domain branching.

SCIP features a so called diving mode which we extensively use for various routines in our solving model. This mode can be started during the solution process, and allows to change the current LP relaxation of the problem. The LP consists of LP rows of the form $l \leq c^T x \leq u$, where $c \in \mathbb{R}^n$ is a vector of coefficients, $x \in \mathbb{R}^n$ represents the variable values, and $l \in \mathbb{R} \cup \{-\infty\}$ and $u \in \mathbb{R} \cup \{+\infty\}$ are the left and right hand side. It is possible to change variable bounds, left and right hand side of LP rows, and the objective function. The LP can be repeatedly changed and solved again. After terminating the diving mode, the LP relaxation is set back to the state before starting the mode.

Especially, we use the diving mode for the execution of OBBT. As we need to solve many LPs during the execution of OBBT, we try to keep these LPs as small as possible. As pointed out in Section 4.3, we only need to include those variables in the LP, that correspond to neurons in layers before the neuron whose bounds shall be optimized. Using the diving mode of SCIP, we do this as follows. First, we relax all LP rows, which means that left and right hand side are set to infinite values such that they do not impose restrictions any longer. Then, all variables are fixed to zero, which effectively removes them from the LP. During the execution of the LP, the fixed variables are released one by one to their orginal domains. This activates the corresponding variable for the next LP solve. Additionally, we reset the left and right hand sides of those LP rows, where all variables with non-zero coefficients have been activated already. These functions are implemented as extensions of PySCIPopt since they require direct access to the LP data structures of SCIP. It should be noted that this approach can be used generally to solve the LP relaxation in SCIP only on a subset of variables, i.e. it is not limited to our application.

We implement two different propagators that are used to tighten variable bounds throughout the solving process. One of the propagators is used with the formulation of the verification problem as satisfiability problem, whereas

the other one corresponds to the formulation as optimization problem. Both propagators can work with several different bound computation approaches as laid out in Chapter 4. The propagators call the corresponding functions, e.g. for the execution of OBBT, and tighten the variable bounds. Our implementation also contains the bound computation approach proposed by Wang et al. [56], based on their approximation (4.3). However, the implementation of this routine is not optimised with respect to fast execution. In case of the formulation as optimization problem, the propagator is also responsible for the inclusion of Langrangian variable bounds (LVBs) as presented in Section 4.3. The generation of LVBs is oriented very closely at the implementation of the OBBT propagator in SCIP as described in Gleixner et al. [24]. As we cannot use the OBBT propagator of SCIP directly, we need access to the public API of SCIP for the creation of LVBs. Therefore, we implement the corresponding functions directly as part of PySCIPopt using Cython.

A primal heuristic is implemented and follows our description in Chapter 5. We provide settings that allow to use the heuristic only at the root node of the branch-and-bound tree, or frequently at many or all nodes of the tree. For the execution of the LP based heuristic, the diving mode of SCIP is used. It is also used for the generation of feasible input vectors for an instance $\Pi = (X, Y, F)$ of the verification problem where X is not a box (cf. Chapter 5).

In Section 3.4 we described how the linear relaxation of a ReLU neural network based on formulation (3.1) can be strengthened by additional cutting planes. These have been suggested by Anderson et al. [3] along with a separation routine, that helps to find useful cutting planes easily. Hence, using the interface PySCIPopt, we implement a separator which executes this separation routine and adds cutting planes if possible. SCIP provides various options which we use to control the behaviour of the separator, we explain these in the subsequent section.

Eventually, if the formulation of the verification problem as optimization problem is used, we include an event handler that interrupts the solving process if the global dual bound is greater than zero or the primal bound is lower than zero. As we described with respect to the generic algorithm in Chapter 6, the verification problem is already solved in these cases.

7.3 Parameter settings

In the following we provide an overview of the parameter settings that are available for our solving model and give short explanations. Not all parameters are always relevant. The parameters marked with * are only considered if the corresponding technique is activated.

Parameter	Default	Explanation
eps	5e-08	numerical tolerance for various comparisons
timelimit	10000	time limit in seconds; not used for our experiments
presolving_rounds	0	presolving is always disabled, i.e. value set to 0
use_opt_mode	True	True: formulation as optimization problem is used False: formulation as feasibility problem is used

Parameter	Default	Explanation
build_optimize_nodes	False	True: use OBBT (possibly on MIP) for computation of initial neuron bounds
build_use_symbolic	False	True: use bound computation of Wang et al. [56] for initial neuron bounds
use_linear_model	True	Only relevant if build_optimize_nodes is True. In that case, OBBT is applied on LP if True, or on MIP if False.
delete_linear_cons	True	Should redundant linear constraints be removed after MIP constraints are added?
sampling_heuristic_local_max_iter	-1000	number of iterations for sampling heuristic except at the root node, negative value to disable
sampling_heuristic_local_freq	1	frequency of heuristic application in branch-and-bound tree, cf. SCIP documentation
sampling_heuristic_local_maxdepth	8	maximum depth in the branch-and-bound tree, up to which heuristic is executed
sampling_heuristic_max_iter	1000	number of iterations for sampling heuristic at root node, negative value to disable
sampling_heuristic_freq	1	should be 1
sampling_heuristic_maxdepth	0	should be 0
sampling_heuristic_bound_for_lp_heur	100000.0	LP heuristic is only executed, if sampling heuristic found a solution with objective value smaller than this value (only if use_opt_mode is True)
sampling_heuristic_max_iter_lp_heur	1000	maximum number of LP solves in LP heuristic
sampling_heuristic_use_lp_sol_gen	False	Should be True if input domain of the instance is not a box, else False.
use_domain_branching	True	Should domain branching rule be used?
*domain_branching_split_mode	gradient	selection of branching variable, cf. Section 6.1, either "gradient" or "standard"
*domain_branching_priority	100000	priority of the branching rule, cf. SCIP documentation
*domain_branching_maxdepth	20	maximum depth in branch-and-bound tree up to which the branching rule is applied
*domain_branching_maxbounddist	1	float value between 0 and 1, cf. SCIP documentation, 1 means that branching rule is applied at each node
use_relu_branching	False	Should ReLU branching be used?
*relu_branching_split_mode	standard	selection of branching variable, cf. Section 6.2, either "gradient" or "standard"
*relu_branching_priority	100000	priority of the branching rule, cf. SCIP documentation
*relu_branching_maxdepth	10	maximum depth in branch-and-bound tree up to which the branching rule is applied
*relu_branching_maxbounddist	1	float value between 0 and 1, cf. SCIP documentation, 1 means that branching rule is applied at each node
use_obbt_propagator	True	Should our propagator be used? (OBBT is not necessarily executed.)
obbt_maxdepth	20	maximum depth in branch-and-bound tree up to which the propagator is executed
obbt_optimize_nodes	True	Should OBBT be applied? Needs use_obbt_propagator to be True so that OBBT is executed.
obbt_use_genvbounds	False	Should LVBs be generated? (needs that the parameters obbt_optimize_nodes and use_opt_mode are True)

Parameter	Default	Explanation
use_obbt_two_variables	False	Should OBBT2 be applied? (needs that the parameters obbt_optimize_nodes and use_obbt_propagator are True)
*obbt_k	10	value of k for OBBT2
*obbt_l	10	value of l for OBBT2
*obbt_sort	True	Should variable pairs for OBBT2 be selected according to the order based on our error analysis? (cf. Section 4.7)
obbt_use_symbolic	False	Should bound computation approach of Wang et al. [56] be applied in propagator?
obbt_bound_for_opt	-200	positive value: OBBT is only executed at neurons with bounds $l < 0 < u$, if $\|l\| + \|u\|$ smaller than the parameter value negative value: OBBT is always executed if bounds fulfill $l < 0 < u$
bfs_from_all_inputs	True	Should be True if not all neurons are reachable from each input neuron (i.e. via non-zero weights), else False.
use_ideal_separator	False	Should separator based on work of Anderson et al. [3] be used?
*sepa_freq	1	frequency of separator application in branch-and-bound tree, cf. SCIP documentation
*sepa_priority	100	priority of the separator, cf. SCIP documentation
*sepa_maxbounddist	0.0	float value between 0 and 1, cf. SCIP documentation, 0.0 means that the separator is only applied at the node with the best lower bound
*sepa_delay	False	Should the separator be delayed if other separators found cuts? (cf. SCIP documentation)

8 Description of Test Instances

In order to evaluate our model computationally and to compare it to other solvers for neural network verification, we need a set of reasonable test instances. As pointed out in Remark 6, research in the area of neural network verification has focused on instances $\Pi = (X, Y, F)$ where the input polytope X is in fact a box. Indeed, we are not aware of any publicly available instances of the verification problem where X is not a box. Therefore, we define such instances to show the capabilities of our solving model. As a basis we use the neural networks of the ACAS Xu system, which were used by Katz et al. [35] to create instances of the verification problem. The ACAS Xu system is designed to prevent collisions of (autonomous) aircrafts. We also perform our evaluations on the original instances published by Katz et al. [35]. However, all of these instances are very similar with respect to the network structure, as all of the neural networks have input and output dimension five and contain 300 ReLU neurons. Therefore, we also use test instances with neural networks that are trained on the well known MNIST handwritten digit data set (cf. LeCun et al. [36]). In fact, we use the neural networks as pusblished by Wang et al. [56] and verify robustness of classifications using the L_∞ norm. In contrast to the neural networks of the ACAS Xu system, the input dimension of the MNIST networks is 784, i.e. relatively high. This difference is especially interesting with respect to the performance of input domain branching as presented in Section 6.1. However, based on Katz et al. [35], we start this chapter with a description of the ACAS Xu system, which is a natural candidate for the application of neural network verification. Subsequently, we explain the definition of corresponding test instances for the verification problem. After that, we lay out the definition of test instances based on the MNIST data set. In the last section of this chapter, we explain the selection of two subsets of the various test instances. We use these subsets to reduce the computational cost for the evaluation of several parameter settings in our model.

8.1 ACAS Xu system

The ACAS Xu system is a variant of the Airborne Collision Avoidance System X (ACAS X) for unmanned aircrafts. Julian et al. [34] investigate how the decision making logic of ACAS Xu can be represented by a neural network instead of a very large table that requires several gigabytes of memory. This is taken as motivation by Katz et al. [35] to verify desirable properties of the neural networks for the ACAS Xu system using their solver Reluplex. Seven input variables, as defined in Table 8.1, are regarded in the ACAS Xu system, which models an encounter of two aircrafts. The purpose of the system is the proposal of an appropriate action for the so called ownship, whose flight is possibly affected by an intruder, i.e. another aircraft nearby. Of course, the ACAS Xu system can also state that in a situation there is no reason to expect a collision.

ρ	distance between ownship and intruder
θ	angle to intruder relative to ownship heading direction
ψ	heading angle of intruder relative to ownship heading direction
v_{own}	speed of ownship
v_{int}	speed of intruder
τ	time until loss of vertical separation
α_{prev}	previous advisory

Table 8.1: Input variables for ACAS Xu system (cf. Katz et al. [35]).

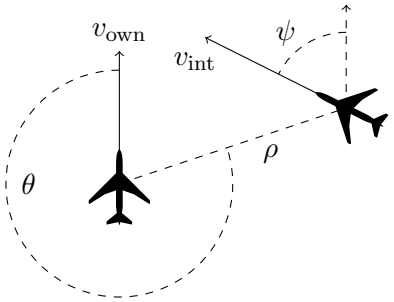

Figure 8.1: Visualization of geometric relations in the ACAS Xu system.

Based on Katz et al. [35] we give an outline of the functioning. Given an assignment of feasible values to the input variables as listed in Table 8.1, ACAS Xu computes scores for five different action items. These are Clear-of-Conflict (COC), weak left, weak right, strong left, or strong right. The action with the lowest score is then recommended by the ACAS Xu system. In fact, input variable τ is discretized to take one of the values $0, 1, 5, 10, 20, 40, 60, 80$ or 100 seconds, while α_{prev} is a discrete variable as it represents the action which was previuosly recommended by ACAS Xu. These variables are integrated into the neural networks and hence there are 45 neural networks, where each of these represents one of the possible combinations of τ and α_{prev}. Subsequently, each network has input dimension five due to the five remaining input variables $\rho, \theta, \psi, v_{\text{own}}$, and v_{int}. The output dimension is also five for each network, as there are five possible actions. Furthermore, each network has six hidden layers, where each of these contains 50 neurons with ReLU activations. A depiction of the geometric relations between the different input variables can be seen in Figure 8.1, and Table 8.2 shows the naming scheme for the neural networks.

α_{prev}	COC	weak left	weak right	strong left	strong right
i	1	2	3	4	5

τ in s	0	1	5	10	20	40	60	80	100
j	1	2	3	4	5	6	7	8	9

Table 8.2: ACAS Xu neural networks are denoted as i_j where i and j correspond to the values of τ and α_{prev} as indicated by the tables.

8.2 Neural network verification for ACAS Xu system

Katz et al. [35] define ten different properties for the ACAS neural networks, where some properties are defined for all 45 networks, and others only for one of these. We refer to Katz et al. [35] for details on the definitions of these properties. Apparently, the definition of Property 8 in Katz et al. [35] does not match the corresponding data which is released on Github by Katz et al. [35]. For our evaluations we stick to the released data, which was also used by Bunel et al. [12] in their comparison. This implies that the instance for Property 8 is a disjunction instance as defined in Remark 7, which is also the case for Properties 1, 2, 3, and 4. The definition of Property 6 does not exactly meet our definition of the verification problem, as the set of allowed input vectors is not a polytope. Nevertheless, Katz et al. [35] already split this property into Properties 6a and 6b, both of which are then feasible conjunction instances. Furthermore, Property 7 consists of two disjunction instances that are combined as in Remark 5. The other properties, i.e. 5, 9, and 10 are plain conjunction instances. While Properties 1, 2, 3, and 4 are applied to all or most of the 45 neural networks, for the others, each property is tested on only one network. Therefore, we name the instances like `property4_3_7`, where the first digit indicates the property and the last two digits denote the neural network as defined in Table 8.2. For Properties 5 to 10, which are defined on only one network each, we leave out the name of the neural network. Yet, if we split one of the conjunction instances 5, 6a, 6b, 7, 9, or 10 into separate instances as suggested in Remark 7, we may add a digit to the name to differentiate between these. For example, `property5_property` is the actual Property 5 as one (conjunction) instance, while `property5_property_0` is one of the instances obtained by splitting the original property as explained in Remark 7.

Both Bunel et al. [11, 12] and Wang et al. [57, 56] use this set of instances to evaluate their solving models for neural network verification. Therefore, we also use this set of instances, which is the most relevant benchmarking set for verification of neural networks, for our computational experiments. In fact, detailed runtimes for the approaches of Wang et al. [56] and Bunel et al. [12] are not provided in the respective papers, and the hardware equipment is different. Therefore, we also include their programs in our evaluation.

Moreover, we use the neural networks of the ACAS Xu system for the definition of instances $\Pi = (X, Y, F)$ where X is not a box. We define different properties which are called `lin_opp`, `int_away`, and `var_dist`. Property `lin_opp(_dir)` models the situation that the intruder is close and heads directly or roughly towards the ownship, which means that COC should not be the recommended output. In case of `int_away`, the intruder is faster than the ownship and ahead of it, and flies roughly in the same direction. Therefore, a strong turn should not be the recommended action. The situation modelled for property `var_dist` is similar. However, in this case both aircrafts are heading in upward direction in Figure 8.1 and the intruder is ahead of the ownship. Moreover, if the distance between the aircrafts is smaller, the intruder must be

faster compared to the ownship. So, for `var_dist` we demand that COC has the minimal score. It should be noted, that the instances based on property `var_dist` are conjunction instances, while the other instances are disjunction instances. Formal definitions of these properties can be found in Appendix A.

8.3 MNIST based test instances

LeCun et al. [36] introduced the famous MNIST data set for handwritten digit recognition which has been used extensively for benchmarking of machine learning models. Each image features a handwritten digit between zero and nine, which should be recognized correctly by machine learning models that were trained on the data set. In fact, it is possible to train relatively small neural networks on this data set, as the size of the grayscale images is only 28×28 pixels. Usually, neural networks for this classification task have ten output neurons. The one with the highest value indicates which digit is shown in the input image, according to the neural network.

It would be desirable to give a formal definition of each digit, and then verify whether a neural network classifies the digits correctly according to these definitions. However, it is very difficult to give a proper definition of a handwritten digit. Therefore, we regard the robustness of the classification of the images as in Wang et al. [56]. We assume that the classification of an image should not change, if each pixel of the image is allowed to change its value only slightly. Each pixel has an integer value between 0 and 255 which indicates the intensity of the pixel between white and black. To generate instances of the verification problem, we allow a perturbation of each pixel value by at most 1, 5, 10, or 20, of course limited by the general bounds of 0 and 255. Using these constraints, we obtain a polytope $X \subset \mathbb{R}^{784}$ of allowed input vectors for the neural network. Then we impose the condition, that the classification of the image should remain the same, with respect to the original (and correct) classification. Verifying this property implies that we obtain a conjunction instance of the verification problem. In fact, we demand that the value of the output neuron corresponding to the correct classification is greater than the values of all other output neurons. Assume that zero is the correct classification and y_0 is the corresponding output neuron, and the other output neurons are $y_1, \ldots, y_9$. Then we define the robustness property

$$Y := \bigcap_{i=1}^{9} \{(y_0, \ldots, y_9)^T \mid y_0 > y_i\} \subset \mathbb{R}^{10},$$

so Y is the intersection of open halfspaces. To obtain an instance of the verification problem, we need a neural network F. Wang et al. [56] provide three (fully connected) ReLU neural networks which are trained on the MNIST data set. Each of these networks has two hidden layers of 24, 50, or 512 ReLU neurons each, and 10 neurons without activation function in the output layer. We only use the networks with 24 and 512 neurons per layer, for which Wang et al. [56] report classification accuracies of 96.59 % and 98.27 % on the MNIST test set [36]. From the MNIST images provided by Wang et al. [56] on Neurify's Github repository [55], we select the images numbered 2, 4,

and 11 showing the digits one, four, or six, respectively. Combined with the four different perturbation bounds, we obtain 12 instances (X, Y, F) of the verification problem for each of the two neural networks. As these instances are quite similar among each other, and verifying robustness on images of digits is a somewhat artificial task, it seems unreasonable to create a very big number of such instances. However, also these instances are based on neural networks that arise from a training process on real data. This gives hope that verification of these is similar to verification of other neural networks with similar architectures.

According to our definition in Remark 7, all of these instances are conjunction instances. Hence, some solving methods require to split each of the instances into nine separate instances.

