# Techniques for Verifying Robustness of Neural Networks

Kishor Jothimurugan
University of Pennsylvania

**Abstract**

Deep neural networks have emerged as a powerful tool in solving many real-world problems. However, their applicability to safety-critical applications such as autonomous driving and malware detection is challenged by the complexity in verifying safety properties of such neural networks. One such property of interest is local adversarial robustness, the ability of a neural network to classify certain inputs correctly in the presence of adversarial noise. We survey three recently proposed techniques for certifying robustness of neural networks that use the Rectified Linear Unit (ReLU) activation function: an SMT based approach [16], an optimization based approach that uses Semi-Definite Programming (SDP) relaxation [21] and an approach that applies abstract interpretation to neural networks [9]. In this survey, we analyze these papers giving a summary of the key ideas, their strengths and limitations.

## 1 Introduction

Deep Neural Networks (DNNs) have become popular for their applicability in a variety of domains to solve many real-world computational problems such as image classification, game playing, natural language processing among many others. However, in order to use DNNs in safety-critial applications, one needs to be able to verify safety properties about these neural networks. This task of verification is difficult for two reasons: lack of human interpretability and the size and complexity of DNNs used in practice (DNNs typically use non-linear activation functions which make analyzing them computationally hard [16, 14]).

In this survey, we look at a specific propery of DNNs that are used in classification tasks, known as local adversarial robustness (henceforth referred to as robustness). Many state-of-the-art DNN classifiers have been shown [24] to change predictions in the presence of small adversarial perturbations. In the case of images, such perturbations are imperceptible to the human eyes, yet, cause DNNs to predict incorrectly. Following this, many *attacks* have been proposed to generate adversarial examples for existing DNNs. A recent survey of these attacks can be found in [26]. Many *defenses* against these attacks have been proposed which modify the training procedures to be more robust with respect to specific attacks. Unfortunately, such defenses are broken using new attacks [1].

In order to guarantee safety from all attacks, one needs to verify that the DNN is robust to *any* small perturbation to the input. Recently, there has been a lot of research work on this topic. These works can be broadly categorized based on the attack model (the kind of perturbations that are allowed), the kind of neural networks they deal with and the technical tools they use. One line of work [16, 9, 7, 19, 18] aims to apply formal verification techniques such as SMT solvers and abstract interpretation whereas another line of work [21, 20, 17, 25] is based on (convex and non-convex) optimization.

We focus on three recent works [16, 21, 9] each of which uses a different approach to verify robustness of DNNs. Some of these approaches are more general and can be applied to verify a richer class of properties. These approaches assume that the DNNs use the Rectified Linear Unit (ReLU) activation function which is piecewise linear and is commonly used in modern DNNs. These verification techniques can be applied

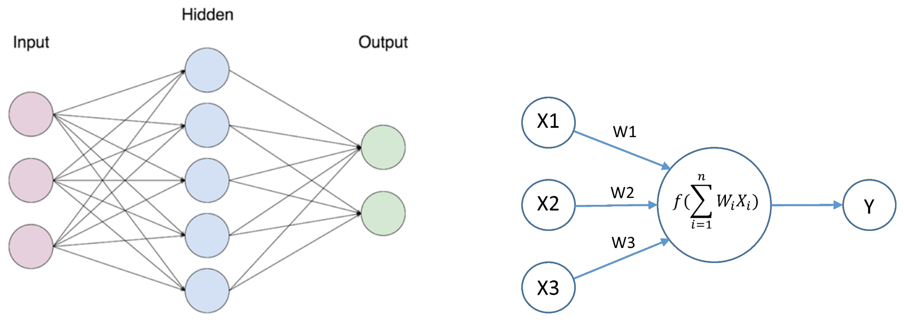

Figure 1: A neural network with two layers: one hidden and one output layer (left) followed by a closer look of a single neuron (right).

to certify robustness in the $\ell_\infty$ attack model in which the adversarial perturbations are restricted to have small $\ell_\infty$ norm. In Section 2 we formally define the verification problem. In Sections 3, 4 and 5 we discuss the three different approaches to solve this problem and in Section 6 we discuss some directions for future research.

## 2 The Verification Problem

Let us consider multi-layered neural networks that classify inputs into a fixed number of classes. Let the input space be $\mathbb{R}^n$ and the set of classes be $\{1, ..., m\}$. A neural network takes as input a vector in $\mathbb{R}^n$ and outputs a vector in $\mathbb{R}^m$ and is organized as a sequence of layers where output of one layer is the input to the next layer. Formally, a neural network $N$ is a sequence of layers $L^1, \ldots, L^k$ where each layer $L^i$ defines a mapping from $\mathbb{R}^{n_{i-1}}$ to $\mathbb{R}^{n_i}$ with $n_0 = n$ and $n_k = m$. Typically, a (fully connected) layer is a linear function composed with a non-linear activation function, i.e., $L^i(x) = f(W^i x + b^i)$ where $W^i$ is an $n_{i-1} \times n_i$ matrix, $b^i \in \mathbb{R}^{n_i}$ and $f$ is the activation function. One common activation function is the ReLU function given by $\text{ReLU}(x) = \max(0, x)$ where "max" denotes the pointwise maximum. A layer $L^i$ is also thought of to be composed of $n_i$ *neurons*, each computing one component of the layer's output (Figure 1). The *weights* $\{W^i \mid 1 \leq i \leq k\}$ and *biases* $\{b^i \mid 1 \leq i \leq k\}$ are obtained during training typically by minimizing some loss function based on the training data. The function computed by the neural network $N$ is given by $N(x) = L^k(L^{k-1}(\cdots(L^1(x)))$. The class assigned to $x$ by $N$ is defined by $C_N(x) = \text{argmax}_y N(x)_y$ where $N(x)_y$ denotes the $y^{th}$ component of $N(x)$ and is the score assigned to label $y$ given input $x$.

A neural network is said to be robust at an input $\overline{x}$ if it does not change its predicted class on adding some small adversarial noise. Let us denote by $B_\epsilon(\overline{x})$ the set $\{x \mid \|x - \overline{x}\|_\infty \leq \epsilon\}$.

**Definition.** Given a neural network $N$, an input vector $\overline{x}$ and an $\epsilon > 0$, $N$ is defined to be $\epsilon$-*robust* at $\overline{x}$ if and only if for all $x \in B_\epsilon(\overline{x})$, $C_N(x) = C_N(\overline{x})$.

