Machine Learning (2023) 112:3685–3712
https://doi.org/10.1007/s10994-021-06050-2

# Global optimization of objective functions represented by ReLU networks

**Christopher A. Strong**[1] · **Haoze Wu**[2] · **Aleksandar Zeljić**[2] · **Kyle D. Julian**[3] · **Guy Katz**[4] · **Clark Barrett**[2] · **Mykel J. Kochenderfer**[3]

Received: 2 October 2020 / Revised: 30 July 2021 / Accepted: 27 August 2021 /
Published online: 20 October 2021
© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2021

**Abstract**

Neural networks can learn complex, non-convex functions, and it is challenging to guarantee their correct behavior in safety-critical contexts. Many approaches exist to find failures in networks (e.g., adversarial examples), but these cannot guarantee the absence of failures. Verification algorithms address this need and provide formal guarantees about a neural network by answering "yes or no" questions. For example, they can answer whether a violation exists within certain bounds. However, individual "yes or no" questions cannot answer qualitative questions such as "what is the largest error within these bounds"; the answers to these lie in the domain of optimization. Therefore, we propose strategies to extend existing verifiers to perform optimization and find: (i) the most extreme failure in a given input region and (ii) the minimum input perturbation required to cause a failure. A naive approach using a bisection search with an off-the-shelf verifier results in many expensive and overlapping calls to the verifier. Instead, we propose an approach that tightly integrates the optimization process into the verification procedure, achieving better runtime performance than the naive approach. We evaluate our approach implemented as an extension of Marabou, a state-of-the-art neural network verifier, and compare its performance with the bisection approach and MIPVerify, an optimization-based verifier. We observe complementary performance between our extension of Marabou and MIPVerify.

**Keywords** Neural network verification · Optimization · Adversarial examples · Marabou

## 1 Introduction

Artificial deep neural networks (DNNs) have demonstrated great promise in a wide variety of applications (Schmidhuber 2015; Liu et al. 2017). These applications include image recognition (Krizhevsky et al. 2012), control (Hunt et al. 1992), and natural language processing (Otter et al. 2020), among many others. Because of these successes, there is

Editors: Daniel Fremont, Alessio Lomuscio, Dragos Margineantu, Cheng Soon-Ong.

✉ Christopher A. Strong
christopher_strong@berkeley.edu

Extended author information available on the last page of the article

naturally interest in incorporating DNNs into other applications, including safety-critical systems (Bojarski et al. 2016; Julian et al. 2016). Although DNNs are obtaining unprecedented results, their opacity poses significant challenges—especially in the context of safety-critical systems, where mistakes can endanger lives and cause significant damage. A notable example includes DNNs in autonomous driving systems, where unexpected behavior of the DNN could harm passengers or pedestrians. Consequently, it is especially desirable to formally reason about DNNs, providing rigorous guarantees about their behaviors.

Recent research has focused on *neural network verification* (Huang et al. 2017; Katz et al. 2017; Gehr et al. 2018; Wang et al. 2018b). Verification involves answering "yes or no" questions about DNNs, and can be used to rule out undesirable behaviors. For example, a verification query for an autonomous driving DNN could ask whether an input exists that encodes a situation in which the autonomous vehicle is approaching an obstacle, but for which the DNN advises the vehicle to maintain the current course. If the verification engine answers no, we are guaranteed that this particular behavior can never happen for *any possible input*. If it answers yes, then it returns an input that leads to the undesirable outcome. The verification problem has been shown to be NP-complete (Katz et al. 2017); however, large strides have been made in recent years in solving networks that arise in practice (Wang et al. 2018a; Weng et al. 2018; Katz et al. 2019; Tran et al. 2020b; Wu et al. 2020).

Although tremendous effort has been put into answering yes or no questions about DNNs, formally answering quantitative questions about them has received less attention. Such questions can be highly important when verifying a system: for example, we may want to know *how close* an obstacle can be before the DNN controller turns the vehicle, or *how much* the steering of the car can be affected by small errors in the input image (e.g., caused by a malfunctioning camera). Finding answers to these questions requires an *optimization* process. Existing techniques can provide bounds on the answers to these questions, but the bounds may be too loose to effectively reason about the performance of the system (Singh et al. 2018a; Wang et al. 2018a, b; Weng et al. 2018; Zhang et al. 2018; Boopathy et al. 2019; Liu et al. 2021). For example, knowing that the most extreme steering angle for a self-driving car is somewhere between $-40°$ and $80°$ is likely not sufficiently informative to deduce guarantees about the car's behavior.

Global optimization of neural networks is also useful in the context of interpreting the patterns that a network has learned. For example, the activation of a hidden node can be maximized or minimized with respect to the input, in order to acquire a sense of the input properties that that node is tuned for. This has been done with local optimizers, but to our knowledge has not been explored with global optimizers (Le 2013; Ribeiro et al. 2016).

Many approximate techniques for answering these questions use heuristics and local optimization to find inputs that lead to undesired behavior from the network. Such techniques are referred to as *adversarial attacks*, and the corresponding inputs that lead to the undesired behavior are known as *adversarial examples*. Various techniques have been proposed, both for performing such attacks and defending against them (Goodfellow et al. 2015; Carlini and Wagner 2017; Chakraborty et al. 2021; Yuan et al. 2019). Additionally, a variety of techniques have been proposed to find certified lower bounds on the perturbation required to produce an adversarial example (Weng et al. 2018; Zhang et al. 2018; Boopathy et al. 2019). Although approximate techniques typically scale much more effectively to large networks (Müller et al. 2020), we may need a stronger guarantee on the behavior of a safety-critical system than we can find with existing approximate techniques. Existing strategies to solve global optimization problems on neural networks include performing a binary search for the optimal value by repeatedly calling neural verifiers, as well as mixed

integer programming (MIP) approaches that explicitly encode the network as constraints in an optimization problem. Repeatedly calling a neural verifier is often computationally prohibitive, while MIP approaches work well on some problem types but struggle with others (Carlini et al. 2017; Tjeng et al. 2019). It is thus desirable to develop new approaches for better accommodating different problem domains.

In order to have a wider variety of approaches for neural optimization, we introduce a framework for converting existing verifiers into optimizers. We demonstrate our framework by extending the Marabou verifier, which implements the Reluplex algorithm (Katz et al. 2019). Our extended version of Marabou, which we refer to as MarabouOpt, performs a branch-and-bound search over the activation space of the network. We compare the runtime of our solver to MIPVerify, a MIP approach that also solves global optimization problems, and we find that the two approaches are complementary to each other on the benchmarks tested. Additionally, we compare the approximate optima found by adversarial attack algorithms to the true optima found by MarabouOpt.

This paper is organized as follows: Sect. 2 provides background and descriptions of the notation used in the rest of the paper; Sect. 3 describes high-level approaches for modifying four categories of verifiers to perform optimization; Sect. 4 describes in detail how the Reluplex algorithm can be modified to perform optimization; Sect. 5 presents our experimental setup and results; Sect. 6 summarizes related work; and Sect. 7 concludes and suggests future research directions.

## 2 Background and problem formulation

**Neural networks** We denote the function represented by a neural network $N$ with $n$ inputs and $m$ outputs as $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Let a network $N$ with $K$ layers, including the input and output layers, have input to layer $\ell$ denoted by $\hat{\mathbf{z}}_\ell$ and output of layer $\ell$ denoted by $\mathbf{z}_\ell$. The input to each layer (besides the first) is computed by applying an affine transformation to the previous layer followed by an *activation function*. We consider two activation functions in this paper: the identity function and the rectified linear unit (ReLU). For a rectified linear unit (ReLU) layer $\ell$, we have

$$\mathbf{z}_\ell = \max(0, \hat{\mathbf{z}}_\ell). \tag{1}$$

For identity layers, we have

$$\mathbf{z}_\ell = \hat{\mathbf{z}}_\ell. \tag{2}$$

Let $\mathbf{W}_\ell$ and $\mathbf{b}_\ell$ be the weights and biases connecting layer $\ell$ to layer $\ell + 1$ such that

$$\hat{\mathbf{z}}_{\ell+1} = \mathbf{W}_\ell \mathbf{z}_\ell + \mathbf{b}_\ell. \tag{3}$$

The output of the network will be referred to as $\mathbf{y}$, with

$$\mathbf{y} = \mathbf{z}_K. \tag{4}$$

This section introduces notation and defines the neural network verification and neural network optimization problems. It also provides a categorization of neural network verification algorithms and explains existing local optimizers.

Let $\mathcal{L}$ denote the set of indices of ReLU layers and $\mathcal{I}$ denote the set of indices of identity layers. A ReLU is considered *active* if its input is greater than or equal to 0, and *inactive*

otherwise. An *activation state* is a representation of whether a node is active or not for each ReLU in a network. For a given activation state, let $\mathcal{A}$ be the set of indices $(i, j)$ of active ReLUs where $i$ is the layer and $j$ is the node in the layer. Similarly, let $\mathcal{N}$ be the set of indices $(i, j)$ of inactive ReLUs. A *partial activation state* is an activation state where some nodes are left unknown. In this case, let $\mathcal{U}$ be the set of indices $(i, j)$ of undetermined nodes for the partial activation state.

**Geometric objects** We refer to sets described by the intersection of affine inequalities as *polytopes*. A polytope $\mathcal{P}$ can be described by a matrix $\mathbf{A}$ and vector $\mathbf{b}$ as $\mathcal{P} = \{\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}\}$. We refer to the complement of a polytope as a *polytope complement*. Polytope complements can be used to represent non-convex and unbounded spaces. *Hyperrectangles* are convex polytopes that can be described by an upper and lower bound on each variable. The radius of a hyperrectangle is a vector $r$ containing values equal to half the interval between the upper and lower bound for each dimension. We assume that the domain of $f$ given by dom($f$) is a hyperrectangle, although this domain is not included in the formulations presented here. *Hypercubes* are hyperrectangles with a uniform radius. Let

$$\mathcal{X} \subseteq \mathbb{R}^n \qquad \mathcal{Y} \subseteq \mathbb{R}^m \tag{5}$$

be input and output sets which we will use in our problem definitions.