8.4 Selection of evaluation subsets

To evaluate various settings of our solving model, we use a subset of the instances that we presented in Sections 8.2 and 8.3. In fact, we use two subsets, one contains SAT instances and the other one UNSAT instances. Both are chosen such that they contain a diverse selection of all the instances. Mostly we use the set of UNSAT instances, since the majority of settings only affects the computation of the dual bound. To evaluate these settings, it is not useful to compare results on SAT instances, as these almost only depend on the performance of the primal heuristic. In both sets, there are conjunction and disjunction instances. As we also perform experiments using the formulation of the verification problem as feasibility problem, we exclude the conjunction instances in this case, because they cannot be solved directly by the model.

For the SAT subset, we select three of the instances that we defined on the ACAS neural networks (see Section 8.2 and Appendix A). In addition to that, we use two MNIST instances with 24 neurons per layer, and two with 512 neurons per layer. From the ACAS instances of Katz et al. [35], we use three instances belonging to Property 2, the two instances that constitute Property 7, and the instance for Property 8. There are no SAT instances belonging to other properties. A detailed list of the resulting 13 instances can be found in Section B.2 of the appendix, where we present results on the application of primal heuristics.

For the UNSAT subset, we select 23 instances in total. Four instances are selected out of our self defined ACAS instances, one of these is `lin_acas_3_1_var_dist`. This is one of the two conjunction instances of our self defined instances as listed in Appendix A. Then we include five MNIST instances, three where the neural network has 24 neurons per layer, and two with networks of 512 neurons per layer. Eventually we choose some of the instances introduced by Katz et al. [35], such that each property is represented, if corresponding UNSAT instances exist. We select two instances of Property 1 which have medium difficulty. For Property 2, there are only two UNSAT instances, which we both include. Furthermore, we include two instances each of Property 3 and 4. These are rather easy to solve, similar to all other instances of Properties 3 and 4. Properties 5 and 10 are included as one conjunction instance each. Regarding Properties 6a and 6b, for each

we choose one of the instances that are obtained by splitting the instances according to Remark 7. Furthermore, we select two of the instances that Property 9 can be split into. In fact, the conjunction instances are quite hard to solve. Hence, it should be noticed, that one of the instances after splitting tends to be significantly easier to solve than the whole conjuntion instance (at least for UNSAT instances). This is the reason to include such instances in the evaluation subset, because otherwise most of the instances cannot be solved within a time limit of two hours. Regarding Properties 7 and 8, there are no corresponding UNSAT instances. Finally, the tables in Appendix C list all instances which are contained in our UNSAT evaluation subset.

9 Computational Evaluation

In this chapter we present empirical results for various experiments that we conducted on the test instances which we presented in the last chapter. First, we provide an empirical comparison of the quality of the bounds which are computed by various methods as laid out in Chapter 4. Then we give an overview of various configurations of our solving model and their influence on its performance. We also use these experiments to determine good settings for our solving model that should be used to reach the best performance. In the last section, we present detailed runtime results of our solving model and the programs of Bunel et al. [12], Wang et al. [56], and Katz et al. [35].

The evaluation of computational experiments requires the consideration of various result values for many test instances. Average values can often help to gain relevant insights. Though, given several non-negative values $v_1, \ldots, v_k \in \mathbb{R}_{\geq 0}$, an average of these values can be computed by different means. Especially, if the values have different orders of magnitude, these differences can be quite substantial. While the arithmetic mean $\frac{1}{k}\sum_{i=1}^{k} v_i$ is most commonly used for the computation of average values, it is heavily influenced by high values. Therefore, single (high) outlier values may have an unintentionally strong impact on the average value. Especially for the computation of averaged runtimes we use the *shifted geometric mean* as defined in Hendel [30]. Based on a shift value $s \in \mathbb{R}_{\geq 0}$, it is given by

$$\Big(\prod_{i=1}^{k} v_i + s\Big)^{\frac{1}{k}} - s.$$

The shifted geometric mean does not only reduce the influence of outliers with high values, but also limits the influence of small values. The latter is in contrast to the geometric mean without shift. We refer to Hendel [30] and Achterberg [1] for a more detailed discussion and example calculations. If not stated differently, runtimes are always reported in seconds and for the computation of corresponding mean values we use a shift value of 10 (seconds). Average values for the number of solving nodes are also computed as shifted geometric mean with a shift value of 10.

9.1 Empirical comparison of bound computation approaches

We use our evaluation set of UNSAT instances, as described in Section 8.4, for a numeric comparison of the bounds which are obtained by various bound computation approaches. Considering an instance $\Pi = (X, Y, F)$ of the verification problem, we asssume that the ReLU neural network F has neurons $1, \ldots, N$. For all neurons we compute lower and upper bounds $[l_i, u_i]$, $i \in [N]$, based on the feasible input domain X. The bounds are computed layerwise from the first to the last layer. Branching is not applied, i.e. we consider the

approximation which assumes the whole polytope X as feasible input domain. Then we compute the shifted geometric mean of $\{u_1 - l_1, \ldots, u_N - l_N\}$ with shift value 1 for each instance. The mean value indicates how good the corresponding bound computation approach is. We use a relatively low shift value of 1, because small bound improvements can often be important. Clearly, it is desirable that the difference $u_i - l_i$ is as small as possible for all neurons $i \in [N]$. Hence, we can use the averaged values to compare the quality of the bounds that are computed by different methods. In view of Table 9.1, which shows the results, it is apparent that the quality of the computed bounds differs vastly. Of course, in general the better methods do come with a higher computational cost.

Instance	Naive IA	Sym. IA	Symbolic equations	OBBT LP	OBBT2 k=2, l=5	OBBT2 k=10, l=10	OBBT MIP
lin_acas_1_1_int_away	214.36	178.85	119.17	36.51	35.95	34.67	4.23
lin_acas_1_1_lin_opp2	182.90	111.48	60.60	15.60	15.49	15.01	3.55
lin_acas_1_1_lin_opp_dir	175.81	98.07	57.77	16.79	16.75	16.14	4.24
lin_acas_3_1_var_dist	155.95	110.24	73.78	15.38	15.14	14.32	1.25
mnist_24_image11_5	1183.50	888.54	755.85	732.07	732.07	732.07	670.67
mnist_24_image2_5	1150.58	834.30	743.92	718.20	718.20	718.20	634.58
mnist_24_image4_1	240.24	141.79	136.55	136.17	136.17	136.17	134.88
mnist_512_image11_5	895.37	654.84	475.09	442.19	442.19	442.19	440.36
mnist_512_image2_1	174.50	92.88	77.00	73.98	73.98	73.98	72.95
property10_property	189.95	93.06	52.47	18.49	18.29	17.49	6.43
property1_1_1	156.76	142.13	101.46	33.10	32.24	30.64	6.98
property1_2_2	251.37	246.90	141.17	52.89	52.41	50.27	12.54
property2_3_3	276.92	256.99	145.08	47.43	46.86	45.11	12.20
property2_4_2	169.18	167.75	100.07	36.38	36.18	35.26	8.79
property3_4_3	35.07	10.03	5.38	1.42	1.37	1.34	0.86
property3_4_4	46.96	18.14	5.87	1.18	1.17	1.17	0.85
property4_2_2	20.35	8.58	5.06	0.83	0.82	0.81	0.54
property4_3_7	54.87	24.96	8.84	2.91	2.83	2.81	2.27
property5_property	63.30	37.02	24.38	5.90	5.79	5.52	2.39
property6_6a_property_3	155.80	123.15	67.99	27.09	26.81	25.62	5.87
property6_6b_property_1	179.81	135.86	72.60	27.22	26.95	25.79	5.72
property9_property_0	66.15	33.42	17.63	6.21	6.15	5.89	2.61
property9_property_4	66.15	33.42	17.63	6.21	6.15	5.89	2.62
shifted geo. mean (shift=1)	156.54	98.30	60.23	24.83	24.61	23.98	10.38

Table 9.1: Averaged differences between neuron bounds in our UNSAT test set for different bound computation methods. "IA" stands for interval arithmetic, i.e. symbolic IA is the approach of Wang et al. [57]. "Symbolic equations" refers to the improved method of Wang et al. [56]. We evaluate OBBT on the LP relaxation as well as on the MIP directly, and also our new technique OBBT2 with two different parameter settings. For OBBT on the MIP model, a time limit of five seconds is set for each MIP which is solved. In fact, the instances "property9_property_0" and "property9_property_4" differ only in the property to be verified, which explains the coinciding numbers.

The results in Table 9.1 show that the best bounds are computed by OBBT on the MIP model. However, this approach implies very long runtimes. Here we only consider the quality of the bounds and refer to Section 9.2 for experiments which evaluate the final performance of various bound computation techniques. While OBBT2 computes better bounds compared to OBBT on

the LP relaxation, the improvements are unfortunately quite minor. The configuration $k = 10$, $l = 10$ implies that 100 additional LPs need to be solved for each layer of the neural network, compared to 10 in the case of $k = 2$, $l = 5$. We also see that the approaches of Wang et al. [57, 56] are clearly superior to naive interval arithmetic. Yet, they are not competitive with OBBT if regarding only the quality of the computed bounds.

9.2 Comparison of different techniques in our model

In this section we provide an overview of the performance of our solving model in various configurations. Especially, we evaluate different settings for OBBT2 and for the separator based on the work of Anderson et al. [3]. Furthermore, we compare many different settings and evaluate the performance of our solving model with these settings. While the formulation of the verification problem as optimization problem is used in most cases, we also provide some results using the formulation as satisfiability problem. The experiments in this section are run for the instances of our evaluation set of UNSAT instances as described in Section 8.4. All results are obtained on cluster nodes with Intel Xeon CPUs E5-2670 which have a clock rate of 2.5 GHz. Each experiment is run exlusively on one cluster node and a memory limit of 32 GB is set. For our time measurements (in wall clock time), we use a script which is provided by Bunel et al. [12] and was used for their computational study. In general we set a time limit of two hours, i.e. 7200 seconds for all experiments. If the time limit is hit during the solving process, we assume the time limit of 7200 seconds as runtime for the corresponding instance. This allows the computation of average runtimes over all instances, also in the presence of instances for which the time limit is hit. Clearly, we possibly underestimate the real average runtime, but still we obtain a lower bound on the average runtime. The same approach is also taken in Achterberg [1].

We report results both for the whole UNSAT test set, and separately for the instances that are based on the ACAS XU system (cf. Section 8.2), and the instances that are based on the MNIST [37] data set. Since the structure of the corresponding neural networks is quite different, the performance of a certain configuration may be bad on the ACAS instances, but good on the MNIST instances, or vice versa. Comparing ACAS and MNIST based instances, the main differences are the number of input neurons (5 vs. 784) and the number of layers in the neural networks (6 vs. 2). Detailed results for all experiments presented in this Section can be found in Appendix C.

For the experiments in this section we use a baseline configuration "no_heur_base" of our solving model. In this configuration, domain branching is used up to a depth of 20 in the branch-and-bound tree. It should be noted that this depth is usually not exceeded, if domain branching is used. For the selection of the branching variable, we apply the gradient based selection rule as described in Section 6.1. Furthermore, OBBT is applied to the LP relaxation at each solving node up to a depth of 20 in the branch-and-bound tree. The primal heuristic is enabled only at the root node. Further techniques

(e.g. the separator based on the work of Anderson et al. [3]) are not applied. For a list of the detailed parameter settings we refer to Section C.3 of the appendix.

Computational results for our relaxation tightening technique OBBT2 are reported in Tables 9.2 and 9.3. We evaluate four different configurations of OBBT2 and compare them to the baseline configuration "no_heur_base". Especially, we investigate our selection rule for the pairs of neurons to which OBBT2 is applied. As described in Section 4.7, two parameters $k, l \in \mathbb{N}$ determine the number of selected neuron pairs. Indeed, per layer of the neural network, kl pairs are then chosen for the execution of OBBT2. This implies that kl LPs per layer of the neural network must be solved additionally for the execution of OBBT2. We try two different parameter settings, $k = 2$, $l = 5$ on the one hand, and $k = 10$, $l = 10$ on the other hand. Moreover, to evaluate the performance of our selection rule, we alternatively choose kl neuron pairs in a fixed order. We use this as a baseline selection rule, since this selection order of the neuron pairs is based on the input description of the neural network. In contrast to that, our selection rule is based on the error analysis as laid out in Section 4.7. The suffix "_nosort" in the name of the configuration indicates that the baseline selection rule is used. Furthermore, "_10" means that $k = 10$, $l = 10$ is set while otherwise it holds $k = 2$, $l = 5$. In Table 9.2 averaged runtimes are reported for the different configurations of OBBT2 and the baseline configuration "no_heur_base" without OBBT2. Clearly, in many cases the application of OBBT2 leads to increased runtimes. Especially in the case $k = 10$, $l = 10$ the cost for solving 100 additional LPs per network layer is not outweighed by the improvements to the LP relaxation. Unfortunately, the results also indicate that our selection rule cannot outperform the baseline selection rule. Apparently, our error analysis does not help to select neuron pairs at which OBBT2 is more effective. However, the results in Table 9.3 clearly show that OBBT2 works in principle. The average number of solving nodes in the branch-and-bound tree, which is computed only for instances that were solved within the time limit, is significantly reduced by OBBT2. Furthermore, in the case $k = 10$, $l = 10$ the reduction is clearly larger than in the case $k = 2$, $l = 5$ which can be attributed to the stronger LP relaxation. However, the runtime increases heavily in this case. See Section C.1 of the appendix for detailed results of this experiment. The results for the baseline configuration "no_heur_base" can be found in Section C.3.

Subset (number of instances)	**All (23)**		**ACAS (18)**		**MNIST (5)**	
Configuration	Time	Timeouts	Time	Timeouts	Time	Timeouts
no_heur_base	854.9	5	771.2	2	1237.3	3
no_heur_base_obbt2_nosort	888.0	5	816.8	2	1199.1	3
no_heur_base_obbt2	892.8	5	821.0	2	1206.2	3
no_heur_base_obbt2_10_nosort	1225.2	9	1230.0	6	1208.1	3
no_heur_base_obbt2_10	1249.5	9	1261.8	6	1206.0	3

Table 9.2: Runtime results for the application of OBBT2. In the column "Timeouts" we report for how many of the instances the execution was stopped due to the time limit of two hours.

Configuration	Nodes	Time
no_heur_base_obbt2_10_nosort	48.2	387.4
no_heur_base_obbt2_10	48.7	400.3
no_heur_base_obbt2	51.2	252.1
no_heur_base_obbt2_nosort	51.7	249.6
no_heur_base	55.6	236.3

Table 9.3: OBBT2 clearly reduces the number of solving nodes, which is computed as shifted geometric mean with shift value 10 (as the runtime mean values). We regard only those 14 instances, which are solved within the time limit by all methods. If the execution of a method is stopped due to the time limit, it is not reasonable to compare the number of solving nodes between different methods.

In the following, we report computational results regarding the application of the separator in our solving model, which is based on the work of Anderson et al. [3]. As mentioned in Section 7.2, we implement a separator in PySCIPopt according to our description in Section 3.4. In SCIP there are various settings available that control when and how often this separator is executed. Since the execution of the separator is not necessary for the correctness of the solving model, we can freely adapt these settings to reach a maximum reduction of the overall runtime. In Table 9.4 we see that the execution of the separator at some of the solving nodes reduces the average runtime of our solving model. We use the same baseline configuration "no_heur_base" as in our evaluation of OBBT2 as a reference for comparison. Detailed computational results and the settings for the various configurations can be found in Section C.2 of the appendix. In the configuration "no_heur_sepa0_freq5", which leads to the shortest average runtime, the separator is applied only at each fifth depth level of the branch-and-bound tree. Moreover, then it is only applied to the solving node with the best, i.e. highest, dual bound. In all other configurations (except "no_heur_base"), the separator is applied at each level of the branch-and-bound tree. However, the separator may be applied to all solving nodes as in "no_heur_sepa1", or only to those with a sufficiently good lower bound (other configurations).

Subset (number of instances)	All (23)		ACAS (18)		MNIST (5)	
Configuration	Time	Timeouts	Time	Timeouts	Time	Timeouts
no_heur_sepa0_freq5	839.2	5	757.4	2	1213.1	3
no_heur_sepa1	841.2	5	760.5	2	1208.7	3
no_heur_sepa0	842.9	5	757.4	2	1237.0	3
no_heur_sepa0_high	849.9	5	771.9	2	1200.9	3
no_heur_base	854.9	5	771.2	2	1237.3	3
no_heur_sepa	855.0	5	776.4	2	1208.6	3

Table 9.4: Comparison of settings for the separator as suggested by Anderson et al. [3]. In the column "Timeouts" we report for how many of the instances the execution was stopped due to the time limit of two hours.

In Table 9.5 we analyse how the application of the separator impacts the number of necessary solving nodes. For this comparison, we regard only those

18 instances of the UNSAT test set which are solved within the time limit in this experiment. Table 9.5 shows that the number of solving nodes is reduced due to the application of the separator. Since we cut off nodes whenever the dual bound is positive, the processing order of the solving nodes also impacts the total number of necessary solving nodes. Therefore, the number of solving nodes can be smaller although the separator is executed less often.

Configuration	Nodes	Time
no_heur_sepa0_freq5	108.9	458.8
no_heur_sepa0	109.0	461.4
no_heur_sepa	109.1	469.9
no_heur_sepa1	109.2	460.2
no_heur_sepa0_high	109.4	466.4
no_heur_base	112.6	469.9

Table 9.5: Mean values for runtime and number of solving nodes over the 18 instances of the UNSAT test set that are solved within the time limit by all configurations in the table. In both cases, the shifted geometric mean over the 18 instances with shift value 10 is used.

In Table 9.6 we report mean runtimes on the UNSAT test set for several different configurations of our solving model. With respect to OBBT2 and the separator, we use those configurations that led to the best results in the corresponding comparisons. Detailed computational results for each test instance and the parameter settings of each configuration can be found in Appendix C. Here we try to give an impression of the differences between the configurations which we tested. Except for two cases, the primal heuristic is used only at the root node and not at any other solving nodes. We are mostly interested in the computation of the dual bound by various methods. Since the primal heuristic contains some random influence, we want to keep this influence small. Though, we do employ the primal heuristic at the root node to obtain a primal bound for the problem, which corresponds to the usual solving process with heuristic enabled. Otherwise, in most cases no primal solution is found at all. We try this with the configuration "no_heur_atall", where the primal heuristic is also disabled at the root node. Contrary to this, in the configuration "heur_base_20" the primal heuristic is applied at all solving nodes of the branch-and-bound tree up to a depth of eight. In fact, most of the other configurations are adapted from our baseline configuration "no_heur_base" by including additional solving techniques.