Note that $N$ is $\epsilon$-robust at $\overline{x}$ if an only if for all $x \in B_\epsilon(\overline{x})$ and all $y \neq C_N(\overline{x})$, $N(x)_y < N(x)_{C_N(\overline{x})}$. Given a network $N$, an input $\overline{x}$ and an $\epsilon > 0$, the robustness verification problem is to decide whether $N$ is $\epsilon$-robust at $\overline{x}$.

In the following sections, we will look at three recent approaches to tackle the robustness verification problem (some of them can be applied to verify other properties of neural networks as well). Let us assume, for now, that all the layers of the given neural network are fully connected (convolutional layers can be treated as fully connected layers for the purpose of verification) and use the ReLU activation function (linear layers with the identity activation function can also be used, but we assume all layers use the ReLU function for simpler presentation). In Section 3 we look at Reluplex [16], an SMT solver based on the simplex algorithm for satisfiability of linear constraints that also handles the (non-linear) ReLU constraints of the form $v_1 = \text{ReLU}(v_2)$. In Section 4 we present a technique intorduced in [21], which uses a Semi-Definite Programming (SDP) relaxation for the problem of maximizing $N(x)_y - N(x)_{C_N(\overline{x})}$ for $x \in B_\epsilon(\overline{x})$ where $y \neq C_N(\overline{x})$. In Section 5 we look at a way to apply abstract interpretation to neural networks [9] to approximate the range of outputs of $N$ given a set of inputs (here, $B_\epsilon(\overline{x})$).

## 3 ReLUplex: An SMT Solver

Satisfiability Modulo Theory (SMT) solvers are a powerful tool in formal verification. ReLU based neural networks can be directly encoded as logical formulas in linear real arithmetic due to the piecewise linear nature of the ReLU function and hence one can use existing SMT solvers to verify properties of such neural networks. But this does not scale well to large networks as each neuron introduces a disjunction, i.e., the constraint $v_1 = \text{ReLU}(v_2)$ would be represented as $(v_2 \leq 0 \wedge v_1 = 0) \vee (v_2 \geq 0 \wedge v_1 = v_2)$; this causes most SMT solvers to perform many case-splittings. To overcome this issue, Reluplex [16] extends the theory of linear real arithmetic with the ReLU function and proposes a solver for this theory.

### 3.1 The Algorithm

Let $V = \{v_1, ..., v_\ell\}$ be a set of variables. An *atomic predicate* over $V$ is either a linear constraint of the form $\sum_i a_i v_i \sim b$ where $\sim \in \{=, \leq, \geq\}$ and each $a_i$ and $b$ are rational constants or a ReLU constraint of the form $v_i = \text{ReLU}(v_j)$. Reluplex takes as input a conjunction of atomic predicates $\varphi = p_1 \wedge \cdots \wedge p_t$ and decides if it is satisfiable, i.e., whether there is a valuation to the variables $\alpha : V \to \mathbb{R}$ such that $\alpha \models p_i$ for all $i$. Given a neural network $N$, an input vector $\overline{x}$ and an $\epsilon > 0$, a formula $\varphi_{N,\overline{x},\epsilon}$ is constructed such that $\varphi_{N,\overline{x},\epsilon}$ is satisfiable if and only if $N$ is *not* $\epsilon$-robust at $\overline{x}$. We first encode the neural network as a conjunction of atomic predicates $\varphi_N$, with one linear and one ReLU constraint for each neuron, using variables $\{x_1, ..., x_n\}$ to represent the input, $\{z_1, ..., z_m\}$ for the output and some intermediate variables. Then, $\varphi_{N,\overline{x},\epsilon}$ is given by a disjunction over $y \neq C_N(\overline{x})$ of $\varphi_y = \varphi_N \wedge \bigwedge_i (\overline{x}_i - \epsilon \leq x_i \leq \overline{x}_i + \epsilon) \wedge (z_y \geq z_{C_N(\overline{x})})$. Since each $\varphi_y$ is a conjunction of atomic predicates, the satisfiability of $\varphi_{N,\overline{x},\epsilon}$ can be solved using Reluplex.

**The Data Structure.** Reluplex is described as an abstract calculus over a data structure called *configurations*. A configuration over a set of variables $V$ is six tuple $(B, T, l, u, R, \alpha)$ where $B \subseteq V$ is a set of *basic* variables, $T$ is a *tableau* consisting of $|B|$ linear equations, one for each basic variable $v_i \in B$, of the form $v_i = \sum_{v_j \notin B} a_j v_j$. $l : V \to \mathbb{R}$ and $u : V \to \mathbb{R}$ represent the lower and upper bounds for the variables respectively, $R$ is a set of ReLU constraints of the form $v_i = \text{ReLU}(v_j)$ and $\alpha : V \to \mathbb{R}$ is a valuation of the variables such that $\alpha$ satisfies all linear equations in the tableau but not necessarily satisfy the upper and lower bounds or the ReLU constraints. Given a conjunction of atomic predicates $\varphi$, the initial configuration is constructed as follows: For each linear constraint $p$ appearing in $\varphi$ of the form $\sum_i a_i v_i \sim b$, a new variable $v_p$ is added and the set of basic variables $B$ is taken to be the set of newly added variables giving one basic variable for each linear constraint in $\varphi$. The linear constraint $p$ is represented in the tableau as an equation $v_p = \sum_i a_i v_i$ and either $l(v_p)$, $u(v_p)$ or both are set to $b$ depending on whether $\sim$ is $\geq$, $\leq$ or $=$ respectively. $R$ is taken to be the set of ReLU constraints appearing in $\varphi$. The initial valuation $\alpha$ maps every variable to zero.