**Neural verification problem** One approach to verifying a network is to show that an input-output property holds. Such a property can be specified as

$$\mathbf{x} \in \mathcal{X} \implies \mathbf{y} = f(\mathbf{x}) \in \mathcal{Y} \tag{6}$$

We will refer to any algorithm that solves verification problems as a *verifier*. Different verifiers can handle different types of input and output sets. There are approaches to verification that are *sound*, *complete*, or both. If a sound algorithm reports that a property holds, then it must actually hold. If a complete algorithm reports that a property is violated, then it must actually be violated. An algorithm that is both sound and complete must always give a correct answer if it terminates.

### 2.1 Approaches to verification

In this section, we present several categories of verification algorithms described by Liu et al. (2021). Each approach can be extended to perform optimization as described in Sect. 3. There are four categories: reachability, optimization, search with reachability, and search with optimization. Although we focus on sound and complete verifiers, there are many interesting incomplete verifiers in these categories as well.

#### 2.1.1 Reachability

Complete reachability methods compute an exact output reachable set, then use this reachable set to solve verification problems. The reachable set is found by propagating the input set through the network layer by layer. Once an exact output reachable set is found, the verification problem can be solved by checking whether the reachable set is contained within the set $\mathcal{Y}$. If the input set is a polytope or union of polytopes, then the reachable set will

also be a union of polytopes (Xiang et al. 2018). In this case, the test for inclusion requires solving several linear programs (LPs). If the exact reachable set is contained within the output set $\mathcal{Y}$, the property holds. If it is not, then the property does not hold.

ExactReach is a reachability method which propagates polytopes through the network (Xiang et al. 2018). The more recent NNV uses other set representations, for example *star sets*, to greatly improve the efficiency of the propagation (Tran et al. 2019, 2020a, b).

#### 2.1.2 Optimization

Optimization methods encode the network and property as a constrained optimization problem. They typically constrain the input to be in $\mathcal{X}$ and the output to be in $\mathcal{Y}^{\complement}$, representing the region outside of the output region of the property (Liu et al. 2021; Lomuscio and Maganti 2017). If the resulting optimization problem is feasible, then we know some input in the input space $\mathcal{X}$ can reach outside of the output space $\mathcal{Y}$, and the property must not hold. Conversely, if the optimization problem is infeasible then there must be no input in $\mathcal{X}$ that reaches outside of the output set and the property holds.

*NSVerify* is an optimization method which uses a mixed integer encoding for the network and linear input and output constraints (Lomuscio and Maganti 2017). *MIPVerify* is another optimization method which improves on NSVerify in two ways. It adopts a tighter encoding of the ReLU and it performs a progressive bound tightening process. MIPVerify has been used to solve both output and minimum adversarial perturbation optimization problems described in Sect. 2.3.

#### 2.1.3 Search

Search with reachability and search with optimization methods search for a counter-example to the property by breaking the problem into a series of subproblems. The search space commonly consists of input ranges or neuron activations (Katz et al. 2017; Wang et al. 2018a, b; Liu et al. 2021; Botoeva et al. 2020). The search ends immediately if it finds an input that violates the property. If a region is proven to satisfy the property, the search proceeds to the next region. If neither conclusion is reached, the region is broken down further. At each step, a reachability or optimization approach is taken to determine whether the property in the region holds or is violated.

*Neurify* is an example of a search with reachability algorithm. It performs a type of symbolic reachability analysis called *symbolic linear relaxation* to find an approximate reachable set and reason about the property for a given region (Wang et al. 2018a). *Reluplex* is an example of a search with optimization algorithm which searches the activation space, solving a constrained linear program at each step to reason about the region (Katz et al. 2017). An extension of the Reluplex algorithm to perform optimization is described in more detail in Sect. 2.2.

### 2.2 The Reluplex algorithm

The Reluplex algorithm searches for a counter-example to the property, i.e., an input $\mathbf{x} \in \mathcal{X}$ such that the output $\mathbf{y} \in \mathcal{Y}^{\complement}$. The Reluplex algorithm explores the activation space by fixing ReLUs to be either inactive or active, one at a time. The search space is a binary tree, where each node represents a set of fixed ReLUs, which is equivalent to a partial activation state. Each edge leaving a node represents another ReLU becoming fixed to be active or

inactive respectively. At each node, a relaxed linear feasibility problem is solved using the simplex algorithm. If a satisfying assignment is obtained, it is checked against the remaining non-linear constraints. If the obtained assignment satisfies the non-linear constraints, a counterexample has been found and the search stops. However, if the assignment violates some non-linear constraints, then Reluplex can either fix the assignment and continue solving the linear relaxation, or perform a *case split*. A case split fixes the phase of a single ReLU and adds its—now linear—constraints to the relaxation, simplifying the problem. Reluplex judiciously explores and trims the search space until it either finds a satisfying assignment or proves that one does not exist.

Note the similar pattern of search in Reluplex and Neurify—both decompose the problem into smaller problems and then solve each independent feasibility problem. This search structure enables these algorithms to perform optimization, as we will show in Sect. 3.

### 2.3 Neural optimization problem

Neural verification problems allow us to answer yes or no questions about properties of the network. However, we would like to be able to answer qualitative questions. To that end, we define an *output optimization problem* and a *minimum adversarial perturbation problem*. The first consists of optimization on the output of the network subject to constraints on the input. The second consists of optimization on the input of the network subject to constraints on the output. Both can be used to answer questions that provide insight into the robustness of a system. The first problem can be used to ask questions like, "If there is at most 5% error in each pixel in an image used to control the steering wheel of a car, how drastically could the wheel mistakenly be turned?" The second problem can be used to ask questions like, "What is the smallest perturbation to my input image which would lead to me misclassifying a person as a stop sign?" For our problems, we will consider input and output constraints given by polytopes. These can each be represented with a set of linear inequalities.

1. **Output optimization problem**: We would like to find the maximum of a linear function of the output given constraints on the input. Let our objective be $g(\mathbf{x}) = \mathbf{c}^\top f(\mathbf{x})$, described by the user-defined parameter $\mathbf{c} \in \mathbb{R}^m$. More complex non-linear objectives can be approximated by augmenting the network with extra layers. For example, for a single output network with output $y$, if we wanted to maximize $y^2$ we could add on layers which approximate the function $f(y) = y^2$ and then perform verification on this augmented network. The problem formulation is

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & \mathbf{c}^\top f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \end{aligned} \tag{7}$$

   with corresponding optimal value $p^*$ and optimizing input $\mathbf{x}^*$.
2. **Minimum adversarial perturbation problem**: We would like to find the minimum adversarial perturbation to some original input $\mathbf{x}_0$ that causes undesired behavior. What it means to be a small perturbation can differ between applications and algorithms, with typical distance metrics including the $L_1$, $L_2$, and $L_\infty$ norms (Carlini et al. 2017; Tjeng et al. 2019). Typically, algorithms search for the smallest possible perturbation that causes a mistake (Szegedy et al. 2014; Carlini and Wagner 2017; Yuan et al. 2019). In our case, we will focus on the $L_\infty$ norm for simplicity. The $L_1$ norm is also commonly

used, as it can also be represented with linear constraints (Tjeng et al. 2019). A perturbed input $\mathbf{x}$ is considered adversarial if it is in some output set $\mathcal{Y}$ which can be used to represent undesired behavior. Our problem is then

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x} - \mathbf{x}_0\|_\infty \\ \text{subject to} \quad & f(\mathbf{x}) \in \mathcal{Y} \end{aligned} \tag{8}$$

with corresponding optimal value $p^*$ and optimal input $\mathbf{x}^*$.
These two general problems can be used to represent the two approaches for limiting the size of the perturbation and requiring adversarial behavior in the taxonomy of adversarial examples given by Yuan et al. (2019). Adversarial examples must be close to a nominal input and lead to undesirable behavior. The first problem can achieve this "closeness" by constraining the input set, and the undesirable behavior through the linear objective, while the second can achieve "closeness" by optimizing the size of the perturbation and the undesirable behavior through the output constraints.

### 2.4 Approximate methods for optimization

There are a variety of approximate approaches for solving neural optimization problems. We highlight several that we use to compare to MarabouOpt. Approximate methods provide bounds on the optima for these problems.

Many adversarial attacks provide approximate solutions to one or the other of our optimization problems (Yuan et al. 2019). The Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) provide approximate solutions to the output optimization problem by making use of the gradient of the objective with respect to the input (Goodfellow et al. 2015; Madry et al. 2018). FGSM takes a single step in the direction of the gradient to the boundaries of the input set, while PGD takes many gradient steps, projecting into the input set after each step. LBFGS, a quasi-Newtonian optimization method (Zhu et al. 1997), can also be used to find an approximate solution to the output optimization problem. We compare MarabouOpt on these problems to these three methods.

## 3 Strategies for converting verifiers into optimizers

The main observation of this paper is that many existing approaches to neural network verification can be extended to solve optimization problems. To substantiate our claim, we illustrate strategies for four major categories of verification algorithms outlined in the survey of verification methods (Liu et al. 2021). As summarized in Sect. 2.1, these categories are reachability, optimization, search with reachability, and search with optimization. Each of these categories poses different advantages and challenges when being extended to support optimization. We describe how to modify algorithms in each of these categories, and as a proof of concept, we showcase the extension of Marabou, a state-of-the-art search and optimization verifier (Katz et al. 2019), and evaluate the performance of the resulting optimizer. Although in this work we only extend Marabou, there are a wide variety of other verifiers that once extended could provide a suite of optimizers each with their own disadvantages or advantages (Johnson and Liu 2020; Bak et al. 2021; Liu et al. 2021).

First, however, we present how to solve optimization problems by combining a decision procedure with bisection search. This is a well-established approach that serves as a

baseline for comparison with the integrated approaches that we propose. While our focus is on complete verification procedures, we discuss extensions of incomplete approaches in Sect. 7. Throughout the section we consider a network $N$ represented by function $f$ with $n$ inputs, $m$ outputs, and $K$ layers.