It is apparent to see in Table 9.6, that there is a clear difference between solving MNIST based and ACAS based instances. In fact, the configuration "no_heur_base_genv" is the fastest in total because it performs well on both types of instances. However, it is not the best configuration for either of the two subsets (although close to the best configuration for ACAS instances). The best configurations with respect to the MNIST instances ("no_heur_relu_genv" with respect to number of timeouts, "mnist_base" with respect to mean runtime) do not perform well on the ACAS instances.

The configurations in Table 9.6 are sorted by the total number of timeouts on the UNSAT evaluation set. In the following, we give short descriptions of

the configurations in this order. The configuration "no_heur_base_genv" corresponds to the baseline configuration with the additional use of Langrangian variable bounds (LVBs) as described in Section 4.3. Obviously, the LVBs are quite beneficial for solving MNIST based instances as it can be seen in Table 9.6. As discussed before, the configuration "no_heur_sepa0_freq5" is the best one we found when investigating the performance of the separator. Next in Table 9.6 follows our baseline configuration which is hence quite good already. Indeed, "no_heur_base_obbt2_nosort" is the best of our configurations that use OBBT2 and it has a higher mean runtime. The configuration "heur_base_20" is identical to the baseline configuration "no_heur_base", except that the primal heuristic is also executed locally. This happens at all solving nodes up to a depth of eight in the branch-and-bound tree. While the additional executions of the heuristic do increase the runtime, these are often necessary to find counterexamples successfully. Although the heuristic may also help to compute tighter neuron bounds, we see that the runtime effect of its extensive application is negative on our UNSAT evaluation set. In contrast, the configuration "no_heur_atall" does not use the primal heuristic at all, and therefore times out on one more ACAS instance than the configurations considered so far. The configuration "no_heur_base_200" is very similar to the baseline configuration, however OBBT is only applied to neurons for which the difference between upper and lower bound is less than 200. Apparently this strategy is not beneficial on our benchmark set. Regarding the mean runtime compared to the number of timeouts, the configuration "no_heur_base_mip" is not in line with the other configurations. Indeed, this configuration applies OBBT to the MIP model in the beginning of the solving process to compute initial neurons bounds. This comes with a high computational cost, also for rather easy instances. However, for more difficult instances, this stratey is not that bad as indicated by the relatively low number of timeouts.

Subset (number of instances)	**All (23)**		**ACAS (18)**		**MNIST (5)**	
Configuration	Time	Timeouts	Time	Timeouts	Time	Timeouts
no_heur_base_genv	586.3	3	765.4	2	221.6	1
no_heur_sepa0_freq5	839.2	5	757.4	2	1213.1	3
no_heur_base	854.9	5	771.2	2	1237.3	3
no_heur_base_obbt2_nosort	888.0	5	816.8	2	1199.1	3
heur_base_20	930.2	5	855.7	2	1255.5	3
no_heur_atall	843.5	6	764.2	3	1202.3	3
no_heur_base_200	952.3	6	757.3	2	2165.0	4
no_heur_base_mip	1556.5	6	1578.2	3	1480.9	3
no_heur_relu_genv	750.3	7	1180.0	7	141.6	0
no_heur_relu	970.2	9	1334.2	8	304.5	1
domain_relu	939.4	10	1054.5	8	618.8	2
no_heur_relu_gradient	1348.9	13	1907.9	12	383.1	1
no_heur_std_branch	1615.6	13	1756.0	10	1196.3	3
no_heur_sym	3416.8	18	4651.6	15	1121.8	3
mnist_base	2753.5	19	7200.0	18	77.5	1

Table 9.6: Runtime results on the UNSAT test set using the formulation as optimization problem and various configurations.

In the configuration "no_heur_relu_genv", ReLU branching is combined with OBBT on the LP relaxation and the creation of LVBs. Most notably,

this configuration is the only one that solves all MNIST instances in the evaluation set within the time limit. Though, the performance on the ACAS based instances is rather mediocre. Due to the low number of input neurons of the ACAS neural networks, input domain branching is more efficient for these instances. The same holds for the configuration "no_heur_relu" which does not include the LVBs. Therefore, it performs not as good on both subsets of our benchmark set. In the configuration "domain_relu", domain branching is used up to a depth of six in the branch-and-bound tree and ReLU branching afterwards. Compared to the baseline configuration which uses only domain branching, it therefore performs better on the MNIST instances. The configuration "no_heur_relu_gradient" applies ReLU branching and uses the gradient based selection of the branching variable as described in Section 6.2. Clearly, the gradient based selection rule is inferior to the selection rule "standard" which is used by the other configurations with ReLU branching. In these configurations, branching variables are selected primarily from the first layers of the neural networks. Though, the selection of the branching variable for domain branching based on a fixed scheme, as suggested in Bunel et al. [11], is not competitive with the gradient based selection (cf. Section 6.1). This is shown clearly by the performance of the configuration "no_heur_std_branch". Gradient based selection is used in all other configurations which use domain branching, i.e. also in "no_heur_base". In the configuration "no_heur_sym" OBBT is not used and instead the bound computation approach of Wang et al. [56] as described in Section 4.1 (see also "symbolic equations" in Section 9.1). It should be noted that our implementation of this method is not very optimized, thus that the computations are quite slow and hence many instances cannot be solved within the time limit. Domain branching is applied in this configuration as in our baseline configuration "no_heur_base". Eventually, the configuration "mnist_base" uses neither domain branching nor ReLU branching, and OBBT is also disabled. Initial neuron bounds are computed by naive interval arithmetic. Subsequently, in this configuration the solving process is mostly limited to the application of standard MIP techniques. Notably enough, this configuration has the lowest mean runtime on the subset of MNIST based instances. On the other hand, it times out on all ACAS based instances. This vast difference can probably be attributed to the different number of layers in the neural networks. As the MNIST neural networks have only two layers, the initial neuron bounds, which are computed by naive IA, are sufficiently good to solve the instances. Though, the ACAS neural networks have six layers, and the neuron bounds, which are computed by naive IA, are therefore extremely loose in rear layers. Hence, these instances cannot be solved successfully without specialized bound computation methods (and branching rules).

While all the configurations for which runtimes are reported in Table 9.6 solve the verification problem as an optimization problem, we also evaluate our solving model with the formulation as satisfiability problem. Indeed, we try three such configurations as can be seen in Table 9.7. Though, with the formulation as satisfiability problem not all of the instances in our evaluation set of UNSAT instances can be solved directly. Although it would be possible to split such instances into various ones, which could then be solved as laid

Configuration	Time	Timeouts
no_heur_sepa_nonopt	555.3	1
no_heur_base_nonopt	555.9	1
no_heur_base	590.1	1
no_heur_relu_nonopt	1040.2	6

Table 9.7: Comparison of various configurations using the formulation as feasibility problem with the baseline configuration of the formulation as optimization problem ("no_heur_base"). Detailed results and the applicable instances can be found in Section B.4 of the appendix.

out in Section 3.1, we simply omit these from our evaluation. The remaining 15 instances can be found in the tables in Section B.4 of the appendix. We create a configuration "no_heur_base_nonopt" that is identical to our baseline configuration "no_heur_base" with the only difference that the formulation as feasibility problem is used. In fact, we also use the same gradient based selection of the branching variable for the domain branching rule, which is actually based on the formulation as optimization problem. However, the implementation of the selection rule does not affect the MIP formulation of the verification problem. Moreover, we try to include the separator ("no_heur_sepa_nonopt") and ReLU branching instead of domain branching ("no_heur_relu_nonopt"). LVBs cannot be created for the formulation as satisfiability problem as there is no objective function. It can be seen that two of the configurations which use the formulation as feasibility problem perform indeed better than our baseline configuration "no_heur_base". Therefore, we also use a configuration which is based on the formulation as satisfiability problem for the comparison with other solvers in the next section.

9.3 Comparison with other solvers

In this section we provide detailed results of computational experiments which we conducted to investigate the performance of different solvers for neural network verification. Besides our own solving model, which we regard in various configurations, we include the programs of Bunel et al. [12], Wang et al. [56], and Reluplex by Katz et al. [35]. The work of Wang et al. [56] is implemented in two solvers, Neurify and ReluVal. ReluVal is used on the ACAS instances of Katz et al. [35], whereas Neurify can be applied to the MNIST instances. Our benchmark set consists of all the instances described in Chapter 8, which can be grouped into three categories. First, we have the ACAS instances as defined by Katz et al. [35]. Second, we have MNIST instances, for which we use the corresponding neural networks provided by Wang et al. [56]. Third, we have our self defined instances which are also based on the ACAS neural networks but do not have a box as feasible input domain. Due to that, the instances of the last category can only be solved by our solving model. As mentioned in Section 2.2, the solver Sherlock of Dutta et al. [16] for output range analysis is in principle capable to solve such instances. However, it fails in certain cases (cf. Section 2.2) and we therefore exclude it from the evaluation. In principle, the solving approaches of Katz et

al. [35] and Bunel et al. [12] allow that the feasible input domain is not a box. Though, this is not reflected in their corresponding implementations such that their programs cannot be applied to our newly defined instances (cf. Section 6.3). We always check whether any alleged counterexample presented by some solver is indeed a feasible counterexample for the corresponding instance. In general, we perform these checks with a numerical tolerance of 10^{-8}. Several counterexamples are only feasible though, if a higher numerical tolerance is allowed, which we report in that case.

A different hardware setup than in the previous section is used to conduct the experiments in this section. However, this affects only the type of CPU which is used. We still set a memory limit of 32 GB and use the script of Bunel et al. [12] to measure the wall clock time of program executions. Each experiment is run exclusively on one cluster node with an Intel Xeon Gold 5122 CPU which operates at a clock rate of 3.6 GHz. Subsequently, we can expect lower runtimes compared to the experiments described in Section 9.2.

Besides the program names *ReluVal*, *Neurify* and *Reluplex*, we use the following terms to denote the various solving models. *Adv* refers to *ReluVal* or *Neurify* using the adversary check mode as described in Section 6.3. *BaB* denotes the branch-and-bound method as implemented by Bunel et al. [12]. Eventually, we use *NonOpt* to describe that our solving model is used with the formulation of the verification problem as feasibility problem. *Joint* refers to our solving model using the formulation as optimization problem, which can also solve conjunction instances (cf. Remark 7). On the other hand, *Separate* indicates that we solve the verification problem as optimization problem, but conjunction instances are split into several disjunction instances which are then solved. This splitting is explained in Remark 7. As explained in Section 6.3, conjunction instances also have to be splitted for BaB and Reluplex.

Based on the results of our evaluation experiments in Section 9.2, we choose the configuration "no_heur_sepa0_freq5" as the best for the ACAS instances. Though, in order for a good performance on refutable instances, we do adapt this configuration to execute the primal heuristic also locally up to a depth of eight in the branch-and-bound tree. Compared to the configuration "no_heur_sepa0_freq5", the only difference is that the parameter "sampling_heuristic_local_max_iter" is set to 1000. For *NonOpt*, the parameter "use_opt_mode" is set to False, for *Joint* it is True. We refer to our hints in the beginning of Appendix C on how the parameter settings must be chosen for the different types of instances. Moreover, the detailed parameter settings of the various configurations can be found in Appendix C, too.

Instance	Result	ReluVal	Adv	NonOpt	Joint	BaB	Reluplex
property1_1_1	UNSAT	0.4	0.1	88.0	95.9	329.4	822.3
property1_1_2	UNSAT	0.5	0.2	146.0	155.3	703.7	1300.9
property1_1_3	UNSAT	1.9	1.5	359.6	382.5	1349.5	timelimit
property1_1_4	UNSAT	1.7	1.7	141.1	148.1	791.9	2260.2
property1_1_5	UNSAT	0.3	0.3	100.0	105.9	60.4	1750.7
property1_1_6	UNSAT	0.4	0.2	127.7	130.6	68.3	999.0
property1_1_7	UNSAT	0.1	0.1	81.8	83.5	66.0	398.0
property1_1_8	UNSAT	0.1	0.1	15.1	17.1	15.2	741.2
property1_1_9	UNSAT	0.1	0.1	15.9	14.6	16.5	204.9
property1_2_1	UNSAT	0.6	0.6	246.5	260.0	974.8	3015.1
property1_2_2	UNSAT	1.1	1.2	968.3	1029.9	1687.5	5351.5

Instance	Result	ReluVal	Adv	NonOpt	Joint	BaB	Reluplex
property1_2_3	UNSAT	1.6	1.5	928.8	988.1	818.8	4123.8
property1_2_4	UNSAT	0.6	0.6	116.5	125.0	603.2	1646.8
property1_2_5	UNSAT	3.8	3.4	1187.3	1148.5	timelimit	timelimit
property1_2_6	UNSAT	2.7	2.7	2319.3	2443.8	timelimit	timelimit
property1_2_7	UNSAT	11.2	10.3	4368.0	4620.1	timelimit	6329.2
property1_2_8	UNSAT	3.5	3.5	3557.8	3713.8	timelimit	timelimit
property1_2_9	UNSAT	8.9	7.2	timelimit	timelimit	timelimit	timelimit
property1_3_1	UNSAT	0.4	0.5	184.0	195.0	818.1	886.6
property1_3_2	UNSAT	0.8	0.8	215.8	269.1	1224.7	2200.6
property1_3_3	UNSAT	1.1	1.3	211.0	220.9	1007.5	2235.1
property1_3_4	UNSAT	0.6	0.6	121.9	130.0	604.1	2025.7
property1_3_5	UNSAT	1.5	1.5	459.9	480.5	4595.1	2737.0
property1_3_6	UNSAT	28.4	21.9	1124.0	1174.8	timelimit	timelimit
property1_3_7	UNSAT	12.1	10.9	1081.2	1142.7	timelimit	timelimit
property1_3_8	UNSAT	7.8	6.7	2540.9	2665.7	timelimit	timelimit
property1_3_9	UNSAT	11.7	8.5	982.6	1052.0	timelimit	timelimit
property1_4_1	UNSAT	17.0	16.0	567.0	597.6	2525.6	timelimit
property1_4_2	UNSAT	2.5	2.5	490.0	501.4	1287.9	5064.6
property1_4_3	UNSAT	1.1	1.2	1344.5	1370.2	940.5	3106.0
property1_4_4	UNSAT	0.8	1.0	134.9	137.9	829.4	2679.3
property1_4_5	UNSAT	2.9	3.1	1927.2	2002.1	timelimit	timelimit
property1_4_6	UNSAT	22.6	15.1	timelimit	timelimit	timelimit	timelimit
property1_4_7	UNSAT	22.3	17.6	3869.8	4205.4	timelimit	timelimit
property1_4_8	UNSAT	48.9	16.8	1786.9	1856.2	timelimit	timelimit
property1_4_9	UNSAT	21.2	15.8	timelimit	timelimit	timelimit	timelimit
property1_5_1	UNSAT	0.6	0.6	255.4	301.4	1204.2	1424.1
property1_5_2	UNSAT	0.6	0.6	204.2	171.7	809.2	3187.3
property1_5_3	UNSAT	0.3	0.3	165.2	177.5	794.2	1461.0
property1_5_4	UNSAT	0.4	0.5	111.0	124.8	594.5	3011.9
property1_5_5	UNSAT	1.2	1.2	624.0	652.0	timelimit	6622.6
property1_5_6	UNSAT	15.9	13.7	3092.2	3248.8	timelimit	timelimit
property1_5_7	UNSAT	3.4	3.4	3427.0	3558.0	timelimit	timelimit
property1_5_8	UNSAT	16.1	11.0	3538.5	3816.2	timelimit	timelimit
property1_5_9	UNSAT	7.8	7.4	4706.9	4995.1	timelimit	timelimit
property2_2_1	SAT	0.1	0.1	24.8	1.4	17.2	386.6
property2_2_2	SAT	0.3	0.2	1.4	1.2	15.5	9.9
property2_2_3	SAT	0.2	0.1	24.8	1.2	15.3	43.3
property2_2_4	SAT	0.1	0.1	1.2	1.2	13.0	29.6
property2_2_5	SAT	0.1	0.1	1.2	1.4	16.4	238.8
property2_2_6	SAT	0.2	0.1	1.3	1.2	15.6	5195.4
property2_2_7	SAT	0.1	0.1	1.2	1.2	16.2	4498.2
property2_2_8	SAT	0.1	0.1	1.2	1.2	17.1	697.7
property2_2_9	SAT	timelimit	0.3	1.2	1.4	16.6	(wrong)
property2_3_1	SAT	0.2	0.2	1.2	1.1	13.4	583.1
property2_3_2	SAT	4252.8	0.2	23.7	1.5	88.1	777.6
property2_3_3	UNSAT	5382.9	95.6	3548.5	3964.8	timelimit	timelimit
property2_3_4	SAT	0.4	0.1	1.2	1.2	13.4	1463.1
property2_3_5	SAT	0.1	0.1	1.2	1.2	14.3	3761.4
property2_3_6	SAT	0.2	0.1	1.2	1.3	15.0	149.3
property2_3_7	SAT	87.2	3.2	1.2	1.2	17.1	595.6
property2_3_8	SAT	0.3	0.2	1.3	1.2	15.3	536.5
property2_3_9	SAT	0.2	0.1	4.7	1.2	15.1	67.2
property2_4_1	SAT	0.1	0.1	1.4	1.2	13.5	700.8
property2_4_2	UNSAT	timelimit	205.2	timelimit	timelimit	timelimit	timelimit
property2_4_3	SAT	0.2	0.1	1.3	6.0	18.0	10.6
property2_4_4	SAT	0.1	0.1	1.3	1.3	12.9	75.9
property2_4_5	SAT	0.2	0.1	1.1	1.4	16.4	3430.9
property2_4_6	SAT	0.1	0.1	1.2	1.3	15.8	1169.8
property2_4_7	SAT	0.1	0.1	1.3	1.2	16.1	3470.2
property2_4_8	SAT	0.1	0.1	1.2	1.3	16.7	4274.8
property2_4_9	SAT	50.6	0.1	1.2	1.2	16.7	5354.9
property2_5_1	SAT	0.1	0.1	1.2	1.2	15.9	1456.7
property2_5_2	SAT	0.2	0.1	1.4	1.2	14.1	290.1
property2_5_3	SAT	timelimit	(UNSAT)	691.7	2.1	1199.7	(wrong)
property2_5_4	SAT	0.2	0.2	4.1	1.2	16.2	33.0
property2_5_5	SAT	0.2	0.1	1.2	1.2	15.3	932.4