**The Derivation Rules.** Reluplex maintains a set of current configurations initialized to the singleton set consisting of the initial configuration and updates them by selecting a configuration and applying one of the rules given in Figures 2, 3 and 4. A rule can be applied to a configuration $\mathcal{C}$ if the rule's *guard* conditions hold for $\mathcal{C}$ and some elements of $\mathcal{C}$ are updated on applying the rule. All rules except the `ReluSplit` rule replace a current configuration by a new one whereas `ReluSplit` generates two configurations from the

$$\frac{\forall v_i \in V.\ l(v_i) \le \alpha(v_i) \le u(v_i), \qquad \forall (v_f = \mathrm{ReLU}(v_b)) \in R.\ \alpha(v_f) = \mathrm{ReLU}(\alpha(v_b))}{\mathtt{SAT}} \quad \mathtt{ReluSuccess}$$

$$\frac{v_i \in B, \qquad (\alpha(v_i) < l(v_i) \wedge \mathtt{slack}^+(v_i) = \emptyset) \vee (\alpha(v_i) > u(v_i) \wedge \mathtt{slack}^-(v_i) = \emptyset)}{\mathtt{UNSAT}} \quad \mathtt{Failure}$$

Figure 2: Success and Failure Rules

$$\frac{v_i \in B, \qquad \alpha(v_i) < l(v_i), \qquad v_e \in \mathtt{slack}^+(v_i)}{T := \mathtt{pivot}(T, v_i, v_e), \qquad B := B \cup \{v_e\} \setminus \{v_i\}} \quad \mathtt{Pivot}_1$$

$$\frac{v_i \in B, \qquad \alpha(v_i) > u(v_i), \qquad v_e \in \mathtt{slack}^-(v_i)}{T := \mathtt{pivot}(T, v_i, v_e), \qquad B := B \cup \{v_e\} \setminus \{v_i\}} \quad \mathtt{Pivot}_2$$

$$\frac{v_i \in B, \qquad \exists v_j.\ (v_j = \mathrm{ReLU}(v_i)) \in R \vee (v_i = \mathrm{ReLU}(v_j)) \in R, \qquad v_e \notin B, T_{i,e} \neq 0}{T := \mathtt{pivot}(T, v_i, v_e), \qquad B := B \cup \{v_e\} \setminus \{v_i\}} \quad \mathtt{ReluPivot}$$

Figure 3: Pivot Rules

current configuration. A `SAT` is returned if the rule `ReluSuccess` can be applied to *atleast one* of the current configurations. On applying the rule `Failure` to one of the configurations, that configuration is removed from the current set. An `UNSAT` is returned if the current set of configurations is empty. The rule `ReluSuccess` can be applied when the valuation $\alpha$ satisfies all lower and upper bounds given by $l$ and $u$ as well as all the ReLU constraints in $R$.

Before describing the other rules, let us look at the `pivot` operation which is used to switch a basic variable (the *leaving* variable) with a non-basic variable (the *entering* variable). This is needed to update a basic variable, because, in order to maintain an assignment $\alpha$ that satisfies all equations in the tableau, updates can only be applied to non-basic variables (while updating the basic variables based on the equations). Given a tableau $T$, let us denote by $T_{i,j}$ the coefficient of $v_j$ in the equation for $v_i$. Let the equation for a basic variable $v_i$ be $v_i = \sum_{v_j \notin B} a_j v_j$. Let $v_e \in V \setminus B$ be such that the coefficient of $v_e$ in this equation is non-zero, i.e., $T_{i,e} = a_e \neq 0$. Then $\mathtt{pivot}(T, v_i, v_e)$ returns a tableau in which the equation of $v_i$ is replaced by an equation for $v_e$, $v_e = \frac{1}{a_e} v_i - \sum_{v_j \notin B, v_j \neq v_e} \frac{a_j}{a_e} v_j$ and $v_e$ is replaced by the right hand side of the above equation in all ther equations of $T$. The new tableau represents the same set of constraints but $v_e$ is now a basic variable and $v_i$ is not.

The rules $\mathtt{Pivot}_1$ and $\mathtt{Pivot}_2$ can be applied only if entering variable provides a *slack* for the leaving variable. Suppose in the current configuration, $v_i$ is a basic variable with $\alpha(v_i) < l(v_i)$. A `pivot` on $v_i$ is required to increase the value of $\alpha(v_i)$. Let us suppose we choose the entering variable to be $v_e$. Increasing the value of $v_i$ would force the value of $v_e$ to increase or decrease depending on whether $T_{i,e} > 0$ or $T_{i,e} < 0$ and to make sure this does not introduce new violations of the lower and upper bounds on $v_e$, we need $\alpha(v_e) < u(v_e)$ or $\alpha(v_e) > l(v_e)$ depending on the sign of $T_{i,e}$. This intuition is captured in the conditions on the `Pivot` rules with the following definitions of the `slack` sets: For a basic variable $v_i$,

$$\mathtt{slack}^+(v_i) = \{v_j \notin B \mid (T_{i,j} > 0 \wedge \alpha(v_j) < u(v_j)) \vee (T_{i,j} < 0 \wedge \alpha(v_j) > l(v_j))\}$$

$$\mathtt{slack}^-(v_i) = \{v_j \notin B \mid (T_{i,j} < 0 \wedge \alpha(v_j) < u(v_j)) \vee (T_{i,j} > 0 \wedge \alpha(v_j) > l(v_j))\}$$

The rule `ReluPivot` also allows to apply the pivot operation on a basic variable $v_i$ if it appears in a ReLU constraint in $R$. The $\mathtt{update}(\alpha, v_j, \delta)$ operation updates the current variable valuation $\alpha$ by adding $\delta$ to $\alpha(v_j)$ and updating the values of all the basic variables based on their equations in the tableau. The rule `Update` allows updating a non-basic variable whose current value violates either the upper or lower bound

$$\frac{v_j \notin B, \qquad \alpha(v_j) < l(v_j) \vee \alpha(v_j) > u(v_j), \qquad l(v_j) \leq \alpha(v_j) + \delta \leq u(v_j)}{\alpha := \texttt{update}(\alpha, v_j, \delta)} \; \texttt{Update}$$

$$\frac{v_j \notin B, \qquad (v_j = \mathrm{ReLU}(v_i)) \in R, \qquad \alpha(v_j) \neq \mathrm{ReLU}(\alpha(v_i))}{\alpha := \texttt{update}(\alpha, v_j, \mathrm{ReLU}(\alpha(v_i)) - \alpha(v_j))} \; \texttt{Update}_f$$

$$\frac{v_i \notin B, \qquad (v_j = \mathrm{ReLU}(v_i)) \in R, \qquad \alpha(v_j) \neq \mathrm{ReLU}(\alpha(v_i)), \qquad \alpha(v_j) \geq 0}{\alpha := \texttt{update}(\alpha, v_i, \alpha(v_j) - \alpha(v_i))} \; \texttt{Update}_b$$

$$\frac{(v_j = \mathrm{ReLU}(v_i)) \in R, \qquad l(v_i) < 0, \qquad u(v_i) > 0}{u(v_i) := 0 \qquad l(v_i) := 0} \; \texttt{ReluSplit}$$

Figure 4: Update and Split Rules

for the variable by adding a $\delta$ that would cause its value to be within bounds. If a non-basic variable $v_i$ appears in a ReLU constraint in $R$ that is currently violated, then either $\texttt{Update}_f$ or $\texttt{Update}_b$ can be applied depending on whether $v_i$ is a *forward* variable (appearing in LHS of the ReLU constraint) or a *backward* variable (appearing in the RHS of the ReLU constraint).