### 3.1 Optimization using bisection search

A complete verifier answers yes or no to the question: "Does $\mathbf{x} \in \mathcal{X}$ imply $\mathbf{y} \in \mathcal{Y}$?" An optimization problem seeking to maximize some function $g(\mathbf{x})$ subject to either $\mathbf{x} \in \mathcal{X}$ or $\mathbf{y} \in \mathcal{Y}$ can be solved by asking a series of yes/no questions where a verification problem is constructed to represent the question, "can the optimal value $p^*$ be greater than $d$" for some value $d$. If we start with inital bounds on $p^*$, $\ell \leq p^* \leq u$, we can then select $d$ to perform bisection and update the bounds on the optimal value, halving the remaining search space with each step. If the answer to a query is yes, the lower bound $\ell$ is strengthened; otherwise, the upper bound $u$ is weakened. This procedure finds the optimal solution when $\ell = u$; alternatively, it can be made to halt when a user-defined minimum optimality gap is achieved. This approach has been used to find optimal solutions (Julian et al. 2020; Carlini et al. 2017) and often relies on bracketing techniques (Kochenderfer and Wheeler 2019) to narrow in on the optimal solution. Meta strategies can be applied to further speed up the convergence, like solving multiple instances in parallel, and splitting the region into sets of disjoint intervals, rather than two halves (Julian et al. 2020). This algorithm always terminates if the bounds are represented with floating point numbers but may not terminate if they are represented with real numbers.

### 3.2 Reachability

Reachability methods operate on sets of points rather than individual points in order to compute an exact output reachable set for a network. Sets of points are represented using abstract domains, such as polytopes or star sets, with efficient operations to propagate these domains through the layers of the network. The verification property is of the form $\mathbf{x} \in \mathcal{X} \implies \mathbf{y} \in \mathcal{Y}$, where $\mathcal{X}$ is the input and $\mathcal{Y}$ the output set. Reachability methods propagate abstract representations of the input set $\mathcal{X}$ through the network layers, until a reachable set $\mathcal{R}$ for the output layer is computed. The property holds if $\mathcal{R} \subseteq \mathcal{Y}$. To reason soundly about the output set, the abstract domain needs to be either: 1. exact—no values are lost or added during the computation, or 2. an over-approximation—no values are lost during the computation. In case of over-approximations, the subset check may fail even though the property holds. Over-approximated output reachable sets can still be used in complete methods when incorporating search in the process. The remainder of this section focuses on methods using exact representation. ExactReach (Xiang et al. 2018) and NNV (Tran et al. 2020b) represent reachable sets using a union of convex sets—polytopes and star sets, respectively. A star set is a polytope encoding that supports efficient computation of propagation through a network.

The reachable set can be represented as

$$\mathcal{R} = \bigcup_{i=1}^{k} P_i, \tag{9}$$

where $P_i$ are convex sets for $i \in \{1, 2, \dots, k\}$. Therefore, there are two equivalent reachability problems to check whether the property holds:

$$(\mathbf{x} \in \mathcal{X} \implies \mathbf{y} \in \mathcal{Y}) \iff (\mathcal{R} \subseteq \mathcal{Y}) \iff \left(P_i \subseteq \mathcal{Y},\ i = 1, \dots, k\right) \tag{10}$$

Checking whether $\mathcal{R} \subseteq \mathcal{Y}$ is challenging because $\mathcal{R}$ may be non-convex. However, we can check whether a convex set $P_i$ is contained within the polytope $\mathcal{Y}$ in polynomial time. The polytope subset problem $P \subseteq P'$, where $P$, $P'$ are polytopes and $P'$ consists of $j$ linear constraints, is answered by solving $j$ linear programs (LPs). Suppose $\mathcal{Y}$ consists of $\ell$ linear constraints, then solving a total of $k \cdot \ell$ LPs answers the verification problem in polynomial time.[1]

Given the basic overview of reachability techniques, how do we solve an output optimization problem using the exact reachable set $\mathcal{R}$? Let $\mathbf{c}^\top y$ with $\mathbf{c} \in \mathbb{R}^m$ be the objective function. The optimal value $p^*$ can be expressed in several ways:

$$p^* = \max_{\mathbf{x} \in \mathcal{X}} \mathbf{c}^\top f(\mathbf{x}) \tag{11}$$

$$= \max_{\mathbf{y} \in \mathcal{R}} \mathbf{c}^\top \mathbf{y} \tag{12}$$

$$= \max_{i \in \{1,2,\dots,k\}} \max_{\mathbf{y} \in P_i} \mathbf{c}^\top \mathbf{y} \tag{13}$$

There are two ways to answer the optimizing query using reachability: 1. using the non-convex set $\mathcal{R}$ as shown in Eq. (12), or 2. using the set of convex sets $P_i$, $i \in \{1, 2, \dots, k\}$ as shown in Eq. (13). In the latter case, the inner maximization of the final expression $\max_{y \in P_i} \mathbf{c}^\top \mathbf{y}$ has a linear objective and linear constraints, so it is an LP. Consequently, solving an LP for each index $i$ yields an exact value of $p^*$.

Note that, whereas solving the subset problem uses $\ell$ LPs per polytope, a single LP per polytope $P_i$ is sufficient to find the optimal value. The maximum of the local maxima for each $P_i$ yields the global optimum. Given the reduction in the number of LP problems solved, there are likely to be cases where the optimization problem is more efficient than the verification problem with the same input set. Understanding when this is the case is an interesting area for future investigation. We also expect this approach to outperform the bisection approach, as the latter requires many more calls to the verifier (albeit with smaller input sets each time).

Exact reachability-based methods can readily be extended to solve output optimization problems. This means that ExactReach and NNV with minimal modifications could be applied to an output optimization problem; the only change required is replacing the polytope subset check with a single optimizing LP call.

Although the approach outlined above guarantees an optimal value, it does not provide a method to find the corresponding optimizing input. Doing so would require tracking additional information linking input regions to the $P_i$ polytopes. Specific implementations of reachability-based approaches may or may not track this information. Those that do typically maintain a one-to-one correspondence between polytopes in the input space and

[1] Solving each LP takes polynomial time, but note that $k$ may be exponentially large compared to the input representation. This exponential growth in the number of output sets is a challenge for both reachability verifiers and optimizers.

output space (Tran et al. 2020b; Vincent and Schwager 2021). Tracking this correspondence provides an additional benefit: we can find the exact input set that maps to a specified output polytope. This is accomplished by mapping the intersection of the output polytope with the exact reachable set back to the input space as a union of polytopes (Vincent and Schwager 2021). We can use this to solve the minimum adversarial perturbation problem. We first find the pre-image of the output set $\mathcal{Y}$ in the input space. We then apply the same approach to analyzing this union of polytopes as we did with the output optimization problem: we can iterate one by one through the polytopes, solving a convex program at each step. In this case the objective is to minimize $\|\mathbf{x} - \mathbf{x}_0\|_\infty$ over each polytope. As a result, we can extend exact reachability-based methods that maintain a correspondence between the input and output sets to solve the minimum adversarial perturbation problem.

### 3.3 Optimization

In this section, we discuss the extension of optimization-based verification approaches to solve optimization problems. This is a fairly direct modification from a theoretical standpoint. Optimization approaches, such as NSVerify and MIPVerify (Lomuscio and Maganti 2017; Tjeng et al. 2019), exactly encode the network as a mixed integer program (MIP). The input set $\mathcal{X}$ and the complement of the output set $\mathcal{Y}$ are added as linear constraints to the MIP, resulting in a feasibility problem

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & 0 \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \\ & \mathbf{y} = f(\mathbf{x}) \\ & \mathbf{y} \in \mathcal{Y}^{\complement} \end{aligned} \tag{14}$$

The original property $\mathbf{x} \in \mathcal{X} \implies \mathbf{y} \in \mathcal{Y}$ does not hold if and only if the MIP is feasible. These optimization-based approaches are sound and complete.

As the problem statement in Eq. (14) suggests, extending this formulation to support optimization problems only requires the embedding of a goal function. Indeed, MIPVerify uses this approach to solve these classes of problems (Tjeng et al. 2019).

For output optimization problems, the output constraint is removed and an objective is added to the mixed integer program, resulting in

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximize}} \quad & \mathbf{c}^\top \mathbf{y} \\ \text{subject to} \quad & \mathbf{x} \in \mathcal{X} \\ & \mathbf{y} = f(\mathbf{x}) \end{aligned} \tag{15}$$

The input set and network constraints remain unchanged. This is still an MIP, since we assume a linear output objective. Most MIP solvers readily admit MIPs with an objective, requiring no further modification in practice.

The process is similar for encoding a minimum adversarial perturbation problem described by output set $\mathcal{Y}$ and original point $\mathbf{x}_0$. In this case, an objective is added for the input and the input constraint is removed, while the network and output constraints remain unchanged. Alternatively, we can also replace the output constraint with $\mathbf{y} \in \mathcal{Y}_{\simeq}$, where $\mathcal{Y}_{\simeq}$ represents a target set of adversarial behaviors, resulting in

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \|\mathbf{x} - \mathbf{x}_0\|_\infty \\ \text{subject to} \quad & \mathbf{y} = f(\mathbf{x}) \\ & \mathbf{y} \in \mathcal{Y}_{\simeq} \end{aligned} \tag{16}$$

As shown in this section, optimization-based verifiers need an appropriate objective added in order to solve output optimization or minimum adversarial input problems. This approach translates directly to search with optimization solvers as well. In practice, this requires minimal modification to the source in order to solve a whole new class of problems.

### 3.4 Search

Search-based approaches break down the space into smaller regions, typically by constraining the input or the activation space (Liu et al. 2021). Algorithm 1 gives the pseudocode for search-based verification. At each search state, a procedure VIOLATED($S$, $P$) is invoked. This procedure takes as input a state in the search space $S$ and problem $P$ and returns one of the following three outputs:

1. A status of VIOLATED and an assignment that violates the property;
2. A status of HOLDS, indicating that the property holds in this region;
3. A status of UNKNOWN, indicating that it is unknown whether the property holds in this region.

If a violating assignment is found, the search returns the discovered solution. If the answer is inconclusive, the search state is decomposed into multiple smaller—i.e., further constrained—states and the search continues. If the property holds in a state, the search proceeds with the next unexplored state. If all states have been explored, the search procedure determines that the property holds.