Instance	Result	ReluVal	Adv	NonOpt	Joint	BaB	Reluplex
property2_5_6	SAT	0.1	0.1	1.1	1.4	17.1	1857.9
property2_5_7	SAT	0.2	0.1	1.3	1.2	18.0	1211.9
property2_5_8	SAT	0.1	0.1	1.2	1.3	16.5	5435.2
property2_5_9	SAT	0.1	0.1	1.3	1.4	20.4	(wrong)
property3_1_1	UNSAT	112.2	73.1	40.0	42.3	79.9	6946.6
property3_1_2	UNSAT	2.4	2.5	10.9	13.5	13.2	5588.8
property3_1_3	UNSAT	3.7	3.7	87.8	92.9	105.9	1277.5
property3_1_4	UNSAT	0.3	0.3	3.4	4.4	6.5	595.9
property3_1_5	UNSAT	0.2	0.2	2.9	2.9	6.1	359.8
property3_1_6	UNSAT	0.1	0.1	2.4	3.4	5.4	74.2
property3_2_1	UNSAT	21.5	15.6	7.8	14.8	9.8	1367.4
property3_2_2	UNSAT	8.1	8.2	11.4	12.8	18.8	739.7
property3_2_3	UNSAT	0.2	3.2	40.8	44.8	159.1	1184.2
property3_2_4	UNSAT	0.6	0.6	2.6	2.6	4.8	47.5
property3_2_5	UNSAT	0.4	0.4	2.8	4.1	5.5	268.8
property3_2_6	UNSAT	0.1	0.1	2.4	3.4	5.7	90.4
property3_2_7	UNSAT	0.2	0.3	2.4	3.7	5.6	132.9
property3_2_8	UNSAT	0.4	0.1	2.2	3.2	5.5	93.8
property3_2_9	UNSAT	0.1	0.1	1.9	2.1	4.6	31.5
property3_3_1	UNSAT	3.0	2.8	3.5	4.5	4.9	202.0
property3_3_2	UNSAT	6.1	6.1	19.1	21.0	69.2	1772.9
property3_3_3	UNSAT	0.3	0.3	3.5	4.4	5.1	1234.8
property3_3_4	UNSAT	0.5	0.5	7.0	8.3	12.7	238.8
property3_3_5	UNSAT	2.1	2.1	2.8	3.9	5.4	94.5
property3_3_6	UNSAT	22.0	20.8	3.5	8.0	13.6	287.9
property3_3_7	UNSAT	0.2	0.1	2.0	3.1	4.3	40.5
property3_3_8	UNSAT	3.5	3.6	2.8	3.8	5.4	205.1
property3_3_9	UNSAT	2.7	2.7	2.5	3.8	4.7	134.2
property3_4_1	UNSAT	8.5	8.4	8.3	10.0	28.9	231.8
property3_4_2	UNSAT	106.8	81.0	22.8	24.6	64.4	3240.4
property3_4_3	UNSAT	2.3	2.3	20.3	22.3	90.8	1990.7
property3_4_4	UNSAT	0.2	0.2	2.3	3.5	4.8	100.7
property3_4_5	UNSAT	0.1	0.1	2.5	2.6	6.1	39.9
property3_4_6	UNSAT	0.2	0.2	4.6	5.9	6.8	346.4
property3_4_7	UNSAT	0.5	0.5	2.5	3.6	5.7	144.9
property3_4_8	UNSAT	1.8	1.8	2.9	4.1	6.0	165.3
property3_4_9	UNSAT	0.1	0.1	3.1	4.4	6.1	162.3
property3_5_1	UNSAT	22.8	20.9	24.0	29.0	72.0	1278.7
property3_5_2	UNSAT	2.3	2.2	3.0	3.5	5.3	170.5
property3_5_3	UNSAT	0.2	0.2	3.7	4.8	5.5	388.6
property3_5_4	UNSAT	0.2	0.2	5.5	3.6	5.0	72.0
property3_5_5	UNSAT	0.5	0.5	3.2	4.4	5.4	98.6
property3_5_6	UNSAT	0.9	1.0	2.8	3.9	7.0	329.5
property3_5_7	UNSAT	0.1	0.1	2.3	3.2	4.8	41.2
property3_5_8	UNSAT	0.1	0.1	2.6	3.7	5.9	353.1
property3_5_9	UNSAT	0.1	0.1	2.2	2.2	4.9	22.7
property4_1_1	UNSAT	1.2	1.1	8.7	11.0	12.7	1463.5
property4_1_2	UNSAT	1.3	1.3	15.3	17.3	13.4	1437.2
property4_1_3	UNSAT	0.4	0.4	25.8	27.6	46.8	1328.9
property4_1_4	UNSAT	0.3	0.3	6.5	8.4	12.9	121.2
property4_1_5	UNSAT	0.5	0.5	6.3	7.8	5.7	406.0
property4_1_6	UNSAT	0.2	0.3	3.2	4.4	5.8	249.7
property4_2_1	UNSAT	0.8	0.9	8.9	11.4	6.8	371.8
property4_2_2	UNSAT	2.1	2.1	7.6	10.0	6.0	471.0
property4_2_3	UNSAT	0.9	1.0	3.1	3.7	6.0	284.0
property4_2_4	UNSAT	0.2	0.2	3.2	4.2	5.3	98.4
property4_2_5	UNSAT	0.4	0.4	3.2	4.1	5.5	174.0
property4_2_6	UNSAT	0.3	0.3	3.9	5.0	6.5	135.1
property4_2_7	UNSAT	0.1	0.1	2.3	3.1	5.1	39.6
property4_2_8	UNSAT	0.1	0.1	7.8	9.3	14.8	623.0
property4_2_9	UNSAT	0.1	0.1	2.1	7.2	4.9	59.3
property4_3_1	UNSAT	1.1	1.2	4.4	5.5	5.7	556.1
property4_3_2	UNSAT	0.5	0.4	3.8	5.0	5.7	125.7
property4_3_3	UNSAT	0.1	0.1	3.2	4.2	4.7	131.6
property4_3_4	UNSAT	0.2	0.2	3.0	4.0	6.1	85.9
property4_3_5	UNSAT	1.4	1.1	4.2	5.3	6.3	233.2

Instance	Result	ReluVal	Adv	NonOpt	Joint	BaB	Reluplex
property4_3_6	UNSAT	1.4	1.3	3.7	4.8	6.3	161.4
property4_3_7	UNSAT	0.4	0.3	2.6	2.8	5.0	346.1
property4_3_8	UNSAT	0.3	0.3	6.6	7.7	44.7	148.9
property4_3_9	UNSAT	1.5	1.5	3.8	3.9	5.8	703.1
property4_4_1	UNSAT	3.2	3.1	3.4	4.3	5.7	55.8
property4_4_2	UNSAT	2.3	2.2	4.0	4.5	6.0	277.5
property4_4_3	UNSAT	1.3	1.3	4.0	4.6	5.9	277.7
property4_4_4	UNSAT	1.5	1.5	7.2	8.5	48.9	235.8
property4_4_5	UNSAT	1.5	1.6	3.7	4.7	6.7	246.1
property4_4_6	UNSAT	0.1	0.1	3.4	4.5	5.7	200.5
property4_4_7	UNSAT	0.2	0.2	2.6	3.7	5.4	53.9
property4_4_8	UNSAT	0.2	0.2	3.4	3.7	6.0	221.5
property4_4_9	UNSAT	0.1	0.1	4.0	4.5	6.2	480.2
property4_5_1	UNSAT	2.9	2.9	3.9	4.9	6.2	583.3
property4_5_2	UNSAT	0.7	0.6	3.8	4.7	5.5	239.0
property4_5_3	UNSAT	0.2	0.2	3.6	4.6	5.5	141.4
property4_5_4	UNSAT	0.2	0.2	2.9	3.9	5.0	164.5
property4_5_5	UNSAT	0.6	0.6	3.7	4.8	5.9	130.2
property4_5_6	UNSAT	0.4	0.4	3.1	3.3	5.7	182.7
property4_5_7	UNSAT	0.1	0.1	2.4	3.4	5.4	44.2
property4_5_8	UNSAT	0.1	0.1	3.2	4.3	5.7	126.3
property4_5_9	UNSAT	0.2	0.1	2.5	3.5	5.4	133.1
property8	SAT	2939.7	74.5	22.9	1.2	20.3	timelimit
shifted geo. mean		5.7	3.1	34.1	34.1	67.5	669.0

Table 9.8: Runtimes on ACAS properties. In adversary check mode, Reluval fails on property2_5_3 and reports UNSAT instead of SAT. We set the runtime to 7200 s for the mean computation in this case. We also assume a runtime of 7200 s for those three cases, for which Reluplex reports counterexamples that are not even valid with a numerical tolerance of 10^{-3}. These are denoted as "wrong" in the table.

Table 9.8 contains the runtime results for all ACAS instances of Properties 1, 2, 3, 4, and 8 as defined by Katz et al. [35]. It should be noted that these are all disjunction instances. The mean runtime of Reluplex is at least one order of magnitude larger than the mean runtimes of all other solvers. ReluVal is clearly superior to all other solvers, especially if its adversary check mode is applied. We see that our solving model, which uses SCIP to strongly integrate the bound computations into a MIP framework, performs significantly better than the rather similar approach of Bunel et al. [12].

Instance	Result	Reluval	Adv	NonOpt	Joint	Separate	BaB	Reluplex
property5	UNSAT	12.3	9.1	3922.2	timelimit	4104.8	timelimit	4883.4
property6a	UNSAT	3.3	3.2	7078.8	6248.2	3745.0	timelimit	timelimit
property6b	UNSAT	1.6	1.5	5935.9	4207.5	6179.1	timelimit	timelimit
property7	SAT	memlimit	1099.1	timelimit	3012.7	3012.7	timelimit	timelimit
property9	UNSAT	411.5	184.5	5672.0	timelimit	5921.1	4214.8	timelimit
property10	UNSAT	1.3	1.1	timelimit	3268.3	5335.3	4567.3	5516.3
sh. geo. mean		60.0	33.6	6053.9	4875.5	4564.6	6107.5	6456.0

Table 9.9: Runtimes on ACAS properties. If Reluval is run in normal check mode, it terminates prematurely on Property 7 due to a lack of memory. We use the time limit of 7200 seconds for the mean computation in this case.

A similar picture of the performance of the various solvers can be seen in

Table 9.9, which shows the results on the ACAS instances of Properties 5, 6, 7, 9, and 10. Especially here it should be noted that we underestimate the runtime of instances for which the solving process is stopped due to the time limit. Also, remind that Property 7 is neither a conjunction nor a disjunction instance directly as we commented in Section 8.2. Therefore, we do have to split the instance to be solvable with our implementation. The corresponding runtime for solving method Joint in Table 9.9 is in fact obtained with the method Separate. We include the value also for method Joint, since this allows the computation of a meaningful mean runtime.

In Tables 9.10 and 9.11 we report runtimes for the MNIST instances. We remind that all of these are conjunction instances by definition. For our solving model we use the configuration "mnist_base" as presented in Section 9.2 and adapt it by two small changes. We enable the local primal heuristic, however it is only executed at the root node, so that the primal heuristic is executed twice at the root node. This is achieved by setting "sampling_heuristic_local_max_iter" to 1000 and "sampling_heuristic_local_freq" to 10. In fact, the configuration "no_heur_relu_genv" would also be a good choice. Using that configuration, the instance mnist_512_image11_5 is solved clearly within the time limit which is not the case if if the configuration "mnist_base" is used. Though, the latter configuration (with the additional execution of the primal heuristic as described above), leads to a lower mean runtime on the MNIST instances with 512 neurons per layer. For that reason, we present the results which are obtained with the configuration "mnist_base".

The instances which are based on neural networks with two layers of 24 neurons are solved very quickly by most solvers, as can be seen in Table 9.10. Though, domain branching is apparently not a good strategy to solve these instances, which feature 784 input neurons. This is shown by the high number of timeouts of the solving method BaB of Bunel et al. [12] on these MNIST instances (see Table 9.10).

Clearly, the MNIST instances with 512 neurons per layer are much more challenging. In Table 9.11 we see that our solving model performs quite good when the approach *Joint* is taken, i.e. conjunction instances are solved directly as one optimization problem. In several cases Neurify aborts the solving process with no result. For the corresponding instances we assume the time limit of 7200 seconds as runtime in order to compute the mean runtime. If the conjunction instances are split into separate disjunction instances, our solving model mostly fails to find counterexamples within the time limit (see *NonOpt* and *Separate*). This can be explained by the fact that the instances, which are obtained after splitting, are solved sequentially one after the other. However, if a verifiable instances is processed first, this may already lead to a timeout. Reluplex is only able to solve two of the easiest instances within the time limit, and the branch-and-bound method of Bunel et al. [12] also performs considerably worse than Neurify and our solving model. Though for all MNIST instances, it should be noted that the counterexamples which are produced by Neurify are only feasible if one applies a large numerical tolerance of 10^{-3}. It is clear to see, that the instances with the lowest perturbation radius of 1 are easy to verify. On the other hand, for the instances with a high perturbation radius of 10 or 20, counterexamples are found in most cases. Apparently, the

Instance	Result	Neurify	Adv	NonOpt	Joint	Separate	BaB	Reluplex
mnist_24_image11_1	UNSAT	0.4	0.4	10.6	8.1	28.0	25.7	0.5
mnist_24_image11_10	UNSAT	2.0	2.1	31.0	15.5	35.3	timelimit	4824.5
mnist_24_image11_20	SAT	0.4	0.5	6.0	7.5	7.8	timelimit	3437.6
mnist_24_image11_5	UNSAT	0.4	0.4	860.7	6.1	31.6	22.4	189.9
mnist_24_image2_1	UNSAT	0.4	0.4	25.1	3.8	27.1	21.5	1.6
mnist_24_image2_10	SAT	0.4	0.5	2.6	2.1	2.0	timelimit	24.7
mnist_24_image2_20	SAT	0.4	0.5	2.7	2.0	2.0	timelimit	19.2
mnist_24_image2_5	UNSAT	0.4	0.4	31.6	13.6	36.8	timelimit	timelimit
mnist_24_image4_1	UNSAT	0.4	0.6	12.4	2.5	25.8	22.3	0.9
mnist_24_image4_10	SAT	0.4	0.5	7.9	2.0	2.1	timelimit	70.5
mnist_24_image4_20	SAT	0.5	0.6	2.7	2.3	5.3	timelimit	11.9
mnist_24_image4_5	SAT	0.4	0.5	23.1	2.0	24.0	timelimit	536.1
shifted geo. mean		0.4	0.5	18.8	4.4	14.1	1006.3	86.1

Table 9.10: Runtimes on the MNIST 24 data set. In contrast to all other solvers, Neurify reports SAT for instance mnist_24_image11_10. However, all counterexamples that Neurify produces, are only valid when applying a large numerical tolerance of 10^{-3}. We regard the instance mnist_24_image11_10 as verifiable and exclude it from the mean computation. Counterexamples produced by Reluplex are valid with a numerical tolerance of 10^{-5}.

perturbation radius 5 poses the biggest difficulties for the solvers, as the corresponding instances are probably on the borderline between SAT and UNSAT.

Instance	Result	Neurify	Adv	NonOpt	Joint	Separate	BaB	Reluplex
mnist_512_image11_1	UNSAT	0.2	0.5	80.0	39.9	236.3	4446.4	371.6
mnist_512_image11_10	-	no result	no result	timelimit	timelimit	timelimit	timelimit	timelimit
mnist_512_image11_20	SAT	0.6	0.8	timelimit	28.6	timelimit	timelimit	timelimit
mnist_512_image11_5	UNSAT	0.2	0.5	timelimit	timelimit	timelimit	4908.0	timelimit
mnist_512_image2_1	UNSAT	0.5	0.5	92.0	97.4	498.6	4854.3	timelimit
mnist_512_image2_10	SAT	no result	no result	timelimit	30.9	timelimit	timelimit	timelimit
mnist_512_image2_20	SAT	0.4	0.7	timelimit	12.4	56.2	timelimit	timelimit
mnist_512_image2_5	-	no result	no result	timelimit	timelimit	timelimit	timelimit	timelimit
mnist_512_image4_1	UNSAT	0.3	0.5	84.3	32.8	290.4	4512.2	1132.2
mnist_512_image4_10	SAT	no result	no result	timelimit	22.9	timelimit	timelimit	timelimit
mnist_512_image4_20	SAT	0.4	0.6	timelimit	21.6	122.3	timelimit	timelimit
mnist_512_image4_5	-	no result	no result	timelimit	timelimit	timelimit	timelimit	timelimit
shifted geo. mean		148.5	150.4	2434.9	220.7	1612.8	6235.3	4830.5

Table 9.11: Runtimes on the MNIST 512 data set. The instances for which Neurify returns no result are treated as time limit instances in the mean computation. Counterexamples of Neurify are only valid when applying a large numerical tolerance of 10^{-3}.

Eventually, we present runtime results for the ACAS based instances which we defined as part of this thesis (cf. Section 8.2 and Appendix A). As explained before, these instances cannot be processed by the other solvers in our computational study. Therefore, we only report the results of our solving model using the configuration "heur_sepa0_freq5" and both the formulation as optimization problem, and as feasibility problem. Some instances cannot be solved directly with the formulation as feasibility problem and are therefore left out in that case. Regarding the other instances, the performance of both formulations is

quite similar. See Table 9.12 for the results which are obtained with the formulation as optimization problem, and Table 9.13 for the results corresponding to the formulation as feasibility problem.

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
1_1_int_away	0.0057	0.0057	229	UNSAT	optimal	415.6
1_1_int_away2	0.015	0.015	403	UNSAT	optimal	560.7
1_1_lin_opp	0.91	0.91	315	UNSAT	optimal	1316.3
1_1_lin_opp2	0.6	0.6	249	UNSAT	optimal	983.4
1_1_lin_opp2_dir	0.6	0.6	583	UNSAT	optimal	2404.9
1_1_lin_opp_dir	0.69	0.69	543	UNSAT	optimal	2488.6
1_2_int_away	-1.7e+04	-0.11	2	SAT	bound	23
1_2_int_away2	-1.7e+04	-0.1	2	SAT	bound	22.3
2_1_var_dist	0.13	0.16	483	UNSAT	bound	1679.2
2_2_lin_opp	-2.5e+06	-2.8	1	SAT	bound	1.7
2_2_lin_opp2	-4.3e+02	-0.21	41	SAT	bound	537.9
3_1_var_dist	0.12	0.26	285	UNSAT	bound	629.2

Table 9.12: Linear ACAS instances run with our model using the formulation as optimization problem. In the column status, "optimal" means that the problem was solved to optimality, while "bound" implies that the solving process was interrupted due to a positive dual or negative primal bound.