The rule `ReluSplit` allows for case-splitting on a ReLU constraint generating two configurations, one in which the upper bound of the backward variable is set to 0 and one in which the lower bound of the backward variable is set to 0 corresponding to the cases when the ReLU function acts as a constant function (mapping to 0) and the identity function respectively. Finally, a configuration is unsuccessful if the current value of a basic variable is out of bounds and no non-basic variable provides a slack, which is when the rule `Failure` can be applied.

**Implementation Strategies.** In order to achieve good performance, Reluplex applies some tricks commonly used in most SMT solvers. It adds two new rules called `DeriveLowerBound` and `DeriveUpperBound` which derive tighter lower and upper bounds based on the current bounds and the tableau equations. For example, if $v_1$ and $v_2$ are both bound to be within $[0, 1]$, then the equation $v_3 = v_1 - v_2$ implies that $v_3$ is bound to the range $[-1, 1]$. If such derived bounds for a variable is such that the upper bound is smaller than the lower bound, one can immediately conclude `UNSAT` for that specific configuration. Reluplex also uses floating point numbers instead of full precision numbers to improve performance at the cost of soundness (though it uses mechanisms to limit the errors). Reluplex also tries to satisfy the ReLU constraints by using `ReluPivot`, $\texttt{Update}_f$ and $\texttt{Update}_b$ rules before performing case-splitting using `ReluSplit`.

## 3.2 Discussion

Reluplex is based on the simplex algorithm used to solve linear programming problems. Simplex can also be viewed as an abstract calculus for deciding satisfiability of a conjunction of linear inequalities. Reluplex is obtained from simplex by adding the ReLU component $R$ as a part of the configuration and adding rules $\texttt{Update}_f$, $\texttt{Update}_b$, `ReluPivot` and `ReluSplit` and updating the success rule to also check for satisfaction of the ReLU constraints. This is a clean and elegant extension to handle ReLU functions making it more suitable for the task of verifying properties of neural networks. Reluplex can be applied to verify a rich family of properties about neural networks with robustness being one of them. It is important to note that the problem of verifying properties from this class is NP-complete in general (see Appendix of [16] for a proof).

**Soundness and Completeness.** It is easy to see that the Reluplex calculus is sound (if a `SAT` or `UNSAT` is returned, then the input formula is satisfiable or unsatisfiable respectively). The authors also claim that the calculus is complete (there is a strategy for applying the derivation rules that guarantees termination).

But with only the rules mentioned in the paper, it can be shown to be incomplete.

Consider the following formula: $\varphi = v_1 \geq 2 \wedge v_2 \leq 1 \wedge v_1 = \text{ReLU}(v_2)$. $\varphi$ is not satisfiable but the subformula of $\varphi$ without the ReLU constraint is satisfiable. Let $\alpha^*$ be a valuation that satisfies this subformula ($v_1 \geq 2 \wedge v_2 \leq 1$). Due to soundness, `SAT` can never be derived. Lemma 1 and 2 in the Appendix of the paper [16] guarantee that there is always a configuration in the set of configurations maintained by the algorithm such that $\alpha^*$ satisfies all constraints in the configuration except the ReLU constraints. The rule `Failure` can never be applied to such a configuration and hence the algorithm will never return `UNSAT` on this instance. Their "proof" of completeness uses the following strategy for applying the rules: First split on all ReLU constraints and then apply a strategy that guarantees termination for the simplex algorithm (which exists because of the completeness of the simplex algorithm). But this ignores ReLU constraints after splitting. One way to make this calculus complete is to add the ReLU constraint to the tableau as a linear constraint after splitting (depending on the case of splitting, ReLU is either a constant function or the identity function both of which are linear). To make sure that the current valuation satisfies all tableau equations, it can be reset to the initial valuation that assigns 0 to all variables.

**Empirical Performance.** Reluplex is shown to outperform existing state of the art SMT solvers for verifying/falsifying safety properties of some neural networks used in the experimental ACAS Xu aircraft collision avoidance system [15]. Reluplex can verify/falsify robustness properties of these neural networks for various values of $\epsilon$ and $\overline{x}$. These neural networks have around 300 ReLU neurons and 8 layers. However, even for such networks, Reluplex takes thousands of seconds to terminate making it hard to apply it to larger neural networks with thousands of neurons.

### 3.3 Related Work

In [2] Reluplex has been used to generate adversarial examples that are provably minimally-distorted. The authors also show that Reluplex can handle networks with max-pooling layers by encoding the max function using linear and ReLU functions. [7] gives another SMT based solution to verify robustness of ReLU based neural networks. The authors of [19] propose an SMT based approach for verifying properties of neural networks with sigmoid activation functions which works by approximating the sigmoid function with piecewise linear functions. Apart from SMT solver based verifiers, a number of complete verifiers [25, 3, 5, 8] have been proposed which are based on mixed-interger linear programming (MILP).

## 4 An SDP Relaxation

In this section, we look at an approach that uses a method from convex optimization called semidefinite relaxation which can be used to relax the constraints of certain non-convex optimization problems to over-approximate the feasible set of points. The resulting problem is a Semi-Definite Programming problem which can be solved efficiently using existing methods. The robustness verification problem can also be thought of as a constrained optimization problem in which one wants to maximize the value of $N(x)_y - N(x)_{\overline{y}}$ subject to the constraint that $x \in B_\epsilon(\overline{x})$, where $\overline{y} = C_N(\overline{x})$ is the class assigned to $\overline{x}$ and $y$ is a label different from $\overline{y}$. The neural network $N$ is $\epsilon$-robust at $\overline{x}$ if and only if the optimal value of this optimization problem is negative for all choices of $y$. Using SDP relaxation, one can get upper bounds on these optimal values, which if all negative, would certify the robustness of the neural network.