**Algorithm 1** Search-based verification

```
1:  function CHECK(S, P)
2:      states ← [S]
3:      while states is not empty
4:          state ← states.dequeue();
5:          status, assignment ← VIOLATED(state, P)
6:          if status = VIOLATED
7:              return Counter-example( assignment )
8:          else if status = UNKNOWN
9:              states.enqueue(SPLIT(state, P))
                                ▷ If the status is HOLDS the search continues
10:     return PropertyHolds
```

**Algorithm 2** Search-based optimization, branch and bound

```
1:  function OPTIMIZE(S, P)
2:      states ← [S]
3:      optSoFar ← None
4:      x ← None
5:      while states is not empty
6:          state ← states.dequeue()
7:          status, val, assignment ← OPTIMUMFORREGION(state, P, optSoFar)
8:          if status = Unknown
9:              states.enqueue(SPLIT(state, P))
10:         else if status = Optimal and val > optSoFar
11:             optSoFar ← val
12:             x ← val
                            ▷ If the status is WorseThanOpt the search continues
13:     return optSoFar, x
```

Algorithm 2 extends the search-based verification approach to solve optimization problems instead. *S* is still a state, and *P* is an optimization problem which includes an objective function and constraints. The Violated procedure is replaced by an OptimumForRegion(state, *P*, optSoFar) optimizing procedure. This can return one of the following three outputs:

1. A status of WorseThanOpt which indicates that the optimal value in this region is less than the true optimum. The function can make use of optSoFar to make this assertion.
2. A status of Unknown, indicating that it is unknown whether the true optimum could be contained in this region.
3. A status of Optimal, an objective value guaranteed to be the optimum in this region, and an assignment which achieves this objective value. This indicates that it has found a true global optimum.

In order to be terminating it also must be guaranteed to find the optimal, and not return Unknown, once the region has been split enough times. One example of an implementation of OptimumForRegion(state, *P*, optSoFar) would be to solve a relatively tractable relaxed problem, providing an upper bound on the solution. If this upper bound is less than optSoFar, it can return WorseThanOpt since the upper bound being lower than an objective value that has already been achieved will guarantee the region cannot achieve the optimum. If the assignment from solving the relaxed problem achieves the upper bound, it can return the objective value and assignment along with a status of Optimal. Otherwise, it can return Unknown. The way that the problem is relaxed and the upper bound is computed differs between search with reachability and search with optimization strategies.

Search with reachability verification methods will typically use approximate reachability methods in place of Violated (Wang et al. 2018a, b). To convert these verifiers into optimizers, their Violated procedure can be modified to implement OptimumForRegion in a similar way to the extension of pure reachability-based methods covered in Sect. 3.2. This could be applied to extend verifiers such as Neurify and ReluVal, which compute an approximate reachable set at each step (Wang et al. 2018a, b). Similarly, search with optimization methods use constrained optimization problems to implement Violated (Katz et al. 2017; Botoeva et al. 2020), which naturally extend to optimization as discussed in Sect. 3.3. In our extension of Reluplex we take this approach, solving a relaxed problem

that can provide an upper bound on the true optimal value for a partial activation state (Katz et al. 2017). This is described in more detail in Sect. 4.

In practice, the search structure largely remains the same as verification, with the exception that finding a counter-example does not end the search. Instead, such intermediate values are used to facilitate the branch-and-bound strategy.

## 4 Extending Reluplex to solve optimization problems

Reluplex is a search with optimization technique. As such, we can make use of the strategy presented in Sect. 3.4 to convert Reluplex from a verifier into an optimizer. First, we define the SPLIT($S$, $P$) and VIOLATED($S$, $P$) functions used in the original Reluplex; then, we discuss how to convert the VIOLATED($S$, $P$) function into a corresponding OPTIMUMFORREGION($S$, $P$, optSoFar) function.

In order to solve a verification problem with a search with optimization technique, we need to define two functions: SPLIT($S$, $P$) and VIOLATED($S$, $P$). The *Split* function is meant to take a state and break it down into two (or more) states which will be easier to solve. This guides the search process. Reluplex searches over the activation states of the network. It does this in an incremental fashion, starting at a state where no ReLUs are fixed and then proceeding to fix ReLUs one by one during the search. Here, the state $S$ will be a partial activation state. The implementation of *Split* will choose a node $(i, j)$ that is undetermined in the current state and return two new states: the first state will be the current state with the additional node $(i, j)$ fixed to be active, and the second state will be the same except with node $(i, j)$ fixed to be inactive. There are a variety of possible strategies to pick this node to fix, including choosing the earliest unfixed ReLU or choosing the ReLU with the largest violation. Bounds on the variables may fix some ReLUs to be a certain phase, in which case those ReLUs will not need to be split.

The function VIOLATED($S$, $P$) should reason about whether a property $P$ holds for the partial activation state $S$. Reluplex accomplishes this by relaxing each undetermined ReLU from $z = max(0, \hat{z})$ to $z \geq 0 \wedge z \geq \hat{z}$. A variety of other relaxations for a ReLU are possible (Liu et al. 2021). Using this relaxation means that these undetermined ReLUs can each be written with two linear constraints, allowing the relaxed feasibility problem to be an LP. Recall that $\mathcal{L}$ represents the indices of ReLU layers, $\mathcal{I}$ represents the indices of identity layers, $\mathcal{A}$ gives the set of active nodes in the activation state, $\mathcal{N}$ gives the set of inactive nodes for the activation state, and $\mathcal{U}$ gives the set of undetermined nodes for the activation state. We also assume we have upper and lower bounds on each variable denoted as $\hat{U}_{(i,j)}$ and $\hat{L}_{(i,j)}$ for pre-activation variables, and $U_{i,j}$ and $L_{i,j}$ for post activation variables for each node $(i, j)$. With all of these variables defined, the relaxed feasibility problem can now be formulated as

$$
\begin{aligned}
\underset{\mathbf{x},\hat{\mathbf{z}},\mathbf{z}}{\text{maximize}} \quad & 0 \\
\text{subject to} \quad & \mathbf{x} \in \mathcal{X} \\
& \mathbf{z}_K \in \mathcal{Y}^{\complement} \\
& \hat{\mathbf{z}}_{i+1} = \mathbf{W}_i \mathbf{z}_i + \mathbf{b}_i, \ i = 1, \dots, K-1 \\
& \mathbf{z}_i = \hat{\mathbf{z}}_i, \ i \in \mathcal{I} \\
& z_{i,j} = \hat{z}_{i,j}, \ (i,j) \in \mathcal{A} \\
& z_{i,j} = 0, \ (i,j) \in \mathcal{N} \\
& z_{i,j} \geq z_{i,j} \ (i,j) \in \mathcal{U} \\
& z_{i,j} \geq 0 \ (i,j) \in \mathcal{U} \\
& \hat{L}_{i,j} \leq \hat{z}_{i,j} \leq \hat{U}_{i,j} \\
& L_{i,j} \leq z_{i,j} \leq U_{i,j}
\end{aligned} \tag{17}
$$

Recall that the complement of the output set is applied as a constraint, representing the search for a counter-example.

Since this problem is a relaxation, if the problem is infeasible then the exact problem must also be infeasible. If the problem is feasible, then the satisfying assignment from the LP can be checked to see if it is consistent with the exact problem. If it is, then we have found a counter-example. If not, then the feasibility of the exact problem remains unknown. These observations can be used to construct the VIOLATED function.

To extend the Reluplex algorithm to perform optimization, we need to extend the functionality of the VIOLATED($S$, $P$) procedure to create a OPTIMUMFORREGION($S$, $P$) procedure. We will convert the feasibility problem used to implement VIOLATED($S$, $P$), Eq. (17) into a linear program which will provide us an upper bound on the objective. This requires adding an objective to the problem being solved just like with purely optimization approaches in Sect. 3.3. If we have an objective function $g(\mathbf{x})$ and assume without loss of generality the objective is being maximized, we arrive at the formulation

$$
\begin{aligned}
\underset{\mathbf{x},\hat{\mathbf{z}},\mathbf{z}}{\text{maximize}} \quad & g(\mathbf{x}) \\
\text{subject to} \quad & \mathbf{x} \in \mathcal{X} \\
& \mathbf{z}_K \in \mathcal{Y}^{\complement} \\
& \hat{\mathbf{z}}_{i+1} = \mathbf{W}_i \mathbf{z}_i + \mathbf{b}_i, \ i = 1, \dots, K-1 \\
& \mathbf{z}_i = \hat{\mathbf{z}}_i, \ i \in \mathcal{I} \\
& z_{i,j} = \hat{z}_{i,j}, \ (i,j) \in \mathcal{A} \\
& z_{i,j} = 0, \ (i,j) \in \mathcal{N} \\
& z_{i,j} \geq \mathbf{z}_{i,j} \ (i,j) \in \mathcal{U} \\
& z_{i,j} \geq 0 \ (i,j) \in \mathcal{U} \\
& \hat{L}_{i,j} \leq \hat{z}_{i,j} \leq \hat{U}_{i,j} \\
& L_{i,j} \leq z_{i,j} \leq U_{i,j}
\end{aligned} \tag{18}
$$

The objective function, input set, and output set can be used to represent output optimization problems and minimum input perturbation problems. The resulting optimal value $p^*$

**Fig. 1** Example search tree with LP objective values

No ReLUs fixed: 20
1 ReLU fixed: 17, $-\infty$
2 ReLUs fixed: 9, 7

from this LP provides an upper bound on the true optimal value for this activation state since the LP comes from a relaxation of the exact optimization problem for this region.

In order to implement OptimumForRegion($S$, $P$, optSoFar) we first solve the LP in Eq. (18). If the optimal value, representing an upper bound on the objective, is lower than optSoFar, then status WorseThanOpt is returned. Otherwise, the procedure checks whether the found upper bound can actually be achieved—this is done by passing the corresponding input value through the network, or checking whether all ReLU constraints are met. If it does, it is the optimum value for the region, and status Optimal with this optimal objective value and assignment is returned. Otherwise, status Unknown is returned, indicating that the search should continue in sub-regions. The relaxed LP becomes exact when all activation states are defined; therefore, this approach is guaranteed to find the optimum in a fully defined activation state, i.e., in a leaf node of the binary search tree representing the activation space. This ensures that the procedure will return an optimum in every leaf node, ensuring termination of the algorithm.

### 4.1 Example

The optimization process is illustrated in Fig. 1. Each node contains the optimum value of the relaxed LP associated with that partial activation state. Gray shading indicates that the optimum of the relaxed LP satisfies all linear and ReLU constraints and so is a valid candidate for global optimum. This figure shows that the algorithm performs two splits and then finds a solution that satisfies both the linear and nonlinear constraints (the gray shaded node). The maximum value found is 9. This can then be used to trim the branch to its right, where the best value found is only 7. The rightmost node is found to be infeasible, completing the search. The optimal value is 9.