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
1_1_int_away	-	-	227	UNSAT	infeasible	395.3
1_1_int_away2	-	-	515	UNSAT	infeasible	634.2
1_1_lin_opp	-	-	313	UNSAT	infeasible	1217.3
1_1_lin_opp2	-	-	247	UNSAT	infeasible	934.5
1_1_lin_opp2_dir	-	-	583	UNSAT	infeasible	2315.9
1_1_lin_opp_dir	-	-	543	UNSAT	infeasible	2410.8
1_2_int_away	-	-	2	SAT	optimal	17.7
1_2_int_away2	-	-	2	SAT	optimal	17.9
2_2_lin_opp	-	-	1	SAT	optimal	5.5
2_2_lin_opp2	-	-	297	SAT	optimal	1763.2

Table 9.13: Linear ACAS instances run with our model using the formulation as feasibility problem. Using this formulation, there are no meaningful primal and dual bounds. The solution status shows that either infeasibility is detected, or a feasible, i.e. optimal, solution is found.

Summing up, our solving model shows a solid performance in all categories of our benchmark set. This highlights the success of our approach, which combines MIP solving, using the solver SCIP [25], with specialized bound computation and branching techniques. Especially, we can solve instances with general polytopes as input domains which is not possible with the algorithm of Wang et al. [56]. Moreover, our solving model clearly outperforms the solvers of Bunel et al. [12] and Katz et al. [35]. Subsequently, Reluplex cannot be regarded as a state-of-the-art solver for neural network verification anymore. While ReluVal and Neurify from Wang et al. [56] show impressive runtime results for most instances, they rely primarily on branching. For big instances, this can become problematic due to limited numerical accuracy and memory capacity.

10 Conclusions and Future Work

In this thesis we give a profound overview of the field of neural network verification. While we strongly focus on fully connected neural networks with ReLU activation function, many results and techniques can be applied similarly to neural networks of other architectures. For example, this includes the leaky ReLU activation function, dropout and max-pooling layers. The thesis features a theoretical and empirical comparison of various approximation methods for ReLU neural networks. These enable the computation of lower and upper bounds for each neuron in a neural network. Such bounds are necessary for the succesful verification of properties of neural networks, and the quality of the bounds heavily impacts the performance of verification algorithms. Hence, we develop a theoretical framework for the comparison of linear approximation methods. We also present a novel approximation technique which is able to improve the linear relaxation of a ReLU neural network. Besides that, we show how the local search procedure of Dutta et al. [16] can be implemented differently in an MIP solving context such that it serves as a primal heuristic. Additionally, we describe a novel formulation of the verification problem as quadratic program. The newly proposed techniques are also evaluated computationally within our newly implemented solving model.

Eventually, we conduct extensive computational experiments using various solvers on a diverse set of test instances. These experiments help to gain an understanding of the performance and efficiency of various solving techniques on different types of test instances. Especially, we compare the programs of Bunel et al. [12], Wang et al. [56] and Katz et al. [35] with our own solving model. We use a benchmark set for verification of neural networks as introduced by Katz et al. [35]. The corresponding instances are based on the ACAS Xu system for aircraft collision avoidance and have five input neurons. Furthermore, we investigate test instances based on the MNIST data set [37], using neural networks of Wang et al. [56]. In contrast to the instances of Katz et al. [35], these have a significantly higher number of 784 input neurons. Moreover, we define some additional instances based on the ACAS Xu system, which do not have independent input bounds. These instances can only be solved by our solving model, as the other solvers support only instances with independent bounds for all input neurons.

Currently, only neural networks with ReLU activation function can be processed by our solver. Especially, our solving model shows the potential of integrating an MIP model of the verification problem with specialized solving techniques such as bound computation methods and branching rules. For the implementation we use the academic MIP solver SCIP [25] and its Python interface PySCIPopt [39]. In fact, we provide the first available solver for instances of the verification problem that do not have independent bounds for each input neuron. Our solving model is not only applied successfully to such instances but also to benchmark instances from the literature as mentioned above. On these instances, it is only outperformed by the programs of Wang

et al. [56], which are faster by one or two orders of magnitude. However, due to better approximation techniques, in our model branching is performed quite rarely compared to the approach of Wang et al. [56]. In comparison to the solvers of Katz et al. [35] and Bunel et al. [12], our implementation is clearly superior; partially it is faster by more than one order of magnitude. Moreover, it is publicly available at `https://github.com/roessig/verify-nn`.

Although our implementation is limited to neural networks with ReLU activation function, the approach could be easily extended to work with other piecewise-linear activation functions. Examples for that are the leaky ReLU function or max-pooling layers. In addition, the great flexibility of our solving model allows the integration of further techniques such that future improvements may render it even more efficient. For future work, it would be highly interesting to integrate the SDP relaxation of Raghunathan et al. [43] into our solving model, which should be possible using the SCIP plugin SCIP-SDP [22]. Furthermore, the solving approach of Wang et al. [56] could be integrated deeply with our model using an optimized implementation of their bound computation approach. Eventually, the intelligent combination of the various techniques which are already available in our solving model or mentioned here could lead to significant improvements of the solving performance.

In conclusion, it can be said that many challenges remain in the field of neural network verification. The further scalability of current verification approaches is still an open task. Besides, new approaches for the falsification of incorrect properties would be highly interesting. Although we present a new heuristic for that (based on ideas of Dutta et al. [16]), better algorithms could probably be developed. Eventually, it is the aim of this thesis to constitute a solid foundation for the interested reader to pursue these questions.

A Definitions of Additional Properties on ACAS Neural Networks

Here we provide the formal definitions of our properties on the ACAS neural networks. The numbers i_j in the beginning of each name refer to the neural network which is used (cf. Table 8.2 in Chapter 8).

The input constraints for (i) `1_1_lin_opp` and (ii) `1_1_lin_opp2` are given as follows, where either constraint (i) or (ii) is used. Desired output: COC should not be minimal.

$$
\begin{aligned}
1000 \leq \rho \leq 2000 & \\
-3.141593 \leq \theta \leq 0 & \quad \text{(i)} \\
0 \leq \theta \leq 3.141593 & \quad \text{(ii)} \\
-3.141593 \leq \psi \leq 3.141593 & \\
1000 \leq v_{\text{own}} \leq 1200 & \\
800 \leq v_{\text{int}} \leq 1200 & \\
\theta = -\psi & \\
-100 \leq v_{\text{own}} - v_{\text{int}} \leq 100 &
\end{aligned}
$$

Input constraints for (i) `2_2_lin_opp` and (ii) `2_2_lin_opp2`, where either constraint (i) or (ii) is used. Desired output: COC should not be minimal.

$$
\begin{aligned}
1000 \leq \rho \leq 2000 & \\
0 \leq \theta \leq 3.141593 & \quad \text{(i)} \\
-3.141593 \leq \theta \leq 0 & \quad \text{(ii)} \\
-3.141593 \leq \psi \leq 3.141593 & \\
100 \leq v_{\text{own}} \leq 1200 & \\
0 \leq v_{\text{int}} \leq 1200 & \\
\theta = -\psi & \\
v_{\text{own}} = v_{\text{int}} &
\end{aligned}
$$

Input constraints for (i) `1_1_lin_opp_dir` and (ii) `1_1_lin_opp2_dir`, where either constraint (i) or (ii) is used. Desired output: COC should not be minimal.

$$
\begin{aligned}
1000 \leq \rho \leq 2500 & \\
-3.141593 \leq \theta \leq 0 & \quad \text{(i)} \\
0 \leq \theta \leq 3.141593 & \quad \text{(ii)} \\
-3.141593 \leq \psi \leq 3.141593 & \\
800 \leq v_{\text{own}} \leq 1200 & \\
600 \leq v_{\text{int}} \leq 1200 & \\
\theta = -\psi & \\
v_{\text{own}} = v_{\text{int}} &
\end{aligned}
$$

Input constraints for (i) `1_1_int_away` and (ii) `1_1_int_away2`. Desired output: (i) strong left is not minimal, or (ii) strong right is not minimal.

$$
\begin{aligned}
5000 \leq \rho \leq 6000 \\
-3.141593 \leq \theta \leq 3.141593 \\
-3.141593 \leq \psi \leq 3.141593 \\
1000 \leq v_{\text{own}} \leq 1200 \\
500 \leq v_{\text{int}} \leq 1200 \\
v_{\text{int}} \geq v_{\text{own}} + 100 \\
-0.392699 \leq \psi - \theta \leq 0.392699
\end{aligned}
$$

Input constraints for (i) `1_2_int_away` and (ii) `1_2_int_away2`. Desired output: (i) strong left is not minimal, or (ii) strong right is not minimal.

$$
\begin{aligned}
5000 \leq \rho \leq 7000 \\
-3.141593 \leq \theta \leq 3.141593 \\
-3.141593 \leq \psi \leq 3.141593 \\
500 \leq v_{\text{own}} \leq 1200 \\
500 \leq v_{\text{int}} \leq 1200 \\
v_{\text{int}} \geq v_{\text{own}} + 100 \\
-0.392699 \leq \psi - \theta \leq 0.392699
\end{aligned}
$$

Input constraints for `2_1_var_dist` and `3_1_var_dist`. Desired output: COC is minimal.

$$
\begin{aligned}
10000 \leq \rho \leq 60760 \\
-0.141593 \leq \theta \leq 0.141593 \\
-0.141593 \leq \psi \leq 0.141593 \\
300 \leq v_{\text{own}} \leq 1200 \\
0 \leq v_{\text{int}} \leq 1200 \\
v_{\text{int}} \geq v_{\text{own}} + 601 - 0.01\rho
\end{aligned}
$$

B Computational Results for Various Components

In each of the following sections we report the results of a computational experiment which is focused on one special technique or concept we presented in this thesis. Like the experiments in Section 9.2, each experiment is run on a cluster node with Intel Xeon CPU E5-2670 at a clock rate of 2.5 GHz. Moreover, a memory limit of 32 GB is set. Here, we present computational results with respect to four different topics: various orders in which the LPs for OBBT are solved (cf. Section 4.3), the primal heuristic as presented in Chapter 5, the quadratic programming formulation of the verification problem (cf. Section 3.3), and solving the verification problem as a feasibility problem (cf. Section 3.1).

B.1 Ordering strategies for LP solving in OBBT

We compare the different ordering strategies for the LP solves during the execution of OBBT as proposed by Gleixner et al. [24] and presented in Section 4.3. The runtimes in Table B.1 were obtained without the local use of our primal heuristic, as opposed to the runtimes in Table B.2. Both contain the number of solving nodes and the runtimes on all instances of our UNSAT evaluation set. Our standard ordering as explained in Section 4.3 is indicated as "base". Indeed, this ordering is the fastest according to the shifted geometric mean (using shift value 10) which we compute over all instances. Hence, it is apparently not useful to apply the greedy or the min-max strategy. We do not report mean values for the number of solving nodes as this is not meaningful if the solving process is aborted in some cases due to the time limit. See Chapter 9 for some more details on the use of the shifted geometric mean and the execution of experiments. As commented there, we assume 7200 seconds for the runtime if the time limit of two hours is hit during execution.

Instance	Nodes			Time		
	base	greedy	min-max	base	greedy	min-max
lin_acas_1_1_int_away	229.0	223.0	223.0	523.9	766.7	869.0
lin_acas_1_1_lin_opp2	251.0	211.0	211.0	1235	1100.5	1179.0
lin_acas_1_1_lin_opp_dir	547.0	369.0	369.0	3253.1	2344.8	2510.7
lin_acas_3_1_var_dist	299.0	211.0	211.0	754.6	695.8	793.0
mnist_24_image11_5	-	-	-	timelimit	timelimit	timelimit
mnist_24_image2_5	-	-	-	timelimit	timelimit	timelimit
mnist_24_image4_1	1.0	1.0	1.0	8.8	8.0	6.3
mnist_512_image11_5	-	-	-	timelimit	timelimit	timelimit
mnist_512_image2_1	2.0	2.0	2.0	418.3	345.5	355.1
property10_property	909.0	909.0	909.0	4506.9	5365.7	5971.8
property1_1_1	11.0	11.0	11.0	127.1	125.8	123.1
property1_2_2	105.0	105.0	105.0	1615	1695.9	1674.8
property2_3_3	2349.0	-	-	7038.7	timelimit	timelimit
property2_4_2	-	-	-	timelimit	timelimit	timelimit
property3_4_3	13.0	13.0	13.0	28.4	38.0	44.0
property3_4_4	1.0	1.0	1.0	4.7	8.3	6.8
property4_2_2	3.0	3.0	3.0	13.4	15.4	18.1
property4_3_7	1.0	1.0	1.0	4.8	6.8	6.0
property5_property	-	-	-	timelimit	timelimit	timelimit
property6_6a_property_3	1185.0	-	-	5802.2	timelimit	timelimit
property6_6b_property_1	317.0	317.0	317.0	2101.9	2498.8	2756.9
property9_property_0	463.0	463.0	463.0	2301.2	2574.9	2834.7
property9_property_4	783.0	783.0	783.0	3447.8	3996.8	4442.7
shifted geo. mean				854.8	899.6	930.7

Table B.1: Evaluation of ordering strategies without local heuristic.

Instance	Nodes			Time		
	base	greedy	min-max	base	greedy	min-max
lin_acas_1_1_int_away	131.0	137.0	125.0	476.7	623.6	656.7
lin_acas_1_1_lin_opp2	251.0	211.0	211.0	1466.3	1300.2	1470.5
lin_acas_1_1_lin_opp_dir	519.0	355.0	355.0	3536.2	2568.6	2875.3
lin_acas_3_1_var_dist	299.0	211.0	211.0	995.3	878.9	972.9
mnist_24_image11_5	8191.0	8191.0	8191.0	5408.6	6117.2	5956.2
mnist_24_image2_5	-	-	-	timelimit	timelimit	timelimit
mnist_24_image4_1	1.0	1.0	1.0	9.7	10.4	8.8
mnist_512_image11_5	-	-	-	timelimit	timelimit	timelimit
mnist_512_image2_1	2.0	2.0	2.0	423.6	355.2	365.3
property10_property	865.0	865.0	865.0	4889.3	5690.1	6279
property1_1_1	11.0	11.0	11.0	138.6	134.1	135.2
property1_2_2	105.0	105.0	105.0	1724.9	1783	1816.8
property2_3_3	1101.0	1069.0	1039.0	3710.4	4605.6	5105
property2_4_2	-	-	-	timelimit	timelimit	timelimit
property3_4_3	13.0	13.0	13.0	39.2	54.7	58.8
property3_4_4	1.0	1.0	1.0	7.1	9.9	8.9
property4_2_2	3.0	3.0	3.0	16.8	18.7	22.7
property4_3_7	1.0	1.0	1.0	6.8	6.4	8.1
property5_property	-	-	-	timelimit	timelimit	timelimit
property6_6a_property_3	-	-	-	timelimit	timelimit	timelimit
property6_6b_property_1	785.0	645.0	622.0	3609.6	4190.2	4371.9
property9_property_0	463.0	463.0	463.0	2577.1	3023.2	3134.9
property9_property_4	723.0	723.0	723.0	3594.5	4084.3	4694.4
shifted geo. mean				907.5	946.9	989.1

Table B.2: Evaluation of ordering strategies with local heuristic employed.

B.2 Primal heuristics

In this section we present computational results regarding our primal heuristic as laid out in Chapter 5. For all tables, the configuration "base" indicates that our heuristic, as described in Chapter 5, is run both at the root node and locally at each solving node up to a tree depth of eight. We compare this to running the heuristic only at the root node ("root node") or not at all ("none") in Table B.5. Furthermore, we investigate how our LP based heuristic performs compared to using only the sampling heuristic ("no_lp"). The better performance of the LP based heuristic on our evaluation set of SAT instances can be seen in Table B.3. In Table B.4 we report how the LP based heuristic affects the runtime on UNSAT instances.

The number of solving nodes and the runtimes in all tables are obtained as the arithmetic mean over several runs (see table captions). Since the values between different runs do not differ much in general, it is admissible to use the arithmetic mean. We perform several runs in order to obtain reliable results, although the performance of the heuristics depends on randomly generated values. In each table, the column "Timeouts" shows, how many of these runs were interrupted as the time limit of two hours was reached.

Instance	Nodes		Timeouts		Time	
	base	no_lp	base	no_lp	base	no_lp
lin_acas_1_2_int_away2	2.0	2.0	0	0	34.4	29.5
lin_acas_2_2_lin_opp	1.0	1.0	0	0	6.1	3.8
lin_acas_2_2_lin_opp2	43.0	186.0	0	0	826.8	2436.9
mnist_24_image2_20	1.0	1.0	0	0	8.5	7.9
mnist_24_image4_5	1.0	405.0	0	0	9.0	831.2
mnist_512_image11_20	1.0	-	0	5	40.8	timelimit
mnist_512_image4_20	1.0	-	0	5	35.3	timelimit
property2_2_2	1.0	1.0	0	0	5.5	2.9
property2_3_7	1.0	1.0	0	0	5.1	4.8
property2_5_3	8.2	1250.0	0	0	109.3	2294.2
property7_property_3	85.0	-	0	4	4514.0	5773.9
property7_property_4	-	-	5	5	timelimit	timelimit
property8_property	1.0	2.6	0	0	6.5	32.6

Table B.3: Computational results for the LP based heuristic, as described in Chapter 5 ("base"), compared to using only the sampling heuristic ("no_lp"). The results are obtained as the arithmetic mean over five runs on our evaluation set of SAT instances. The shifted geometric mean of the runtimes over all instances is 71.7 s for the "base" configuration, which includes our LP heuristic, and 330.1 s for the configuration "no_lp" without the LP heuristic.

Instance	Nodes		Timeouts		Time	
	base	no_lp	base	no_lp	base	no_lp
lin_acas_1_1_int_away	131.0	143.0	0	0	491.6	511.6
lin_acas_1_1_lin_opp2	251.0	251.0	0	0	1526.7	1544.0
lin_acas_1_1_lin_opp_dir	509.7	512.7	0	0	3672.2	3685.8
lin_acas_3_1_var_dist	299.7	299.0	0	0	1116.6	1024.9
mnist_24_image11_5	8191.0	8191.0	0	0	5504.4	5710.6
mnist_24_image2_5	-	-	3	3	timelimit	timelimit
mnist_24_image4_1	1.0	1.0	0	0	12.7	11.3
mnist_512_image11_5	-	-	3	3	timelimit	timelimit
mnist_512_image2_1	2.0	2.0	0	0	427.8	418.9
property10_property	865.0	865.0	0	0	4932.2	4756.9
property1_1_1	11.0	11.0	0	0	140.4	140.7
property1_2_2	105.0	105.0	0	0	1772.1	1717.9
property2_3_3	1143.6	1116.3	0	0	3800.8	3745.8
property2_4_2	-	-	3	3	timelimit	timelimit
property3_4_3	13.0	13.0	0	0	45.7	39.9
property3_4_4	1.0	1.0	0	0	9.4	5.9
property4_2_2	3.0	3.0	0	0	18.9	15.4
property4_3_7	1.0	1.0	0	0	7.3	6.2
property5_property	-	-	3	3	timelimit	timelimit
property6_6a_property_3	-	-	3	3	timelimit	timelimit
property6_6b_property_1	842.7	780.3	0	0	3775.7	3642.4
property9_property_0	463.0	463.0	0	0	2685.5	2559.3
property9_property_4	723.7	728.3	0	0	3720.4	3710.9

Table B.4: The runtimes and numbers of solving nodes are given as the arithmetic mean over three runs on our evaluation set of UNSAT instances. The shifted geometric mean of the runtimes over all instances is 945.9 s for the "base" configuration, which includes our LP heuristic, and 915.3 s for the configuration "no_lp" without the LP heuristic. That means, the runtime increases only slightly due to the application of our heuristic.

Instance	Nodes			Timeouts			Time		
	base	root node	none	base	root node	none	base	root node	none
lin_acas_1_2_int_away2	2.0	-	-	0	3	3	30.4	timelimit	timelimit
lin_acas_2_2_lin_opp	1.0	1.0	-	0	0	3	3.3	3.8	timelimit
lin_acas_2_2_lin_opp2	43.0	-	1177.0	0	1	0	808.4	6825.9	7056.3
mnist_24_image2_20	1.0	1.0	421.0	0	0	0	6.0	6.7	1162.3
mnist_24_image4_5	1.0	1.0	-	0	0	3	6.3	7.9	timelimit
mnist_512_image11_20	1.0	1.0	-	0	0	3	39.4	46.4	timelimit
mnist_512_image4_20	1.0	1.0	-	0	0	3	33.0	35.6	timelimit
property2_2_2	1.0	1.0	-	0	0	3	2.7	3.7	timelimit
property2_3_7	1.0	1.0	-	0	0	3	2.6	3.2	timelimit
property2_5_3	5.7	977.0	2066.0	0	0	0	66.7	1677.3	2997.1
property7_property_3	77.0	-	-	0	3	3	4042.5	timelimit	timelimit
property7_property_4	-	-	-	3	3	3	timelimit	timelimit	timelimit
property8_property	1.0	1.0	-	0	0	3	3.8	5.0	timelimit

Table B.5: The results are obtained as arithmetic mean over three runs on our evaluation set of SAT instances. It is obvious that without application of any specialized heuristic, SAT instances can hardly be solved.

B.3 Quadratic programming formulation

In this section we present the computational results of our implementation of the quadratic programming model for verification of neural networks. It should be noticed that we use a different SCIP setup for the results in this section. For nonlinear programming problems, it is beneficial to compile SCIP with Ipopt (cf. Wächter and Biegler [54]), which is a software package for nonlinear optimization. The results are therefore obtained with SCIP 6.0.0 using Ipopt 3.12.5, and Soplex 4.0.0 as LP solver. We evaluate the quadratic programming model on the disjunction instances from the UNSAT and SAT evaluation sets. Additionally, we split the instances of the MNIST 24 test set as described in Remark 7 and evaluate the quadratic programming model on these. In contrast to the other experiments in this thesis, we use a time limit of one hour since very few instances can be solved within a reasonable amount of time.

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
linear_acas_1_1_int_away	-7.9e+09	-	1	-	-	timelimit
linear_acas_1_1_lin_opp2	-5.7e+09	-	1	-	-	timelimit
linear_acas_1_1_lin_opp_dir	-5.1e+09	-	1	-	-	timelimit
property1_1_1	-3.1e+09	-	1	-	-	timelimit
property1_2_2	-3.8e+09	-	1	-	-	timelimit
property2_3_3	-4.1e+09	-	1	-	-	timelimit
property2_4_2	-2.1e+09	-	94	-	-	timelimit
property3_4_3	-3.7e+07	-	55	-	-	timelimit
property3_4_4	-9.3e+07	-	1	-	-	timelimit
property4_2_2	-5.2e+06	-	108	-	-	timelimit
property4_3_7	-1.7e+09	-	138	-	-	timelimit
property6_6a_property_3	-3.2e+09	-	1	-	-	timelimit
property6_6b_property_1	-4.4e+09	-	1	-	-	timelimit
property9_property_0	-1.3e+08	-	1	-	-	timelimit
property9_property_4	-1.3e+08	-	1	-	-	timelimit

Table B.6: Computational results for all disjunction instances of our UNSAT evaluation set with time limit of one hour.

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
linear_acas_1_2_int_away2	-5.3e+09	-	1	-	-	timelimit
linear_acas_2_2_lin_opp	-7.6e+09	-	1	-	-	timelimit
linear_acas_2_2_lin_opp2	-8.2e+09	-	1	-	-	timelimit
property2_2_2	-3.4e+09	-	1	-	-	timelimit
property2_3_7	-1.6e+13	-	5929	-	-	timelimit
property2_5_3	-4.9e+09	-	1	-	-	timelimit
property7_property_3	-4.6e+08	-	1	-	-	timelimit
property7_property_4	-4.6e+08	-	1	-	-	timelimit
property8_property	-2.7e+11	-	965	-	-	timelimit

Table B.7: Computational results for all disjunction instances of our SAT evaluation set with time limit of one hour.

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
mnist_24_image11_10_0	-4.7e+07	-	537	-	-	timelimit
mnist_24_image11_10_1	-4.7e+07	7.1e+05	444	-	-	timelimit
mnist_24_image11_1_0	-	-	0	UNSAT	infeasible	1.4
mnist_24_image11_1_1	-	-	0	UNSAT	infeasible	1.4
mnist_24_image11_1_2	-	-	0	UNSAT	infeasible	1.5
mnist_24_image11_1_3	-	-	0	UNSAT	infeasible	1.4
mnist_24_image11_1_4	-	-	0	UNSAT	infeasible	1.6
mnist_24_image11_1_5	-	-	0	UNSAT	infeasible	1.6
mnist_24_image11_1_7	-	-	0	UNSAT	infeasible	1.6
mnist_24_image11_1_8	-	-	0	UNSAT	infeasible	1.4
mnist_24_image11_1_9	-	-	0	UNSAT	infeasible	1.6
mnist_24_image11_20_0	-1.8e+08	-	1	-	-	timelimit
mnist_24_image11_20_1	-1.7e+08	-	130	-	-	timelimit
mnist_24_image11_5_0	-1.1e+07	2e+06	540	-	-	timelimit
mnist_24_image11_5_1	-8.3e+06	-	593	-	-	timelimit
mnist_24_image2_10_0	-5.5e+07	-	1	-	-	timelimit
mnist_24_image2_10_2	-5.6e+07	-	1	-	-	timelimit
mnist_24_image2_1_0	-	-	0	UNSAT	infeasible	1.5
mnist_24_image2_1_2	-	-	1	UNSAT	infeasible	96.0
mnist_24_image2_1_3	-	-	0	UNSAT	infeasible	1.4
mnist_24_image2_1_4	-	-	0	UNSAT	infeasible	1.5
mnist_24_image2_1_5	-	-	1	UNSAT	infeasible	114.5
mnist_24_image2_1_6	-	-	0	UNSAT	infeasible	1.4
mnist_24_image2_1_7	-	-	1	UNSAT	infeasible	97.2
mnist_24_image2_1_8	-	-	1	UNSAT	infeasible	96.5
mnist_24_image2_1_9	-	-	0	UNSAT	infeasible	1.5
mnist_24_image2_20_0	-1.7e+08	-	1	-	-	timelimit
mnist_24_image2_5_0	-1.5e+07	-	460	-	-	timelimit
mnist_24_image2_5_2	-1.6e+07	-	600	-	-	timelimit
mnist_24_image4_10_0	-5.4e+07	0.0	1	SAT	bound	1074.0
mnist_24_image4_10_1	-5.4e+07	-	464	-	-	timelimit
mnist_24_image4_10_2	-5.5e+07	-	1	-	-	timelimit
mnist_24_image4_1_0	-	-	0	UNSAT	infeasible	1.6
mnist_24_image4_1_1	-	-	0	UNSAT	infeasible	1.5
mnist_24_image4_1_2	-	-	0	UNSAT	infeasible	1.5
mnist_24_image4_1_3	-	-	0	UNSAT	infeasible	1.6
mnist_24_image4_1_5	-	-	0	UNSAT	infeasible	1.6
mnist_24_image4_1_6	-	-	0	UNSAT	infeasible	1.4
mnist_24_image4_1_7	-	-	0	UNSAT	infeasible	1.5
mnist_24_image4_1_8	-	-	0	UNSAT	infeasible	1.6
mnist_24_image4_1_9	-	-	1	UNSAT	infeasible	103.7
mnist_24_image4_20_0	-1.8e+08	-	1	-	-	timelimit
mnist_24_image4_5_0	-1.3e+07	5.8e+05	541	-	-	timelimit
mnist_24_image4_5_1	-1.2e+07	1.6e+06	883	-	-	timelimit

Table B.8: Quadratic programming formulation on MNIST 24 test set with splitted instances. We set the time limit of one hour for each original instance, and report the runtimes for all instances after splitting until the time limit is reached. If zero solving nodes are indicated, infeasibility was detected by presolving. The solving status "bound" indicates that the solving process was aborted as a primal bound of 0.0 was reached. Status "infeasible" means that SCIP detected infeasibility of the quadratic program, which corresponds to an UNSAT instance of the verification problem. Except from the SAT instance "mnist_24_image4_10_0", only very easy instances with a small input domain are solved within the time limit of one hour.

B.4 Solving as feasibility problem

In this section we provide some computational results on the performance of our solving model using the formulation as feasibility problem, which we presented in Section 3.1. We refer to the beginning of Appendix C for some notes on the reported parameter settings.

Instance	Nodes	Result	Status	Time
lin_acas_1_1_int_away	227.0	UNSAT	infeasible	503.0
lin_acas_1_1_lin_opp2	249.0	UNSAT	infeasible	1194.5
lin_acas_1_1_lin_opp_dir	547.0	UNSAT	infeasible	3141.7
property1_1_1	11.0	UNSAT	infeasible	121.8
property1_2_2	105.0	UNSAT	infeasible	1546.1
property2_3_3	1849.0	UNSAT	infeasible	5358.9
property2_4_2	-	-	-	timelimit
property3_4_3	13.0	UNSAT	infeasible	26.4
property3_4_4	1.0	UNSAT	infeasible	4.0
property4_2_2	3.0	UNSAT	infeasible	10.3
property4_3_7	1.0	UNSAT	infeasible	4.7
property6_6a_property_3	1183.0	UNSAT	infeasible	5495.2
property6_6b_property_1	317.0	UNSAT	infeasible	2039.4
property9_property_0	463.0	UNSAT	infeasible	2212.6
property9_property_4	781.0	UNSAT	infeasible	3303.5

Table B.9: Results for configuration "no_heur_base_nonopt" which is the baseline configuration using the formulation of the verification problem as satisfiability problem.

Parameter	Value
use_opt_mode	False
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Nodes	Result	Status	Time
lin_acas_1_1_int_away	1602.0	UNSAT	infeasible	2504.3
lin_acas_1_1_lin_opp2	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	timelimit
property1_1_1	31.0	UNSAT	infeasible	343.7
property1_2_2	121.0	UNSAT	infeasible	1808.3
property2_3_3	-	-	-	timelimit
property2_4_2	-	-	-	timelimit
property3_4_3	41.0	UNSAT	infeasible	66.1
property3_4_4	1.0	UNSAT	infeasible	5.7
property4_2_2	5.0	UNSAT	infeasible	13.7
property4_3_7	1.0	UNSAT	infeasible	4.3
property6_6a_property_3	-	-	-	timelimit
property6_6b_property_1	-	-	-	timelimit
property9_property_0	817.0	UNSAT	infeasible	4129.3
property9_property_4	1463.0	UNSAT	infeasible	5474.0

Table B.10: Results for configuration "no_heur_relu_nonopt", in which ReLU branching is used instead of domain branching.

Parameter	Value
use_opt_mode	False
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	False
use_relu_branching	True
relu_branching_split_mode	standard
relu_branching_priority	100000
relu_branching_maxdepth	20
relu_branching_maxbounddist	1
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Nodes	Result	Status	Time
lin_acas_1_1_int_away	227.0	UNSAT	infeasible	513.5
lin_acas_1_1_lin_opp2	245.0	UNSAT	infeasible	1193.3
lin_acas_1_1_lin_opp_dir	543.0	UNSAT	infeasible	3144.1
property1_1_1	11.0	UNSAT	infeasible	128.3
property1_2_2	91.0	UNSAT	infeasible	1342.7
property2_3_3	2157.0	UNSAT	infeasible	6085.5
property2_4_2	-	-	-	timelimit
property3_4_3	13.0	UNSAT	infeasible	26.0
property3_4_4	1.0	UNSAT	infeasible	4.0
property4_2_2	3.0	UNSAT	infeasible	10.5
property4_3_7	1.0	UNSAT	infeasible	4.3
property6_6a_property_3	1159.0	UNSAT	infeasible	5447.3
property6_6b_property_1	309.0	UNSAT	infeasible	1954.6
property9_property_0	463.0	UNSAT	infeasible	2233.5
property9_property_4	777.0	UNSAT	infeasible	3306.5

Table B.11: Results for configuration "no_heur_sepa_nonopt" which corresponds to the baseline configuration "no_heur_base_nonopt", but additionally includes the separator.

Parameter	Value
use_opt_mode	False
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	1
sepa_priority	100
sepa_maxbounddist	0.5
sepa_delay	False

C Computational Results UNSAT Test Set

In the following sections we report detailed computational results for the experiments that are presented in Section 9.2. Each experiment is conducted on the evaluation set of UNSAT instances as presented in Section 8.4. As laid out in Section 9.2, each experiment is run on a cluster node with Intel Xeon CPU E5-2670 at a clock rate of 2.5 GHz. Moreover, a memory limit of 32 GB is set. On each page we report the results for one configuration and the corresponding settings in our solving model. In all configurations, the verification problem is solved as an optimization problem based on the formulation we presented in Section 3.2. We present runtime, number of solving nodes, the result SAT (refutable) or UNSAT (verifiable), and the values of the dual and primal bound, if these are finite. Additionally, we report a status which is "optimal", if the MIP was solved to optimality or "bound", if the solving process was interrupted due to a positive dual or a negative primal bound. If the status is "infeasible", all solving nodes have been cut off throughout the solving process. With respect to the prameter settings of our solving model, we refer to Section 7.3 for an explanation of these. It should be noted that various parameters are not relevant in certain cases. Therefore, not all parameters are reported for all configurations. If parameter values are not explicitly stated, they can be assumed to take the default values as indicated in Section 7.3.

Two parameters are set differently for various instances within the evaluation set of UNSAT instances. For our self defined instances that are based on the ACAS neural networks (see Section 8.2 and Appendix A), the parameter "sampling_heuristic_use_lp_sol_gen" is set to True, otherwise to False. This is due to the fact that for all other instances the feasible input domain is a box. On the other hand, the parameter "bfs_from_all_inputs" is set to True for all ACAS based instances, and to False for all MNIST based instances. It must be True for the ACAS based instances, because the corresponding neural networks have a neuron without activation function between each input neuron and the first ReLU layer. These neurons in between apply a linear transformation to each input component and can only be reached from the corresponding input neuron. Therefore, a breadth-first search (BFS) must be performed from each input neuron to find all neurons in the network. However, doing so for the MNIST instances would be wasted computational effort. The two parameters are set analogously in all configurations and therefore we do not report their values in the tables of this section.

C.1 OBBT2 options

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	171.0	UNSAT	optimal	522.6
lin_acas_1_1_lin_opp2	1.2	1.2	231.0	UNSAT	optimal	1387.6
lin_acas_1_1_lin_opp_dir	1.7	1.7	527.0	UNSAT	optimal	3806.6
lin_acas_3_1_var_dist	0.11	0.26	283.0	UNSAT	bound	837.9
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.6
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	418.3
property10_property	0.02	0.16	715.0	UNSAT	bound	4540.5
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	160.0
property1_2_2	1.5e+03	1.5e+03	95.0	UNSAT	optimal	1856.2
property2_3_3	0.036	0.12	1499.0	UNSAT	bound	6083.4
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.6	6.6	11.0	UNSAT	optimal	28.4
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	6.4
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	15.3
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.1
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.19	0.19	1043.0	UNSAT	optimal	6376.0
property6_6b_property_1	0.25	0.25	297.0	UNSAT	optimal	2412.0
property9_property_0	0.7	0.7	397.0	UNSAT	optimal	2433.3
property9_property_4	0.54	0.54	667.0	UNSAT	optimal	3552.5

Table C.1: Configuration "no_heur_base_obbt2": Baseline configuration with additional application of OBBT2, where our selection rule as described in Section 4.7 is applied with $k = 2$, $l = 5$.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	True
obbt_k	2
obbt_l	5
obbt_sort	True
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	167.0	UNSAT	optimal	499.1
lin_acas_1_1_lin_opp2	1.2	1.2	231.0	UNSAT	optimal	1376.6
lin_acas_1_1_lin_opp_dir	1.7	1.7	535.0	UNSAT	optimal	3834.8
lin_acas_3_1_var_dist	0.12	0.26	289.0	UNSAT	bound	845.4
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.0
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	419.9
property10_property	0.04	0.16	739.0	UNSAT	bound	4612.2
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	156.6
property1_2_2	1.5e+03	1.5e+03	95.0	UNSAT	optimal	1756.7
property2_3_3	0.028	0.093	1489.0	UNSAT	bound	5988.3
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	32.0
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	5.8
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	14.6
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.7
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1047.0	UNSAT	optimal	6401.7
property6_6b_property_1	0.25	0.25	299.0	UNSAT	optimal	2389.8
property9_property_0	0.7	0.7	413.0	UNSAT	optimal	2499.5
property9_property_4	0.26	0.26	655.0	UNSAT	optimal	3567.3

Table C.2: Configuration "no_heur_base_obbt2_nosort": Baseline configuration with additional application of OBBT2, where the baseline selection rule (cf. Section 9.2) is applied with $k = 2$, $l = 5$.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	True
obbt_k	2
obbt_l	5
obbt_sort	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	165.0	UNSAT	optimal	889.7
lin_acas_1_1_lin_opp2	1.2	1.2	213.0	UNSAT	optimal	2658.7
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	0.12	0.26	257.0	UNSAT	bound	1533.5
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	5.9
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	437.6
property10_property	-	-	-	-	-	timelimit
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	376.4
property1_2_2	1.5e+03	1.5e+03	89.0	UNSAT	optimal	5569.7
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.6	6.6	11.0	UNSAT	optimal	44.1
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	5.7
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	21.2
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	6.2
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	0.23	0.23	269.0	UNSAT	optimal	5070.1
property9_property_0	0.88	0.88	355.0	UNSAT	optimal	4725.2
property9_property_4	0.54	0.54	583.0	UNSAT	optimal	6504.4

Table C.3: Configuration "no_heur_base_obbt2_10": Baseline configuration with additional application of OBBT2, where our selection rule as described in Section 4.7 is applied with $k = 10$, $l = 10$.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	True
obbt_k	10
obbt_l	10
obbt_sort	True
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	151.0	UNSAT	optimal	839.8
lin_acas_1_1_lin_opp2	1.2	1.2	215.0	UNSAT	optimal	2663.0
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	0.12	0.26	263.0	UNSAT	bound	1535.8
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	5.9
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	439.3
property10_property	-	-	-	-	-	timelimit
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	370.0
property1_2_2	1.5e+03	1.5e+03	85.0	UNSAT	optimal	4980.4
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	11.0	UNSAT	optimal	39.8
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.8
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	20.2
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.0
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	0.23	0.23	267.0	UNSAT	optimal	4886.3
property9_property_0	1	1	357.0	UNSAT	optimal	4819.1
property9_property_4	0.54	0.54	573.0	UNSAT	optimal	6515.5

Table C.4: Configuration "no_heur_base_obbt2_10_nosort": Baseline configuration with additional application of OBBT2, where the baseline selection rule (cf. Section 9.2) is applied with $k = 10$, $l = 10$.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	True
obbt_k	10
obbt_l	10
obbt_sort	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

C.2 Separator options

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	530.0
lin_acas_1_1_lin_opp2	1.2	1.2	247.0	UNSAT	optimal	1234.4
lin_acas_1_1_lin_opp_dir	1.7	1.7	543.0	UNSAT	optimal	3283.0
lin_acas_3_1_var_dist	0.11	0.26	289.0	UNSAT	bound	729.8
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.3
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	416.9
property10_property	0.016	0.16	895.0	UNSAT	bound	4511.3
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	137.6
property1_2_2	1.5e+03	1.5e+03	93.0	UNSAT	optimal	1461.4
property2_3_3	0.038	0.12	1885.0	UNSAT	bound	5712.7
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	29.9
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	6.9
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	14.9
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.9
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1159.0	UNSAT	optimal	5761.1
property6_6b_property_1	0.23	0.23	301.0	UNSAT	optimal	1987.4
property9_property_0	1	1	463.0	UNSAT	optimal	2328.9
property9_property_4	0.54	0.54	779.0	UNSAT	optimal	3470.3

Table C.5: Configuration "no_heur_sepa0_high": Baseline configuration with additional execution of the separator with high priority among all separators, but only at the current best node, i.e. the node with the highest dual bound.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	1
sepa_priority	100000
sepa_maxbounddist	0.0
sepa_delay	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	530.0
lin_acas_1_1_lin_opp2	1.2	1.2	247.0	UNSAT	optimal	1241.8
lin_acas_1_1_lin_opp_dir	1.7	1.7	543.0	UNSAT	optimal	3281.5
lin_acas_3_1_var_dist	0.11	0.26	289.0	UNSAT	bound	731.0
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	8.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	418.1
property10_property	0.027	0.16	897.0	UNSAT	bound	4539.6
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	136.2
property1_2_2	1.5e+03	1.5e+03	93.0	UNSAT	optimal	1454.9
property2_3_3	0.054	0.093	1757.0	UNSAT	bound	5639.1
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.6	6.6	13.0	UNSAT	optimal	27.5
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	5.0
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	13.1
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.9
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1149.0	UNSAT	optimal	5729.0
property6_6b_property_1	0.25	0.25	303.0	UNSAT	optimal	1984.6
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2325.8
property9_property_4	0.89	0.89	779.0	UNSAT	optimal	3473.2

Table C.6: Configuration "no_heur_sepa0": Baseline configuration with additional execution of the separator but only at the current best node, i.e. the node with the highest dual bound.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	1
sepa_priority	100
sepa_maxbounddist	0.0
sepa_delay	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	527.7
lin_acas_1_1_lin_opp2	1.2	1.2	249.0	UNSAT	optimal	1241.1
lin_acas_1_1_lin_opp_dir	1.7	1.7	543.0	UNSAT	optimal	3246.1
lin_acas_3_1_var_dist	0.12	0.26	285.0	UNSAT	bound	723.9
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	7.2
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	415.2
property10_property	0.026	0.16	881.0	UNSAT	bound	4372.6
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	130.1
property1_2_2	1.5e+03	1.5e+03	93.0	UNSAT	optimal	1447.6
property2_3_3	0.054	0.093	1775.0	UNSAT	bound	5727.6
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	28.2
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	5.7
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	13.0
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.8
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1149.0	UNSAT	optimal	5649.1
property6_6b_property_1	0.25	0.25	305.0	UNSAT	optimal	2033.3
property9_property_0	0.7	0.7	461.0	UNSAT	optimal	2282.1
property9_property_4	0.26	0.26	777.0	UNSAT	optimal	3438.1

Table C.7: Configuration "no_heur_sepa0_freq5": Baseline configuration with additional execution of the separator but only at the current best node, i.e. the node with the highest dual bound, and each fifth depth level of the branch-and-bound tree.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	5
sepa_priority	100
sepa_maxbounddist	0.0
sepa_delay	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	227.0	UNSAT	optimal	530.9
lin_acas_1_1_lin_opp2	1.2	1.2	247.0	UNSAT	optimal	1233.1
lin_acas_1_1_lin_opp_dir	1.7	1.7	543.0	UNSAT	optimal	3275.5
lin_acas_3_1_var_dist	0.11	0.26	287.0	UNSAT	bound	727.0
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	416.7
property10_property	0.016	0.16	883.0	UNSAT	bound	4509.2
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	135.9
property1_2_2	1.5e+03	1.5e+03	93.0	UNSAT	optimal	1453.3
property2_3_3	0.054	0.093	1831.0	UNSAT	bound	5708.4
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.6	6.6	13.0	UNSAT	optimal	33.2
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	6.7
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	15.6
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	6.3
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.18	0.18	1149.0	UNSAT	optimal	5714.8
property6_6b_property_1	0.23	0.23	307.0	UNSAT	optimal	2010.0
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2330.9
property9_property_4	0.54	0.54	779.0	UNSAT	optimal	3468.9

Table C.8: Configuration "no_heur_sepa": Baseline configuration with additional execution of the separator.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	1
sepa_priority	100
sepa_maxbounddist	0.5
sepa_delay	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	227.0	UNSAT	optimal	534.6
lin_acas_1_1_lin_opp2	1.2	1.2	247.0	UNSAT	optimal	1237.0
lin_acas_1_1_lin_opp_dir	1.7	1.7	543.0	UNSAT	optimal	3283.9
lin_acas_3_1_var_dist	0.11	0.26	289.0	UNSAT	bound	734.4
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.6
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	421.6
property10_property	0.016	0.16	885.0	UNSAT	bound	4491.6
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	133.6
property1_2_2	1.5e+03	1.5e+03	93.0	UNSAT	optimal	1458.6
property2_3_3	0.052	0.12	1825.0	UNSAT	bound	5913.7
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	29.2
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.7
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	13.8
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.7
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1149.0	UNSAT	optimal	5707.0
property6_6b_property_1	0.25	0.25	307.0	UNSAT	optimal	2003.2
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2318.0
property9_property_4	0.54	0.54	779.0	UNSAT	optimal	3477.8

Table C.9: Configuration "no_heur_sepa1": Baseline configuration with additional execution of the separator at each node of the branch-and-bound tree.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	True
sepa_freq	1
sepa_priority	100
sepa_maxbounddist	1.0
sepa_delay	False

C.3 Further configurations

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.081	0.081	2245.0	UNSAT	optimal	1987.7
lin_acas_1_1_lin_opp2	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	0.00096	0.26	1062.0	UNSAT	bound	1274.6
mnist_24_image11_5	1.2e+03	1.2e+03	1.0	UNSAT	optimal	10.9
mnist_24_image2_5	2e+02	2e+02	593.0	UNSAT	optimal	942.6
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	6.1
mnist_512_image11_5	8.6e+03	8.6e+03	1.0	UNSAT	optimal	595.4
mnist_512_image2_1	1e+04	1e+04	1.0	UNSAT	optimal	403.4
property10_property	0.037	0.16	161.0	UNSAT	bound	656.8
property1_1_1	1.5e+03	1.5e+03	31.0	UNSAT	optimal	372.3
property1_2_2	1.5e+03	1.5e+03	125.0	UNSAT	optimal	1982.4
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	57.0	UNSAT	optimal	100.5
property3_4_4	4.5	4.6	2.0	UNSAT	bound	5.5
property4_2_2	5.5	5.5	5.0	UNSAT	optimal	15.2
property4_3_7	2.6	4.4	2.0	UNSAT	bound	5.2
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	0.7	0.7	829.0	UNSAT	optimal	4521.7
property9_property_4	0.54	0.54	1273.0	UNSAT	optimal	5837.1

Table C.10: Configuration "no_heur_relu_genv": ReLU branching combined with generation of LVBs.

Parameter	Value
sampling_heuristic_local_max_iter	0
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	20
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	False
use_relu_branching	True
relu_branching_split_mode	standard
relu_branching_priority	100000
relu_branching_maxdepth	20
relu_branching_maxbounddist	1
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	True
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp2	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	-	-	-	-	-	timelimit
mnist_24_image11_5	1.4e+02	1.2e+03	10.0	UNSAT	bound	6.1
mnist_24_image2_5	0.43	1.6e+02	3750.0	UNSAT	bound	20.0
mnist_24_image4_1	1.2e+03	1.2e+03	4.0	UNSAT	bound	5.1
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	8.7e+03	1e+04	11.0	UNSAT	bound	87.9
property10_property	-	-	-	-	-	timelimit
property1_1_1	-	-	-	-	-	timelimit
property1_2_2	-	-	-	-	-	timelimit
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	-	-	-	-	-	timelimit
property3_4_4	-	-	-	-	-	timelimit
property4_2_2	-	-	-	-	-	timelimit
property4_3_7	-	-	-	-	-	timelimit
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	-	-	-	-	-	timelimit
property9_property_4	-	-	-	-	-	timelimit

Table C.11: Configuration "mnist_base": Except for the primal heuristic at the root node, no specific techniques for verification of neural networks are used. The problem is solved as a plain MIP by SCIP.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	False
use_relu_branching	False
use_obbt_propagator	False
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	523.9
lin_acas_1_1_lin_opp2	1.2	1.2	251.0	UNSAT	optimal	1235.0
lin_acas_1_1_lin_opp_dir	1.7	1.7	547.0	UNSAT	optimal	3253.1
lin_acas_3_1_var_dist	0.11	0.26	299.0	UNSAT	bound	754.6
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	8.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	418.3
property10_property	0.021	0.16	909.0	UNSAT	bound	4506.9
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	127.1
property1_2_2	1.5e+03	1.5e+03	105.0	UNSAT	optimal	1615.0
property2_3_3	0.12	0.12	2349.0	UNSAT	optimal	7038.7
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	28.4
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.7
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	13.4
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.8
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1185.0	UNSAT	optimal	5802.2
property6_6b_property_1	0.23	0.23	317.0	UNSAT	optimal	2101.9
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2301.2
property9_property_4	0.54	0.54	783.0	UNSAT	optimal	3447.8

Table C.12: Configuration "no_heur_base": Our baseline configuration which applies domain branching and OBBT to the LP relaxation.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	8.9e-06	0.019	5448.0	UNSAT	bound	2788.4
lin_acas_1_1_lin_opp2	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	-	-	-	-	-	timelimit
mnist_24_image11_5	8.8	1.2e+03	17.0	UNSAT	bound	18.9
mnist_24_image2_5	1.3e+02	1.6e+02	1393.0	UNSAT	bound	2246.9
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	5.4
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	8.4e+03	1e+04	2.0	UNSAT	bound	414.3
property10_property	0.034	0.16	140.0	UNSAT	bound	593.5
property1_1_1	1.5e+03	1.5e+03	31.0	UNSAT	optimal	382.7
property1_2_2	1.5e+03	1.5e+03	174.0	UNSAT	optimal	3199.0
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	31.0	UNSAT	optimal	59.8
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.7
property4_2_2	5.5	5.5	5.0	UNSAT	optimal	16.1
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	7.6
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	0.11	0.7	853.0	UNSAT	bound	4674.8
property9_property_4	0.00083	0.54	1342.0	UNSAT	bound	5980.2

Table C.13: Configuration "no_heur_relu": Similar to the baseline configuration "no_heur_base", but with ReLU branching instead of domain branching.

Parameter	Value
sampling_heuristic_local_max_iter	0
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	20
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	False
use_relu_branching	True
relu_branching_split_mode	standard
relu_branching_priority	100000
relu_branching_maxdepth	20
relu_branching_maxbounddist	1
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.28	0.28	229.0	UNSAT	optimal	521.9
lin_acas_1_1_lin_opp2	-	-	251.0	UNSAT	infeasible	1241.0
lin_acas_1_1_lin_opp_dir	-	-	547.0	UNSAT	infeasible	3281.2
lin_acas_3_1_var_dist	0.11	-	299.0	UNSAT	bound	757.1
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.3e+03	1.3e+03	2.0	UNSAT	bound	6.6
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	7.9e+03	-	2.0	UNSAT	bound	411.6
property10_property	0.021	0.16	909.0	UNSAT	bound	4484.1
property1_1_1	-	-	11.0	UNSAT	infeasible	126.4
property1_2_2	-	-	105.0	UNSAT	infeasible	1645.4
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	-	-	13.0	UNSAT	infeasible	26.2
property3_4_4	-	-	1.0	UNSAT	infeasible	3.9
property4_2_2	-	-	3.0	UNSAT	infeasible	12.4
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.0
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.29	0.29	1185.0	UNSAT	optimal	5811.9
property6_6b_property_1	0.36	0.36	317.0	UNSAT	optimal	2104.1
property9_property_0	5	5	463.0	UNSAT	optimal	2299.9
property9_property_4	-	-	783.0	UNSAT	infeasible	3466.6

Table C.14: Configuration "no_heur_atall": Baseline configuration but with heuristic completely disabled.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	-1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.00059	0.22	382.0	UNSAT	bound	828.6
lin_acas_1_1_lin_opp2	0.021	1.2	888.0	UNSAT	bound	3608.3
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	0.00086	0.26	419.0	UNSAT	bound	912.1
mnist_24_image11_5	1.2e+03	1.2e+03	511.0	UNSAT	optimal	226.6
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	8.6
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	419.7
property10_property	-	-	-	-	-	timelimit
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	128.0
property1_2_2	1.5e+03	1.5e+03	105.0	UNSAT	optimal	1624.2
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.6	6.6	13.0	UNSAT	optimal	30.4
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.9
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	15.1
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.5
property5_property	0.021	0.37	1709.0	UNSAT	bound	3034.8
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	-	-	-	-	-	timelimit
property9_property_4	-	-	-	-	-	timelimit

Table C.15: Configuration "domain_relu": In this configuration, domain branching is applied up to a depth level of six in the branch-and-bound tree, and ReLU branching afterwards.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	6
domain_branching_maxbounddist	1
use_relu_branching	True
relu_branching_split_mode	standard
relu_branching_priority	10000
relu_branching_maxdepth	20
relu_branching_maxbounddist	1
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp2	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	-	-	-	-	-	timelimit
mnist_24_image11_5	15	1.2e+03	47.0	UNSAT	bound	45.5
mnist_24_image2_5	41	1.6e+02	1653.0	UNSAT	bound	2439.6
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	10.2
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	8.4e+03	1e+04	2.0	UNSAT	bound	464.7
property10_property	0.0037	0.16	145.0	UNSAT	bound	551.0
property1_1_1	-	-	-	-	-	timelimit
property1_2_2	1.5e+03	1.5e+03	127.0	UNSAT	optimal	2252.7
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	257.0	UNSAT	optimal	565.9
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	5.8
property4_2_2	5.5	5.5	9.0	UNSAT	optimal	25.3
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.3
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	-	-	-	-	-	timelimit
property9_property_4	-	-	-	-	-	timelimit

Table C.16: Configuration "no_heur_relu_gradient": In this configuration, ReLU branching is applied and the branching variable is selected based on gradient information.

Parameter	Value
sampling_heuristic_local_max_iter	0
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	20
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	False
use_relu_branching	True
relu_branching_split_mode	gradient
relu_branching_priority	100000
relu_branching_maxdepth	20
relu_branching_maxbounddist	1
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	-	-	227.0	UNSAT	infeasible	1434.5
lin_acas_1_1_lin_opp2	-	-	229.0	UNSAT	infeasible	2031.3
lin_acas_1_1_lin_opp_dir	-	-	431.0	UNSAT	infeasible	3532.5
lin_acas_3_1_var_dist	0.001	-	247.0	UNSAT	bound	1277.7
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.3e+03	1.3e+03	2.0	UNSAT	bound	6.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	8.1e+03	1.1e+04	2.0	UNSAT	bound	1162.7
property10_property	0.012	0.16	805.0	UNSAT	bound	4053.4
property1_1_1	-	-	1.0	UNSAT	infeasible	882.9
property1_2_2	-	-	103.0	UNSAT	infeasible	2550.2
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	-	-	1.0	UNSAT	infeasible	240.5
property3_4_4	-	-	1.0	UNSAT	infeasible	73.2
property4_2_2	-	-	1.0	UNSAT	infeasible	219.3
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	66.3
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	1181.0	UNSAT	infeasible	7169.4
property6_6b_property_1	0.28	0.28	301.0	UNSAT	optimal	2693.9
property9_property_0	-	-	457.0	UNSAT	infeasible	2982.7
property9_property_4	-	-	773.0	UNSAT	infeasible	4157.0

Table C.17: Configuration "no_heur_base_mip": Similar to the baseline configuration "no_heur_base", but initial neuron bounds are computed by OBBT on the MIP formulation.

Parameter	Value
build_optimize_nodes	True
use_linear_model	False
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp2	1.2	1.2	985.0	UNSAT	optimal	5242.6
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	0.1	0.26	789.0	UNSAT	bound	2687.8
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	5.9
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	419.0
property10_property	0.021	0.16	909.0	UNSAT	bound	4483.6
property1_1_1	1.5e+03	1.5e+03	135.0	UNSAT	optimal	1777.8
property1_2_2	-	-	-	-	-	timelimit
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	63.0	UNSAT	optimal	119.5
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.9
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	12.8
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.7
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	-	-	-	-	-	timelimit
property9_property_4	-	-	-	-	-	timelimit

Table C.18: Configuration "no_heur_std_branch": Similar to the baseline configuration "no_heur_base", but the domain branching variable is selected by the rule "standard" (cf. Section 6.1).

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	standard
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp2	-	-	-	-	-	timelimit
lin_acas_1_1_lin_opp_dir	-	-	-	-	-	timelimit
lin_acas_3_1_var_dist	-	-	-	-	-	timelimit
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	5.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	5.1e+03	1e+04	2.0	UNSAT	bound	303.7
property10_property	-	-	-	-	-	timelimit
property1_1_1	3.3e+02	1.5e+03	825.0	UNSAT	bound	460.6
property1_2_2	-	-	-	-	-	timelimit
property2_3_3	-	-	-	-	-	timelimit
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	-	-	-	-	-	timelimit
property3_4_4	4.6	4.6	5579.0	UNSAT	optimal	1892.3
property4_2_2	-	-	-	-	-	timelimit
property4_3_7	4.4	4.4	1001.0	UNSAT	optimal	153.2
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	-	-	-	-	-	timelimit
property6_6b_property_1	-	-	-	-	-	timelimit
property9_property_0	-	-	-	-	-	timelimit
property9_property_4	-	-	-	-	-	timelimit

Table C.19: Configuration "no_heur_sym": In this configuration, neuron bounds are computed using the approach of Wang et al. [56].

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	False
obbt_use_symbolic	True
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	493.9
lin_acas_1_1_lin_opp2	1.2	1.2	251.0	UNSAT	optimal	1219.1
lin_acas_1_1_lin_opp_dir	1.7	1.7	547.0	UNSAT	optimal	3222.0
lin_acas_3_1_var_dist	0.11	0.26	299.0	UNSAT	bound	738.0
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	8.0
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	-	-	-	-	-	timelimit
property10_property	0.11	0.16	1407.0	UNSAT	bound	5551.1
property1_1_1	1.5e+03	1.5e+03	17.0	UNSAT	optimal	122.1
property1_2_2	1.5e+03	1.5e+03	109.0	UNSAT	optimal	1395.8
property2_3_3	0.12	0.12	2341.0	UNSAT	optimal	6889.4
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	28.4
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	4.1
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	12.7
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	4.2
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.22	0.22	1185.0	UNSAT	optimal	5586.9
property6_6b_property_1	0.24	0.24	317.0	UNSAT	optimal	1971.5
property9_property_0	0.88	0.88	463.0	UNSAT	optimal	2284.3
property9_property_4	0.54	0.54	783.0	UNSAT	optimal	3432.3

Table C.20: Configuration "no_heur_base_200": This configuration is similar to the baseline configuration "no_heur_base", but OBBT is applied only at neurons where the difference between upper and lower bound is less than 200.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.0057	0.0057	229.0	UNSAT	optimal	613.7
lin_acas_1_1_lin_opp2	0.6	0.6	251.0	UNSAT	optimal	1462.4
lin_acas_1_1_lin_opp_dir	0.69	0.69	547.0	UNSAT	optimal	3679.1
lin_acas_3_1_var_dist	0.11	0.26	299.0	UNSAT	bound	994
mnist_24_image11_5	-	-	-	-	-	timelimit
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	9.8
mnist_512_image11_5	-	-	-	-	-	timelimit
mnist_512_image2_1	1e+04	1e+04	2.0	UNSAT	bound	426.3
property10_property	0.021	0.16	909.0	UNSAT	bound	4996.3
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	137.9
property1_2_2	1.5e+03	1.5e+03	105.0	UNSAT	optimal	1717.9
property2_3_3	-	-	-	-	-	6820.1
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	38.8
property3_4_4	4.6	4.6	1.0	UNSAT	optimal	6.9
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	16.3
property4_3_7	4.4	4.4	1.0	UNSAT	optimal	5.9
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.066	0.066	1185.0	UNSAT	optimal	6052.3
property6_6b_property_1	0.23	0.23	317.0	UNSAT	optimal	2348.1
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2584.9
property9_property_4	0.26	0.26	783.0	UNSAT	optimal	3775.9

Table C.21: Configuration "heur_base_20": This configuration is similar to the baseline configuration "no_heur_base", but the primal heuristic is also applied locally up to a depth of eight in the branch-and-bound tree.

Parameter	Value
sampling_heuristic_local_max_iter	1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	False
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

Instance	Dual Bound	Primal Bound	Nodes	Result	Status	Time
lin_acas_1_1_int_away	0.22	0.22	229.0	UNSAT	optimal	525.6
lin_acas_1_1_lin_opp2	1.2	1.2	251.0	UNSAT	optimal	1243.2
lin_acas_1_1_lin_opp_dir	1.7	1.7	547.0	UNSAT	optimal	3243.2
lin_acas_3_1_var_dist	0.081	0.26	269.0	UNSAT	bound	706.5
mnist_24_image11_5	1.2e+03	1.2e+03	1.0	UNSAT	optimal	9.8
mnist_24_image2_5	-	-	-	-	-	timelimit
mnist_24_image4_1	1.2e+03	1.2e+03	1.0	UNSAT	optimal	8.2
mnist_512_image11_5	8.6e+03	8.6e+03	1.0	UNSAT	optimal	608.9
mnist_512_image2_1	1e+04	1e+04	1.0	UNSAT	optimal	405.0
property10_property	0.011	0.16	803.0	UNSAT	bound	4148.6
property1_1_1	1.5e+03	1.5e+03	11.0	UNSAT	optimal	126.4
property1_2_2	1.5e+03	1.5e+03	105.0	UNSAT	optimal	1577.0
property2_3_3	0.12	0.12	2325.0	UNSAT	optimal	6989.1
property2_4_2	-	-	-	-	-	timelimit
property3_4_3	6.4	6.4	13.0	UNSAT	optimal	28.2
property3_4_4	4.5	4.6	2.0	UNSAT	bound	5.6
property4_2_2	5.5	5.5	3.0	UNSAT	optimal	12.1
property4_3_7	3	4.4	2.0	UNSAT	bound	5.5
property5_property	-	-	-	-	-	timelimit
property6_6a_property_3	0.066	0.066	1185.0	UNSAT	optimal	5799.3
property6_6b_property_1	0.23	0.23	317.0	UNSAT	optimal	2105.6
property9_property_0	0.7	0.7	463.0	UNSAT	optimal	2295.3
property9_property_4	0.54	0.54	783.0	UNSAT	optimal	3466.8

Table C.22: Configuration "no_heur_base_genv": This configuration corresponds to the baseline configuration "no_heur_base" with additional generation of LVBs.

Parameter	Value
sampling_heuristic_local_max_iter	-1000
sampling_heuristic_local_freq	1
sampling_heuristic_local_maxdepth	8
sampling_heuristic_max_iter	1000
sampling_heuristic_freq	1
sampling_heuristic_maxdepth	0
sampling_heuristic_bound_for_lp_heur	100000.0
sampling_heuristic_max_iter_lp_heur	1000
use_domain_branching	True
domain_branching_split_mode	gradient
domain_branching_priority	100000
domain_branching_maxdepth	20
domain_branching_maxbounddist	1
use_relu_branching	False
use_obbt_propagator	True
obbt_maxdepth	20
obbt_use_genvbounds	True
use_obbt_two_variables	False
obbt_optimize_nodes	True
obbt_use_symbolic	False
obbt_bound_for_opt	-200
use_ideal_separator	False

References

[1] Tobias Achterberg. "Constraint Integer Programming". PhD thesis. 2007. URL: http://dx.doi.org/10.14279/depositonce-1634.

[2] Tobias Achterberg. "SCIP: solving constraint integer programs". In: *Mathematical Programming Computation* 1.1 (July 2009), pp. 1–41. ISSN: 1867-2957. DOI: 10.1007/s12532-008-0001-1.

[3] Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. "Strong convex relaxations and mixed-integer programming formulations for trained neural networks". In: *arXiv* (Nov. 2018). URL: https://arxiv.org/abs/1811.01988.

[4] E. Balas. "Disjunctive Programming and a Hierarchy of Relaxations for Discrete Optimization Problems". In: *SIAM Journal on Algebraic Discrete Methods* 6.3 (1985), pp. 466–486. URL: https://doi.org/10.1137/0606047.

[5] Dimitris Bertsimas and John Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997. ISBN: 1886529191.

[6] Dimitris Bertsimas and Robert Weismantel. *Optimization over Integers*. Dynamic Ideas, 2005. ISBN: 978-0975914625.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006. ISBN: 978-0-387-31073-2.

[8] Helmut Bölcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. "Optimal approximation with sparsely connected deep neural networks". In: *SIAM Journal on Mathematics of Data Science* (2019). URL: http://www.nari.ee.ethz.ch/commth/pubs/p/deep-approx-18.

[9] Siegfried Bosch. *Lineare Algebra*. 4th ed. Springer, 2008. DOI: 10.1007/978-3-642-55260-1.

[10] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521833787.

[11] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. "Piecewise Linear Neural Network verification: A comparative study". In: *arXiv* (2017). URL: https://arxiv.org/abs/1711.00455.

[12] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. "A Unified View of Piecewise Linear Neural Network Verification". In: *Advances in Neural Information Processing Systems 31 (NIPS 2018)*. Ed. by Samy Bengio, Hanna Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 4795–4804. URL: https://arxiv.org/abs/1711.00455v3.

[13] Chih-Hong Cheng, Georg Nührenberg, Chung-Hao Huang, and Harald Ruess. "Verification of Binarized Neural Networks via Inter-neuron Factoring". In: *Verified Software. Theories, Tools, and Experiments - 10th International Conference: Revised Selected Papers*. 2018, pp. 279–290. URL: https://doi.org/10.1007/978-3-030-03592-1_16.

[14] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. "Maximum Resilience of Artificial Neural Networks". In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D'Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 251–268. ISBN: 978-3-319-68167-2.

[15] George B. Dantzig. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, 1963.

[16] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. "Output Range Analysis for Deep Feedforward Neural Networks". In: *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*. 2018, pp. 121–138. DOI: 10.1007/978-3-319-77935-5_9.

[17] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A. Mann, and Pushmeet Kohli. "A Dual Approach to Scalable Verification of Deep Networks". In: *UAI*. AUAI Press, 2018, pp. 550–559.

[18] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3.

[19] Rüdiger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D'Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 269–286. ISBN: 978-3-319-68167-2. DOI: 10.1007/978-3-319-68167-2_19.

[20] Matteo Fischetti and Jason Jo. "Deep Neural Networks and Mixed Integer Linear Optimization". In: *Constraints* 23.3 (July 2018), pp. 296–309. ISSN: 1383-7133. DOI: 10.1007/s10601-018-9285-6.

[21] Martin Fränzle and Christian Herde. "HySAT: An efficient proof engine for bounded model checking of hybrid systems". In: *Formal Methods in System Design* 30.3 (June 2007), pp. 179–198. URL: https://doi.org/10.1007/s10703-006-0031-0.

[22] Tristan Gally, Marc E. Pfetsch, and Stefan Ulbrich. "A framework for solving mixed-integer semidefinite programs". In: *Optimization Methods and Software* 33.3 (2018), pp. 594–632. DOI: 10.1080/10556788.2017.1322081. URL: https://doi.org/10.1080/10556788.2017.1322081.

[23] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 3–18.

[24] Ambros M. Gleixner, Timo Berthold, Benjamin Müller, and Stefan Weltge. "Three enhancements for optimization-based bound tightening". In: *Journal of Global Optimization* 67.4 (2017), pp. 731–757. ISSN: 1573-2916. URL: https://doi.org/10.1007/s10898-016-0450-4.

[25] Ambros Gleixner, Michael Bastubbe, Leon Eifler, Tristan Gally, Gerald Gamrath, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Marco E. Lübbecke, Stephen J. Maher, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Christoph Schubert, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Matthias Walter, Fabian Wegscheider, Jonas T. Witt, and Jakob Witzig. *The SCIP Optimization Suite 6.0*. Technical Report. Optimization Online, July 2018. URL: http://www.optimization-online.org/DB_HTML/2018/07/6692.html.

[26] *GLPK (GNU Linear Programming Kit)*. URL: https://www.gnu.org/software/glpk/ (visited on 03/05/2019).

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

[28] Winfried K. Grassmann and J. Paul Tremblay. *Logic and discrete mathematics - a computer science perspective*. Prentice Hall, 1996. ISBN: 978-0-13-501206-2.

[29] Branko Grünbaum. *Convex Polytopes*. Ed. by Volker Kaibel, Victor Klee, and Günter M. Ziegler. 2nd ed. Springer-Verlag New York, 2003. ISBN: 978-0-387-00424-2.

[30] Gregor Hendel. "Empirical Analysis of Solving Phases in Mixed Integer Programming". MA thesis. 2014, p. 159.

[31] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. "Safety Verification of Deep Neural Networks". English. In: *Computer Aided Verification (CAV)*. Ed. by Rupak Majumdar and Viktor Kunčak. Lecture Notes in Computer Science Part 1. Springer, July 2017, pp. 3–29. DOI: 10.1007/978-3-319-63387-9_1.

[32] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 661–667. ISBN: 978-3-642-02658-4.

[33] Colin Jones, E. C. Kerrigan, and Jan Maciejowski. *Equality Set Projection: A new algorithm for the projection of polytopes in halfspace representation*. Tech. rep. Cambridge: Cambridge University Engineering Dept, 2004. URL: http://infoscience.epfl.ch/record/169768.

[34] Kyle Julian, Jessica Lopez, Jeffrey S. Brush, Michael Owen, and Mykel J. Kochenderfer. "Policy compression for aircraft collision avoidance systems". In: *Digital Avionics Systems Conference (DASC)*. 2016. DOI: 10.1109/DASC.2016.7778091.

[35] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 2017, pp. 97–117. DOI: 10.1007/978-3-319-63387-9_5.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324.

[37] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/ (visited on 02/15/2019).

[38] Alessio Lomuscio and Lalit Maganti. "An approach to reachability analysis for feed-forward ReLU neural networks". In: *arXiv* (2017). URL: http://arxiv.org/abs/1706.07351.

[39] Stephen Maher, Matthias Miltenberger, Joao Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. "PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite". In: *Mathematical Software - ICMS 2016*. Vol. 9725. 2016, pp. 301–307. DOI: 10.1007/978-3-319-42432-3_37.

[40] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying Properties of Binarized Deep Neural Networks". In: *AAAI Conference on Artificial Intelligence*. 2018. URL: https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16898.

[41] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–257. ISBN: 978-3-642-14295-6.

[42] Luca Pulina and Armando Tacchella. "Challenging SMT solvers to verify neural networks". In: *AI COMMUNICATIONS* 25 (Jan. 2012), pp. 117–135. DOI: 10.3233/AIC-2012-0525.

[43] Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. "Semidefinite relaxations for certifying robustness to adversarial examples". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 10877–10887. URL: http://papers.nips.cc/paper/8285-semidefinite-relaxations-for-certifying-robustness-to-adversarial-examples.pdf.

[44] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. "Reachability Analysis of Deep Neural Networks with Provable Guarantees". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 2651–2659. URL: https://doi.org/10.24963/ijcai.2018/368.

[45] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. "Recent Improvements in the SMT Solver iSAT". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Warnemünde, Germany, March 12-14, 2013.* 2013, pp. 231–241.

[46] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. "Towards Verification of Artificial Neural Networks". In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2015, Chemnitz, Germany, March 3-4, 2015.* 2015, pp. 30–40.

[47] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. "Fast and Effective Robustness Certification". In: *NeurIPS*. 2018, pp. 10825–10836.

[48] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. "An abstract domain for certifying neural networks". In: *PACMPL* 3.POPL (2019), 41:1–41:30.

[49] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. "Boosting Robustness Certification of Neural Networks". In: *International Conference on Learning Representations*. May 2019. URL: https://files.sri.inf.ethz.ch/website/papers/RefineAI.pdf.

[50] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. "Intriguing properties of neural networks". In: *International Conference on Learning Representations*. 2014. URL: https://arxiv.org/abs/1312.6199v4.

[51] Vincent Tjeng and Russ Tedrake. "Verifying Neural Networks with Mixed Integer Programming". In: *arXiv* (2017). URL: https://arxiv.org/abs/1711.07356v1.

[52] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *International Conference on Learning Representations*. 2019. URL: https://arxiv.org/abs/1711.07356v3.

[53] Juan Pablo Vielma. "Embedding Formulations and Complexity for Unions of Polyhedra". In: *Management Science* 64.10 (Oct. 2018), pp. 4721–4734. ISSN: 0025-1909. URL: https://doi.org/10.1287/mnsc.2017.2856.

[54] Andreas Wächter and Lorenz T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (Mar. 2006), pp. 25–57. ISSN: 1436-4646. URL: https://doi.org/10.1007/s10107-004-0559-y.

[55] Shiqi Wang. *Neurify Github repository*. 2018. URL: https://github.com/tcwangshiqi-columbia/Neurify (visited on 02/15/2019).

[56] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Efficient Formal Safety Analysis of Neural Networks". In: *32nd Conference on Neural Information Processing Systems (NIPS)*. Montreal, Canada, 2018. URL: https://arxiv.org/abs/1809.08098.

[57] 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 (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi.

[58] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Duane Boning, and Inderjit S. Dhillon AND Luca Daniel. "Towards Fast Computation of Certified Robustness for ReLU Networks". In: *International Conference on Machine Learning (ICML)*. July 2018.

[59] Eric Wong and Zico Kolter. "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, Oct. 2018, pp. 5286–5295. URL: https://arxiv.org/abs/1711.00851.

[60] Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J. Zico Kolter. "Scaling provable adversarial defenses". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 8400–8409. URL: http://papers.nips.cc/paper/8060-scaling-provable-adversarial-defenses.pdf.

[61] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. "Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* (2018). DOI: 10.1109/TNNLS.2018.2808470.

[62] Weiming Xiang, Hoang-Dung Tran, Joel A. Rosenfeld, and Taylor T. Johnson. "Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers". In: *2018 Annual American Control Conference (ACC)*. June 2018, pp. 1574–1579. DOI: 10.23919/ACC.2018.8431048.

[63] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 4939–4948. URL: http://papers.nips.cc/paper/7742-efficient-neural-network-robustness-certification-with-general-activation-functions.pdf.

[64] Huan Zhang, Pengchuan Zhang, and Cho-Jui Hsieh. "RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications". In: *arXiv* (2019). URL: https://arxiv.org/abs/1810.11783.