## 4.1 The Algorithm

Let a neural network $N$, an input $\overline{x}$ and $\epsilon > 0$ be given. First, let us take a closer look at the above optimization problem (without relaxation). Let $L^1, ..., L^k$ be the layers of $N$. We introduce $n$ variables $v_1^0, ..., v_n^0$ representing the input of $N$. For each neuron, we introduce a variable representing the output of the neuron: let us denote by $v_j^i$, the variable corresponding to the $j^{\text{th}}$ component of the output of $L^i$. We use $v^i$ to denote the vector $[v_1^i, ..., v_{n_i}^i]^T$. Given a label $y \neq \overline{y}$, the objective function is given by $v_y^k - v_{\overline{y}}^k$ and $v^0$ is constrained to be in $B_\epsilon(\overline{x})$ and $v^1, ..., v^k$ are constrained to be the output of their corresponding layers. Formally we define,

$$\texttt{OPT}_y(\overline{x}, \overline{y}) = \max_{v^0,...,v^k} v_y^k - v_{\overline{y}}^k$$

subject to,

$$v^i = \text{ReLU}(W^i v^{i-1} + b^i) \text{ for } i = 1, ..., k$$
$$\overline{x}_j - \epsilon \leq v_j^0 \leq \overline{x}_j + \epsilon \text{ for } j = 1, ..., n$$

**Quadratic Constraints for ReLU.** Note that the only source of non-linearity in the above formulation is the ReLU function. Let us look at a constraint of the form $v_2 = \text{ReLU}(v_1)$, where $v_1$ and $v_2$ are scalar variables. The key observation made by the authors of [21] is that this constraint is equivalent to the following three constraints: (1) $v_2 \geq v_1$, (2) $v_2 \geq 0$ and (3) $v_2(v_2 - v_1) = 0$. The first two constraints ensure that the value $v_2$ is atleast the value of $\text{ReLU}(v_1)$ and the last constraint forces $v_2$ to be either 0 or $v_1$. This allows us to write the optimization problem as a quadratically constrained quadratic program (QCQP) as follows:

$$\texttt{OPT}_y(\overline{x}, \overline{y}) = \max_{v^0,...,v^k} v_y^k - v_{\overline{y}}^k$$

$$\text{s.t. for } i = 1, ..., k$$
$$v^i \geq 0$$
$$v^i \geq W^i v^{i-1} + b^i$$
$$v^i \odot v^i = v^i \odot (W^i v^{i-1} + b^i)$$
$$\overline{x} - \overline{\epsilon} \leq v^0 \leq \overline{x} + \overline{\epsilon}$$

where $\odot$ is the component-wise multiplication of vectors, $\overline{\epsilon} \in \mathbb{R}^n$ represents the vector with all components equal to $\epsilon$ and for any two vectors $z^1$ and $z^2$, $z^1 \geq z^2$ if and only if $z_j^1 \geq z_j^2$ for all $j$. The last constraint is also replaced with an equivalent quadratic constraint to get tighter relaxations. Let us use $l^0$ and $u^0$ to denote $\overline{x} - \overline{\epsilon}$ and $\overline{x} + \overline{\epsilon}$ respectively. Note that for any $z \in \mathbb{R}$ and $l \leq u$, $l \leq z \leq u$ if and only if $(z - l)(z - u) \leq 0$. Therefore, the last constraint can be replaced by the following constraint:

$$v^0 \odot v^0 \leq (l^0 + u^0) \odot v^0 - l^0 \odot u^0 \tag{1}$$

**SDP Relaxation.** The above QCQP is non-convex and in order to get an upper bound on $\texttt{OPT}_y(\overline{x}, \overline{y})$ efficiently, the constraints are relaxed to get a convex SDP. Let $v = [1, v^0, ..., v^k]^T$ be the vector of dimension $n' = 1 + \sum_{i=1}^k n_i$ formed be concatenating all variables. Then, we can see that all constraints in the QCQP are linear in the entries of the matrix $vv^T$. Introducing a new set of variables corresponding to the entries of a $n' \times n'$ matrix $P$, the optimization problem reduces to a linear programming problem with an addition constraint that $P = zz^T$ for some vector $z$ with $z_1 = 1$. We use symbolic indexing to index submatrices of $P$. $P[1]$ refers to $P_{1,1}$. For $i, j \in \{1, ..., k\}$, $P[v^i]$ denotes the submatrix corresponding to $v^i$ in the matrix $vv^T$, and $P[v^i(v^j)^T]$ denotes the submatrix corresponding to $v^i(v^j)^T$ in the matrix $vv^T$. For $k = 1$, we have,

$$P = \begin{bmatrix} P[1] & P[(v^0)^T] & P[(v^1)^T] \\ P[v^0] & P[v^0(v^0)^T] & P[v^0(v^1)^T] \\ P[v^1] & P[v^1(v^0)^T] & P[v^1(v^1)^T] \end{bmatrix}$$

Using this notation, the QCQP is relaxed to the following convex semi-definite program:

$$\begin{aligned} \mathtt{OPT}^{\mathrm{SDP}}_y(\overline{x}, \overline{y}) = \max_P \ & P[v^k]_y - P[v^k]_{\overline{y}} \\ \text{s.t. for } i = 1, ..., k & \\ & P[v^i] \geq 0 \\ & P[v^i] \geq W^i P[v^{i-1}] + b^i \\ & \mathrm{diag}(P[v^i(v^i)^T]) = \mathrm{diag}(W^i P[v^{i-1}(v^i)^T]) + P[v^i] \odot b^i \\ & \mathrm{diag}(P[v^0(v^0)^T]) \leq (l^0 + u^0) \odot P[v^0] - l^0 \odot u^0 \\ & P[1] = 1, P \succeq 0 \end{aligned}$$

where $\mathrm{diag}(M)$ denotes the vector formed by the diagonal elements of the matrix $M$. We have relaxed the constraint $\{P = zz^T$ for some $z\}$ to $\{P \succeq 0\}$ which states that $P$ is positive semi-definite. It is known that a $n' \times n'$ matrix $M$ is positive semi-definite if and only if there is a $n' \times n'$ matrix $Z$ such that $M = ZZ^T$. Therefore, the original constraint $\{P = zz^T$ for some vector $z\}$ is replaced by $\{P = ZZ^T$ for some $n' \times n'$ matrix $Z$ with arbitrary rank$\}$. Hence, we can conclude that $\mathtt{OPT}^{\mathrm{SDP}}_y(\overline{x}, \overline{y}) \geq \mathtt{OPT}_y(\overline{x}, \overline{y})$ and if $\mathtt{OPT}^{\mathrm{SDP}}_y(\overline{x}, \overline{y}) < 0$ for all $y \neq \overline{y}$, we can infer that $N$ is $\epsilon$-robust at $\overline{x}$.