## 5 Experiments and results

In this section, we evaluate the performance of MarabouOpt on a diverse set of optimization queries, including safety properties of control systems and robustness properties of perception models.

### 5.1 Implementation and experimental setup

We extend Marabou (Katz et al. 2019), an open-source neural network verification tool implementing the Reluplex algorithm, to support solving optimization queries, using the

method described in Sects. 3.4 and 4. Marabou integrates the symbolic bound tightening techniques introduced in Wang et al. (2018b) and a modified version of the progressive bound tightening preprocessing pass introduced in Tjeng et al. (2019).[2] Note that while the tool supports parallelism in both preprocessing and solving (Wu et al. 2020), the results here do not make use of any parallelism. Integrating our optimizing extension with the parallel features of Marabou is a promising avenue for future work.

We refer to the optimizing extension of Marabou as MarabouOpt. Given an optimization query and a timeout, MarabouOpt returns Infeasible, Timeout, or the variable assignment for the optimal solution. We compare MIPVerify (Tjeng et al. 2019) to MarabouOpt, as it can directly solve the same set of benchmarks. By extending other verifiers according to our framework, for example those described in the survey by Liu et al. (2021) or those in the international verification of neural networks competitions (Johnson and Liu 2020; Bak et al. 2021), it may be possible to create a stronger tool to compare against than MIPVerify.

As an additional baseline, we implement a binary search to solve optimization problems with a series of calls to the Marabou verifier and the approach described in Sect. 3.1. We refer to this optimizer as MarabouBin. For output optimization queries, we first perform calls to the verifier to find an upper bound for the objective, and then begin the binary search. For minimum adversarial input queries we begin with an upper and lower bound and can thus immediately start the binary search in the middle of that interval. In all cases, we continue narrowing the interval until an optimality gap less than $10^{-4}$ is achieved. Since the minimum adversarial input queries chosen have the potential to be unsatisfiable (there is no adversarial input in the given region), the binary search will return that the query is unsatisfiable if it does not find an adversarial example to within $10^{-4}$ of the border of the region. We choose to start in the middle instead of first checking whether the full region is satisfiable, as we expect many queries to be satisfiable and queries that include the full input region to be quite expensive. By starting in the middle, we potentially avoid ever needing to consider substantial portions of the full input region if we find an adversarial example early enough. This is a design choice for MarabouBin which we expect to produce different times for satisfiable and unsatisfiable queries than if we had first checked satisfiability.

MIPVerify tightens the bounds of the input to each neuron with an MIP-solver and then solves the preprocessed queries using a Mixed-integer encoding of the problem. Marabou employs a similar preprocessing pass, except that the tightening is done on both the input variable and the output variable of each neuron. The timeout per preprocessing query for both MarabouOpt and MIPVerify is 1 s, while that for MarabouBin is 0.5 s. We obtain these values by performing a grid search of this parameter for each solver on a subset of the benchmarks.

In addition to complete methods, we also compare against three approximate approaches, including Projected GradientDescent (PGD), Limited-memory BFGS (LBFGS), and the Fast Gradient SignedMethod (FGSM).

[2] Marabou also later integrated the DeepPoly analysis from Singh et al. (2019b), which can derive tighter bounds then the symbolic bound tightening technique. This experimental evaluation was conducted before DeepPoly was made available in Marabou.

### 5.2 Benchmarks

The benchmark sets consist of network-query pairs, with networks from three different application domains: aircraft collision avoidance (ACAS Xu), aircraft localization (TinyTaxiNet), and digit recognition (MNIST). Optimization queries include both output optimization problems and minimum adversarial perturbation problems.

**ACAS Xu Family** The ACAS Xu family of benchmarks, introduced in Katz et al. (2017), implements a prototype aircraft collision avoidance system—advising course corrections based on the relative positions of two aircraft. The system consists of 45 fully-connected feed-forward neural networks, each with 6 hidden layers and 50 ReLU nodes per layer. Each network uses 5 inputs to describe the encounter geometry and produces 5 outputs—the predicted cost of following each action. The system chooses the action with the lowest cost as the advisory. We consider both output optimization queries and minimum adversarial perturbation queries on the 45 networks.

For output optimization queries, the objective is maximizing $y_{real} - y_{adv}$, where $y_{real}$ is the expected output in the given input region, and $y_{adv}$ is an adversarial output. For minimum adversarial perturbation queries, the objective is to minimize the perturbation on one input dimension that would result in an adversarial output. The input regions used in the output optimization queries are from properties 1–4 in Katz et al. (2017), which apply to all 45 networks. To construct the minimum adversarial perturbation queries we adopt the input and output constraints from property 2. We then consider perturbations on a single input at a time. We set the objective for each query to be to minimize the perturbation from the center for a single input dimension. This results in 5 distinct queries per network, one for each input dimension. In total, this yields 180 ($45 \times 4$) output optimization queries and 225 ($45 \times 5$) input optimization queries.

**TinyTaxiNet** The TinyTaxiNet family of benchmarks consist of perception networks that predict the aircraft position on the taxiway relative to the center-line. The output is used by a controller that adjusts the trajectory to correct the position of the aircraft. The input to the network is a gray-scale image compressed to $16 \times 8$ pixels, with values ranging between [0, 1]. The networks produce two outputs: the lateral distance to the runway center-line and the heading angle with respect to the center-line. We evaluated on three network architectures, each consisting of one convolution layer and 2 feed-forward layers. The networks have a total of 32, 64, and 128 ReLUs, respectively.

We consider the problem of output optimization on these benchmarks. The task is to maximize the predicted lateral distance to the runway center-line. The input region is a hyper-cube parameterized by the centroid and the radius. For each network, we generate 60 such queries, with centroids randomly sampled from the training data and the radius sampled evenly from the set $\{0.04, 0.08, 0.016\}$.

**MNIST** We also evaluated MarabouOpt on four fully-connected feed-forward networks trained on the MNIST dataset of handwritten digits. Each network has 784 inputs (representing a grey-scale image) with value range [0, 1] and 10 outputs (each representing a digit). We trained 4 models. MNIST1 and MNIST2 consist of 10 layers each, with 10 and 20 ReLU nodes per layer respectively. MNIST3 and MNIST4 consist of 20 layers each,

with 20 and 40 ReLU nodes per layer respectively. The networks have approximately 95% accuracy on the MNIST test set. The range of widths and depths of these networks, with a maximum of 800 ReLUs, allows for queries of varying computational difficulty. State-of-the-art approximate verification techniques on local robustness queries can often scale to much larger networks, even some with hundreds of thousands of nodes (Müller et al. 2020).

We consider minimum adversarial perturbation queries on the MNIST networks. The task is to minimize the $L_\infty$ perturbation on the input image that results in an adversarial output. We generate 50 such queries for each network by randomly choosing training images, 5 from each class, and corresponding target labels. We set the radius of the input region to be 0.05, which corresponds to a maximal perturbation of 13 pixel-values. These queries will be infeasible if there is no adversarial input within the input region. The extended verifiers we propose can handle infeasible queries.

In summary, the full benchmark set consists of 225 minimum adversarial perturbation queries on ACAS Xu networks (**ACAS In.**), 180 output optimization queries on the ACAS Xu networks (**ACAS Out.**), 180 output optimization queries on the TinyTaxiNets (**Taxi Out.**), and 200 minimum adversarial perturbation queries on the MNIST networks (**MNIST Out.**).

### 5.3 Experimental evaluation

In this section, we present results of the following experiments:

1. Evaluation of the runtime performance of MarabouOpt and MarabouBin on the three benchmark sets. We compare against MIPVerify, a state-of-the-art solver, on the same benchmarks.
2. Comparison of the objective values found by approximate methods with the true optimums found by exact methods.

We run all experiments on a cluster equipped with Intel Xeon E5-2620 v4 cpus running Ubuntu 16.04. One processor with 8GB RAM was allocated for each job, and each job is given a 2-h CPU timeout.

### 5.4 Runtime evaluation

We ran MarabouOpt, MarabouBin, and MIPVerify on all benchmarks. The number of solved instances and total runtime of the solved instances are shown in Table 1. For each benchmark set, we highlight the solver that solves the most instances.

**MarabouOpt versus MarabouBin** As shown by Table 1, MarabouOpt outperforms MarabouBin on three of the four benchmark sets, showing that our integrated optimization extension of the Marabou verifier is overall more effective than the black-box approach using bisection search. Figure 2 shows a scatter plot of the runtime of the two solvers on all benchmarks. Points with value 7200 on the $x$ and $y$ axes denote timeout of the tool on the respective axis. Figure 2 shows that MarabouOpt significantly outperforms Marabou-Bin evidenced by the concentration of points below the dashed diagonal line. Surprisingly,

**Table 1** Number of solved instances and runtime in seconds

| Solver | MNIST In. (200) | | ACAS In. (225) | | Taxi out. (180) | | ACAS Out. (180) | |
|---|---|---|---|---|---|---|---|---|
| | No. of solved | Time | No. of solved | Time | No. of solved | Time | No. of solved | Time |
| MIPVerify | **61** | 99,127 | 138 | 300,558 | **180** | 3150 | 90 | 1169.4 |
| Marabou-Bin | 29 | 53,301 | 123 | 504,615 | 120 | 20,237 | 82 | 107,422 |
| Mara-bouOpt | 14 | 13,209 | **174** | 226,431 | 126 | 6144 | **97** | 61,864 |

The largest number of solved instances for each benchmark set is given in bold

Fig. 2 shows that MarabouBin outperforms MarabouOpt on some of the MNIST and ACAS input queries, which merits further investigation.

**MarabouOpt versus MIPVerify** On the other hand, MarabouOpt and MIPVerify show strengths on different benchmark sets. While MIPVerify outperforms MarabouOpt on the MNIST and TinyTaxiNet benchmark sets, MarabouOpt solves more instances of both input and output optimization queries on the ACAS Xu networks. The complementary nature of the two solvers is further illustrated by Fig. 3. The cluster of queries on the bottom left boundary with $x$ less than 300 and $y$ up to 3000 suggests that a subset of output optimization problems on the ACAS Xu benchmarks can be quickly resolved by MIPVerify while taking non-trivial time for MarabouOpt. On the other hand, the points in the right region with $x$ equal to 7200 suggest that MarabouOpt is able to solve some of the harder ACAS Xu queries that MIPVerify cannot handle. The horizontal cluster with $y$ around 900 suggests that a number of minimum adversarial perturbation benchmarks on ACAS Xu networks take Marabou around 900 s to solve. This is because many of those queries are quickly resolved by MarabouOpt after the preprocessing pass.

**Comparing solved instances for all solvers** Figure 4 shows the number of commonly and uniquely solved benchmarks by the three solvers. There are 86 instances that can be uniquely solved by MIPVerify, and 36 instances that can be uniquely solved by MarabouOpt. The two solvers combined can cover over 99% of the solved instances. Interestingly, there are 3 instances that only MarabouBin can solve. Upon closer examination, they are all minimum adversarial input queries.

In total, 522 out of the 785 benchmarks are solved. MarabouOpt solved 12 infeasible queries, while MIPVerify solved 6. Figures 5 and 6 are two examples of the minimum adversarial inputs found by MarabouOpt on MNIST1. In particular, a hardly discernible perturbation (0.001) can result in a misclassification in Fig. 5, while it requires a perturbation with $L_\infty$ norm of at least 0.048 (corresponding to 12 pixel-values) for the network to mistakenly classify the image as a “7” in Fig. 6, suggesting that the same network can exhibit very different local adversarial robustness properties in different input regions.

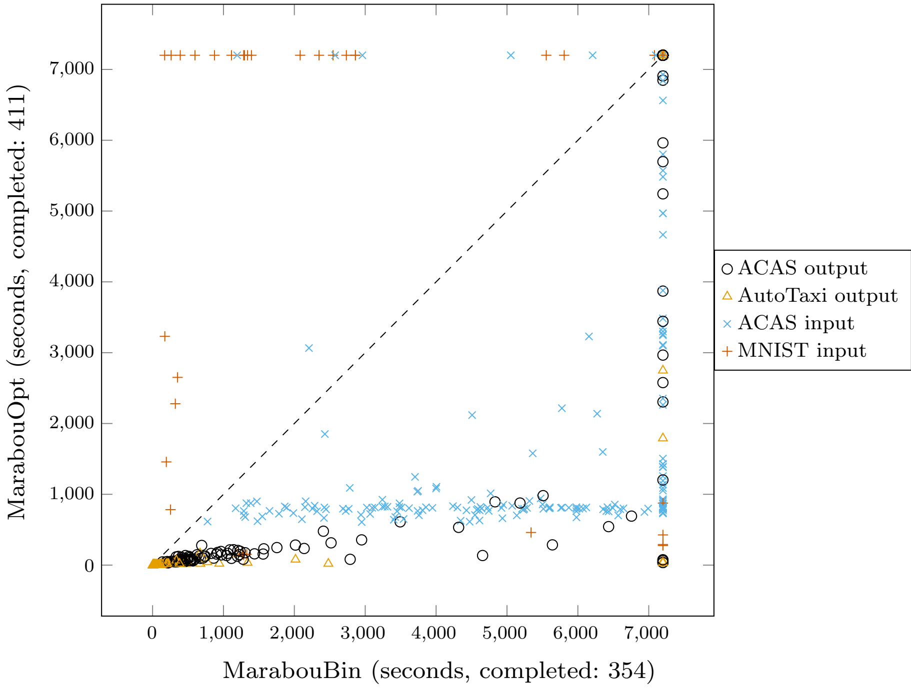


**Fig. 2** Runtime comparison of MarabouOpt and MarabouBin

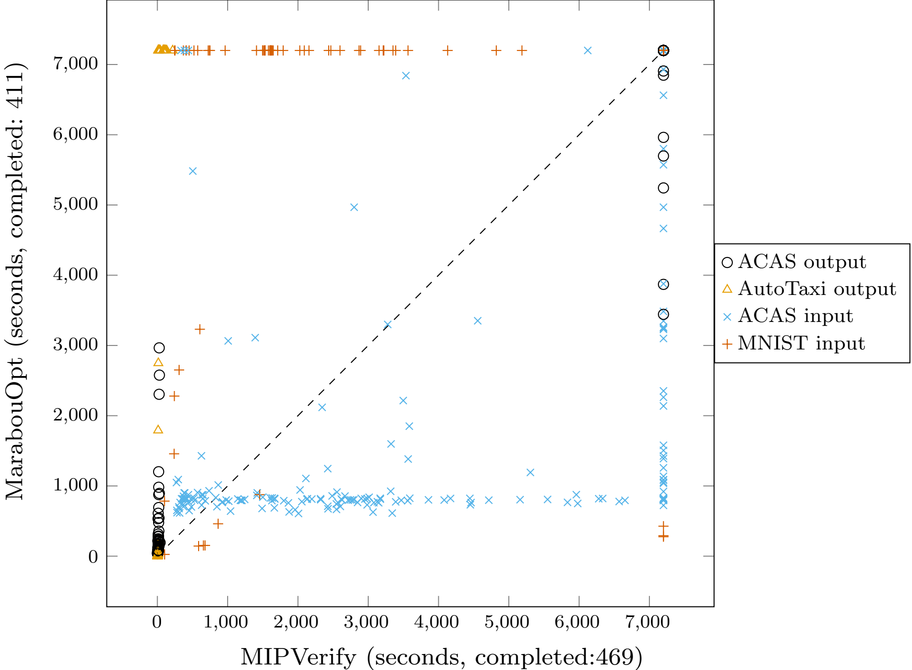


**Fig. 3** Runtime comparison of MarabouOpt and MIPVerify

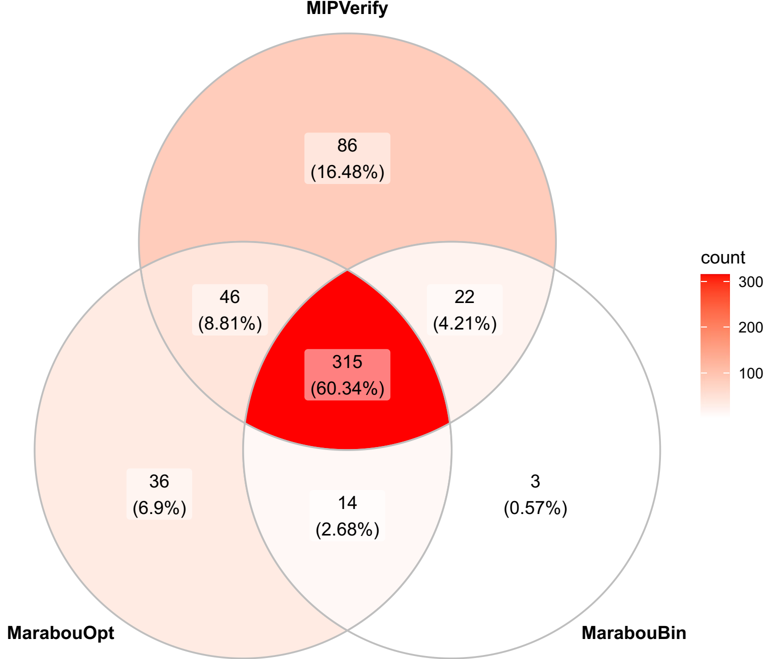


**Fig. 4** Number of commonly and uniquely solved benchmarks

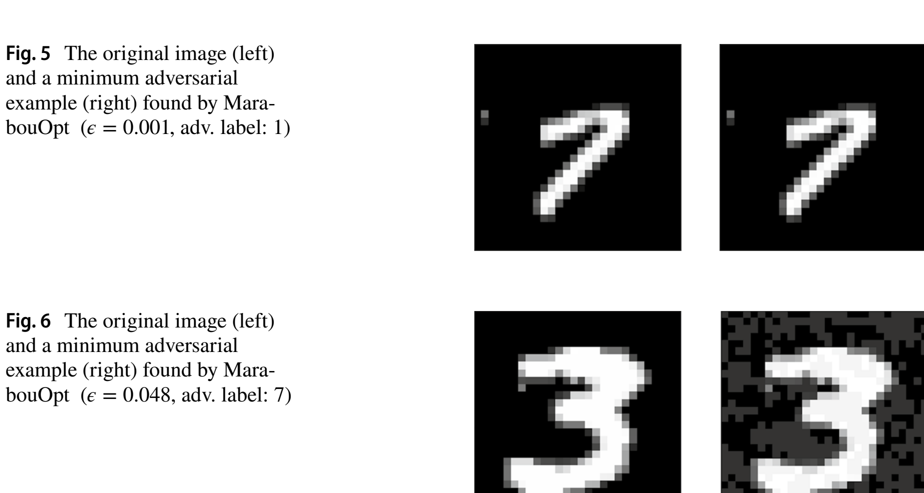

**Fig. 5** The original image (left) and a minimum adversarial example (right) found by MarabouOpt ($\epsilon = 0.001$, adv. label: 1)

**Fig. 6** The original image (left) and a minimum adversarial example (right) found by MarabouOpt ($\epsilon = 0.048$, adv. label: 7)

## 5.5 Objective value evaluation: exact versus approximate optimization

In this subsection, we compare the performance of exact and approximate optimization approaches. We evaluate several tools on output optimization queries, and discuss how additional tools could be applied to minimum adversarial input queries.