**Bounds on intermediate layer outputs.** To make the relaxation tighter, bounds on the variables $v^1, ..., v^k$ are obtained using interval arithmetic. These bounds are obtained inductively. We are already given lower and upper bounds ($l^0$ and $u^0$) on the input variable vector $v^0$. $l^i$ and $u^i$ are obtained from $l^{i-1}$ and $u^{i-1}$ as follows:

$$\begin{aligned} l^i &= W^i_+ l^{i-1} + W^i_- u^{i-1} \\ u^i &= W^i_+ u^{i-1} + W^i_- l^{i-1} \end{aligned}$$

where $(M_+)_{i,j} = \max(M_{i,j}, 0)$ and $(M_-)_{i,j} = \min(M_{i,j}, 0)$. These lower and upper bounds are then added as constraints to the SDP, after converting them to their equivalent quadratic constraints (as in (1)).

**Comparison to LP relaxation.** [17] provides a Linear Programming (LP) relaxation to the above optimization problem. The key idea there is to replace the constraint $v_1 = \mathrm{ReLU}(v_2)$ with three different linear constraints: (i) $v_1 \geq 0$, (ii) $v_1 \geq v_2$ and (iii) $v_1 \leq \mathrm{ReLU}(l) + \frac{v_2 - l}{u - l}(\mathrm{ReLU}(u) - \mathrm{ReLU}(l))$ where $u$ and $l$ are upper and lower bounds on $v_2$ respectively. The soundness of this relaxation follows from the convexity of the ReLU function. It has been shown that in the case of one layer networks, the SDP relaxation has asymptotically (in $n$ and $m$) smaller optimal value (better approximation) than the LP relaxation with high probability, where each element of the weight matrix $W^1$ is chosen uniformly at random from the set $\{-1, 1\}$. This gap is due to the fact that the LP relaxation reasons about different components of the output of a layer independent of each other whereas the semi-definite program given above jointly reasons about all the components. Therefore, there is some evidence to support the use of SDP relaxation instead of the LP relaxation even though linear programs can be solved more efficiently in practice.

## 4.2 Discussion

This method provides an SDP relaxation to upper bound the confidence values on incorrect classes for perturbed inputs close to a given point. Efficient solvers for semi-definite programs can be used to compute the upper bounds in reasonable time.

**Soundness and Completeness.** This approach is sound in the sense that whenever the algorithm certifies a neural network to be robust at a given input, it is indeed robust. Since only upper bounds on $\mathtt{OPT}_y(\overline{x}, \overline{y})$ are computed, this approach is not complete, i.e., failure of the algorithm to certify robustness does not imply that the given neural network is not robust at the given input. In such cases, further analysis has to be done to figure out whether there is an adversarial example.

**Empirical Performance.** The authors of [21] implemented this approach and tested it against three neural networks trained using different robust training procedures [20, 17, 12] on the MNIST dataset of handwritten digits. Out of a 1000 random test inputs, this algorithm certified all three networks to be 0.1-robust at atleast 80% of the test inputs. They also compared with other optimization based certification procedures (one based on LP relaxation [17] and one based on gradient bounds [20]) and showed that this approach certified the neural networks to be robust at significantly more inputs than these procedures even though some of the training methods used are geared towards better certification using these procedures. The neural networks considered have a maximum of 4 hidden layers and 500 hidden nodes. The average SDP computation for these networks took around 25 mins. The authors do not study how this approach scales with increase in the size of the networks. Also, it will be interesting to see how the robustness parameter $\epsilon$ affects the certification rates (fraction of test inputs at which robustness is certified).

### 4.3 Related Work

Closely related to this method is an approach based on LP relaxation [17] which also provides an efficient learning procedure to learn robust networks that are verifiable using the LP relaxation. [20] also proposes a learning procedure coupled with a robustness certification algorithm based on gradient bounds. The authors of [6] provide a Lagrangian relaxation based approach to this problem and also prove theoretical approximation guarantees under some (strong) assumptions.

## 5 Abstract Interpretation for AI

Abstract Interpretation is a classical framework for obtaining sound and computable over-approximations of the set of behaviors of a program. It is commonly used in program analysis to verify properties of programs by checking whether all behaviors in the over-approximated set satisfy the given property. This technique can be used to verify robustness of neural networks by over-approximating the set of outputs of the network given a set of inputs (in our case $B_\epsilon(\overline{x})$).

### 5.1 The Algorithm

Neural networks with ReLU activations can be viewed as programs with affine linear transformations and conditionals. We can therefore encode neural networks as programs and directly apply abstract interpretation to these programs. The authors of [9] propose a language called CAT (conditional affine transformations) functions in which ReLU networks can be encoded:

$$\begin{aligned} f(v) &::= Wv + b \\ &\quad \mid \texttt{case}\ E_1 : f_1(v), \cdots, \texttt{case}\ E_q : f_q(v) \\ &\quad \mid f_1(f_2(v)) \\ E &::= E_1 \wedge E_2 \mid v_i \geq v_j \mid v_i \geq 0 \mid v_i < 0 \end{aligned}$$

where $W$ is a matrix and $b$ is a vector. A CAT function $f$ takes as input a vector $x \in \mathbb{R}^n$ and outputs a vector $f(x)$ in $\mathbb{R}^m$. The basic functions are affine linear transformations ($Wv + b$). They can be composed using conditionals and functional compositions. The conditional (`case`) statement applies the function corresponding to the first condition (from $E_1$ to $E_q$) that is met by the input. The conditions are expressions that are conjunctions of basic linear inequalities. A neural network $N$ can be represented as a corresponding CAT function $f_N$.

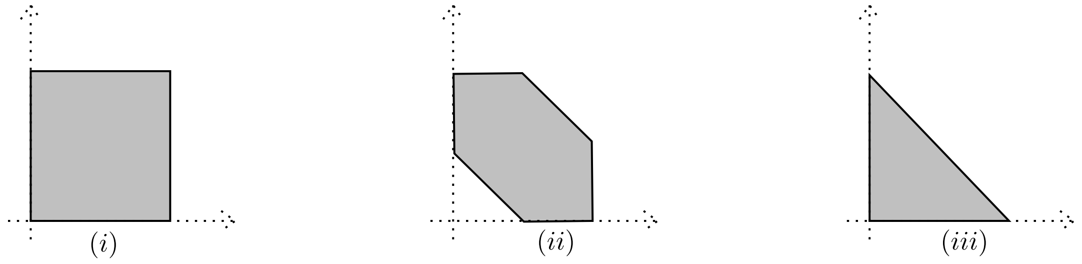

Figure 5: Abstract domains in two dimensions: $(i)$ A box, $(ii)$ A zonotope that is not a box and $(iii)$ A polyhedron that is not a zonotope.

Given a CAT function $f = f_N : \mathbb{R}^n \to \mathbb{R}^m$ for some $N$, $\epsilon > 0$ and an input $\overline{x}$, we want to check if for all $z \in f(B_\epsilon(\overline{x}))$ and all $y \neq N_C(\overline{x}) = \overline{y}$, $z_y < z_{\overline{y}}$. This is a property of $f(B_\epsilon(\overline{x})) \subseteq \mathbb{R}^m$ and if we have a set $S$ such that $f(B_\epsilon(\overline{x})) \subseteq S$ and $S$ satisfies the required property, then we can conclude $N$ is $\epsilon$-robust at $\overline{x}$. Abstract interpretation can be used to find such an $S$.

**Abstract Domains.** To apply abstract interpretation, we need to choose an abstract domain consisting of a set of abstract values. Each abstract value is associated with a set of concrete values. Formally, an abstract domain for $\mathbb{R}^m$ (in general, can be defined for any set of concrete values) is a set $\mathcal{D}$ along with a *concretization* function $\gamma$ mapping an abstract value $d \in \mathcal{D}$ to a set of concrete values $\gamma(d) \subseteq \mathbb{R}^m$ and an *abstraction* function $\alpha$ mapping a set of concrete values $S \subseteq \mathbb{R}^m$ to an abstract value $\alpha(S) \in \mathcal{D}$ such that for all $S \subseteq \mathbb{R}^m$,

$$S \subseteq \gamma(\alpha(S))$$

The above property states that the abstraction of a set $S$ given by $\alpha(S)$ is sound, i.e., represents at-least all the concrete values in $S$ (potentially more). Let us look at a few examples.

**Example** (Interval Domain)**.** For $m = 1$, the interval domain is given by $\mathcal{I} = \{(a, b) \mid a \leq b; a, b \in \mathbb{R} \cup \{-\infty, \infty\}\} \cup \{\emptyset\}$. The concretization function is given by $\gamma((a, b)) = [a, b]$ and the abstraction function is given by $\alpha(S) = (\inf S, \sup S)$ if $S \neq \emptyset$ and $\emptyset$ otherwise. This extends naturally to multiple dimensions by taking products of intervals and is called the box domain.

**Example** (Zonotopes)**.** Zonotopes are affine images of boxes and are more expressive than boxes. A zonotope in $\mathbb{R}^m$ is given by $\{Az + b \mid z \in B, A \text{ is an } m \times p \text{ matrix and } b \in \mathbb{R}^m\}$ for some $p \in \mathbb{N}$ and box $B = [a_1, b_1] \times \cdots \times [a_p, b_p]$. [10] defines the zonotope domain in which an abstract value represents a zonotope along with various operations on this domain.

**Example** (Polyhedra)**.** The polyhedra domain [4] consists of polyhedra, where a polyhedron is an intersection of half spaces and is given by a set of linear constraints of the form $Az \leq b$ for some matrix $A$ and vector $b$. Any zonotope is also a polyhedron and hence this domain is more precise than both the box domain and the zonotope domain (Figure 5).

For the purposes of verifying CAT functions, the abstract domain is assumed to have the *join* $\sqcup$ and *meet* $\sqcap$ operations which are abstract operations for union and intersection. These operations have to be sound, i.e., $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$ and $\gamma(d_1) \cap \gamma(d_2) \subseteq \gamma(d_1 \sqcap d_2)$. Apart from the above standard abstract domains, the authors of [9] also define and use the bounded powerset domain with zonotopes for verifying robustness of neural networks:

**Example.** The zonotope-$K$ domain where $K \in \mathbb{N}$ consists of abstract values each of which represents a union of at-most $K$ zonotopes. The join of two such abstract values may involve union of more than $K$ zonotopes in which case certain heuristics are used to recursively select two zonotopes and then apply the join operation of zonotopes to them to produce a single zonotope, until there are at-most $K$ zonotopes left.

**Approximating CAT Functions.** Let us assume that we have a family of abstract domains $\mathcal{D}^1, \mathcal{D}^2, ...$ one for every dimension (for example, boxes, zonotopes or polyhedra) with the corresponding concretization functions $\gamma^1, \gamma^2, ...$ and abstraction functions $\alpha^1, \alpha^2, ...$ where $\gamma^i : \mathcal{D}^i \to 2^{\mathbb{R}^i}$ and $\alpha^i : 2^{\mathbb{R}^i} \to \mathcal{D}^i$. Given a CAT function $f : \mathbb{R}^n \to \mathbb{R}^m$, we define an abstract function $\widetilde{f} : \mathcal{D}^n \to \mathcal{D}^m$ such that $\gamma^m(\widetilde{f}(d))$ over-approximates the set $f(\gamma^n(d))$. $\widetilde{f}$ is defined inductively as follows:

- The abstract domain is assumed have support for affine linear operations $(Wv + b)$. For a zonotope or polyhedron, applying an affine linear operation gives another zonotope or polyhedron respectively. For boxes, $\widetilde{f}(B) = \alpha^m(f(\gamma^n(B)))$ can be computed as $\alpha^m(S)$ is easy to compute for the box domain if $S$ is a polyhedron.
- Let $f(v) = \texttt{case}\ E_1 : f_1(v), \cdots, \texttt{case}\ E_q : f_q(v)$. Given abstract functions $\widetilde{f_1}, ..., \widetilde{f_q}$ for $f_1, ..., f_q$ the abstract function $\widetilde{f}$ for $f$ is given by,

$$\widetilde{f}(d) = \widetilde{f_1}(d \sqcap \alpha(E_1)) \sqcup \cdots \sqcup \widetilde{f_q}(d \sqcap \alpha(E_q))$$

  where $\alpha(E) = \alpha^n(\{z \in \mathbb{R}^n \mid z \models E\})$. For zonotopes, the join with a set of linear constraints $(E)$ can be performed directly using existing techniques [11].
- Functional composition translates to functional composition of abstract functions.

Given a set of inputs $S$ (in our case $B_\epsilon(\overline{x})$) to $f$, one can now compute an abstract value $d = \widetilde{f}(\alpha^n(S))$ which gives an over-approximation of $f(S)$, i.e., $f(S) \subseteq \gamma^m(d)$.

**Verifying Properties.** Once we obtain an over-approximation $\gamma^m(d)$ of $f(S)$, to verify that a property $P \subseteq \mathbb{R}^m$ is satisfied by all vectors in $f(S)$, it is enough to check if $P$ is satisfied by all vectors in $\gamma^m(d)$. This is done by checking if $\gamma^m(\alpha^m(\mathbb{R} \setminus P) \sqcap d) = \emptyset$. The property $P$ of interest to us is given by $\{z \in \mathbb{R}^m \mid z_y < z_{\overline{y}}\ \forall\ y \neq \overline{y}\}$. To verify this property, for every $y \neq \overline{y}$, we check if $\gamma^m(\alpha^m(\{z \mid z_y \geq z_{\overline{y}}\}) \sqcap d) = \emptyset$. The join with the linear constraint as well as emptiness check is supported by the box, zonotope and the polyhedra domains.

## 5.2 Discussion

This method of certifying robustness of neural networks is a direct application of the framework of abstract interpretation which is widely used in program analysis and verification. It is lightweight and highly scalable.

**Soundness and Completeness.** This approach is sound but not complete. Since only an over-approximation of the set $f(B_\epsilon(\overline{x}))$ is computed, it is possible that the algorithm fails to verify the propery even if the given network is $\epsilon$-robust at $\overline{x}$.

**Empirical Performance.** The authors of [9] implemented this approach and tested on various image classifying networks. The neural networks included one convolutional network each for MNIST and CIFAR-10 datasets with roughly 53000 neurons and 7 fully connected networks for each dataset consisting of a maximum of 1800 neurons and 9 hidden layers. These networks are closer to realistic networks used in practice as compared to the networks used to evaluate the other two approaches. The algorithm has been tested on these networks at 10 input images selected from each dataset for varying values of $\epsilon$. It is able to prove robustness at all input images for small values of $\epsilon$ (0.001,0.005) and around half of the input images for larger values of $\epsilon$ (0.045,0.065) using the zonotope-64 domain. This work has also been compared with Reluplex showing that they both perform similarly on small neural networks whereas on large networks, Reluplex proves only a small fraction of the properties (a property is an input point along with a value of $\epsilon$) as compared to this approach, with a timeout of 1 hour for each property. Hence, this approach is shown to be more scalable.

### 5.3 Related Work

The authors of [18] provide an adversarial learning algorithm to learn neural networks that can be cetified using the above approach. [22] improves upon this approach by using special abstract operators to approximate the ReLU, sigmoid and tanh functions. Finally, [23] uses abstract interpretation with mixed-integer linear programming (MILP) to produce a complete, yet scalable verifier.

## 6 Future Work

The above discussed works provide some preliminary solutions to the problem of verifying local adversarial robustness of neural networks. An immediate challenge is the scalability of these approaches and their ability to verify large neural networks used in the real world. A few possible direction for future research are as follows.

**Computational Complexity.** It can be shown [16] that verifying properties specified as a conjunction of linear inequalities in the input and output variables of a neural network is NP-hard. The robustness verification problem formulated in Section 2 is in co-NP but it is unknown if it is co-NP hard. [6] proves hardness result for neural networks with single hidden layer that use the sigmoid activation function. Also, it might be worth investigating randomized algorithms for the problem which can be used for testing of neural networks for robustness with probabilistic guarantees.

**Approximation Guarantees.** The above discussed sound but incomplete verifiers do not provide any theoretical guarantees on the precision of the approach. It would be interesting to see if one can propose algorithms that provide such approximation guarantees. For example, in the optimization based approach one would like to bound the error introduced due to the relaxation. [6] provides an algorithm with such a guarantee for a very restricted class of neural networks.

**Different Attack Models.** We focused on the $\ell_\infty$ attack model in this survey. Various other attack models are also worth considering. [13, 6] consider the $\ell_2$ attack model in which the perturbations are restricted to have small $\ell_2$ norm. One might also want to add more restrictions on what kind of perturbations are allowed. For instance, it is enough to show that a neural network is robust against small perturbations that can be carried out in real life scenarios. It is hard to deal with such constraints due to lack of mathematical models for them.

**Different Neural Network Architectures.** We only looked at fully connected neural networks that use the ReLU activation function. Other activation functions such as sigmoid and tanh functions are hard to deal with since they are not piecewise linear. One approach is to use abstraction refinement to piecewise linear approximations of such functions [19]. This does not scale well to large networks. [22] provides abstract operators for these functions over the zonotope abstract domain. Most of the existing approaches treat convolutional layers as fully connected layers for the purposes of verification. It might be worth investigating faster approaches to deal with such layers as they are very sparse and structured as compared to fully connected layers.

## References

[1] Anish Athalye, Nicholas Carlini, and David A. Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning (ICML)*, 2018.

[2] Nicholas Carlini, Guy Katz, Clark Barrett, and David L Dill. Provably minimally-distorted adversarial examples. *arXiv preprint arXiv:1709.10207*, 2017.

[3] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis (ATVA)*, 2017.

[4] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, 1978.

[5] Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks. *arXiv preprint arXiv:1709.09130*, 2017.

[6] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A. Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2018.

[7] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *Automated Technology for Verification and Analysis (ATVA)*, 2017.

[8] Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*.

[9] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy (SP)*, 2018.

[10] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *International Conference on Computer Aided Verification*, 2009.

[11] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *International Conference on Computer Aided Verification*, 2010.

[12] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.

[13] Matthias Hein and Maksym Andriushchenko. Formal guarantees on the robustness of a classifier against adversarial manipulation. In *Advances in Neural Information Processing Systems*, 2017.

[14] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 1989.

[15] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. A formally verified hybrid system for the next-generation airborne collision avoidance system. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015.

[16] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, 2017.

[17] J. Zico Kolter and Eric Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning (ICML)*, 2018.

[18] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, 2018.

[19] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, 2010.

[20] A. Raghunathan, J. Steinhardt, and P. Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2018.

[21] Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. Semidefinite relaxations for certifying robustness to adversarial examples. In *Advances in Neural Information Processing Systems*, 2018.

[22] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems*, 2018.

[23] Gagandeep Singh, Timon Gehr, M Puschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[24] 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 (ICLR)*, 2014.

[25] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356*, 2017.

[26] Xiaoyong Yuan, Pan He, Qile Zhu, Rajendra Rana Bhat, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2019.