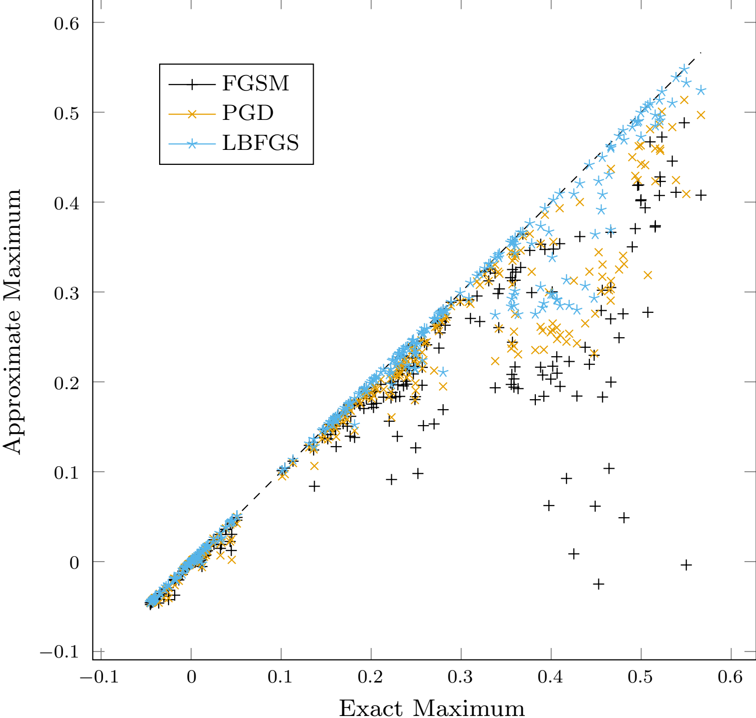


**Fig. 7** Exact versus approximate methods on output optimization problems. For each point, the position on the $x$ axis shows the true maximum found with MarabouOpt, MIPVerify, or MarabouBin while the position on the $y$ axis shows the objective value achieved by the approximate solvers (FGSM, PGD, or LBFGS). The dotted line is the line $y = x$. A perfect "approximate" optimizer that always achieves the true optimum would have points along the dashed line. As a result, the vertical gap between a point and the dotted line represents the gap between the exact maximum and the approximate value

**Output optimization queries** We run three approximate methods on all of the output optimization queries. We then compare the approximate values with the exact values and compare the runtimes. The exact values are obtained from MIPVerify, MarabouOpt, or MarabouBin, while the approximate values are obtained from the gradient-based optimizers, which includes LBFGS, PGD, and FGSM. The values are compared in Fig. 7, with each point corresponding to a single query. The vertical distance between a point and the dotted line represents the gap between the approximate value and the true optimum. We observe that LBFGS most closely matches the exact values. We also observe that the gap between exact and approximate optimizers can be substantial, as evidenced by the points on the right side of the figure. Table 2 provides statistics on the runtime of each approximate optimizer. Note that an issue with our implementation of LBFGS caused several queries to take much longer than the others, skewing its mean upwards. Comparing the medians, we see that FGSM was the fastest, followed by PGD, and then LBFGS. All three approximate solvers typically return within a second and should scale polynomially in the size of networks instead of the exponential scaling we observe for the exact optimizers. These results suggest that for scenarios where a guarantee of optimality is not needed,

**Table 2** Statistics on the runtime of FGSM, PGD, and LBFGS for their solved output optimization query instances

| Optimizer | Minimum (s) | Mean (s) | Median (s) | Maximum (s) |
|---|---|---|---|---|
| FGSM | 0.1 | 0.14 | 0.13 | 0.31 |
| PGD | 0.09 | 0.23 | 0.19 | 0.96 |
| LBFGS | 0.02 | 68.7 | 0.35 | 5893 |

approximate optimizers may often be “close enough” and provide a value in significantly less time. However, if strict guarantees are required, then an exact optimizer may be the better choice.

**Minimum adversarial perturbation queries** A similar experiment could be performed for the minimum adversarial perturbation queries. PGD or LBFGS could be run with a loss that incorporates the size of the perturbation as in the work of Yuan et al. (2019) to upper bound the minimum perturbation. An experiment like this was performed by Carlini et al. (2017) to compare approximate and exact minimum adversarial perturbations. Alternatively, Fast-Lin, Fast-Lip, CROWN, and CNN-Cert are designed to provide a certified lower bound on the minimum perturbation which could be compared with the true minimum (Weng et al. 2018; Zhang et al. 2018; Boopathy et al. 2019). Although we did not run these experiments, they remain of interest for future work.

## 6 Related work

In this section, we summarize existing neural verification and neural optimization methods. We organize the discussion using the categorization of neural verifiers introduced by Liu et al. (2021). For optimization, we focus on works relevant to the two specific optimization problems addressed in this paper: output optimization problems and minimum adversarial input problems (see Sect. 2.3).

**Neural verification methods** Neural verification methods check properties about the input–output relationship of a network. Specifically, given an input set and an output set, a verification method proves that all elements of the input set are mapped by the network into the output set. Section 2 provides notation for this problem.

Liu et al. (2021) separate verification algorithms into four categories: reachability, optimization, search with reachability, and search with optimization. In this work we make particular reference to reachability verifiers ExactReach and NNV (Xiang et al. 2018; Tran et al. 2020b), optimization verifiers NSVerify and MIPVerify (Lomuscio and Maganti 2017; Tjeng et al. 2019), search with reachability verifiers ReluVal and Neurify (Wang et al. 2018b, a), and search with optimization verifiers Reluplex and Venus (Katz et al. 2017, 2019; Botoeva et al. 2020). We refer the reader to Sect. 2.1 or the survey paper (Liu et al. 2021) for a more thorough description and discussion. Additionally, the 1st and 2nd International Verification of Neural Networks Competitions boast an extensive list of participating verification algorithms for a curious reader to explore (Johnson and Liu 2020; Bak et al. 2021). These tools typically fit into the categorization presented by Liu et al. (2021). Although we have not enumerated each solver here in favor of describing a few

representative verifiers from each category, each verifier can be extended to perform optimization using the framework described in this paper. Bunel et al. (2020) also provide a framework for neural verification in terms of a branch and bound search. Both frameworks are very useful for conceptualization of how different verification algorithms function and their common high-level structure.

**Exact optimization methods** There are several existing techniques for solving general optimization problems on neural networks. These problems are directly encoded as mixed integer programs (MIP) (Wolsey 1998; Cheng et al. 2017; Fischetti and Jo 2018; Lomuscio and Maganti 2017) and solved by an MIP solver such as the open source solver GLPK (Makhorin 2004) or the highly optimized commercial solver Gurobi (Gurobi Optimization 2020). MIPVerify (Tjeng et al. 2019) employs a progressive bound-tightening preprocessing step and applies an MIP solver to the problem. This preprocessing pass has a significant impact on computation time, and a similar preprocessing pass is implemented in MarabouOpt. Another approach treats existing neural verifiers as a black box, performing a bisection method using multiple verification calls. This method has been applied in adaptive stress testing of a control network (Julian et al. 2020) and to find minimally distorted adversarial examples (Carlini et al. 2017). Optimization modulo theory solvers can also handle complex, non-convex optimization problems (Bjorner et al. 2015; Sebastiani and Trentin 2015, 2020). However, these solvers are based on satisfiability modulo theories (SMT) technology and perform computation over real arithmetic, which has been reported to scale poorly compared to the less precise floating point optimizers (Katz et al. 2017).

**Approximate optimization methods** There is also a rich body of literature on approximate techniques for both optimization problems, particularly within the field focused on adversarial example discovery. Generation of adversarial examples typically combines a local optimization method with heuristics and applies it to a neural network. Approximate approaches exchange optimality for computational efficiency and often focus on minimum perturbation problems (Szegedy et al. 2014; Carlini and Wagner 2017). Yuan et al. (2019) offer a taxonomy of these techniques in their survey. We address the methods relevant for this work. LBFGS is a quasi-newtonian optimization method that finds an approximate minimum perturbation adversarial input (Zhu et al. 1997; Szegedy et al. 2014). The verification tools Fast-Lin, Fast-Lip, CROWN, and CNN-Cert provide a lower bound on the minimum adversarial distortion and can be considered approximate optimizers as they yield an approximate value (lower bound) for the minimum distortion (Weng et al. 2018; Zhang et al. 2018; Boopathy et al. 2019). FGSM and PGD have been used to directly generate adversarial examples within a small input region, corresponding to our output optimization problem, instead of finding minimum perturbation adversarial examples (Goodfellow et al. 2015; Madry et al. 2018). LBFGS can be used in this way as well.

## 7 Conclusions

This paper presents general strategies for extending different categories of neural verifiers to solve global optimization problems. It focuses on two classes of problems: output optimization and minimum adversarial perturbation problems. We extended the Marabou neural verifier to create an optimizer MarabouOpt and compared its runtime performance against the black-box bisection search and MIPVerify. We observed that on a

significant majority of queries, MarabouOpt substantially outperformed a naive bisection based approach, showing the advantages of tight integration that can be achieved with proposed extension strategies. The comparison of MarabouOpt and MIPVerify shows complementary performance, indicating that different optimizers have both strengths and weaknesses and that extensions of each verifier should be explored. Our comparison of the global optima found by these solvers to local optima from several adversarial attacks found that although the optima were often similar, there were some marked differences.

Neural optimization problems have a wide variety of applications and merit deeper investigation. By encouraging the development of optimization techniques, we hope that more tools will be available to those working to verify safety critical systems.

Further work is needed to determine which verifiers will work best when extended to perform optimization tasks. This could consist of extending several more algorithms and comparing their performance against MarabouOpt and MIPVerify. Additional work would also lie in incorporating some of the best practices from each technique into the others. For example, integrating MIPVerify's optimization-based preprocessing step (Tjeng et al. 2019) as we did with MarabouOpt may provide other verifiers with a performance boost as well.

Incomplete solvers could also be used to obtain bounds on the optima for our optimization problems. For example, Ai2 and its updated version ERAN can compute an approximate reachable set for each layer (Gehr et al. 2018; Singh et al. 2018a, b; Balunovic et al. 2019; Singh et al. 2019a, b). Maximizing an output objective over this approximate reachable set will give us an upper bound on the true optimum. For optimization-based methods, these bounds on the objective can be applied as constraints before solving. For some use cases, these bounds may provide enough information if an exact solution is not needed.

There are several promising directions to improve MarabouOpt. Many of these involve reducing the depth of the tree the algorithm explores. Some directions include further developing our bound-tightening strategies, more intelligently choosing the node to split on, incorporating input splitting like in Marabou's *Split-and-Conquer* strategy (Katz et al. 2019), and incorporating MIPVerify's preprocessing step. In addition, parallelization could be incorporated into the optimizer. Marabou has been able to use parallelism to great effect (Wu et al. 2020). The same extensions could be explored with MarabouOpt.

**Acknowledgements** We would like to acknowledge support from Tomer Arnon, Christopher Lazarus, Changliu Liu, Ahmed Irfan, Chelsea Sidrane, Jayesh Gupta, Alex Usvyatsov, Rianna Jitosho, Eric Luxenberg, and Katherine Strong.

**Funding** Funding in direct support of this work: DARPA under contract FA8750-18-C-0099.

**Availability of data and materials** The networks and datasets used for testing can be found at https://github.com/castrong/NeuralOptimization.jl.

**Code availability** The benchmarking framework is available at https://github.com/castrong/NeuralOptimization.jl. The Marabou verifier can be found at https://github.com/NeuralNetworkVerification/Marabou with the optimization extension at https://github.com/castrong/Marabou/tree/opt_branch_7_30. A wrapper to run MIPVerify on these benchmarks is located at https://github.com/castrong/MIPVerifyWrapper and the original implementation is located at https://github.com/vtjeng/MIPVerify.jl. Implementations of adversarial attacks can be found at https://github.com/jaypmorgan/Adversarial.jl.

## Declarations

**Conflict of interest** The authors declare that they have no conflict of interest.

## References


Bak, S., Liu, C., & Johnson, T. T. (2021). VNN21. https://sites.google.com/view/vnn2021. Accessed 17 July 2021

Balunovic, M., Baader, M., Singh, G., Gehr, T., & Vechev, M. (2019). Certifying geometric robustness of neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 15313–15323).

Bjorner, N., Phan, A. D., & Fleckenstein, L. (2015). *ν*z-an optimizing SMT solver. In *International conference on tools and algorithms for the construction and analysis of systems (TACAS)* (pp. 194–199). Springer.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). End to end learning for self-driving cars. Technical Report. arXiv:1604.07316

Boopathy, A., Weng, T. W., Chen, P. Y., Liu, S., & Daniel, L. (2019). Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. *AAAI Conference on Artificial Intelligence, 33,* 3240–3247.

Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., & Misener, R. (2020). Efficient verification of relu-based neural networks via dependency analysis. In *AAAI conference on artificial intelligence (AAAI).*

Bunel, R., Lu, J., Turkaslan, I., Kohli, P., Torr, P., & Mudigonda, P. (2020). Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research, 21*(2020), 1–39.

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

Carlini, N., Katz, G., Barrett, C., & Dill, D. L. (2017). Provably minimally-distorted adversarial examples. arXiv preprint arXiv:170910207

Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., & Mukhopadhyay, D. (2021). A survey on adversarial attacks and defences. *CAAI Transactions on Intelligence Technology, 6*(1), 25–45.

Cheng, C. H., Nührenberg, G., & Ruess, H. (2017). Maximum resilience of artificial neural networks. In *International symposium on automated technology for verification and analysis* (pp. 251–268). Springer.

Fischetti, M., & Jo, J. (2018). Deep neural networks and mixed integer linear optimization. *Constraints, 23*(3), 296–309. https://doi.org/10.1007/s10601-018-9285-6.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., & Vechev, M. (2018). AI2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE symposium on security and privacy (S&P).*

Goodfellow, I., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *International conference on learning representations*. arxiv.org/abs/1412.6572

Gurobi Optimization L (2020). Gurobi optimizer reference manual. http://www.gurobi.com

Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. (2017). Safety verification of deep neural networks. In *International conference on computer-aided verification* (pp. 3–29).

Hunt, K. J., Sbarbaro, D., Zbikowski, R., & Gawthrop, P. J. (1992). Neural networks for control systems—a survey. *Automatica, 28*(6), 1083–1112.

Johnson, T. T., & Liu, C. (2020). Vnn20. https://sites.google.com/view/vnn20/. Accessed 17 July 2021.

Julian, K., Lopez, J., Brush, J., Owen, M., & Kochenderfer, M. (2016). Policy compression for aircraft collision avoidance systems. In *Digital avionics systems conf. (DASC)* (pp. 1–10).

Julian, K. D., Lee, R., & Kochenderfer, M. J. (2020). Validation of image-based neural network controllers through adaptive stress testing. In *2020 IEEE 23rd international conference on intelligent transportation systems (ITSC)* (pp. 1–7).

Katz, G., Barrett, C., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer-aided verification* (pp. 97–117). Springer.

Katz, G., Huang, D. A., Ibelingm, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., & Zeljić, A., et al. (2019). The marabou framework for verification and analysis of deep neural networks. In *International conference on computer-aided verification* (pp. 443–452). Springer.

Kochenderfer, M. J., & Wheeler, T. A. (2019). *Algorithms for optimization*. London: MIT Press.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 1097–1105).

Le, Q. V. (2013). Building high-level features using large scale unsupervised learning. In *IEEE international conference on acoustics, speech and signal processing*(pp. 8595–8598).

Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., & Kochenderfer, M. J. (2021). Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization,4*(3–4), 244–404. https://doi.org/10.1561/2400000035

Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing, 234,* 11–26.

Lomuscio, A., & Maganti, L. (2017). An approach to reachability analysis for feed-forward relu neural networks. arXiv preprint arXiv:170607351

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International conference on learning representations*. https://openreview.net/forum?id=rJzIBfZAb

Makhorin, A. (2004). GLPK (Gnu linear programming kit), version 4.42. http://www.gnu.org/software/glpk

Müller, C., Singh, G., Püschel, M., & Vechev, M.T. (2020). Neural network robustness verification on gpus. CoRR arxiv.org/abs/2007.10868

Otter, D. W., Medina, J. R., & Kalita, J. K. (2020). A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems,* 1–21.

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why should I trust you?" explaining the predictions of any classifier. In *ACM SIGKDD International conference on knowledge discovery and data mining* (pp. 1135–1144).

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks, 61,* 85–117.

Sebastiani, R., & Trentin, P. (2015). Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In *International conference on tools and algorithms for the construction and analysis of systems (TACAS)* (pp. 335–349). Springer.

Sebastiani, R., & Trentin, P. (2020). Optimathsat: A tool for optimization modulo theories. *Journal of Automated Reasoning, 64*(3), 423–460.

Singh, G., Gehr, T., Mirman, M., Püschel, M., & Vechev, M. (2018a). Fast and effective robustness certification. In *Advances in neural information processing systems (NIPS)* (pp. 10802–10813).

Singh, G., Gehr, T., Püschel, M., & Vechev, M. (2018b). Boosting robustness certification of neural networks. In *International conference on learning representations*.

Singh, G., Ganvir, R., Püschel, M., & Vechev, M. (2019a). Beyond the single neuron convex barrier for neural network certification. In *Advances in neural information processing systems (NIPS)* (pp. 15098–15109).

Singh, G., Gehr, T., Püschel, M., & Vechev, M. (2019b). An abstract domain for certifying neural networks. In *Proceedings of the ACM on programming languages 3(POPL)* (pp. 1–30).

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). Intriguing properties of neural networks. In *International conference on learning representations*. arxiv.org/abs/1312.6199

Tjeng, V., Xiao, K. Y., & Tedrake, R. (2019). Evaluating robustness of neural networks with mixed integer programming. In *International conference on learning representations*. https://openreview.net/forum?id=HyGIdiRqtm

Tran, H. D., Lopez, D. M., Musau, P., Yang, X., Nguyen, L. V., Xiang, W., & Johnson, T. T. (2019). Star-based reachability analysis of deep neural networks. In *International symposium on formal methods* (pp. 670–686). Springer.

Tran, H. D., Bak, S., Xiang, W., & Johnson, T. T. (2020a). Verification of deep convolutional neural networks using imagestars. In *International conference on computer aided verification* (pp. 18–42). Springer.

Tran, H. D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L., Xiang, W., et al. (2020). Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. *Computer Aided Verification, 12224,* 3–17.

Vincent, J. A., & Schwager, M. (2021). Reachable polyhedral marching (rpm): A safety verification algorithm for robotic systems with deep neural network components. In *IEEE international conference on robotics and automation (ICRA)*.

Wang, S., Pei, K., Whitehouse, J., Yang, J., & Jana, S. (2018a). Efficient formal safety analysis of neural networks. In *Advances in neural information processing systems (NIPS)* (pp. 6367–6377).

Wang, S., Pei, K., Whitehouse, J., Yang, J., & Jana, S. (2018b). Formal security analysis of neural networks using symbolic intervals. In *USENIXsecurity symposium* (pp. 1599–1614).

Weng, T. W., Zhang, H., Chen, H., Song, Z., Hsieh, C. J., Boning, D., Dhillon, I. S., & Daniel, L. (2018). Towards fast computation of certified robustness for relu networks. In *International conference on machine learning (ICML)*.

Wolsey, L. A. (1998). *Integer programming* (Vol. 52). London: Wiley.

Wu, H., Ozdemir, A., Zeljić, A., Julian, K., Irfan, A., Gopinath, D., Fouladi, S., Katz, G., Pasareanu, C., & Barrett, C. (2020). Parallelization techniques for verifying neural networks. In *Formal methods in computer aided design (FMCAD)*.

Xiang, W., Tran, H. D., Rosenfeld, J. A., & Johnson, T. T. (2018). Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In *Annual American control conference* (pp. 1574–1579). https://doi.org/10.23919/ACC.2018.8431048

Yuan, X., He, P., Zhu, Q., & Li, X. (2019). Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems, 30*(9), 2805–2824.

Zhang, H., Weng, T. W., Chen, P. Y., Hsieh, C. J., & Daniel, L. (2018). Efficient neural network robustness certification with general activation functions. In *Advances in neural information processing systems (NeurIPS)*.

Zhu, C., Byrd, R. H., Lu, P., & Nocedal, J. (1997). Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS), 23*(4), 550–560.

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Authors and Affiliations

**Christopher A. Strong[1] · Haoze Wu[2] · Aleksandar Zeljić[2] · Kyle D. Julian[3] · Guy Katz[4] · Clark Barrett[2] · Mykel J. Kochenderfer[3]**

Haoze Wu
haozewu@stanford.edu

Aleksandar Zeljić
zeljic@stanford.edu

Kyle D. Julian
kjulian3@stanford.edu

Guy Katz
guykatz@cs.huji.ac.il

Clark Barrett
barrett@stanford.edu

Mykel J. Kochenderfer
mykel@stanford.edu

1 Department of Electrical Engineering, Stanford University, Stanford, CA, USA

2 Department of Computer Science, Stanford University, Stanford, CA, USA

3 Department of Aeronautics and Astronautics, Stanford University, Stanford, CA, USA

4 School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel