CLEVEREST: Accelerating CEGAR-based Neural Network Verification via Adversarial Attacks

Zhe Zhao[1], Yedi Zhang[1], Guangke Chen[1], Fu Song[1(✉)], Taolue Chen[2], and Jiaxiang Liu[3]

[1] ShanghaiTech University, Shanghai, China
{zhaozhe1,zhangyd1,chengk,songfu}@shanghaitech.edu.cn
[2] Birkbeck, University of London, London, UK
t.chen@bbk.ac.uk
[3] Shenzhen University, Shenzhen, China
jiaxiang.liu@szu.edu.cn

Abstract. Deep neural networks (DNNs) have achieved remarkable performance in a myriad of complex tasks. However, lacking of robustness and black-box nature hinder their deployment in safety-critical systems. A large number of testing and formal verification techniques have been proposed recently, aiming to provide quality assurance for DNNs. Generally speaking, testing is a fast and simple way to disprove—but not to prove—certain properties of DNNs, while formal verification can provide correctness guarantees but often suffers from scalability and efficiency issues. In this work, we present a novel methodology, CLEVEREST, to accelerate formal verification of DNNs by synergistically combining testing and formal verification techniques based on the counterexample guided abstraction refinement (CEGAR) framework. We instantiate our methodology by leveraging CEGAR-NN, a CEGAR-based neural network verification method, and a representative adversarial attack method for testing. We conduct extensive experiments on the widely-used ACAS Xu DNN benchmark. The experimental results show that the testing can effectively reduce the usage of formal verification in the check-refine loop, hence significantly improves the efficiency.

1 Introduction

As a new programming paradigm, deep learning has achieved incredible performance in a large number of complex tasks such as computer vision [31], autonomous driving [1] and cyber-security [7,8,51]. Nevertheless, deep neural networks (DNNs) have shown to be intrinsically vulnerable to perturbations [54],

This work is supported by the National Natural Science Foundation of China (62072309 and 62272397), an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), Birkbeck BEI School Project (EFFECT) and the Natural Science Foundation of Guangdong Province (2022A1515011458).

G. Singh and C. Urban (Eds.): SAS 2022, LNCS 13790, pp. 449–473, 2022.
https://doi.org/10.1007/978-3-031-22308-2_20

which significantly hinders their applications in safety-critical domains. Insofar approaches on quality assurance of DNNs can be roughly classified into two (complementary) categories: testing (e.g., [4,6,20,32,38,41,45,54,69]) and formal verification (e.g. [15,18,21,23,24,26,28,33,35,39,40,49,50,59,60,67]). The purpose of testing is to disprove the robustness of DNNs by providing adversarial examples (i.e., counterexamples). In contrast, formal verification is often used to provide theoretical guarantees of DNNs, and, once violated, counterexamples may be provided. Computationally, testing is able to scale up to large DNN models, whereas formal verification is currently limited in scalability.

Early efforts on robustness verification reduce the problem to constraint solving (e.g., SMT [15,24,28,29,48], LP and MILP [5,14,34,55,61,68]). Such techniques are often both sound (i.e., no false negative) and complete (i.e., no false positive), but are limited in scalability. To improve the scalability, abstraction techniques have been proposed including abstract interpretation [18,49,50,56,60,63,65] and network abstraction [2,16,19,47,52]. Abstract interpretation approximates the output ranges of neurons for a given input region while network structure abstraction approximates the network via a smaller network which could be verified using existing verification approaches. Unfortunately, abstraction techniques often compromise accuracy. Refinement techniques thus have been adopted which, guided by counterexamples, refine either the estimated output ranges [50,60,63,65] or the abstract network [16,44]. Despite these advances, scalability remains a major challenge in formal verification of DNNs.

In this work, we propose CLEVEREST (CEGAR neural network verification adversarial attacks), a novel methodology to accelerate robustness verification of DNNs by synergistically combining robustness testing and formal verification in the celebrated counterexample guided abstraction refinement (CEGAR) framework [10]. To the best of our knowledge, this is the first attempt to synergistically integrate efficient testing with formal verification for DNN quality assurance. We note that prior work [66] only utilizes testing methods to find adversarial examples before the complete verification, which is to reduce time overhead, but is simply a sequential composition of testing and verification and cannot improve the verification itself.

The methodology of CEGAR follows an abstract-check-refine paradigm. To verify a DNN N against a property, an over-approximation $\widehat{N}$ of N is built and then the check-refine loop is executed. First, we check if the property holds for $\widehat{N}$. If $\widehat{N}$ satisfies the property, we can conclude that N satisfies the property as well (because $\widehat{N}$ is an over-approximation) and stop. Otherwise a counterexample x is found on $\widehat{N}$. We check if x is also a counterexample for N. If it is, we conclude that N does not satisfy the property and stop. Otherwise, the counterexample is spurious and $\widehat{N}$ is refined to exclude the counterexample x. Note that the existing CEGAR-based DNN verification utilizes computational expensive verification techniques to check the abstract systems and to obtain counterexamples in the check-refine loop [16,44].

Our insight of CEGAR in DNN verification is that the abstract systems in early stages of the check-refine loop are often coarse-grained where counterexamples could be easily found by existing robustness testing techniques. Based

on this observation, we propose to verify the robustness of DNNs by applying an *abstract-test-refine* paradigm. The abstract-test-refine paradigm is similar to the standard abstract-check-refine paradigm, except that the abstract systems are to be checked by testing. If the testing fails to find a counterexample, the check-refine is leveraged after which the test-refine loop is applied again.

Our framework can be instantiated by any robustness testing and CEGAR-based verification technique. To evaluate its effectiveness, we implement a verification tool, named CLEVEREST-NN, by leveraging the preprocessing, abstraction, refinement and verification procedures from the CEGAR-NN framework [16] and the PGD adversarial attack [38] for testing. In particular, we show how to encode properties as loss functions so that an adversarial attack could be leveraged. We also propose an attack guided abstraction which allows us to avoid too coarse abstract systems by leveraging an adversarial attack during the iterative abstraction. We thoroughly conduct experiments on the widely used ACAS Xu benchmark [27,28], an airborne collision avoidance system built for unmanned aircraft. The experimental results based on 45 DNNs show that our tool is very promising. For instance, compared with CEGAR-NN, CLEVEREST-NN solved 21 more (62 vs. 41 out of 90) clear-of-conflict related verification instances within the same time limit. Furthermore, on the verification instances solved by both tools, the average execution time (per verification instance) is reduced by 42% (from 3,584s to 2,076s).

To summarize, our main contributions are as follows.

- We propose CLEVEREST, a methodology to accelerate DNN verification by synergistically combining robustness testing and CEGAR-based verification.
- We implement our methodology based on CEGAR-NN and PGD adversarial attack, giving rise to a new DNN verification tool CLEVEREST-NN.
- We conduct extensive experiments on ACAS Xu. The experimental results show that our method significantly improve the performance of CEGAR-NN.

Outline. Section 2 presents the background for DNNs, their verification and adversarial attacks. We propose our methodology in Sect. 3 and instantiate the methodology for DNN verification in Sect. 4. Section 5 reports experimental results. Section 6 discusses related work. We conclude this work in Sect. 7.

2 Background

In this section, we introduce the background of DNNs as well as their verification and adversarial attacks.

Deep Neural Networks. An ℓ-layer ($\ell \geq 2$) deep neural network (DNN) N is a graph structured in layers (cf. Figure 1), where the first layer is called an *input layer*, the last layer is called an *output layer*, and the $\ell - 2$ intermediate layers are called *hidden layers*. All the nodes in these layers are called *neurons* and neurons in hidden layers are called *hidden neurons*. Each neuron in a non-input layer is associated with a *bias* and could be pointed to by other neurons via

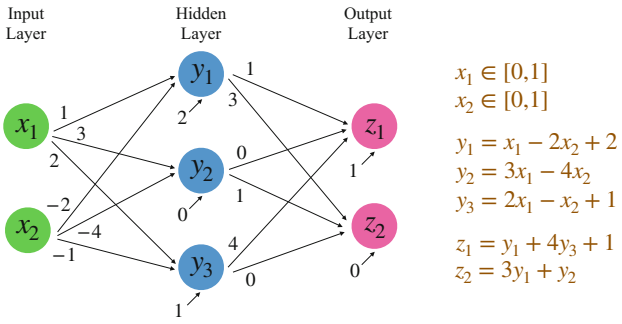

Fig. 1. A fully connected FNN with 2 input nodes (x_1, x_2), 2 output nodes (z_1, z_2) and 1 hidden layers, the activation function is not included. Each edge is associated with a weight value and each node except for inputs is associated with a bias.

weighted, directed edges. The DNN is called a *feed-forward* deep neural network (FNN) if all the weighted, directed edges are from the i-th layer to the $(i+1)$-th layer. An FNN is *fully connected* if each neuron in the i-th layer is connected from all the neurons in the $(i-1)$-th layer. Given an input, the DNN propagates it through the network layer by layer and computes an output. In this work, we consider (fully connected) FNNs, though our methodology is generic.

Formally, an ℓ-layer FNN N is a function $N : X \to Y$, which maps an input vector $\vec{x} \in X$ to an output vector $\vec{y} = N(\vec{x}) \in Y$. Here, $N(\vec{x}) = \vec{W}_\ell \vec{v}_{\ell-1} + \vec{b}_\ell$, and the output vector $\vec{v}_i$ of the i-th layer is recursively defined as follows:

$$\vec{v}_1 = \vec{x}, \quad \vec{v}_i = \sigma(\vec{W}_i \vec{v}_{i-1} + \vec{b}_i) \text{ for } i = 2, \cdots, \ell - 1,$$

where $\vec{W}_i$ and $\vec{b}_i$ (for $2 \leq i \leq \ell$) are the weight matrix and bias vector of the i-th layer respectively, and σ is an activation function (e.g., `ReLU`, `sigmoid`, `tanh`) applied to the input vector entrywise. For classification tasks, the output *class* of a given input $\vec{x}$ is the first index i such that $N(\vec{x})$ at the index i is of the highest value. In this work, we denote by $N_c(\vec{x})$ the output class.

Neural Network Verification. The (neural network) *verification query* for a given FNN N is often formalized as a triple $\langle P, N, Q \rangle$, where the pre-condition P is a property on inputs and the post-condition Q is a property on outputs. The verification query amounts to checking if $N(\vec{x})$ satisfies the post-condition Q for all inputs $\vec{x} \in X$ that fulfil the pre-condition P. A *counterexample* of the verification query $\langle P, N, Q \rangle$ is an input $\vec{x} \in X$ such that $\vec{x}$ satisfies the pre-condition P but $N(\vec{x})$ does not satisfy the post-condition Q. In practice, pre-conditions (resp. post-conditions) are often given as conjunctions of linear constraints on the input values (resp. output values).

Robustness, originated with the study of adversarial attacks [54], is a typical property of DNNs which requires a DNN to produce the same classification result for an input when a small perturbation is added. The perturbation range of an input is usually represented as a ball centered at the input under the L-norm distance. There are three widely-used L-norms: L_0, L_2 and L_∞ norms [6].

Given two inputs $\vec{x}, \vec{x}'$, the L_0 norm distance $\|\vec{x}-\vec{x}'\|_0$ is the number of non-zero elements in the vector $\vec{x}-\vec{x}'$, the L_2 norm distance $\|\vec{x}-\vec{x}'\|_2$ is the Euclidean distance between $\vec{x}$ and $\vec{x}'$, and the L_∞ norm distance $\|\vec{x}-\vec{x}'\|_\infty$ is the maximal entry in the vector $|\vec{x}-\vec{x}'|$. A DNN is (local) *robust* w.r.t. an input $\vec{x} \in X$ and a threshold $\epsilon > 0$ if $N_c(\vec{x}) = N_c(\vec{x}')$ for any $\vec{x}' \in X$ such that $\|\vec{x}-\vec{x}'\|_p \leq \epsilon$. Counterexamples in this setting are often called *adversarial examples*. The robustness property for any L-norm could be expressed as a neural network verification query, where the constraints $\|\vec{x}-\vec{x}'\|_p \leq \epsilon$ for $p = 0, \infty$ and $N_c(\vec{x}) = N_c(\vec{x}')$ for any $\vec{x}' \in X$ can be encoded as conjunctions of linear constraints. Therefore, we define a robustness property as a verification query $\langle P, N, Q\rangle$, where P is given by an input $\vec{x}$ and a threshold $\epsilon > 0$, and Q is given by a conjunction of linear constraints on the output. Towards robustness of DNNs, instead of qualitatively verifying if a given robustness property holds or not, one may have an interest in computing a *maximum robustness radius* ϵ such that $\langle(\vec{x}, \epsilon), N, Q\rangle$ holds but $\langle(\vec{x}, \epsilon'), N, Q\rangle$ does not hold for any $\epsilon' > \epsilon$.

Reachability is another property of DNNs which specifies that inputs from a given input region must produce outputs that lie in a given output region. For example, a DNN model controlling the velocity of an autonomous vehicle may have a safety property specifying that the model never produces a desired velocity value greater than the vehicle's maximum physical speed for any input.

As a convention in neural network verification [16], we say the verification query $\langle P, N, Q\rangle$ is satisfiable (`SAT`) if it *has* a counterexample, otherwise $\langle P, N, Q\rangle$ is unsatisfiable (`UNSAT`) indicating no counterexample can be found.

Adversarial Attacks. Consider a DNN N, an input $\vec{x} \in X$ and a distance threshold ϵ (based on L_p norms), an adversarial attack task is to find an adversarial example $\vec{x}' \in X$ such that $N_c(\vec{x}) \neq N_c(\vec{x}')$ and $\|\vec{x}-\vec{x}'\|_p \leq \epsilon$. Note that it is the same as finding a counterexample that violates the corresponding robustness property. Since the discovery of adversarial examples [54], many adversarial attacks have been invented as efficient methods for testing the robustness of DNNs [4,6,20,32,38,41,45]. We only briefly recap one representative and promising attack, Project Gradient Descent (PGD) adversarial attack [38], which will be used in our implementation.

The PGD adversarial attack is an iterative attack with a randomized start seed. It first adds a Gaussian noise to the input $\vec{x}$, resulting in a randomized seed $\vec{x}_0 \in X$ such that $\|\vec{x}-\vec{x}_0\|_\infty \leq \epsilon$. After that, it iteratively computes a sequence of input samples $\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_m$ until an adversarial example is successfully found or the number of iterations exceeds a given threshold m. Namely, $\vec{x}_{i+1} = \texttt{clip}_{\epsilon,\vec{x}}(\vec{x}_i + \alpha \cdot \texttt{sign}(\nabla_{\vec{x}} J(\vec{x}_i, y)))$ where

- $0 < \alpha < \epsilon$ is a small step size;
- y is the ground-truth class $N_c(\vec{x})$ of the input $\vec{x}$;
- $\texttt{sign}(\cdot)$ is a sign function such that $\texttt{sign}(z)$ is $+1$ if $z > 0$, -1 if $z < 0$ and 0 if $z = 0$; (it is used in the entry-wise way.)
- $\texttt{clip}_{\epsilon,\vec{x}}(\vec{x}')$ is a clip function which performs per-entry clipping of the sample $\vec{x}'$ to ensure that $\|\vec{x}-\vec{x}'\|_\infty \leq \epsilon$;

– $J(\vec{x}, y)$ is a loss function (e.g., the mean-squared error or the categorical cross-entropy of the DNN);
– $\nabla_{\vec{x}}$ is the partial derivative of the loss function $J(\vec{x}, y)$ at $\vec{x}$.

Intuitively, the attack is to search an input sample $\vec{x}' \in X$ to maximize the loss function. To prevent the attack from trapping in local optima, the above search of an adversarial example is often repeated multiple times. The details of the PGD adversarial attack algorithm are given in Appendix A.1.

Example 1. Consider the illustrative example shown in Fig. 1. we can obtain the computational flow of the neural network in terms of the specific weights and biases. Suppose we want to verify if $z_1 > z_2$, from the neural network verification point of view, we can treat these equations and properties as constraints for SMT solving [28], or perform symbolic interval analysis from the input layer-by-layer [59], etc. From the adversarial attack point of view, we simply need to find a counterexample in the input interval to disprove $z_1 > z_2$. When this problem is easy to disprove, the use of attack algorithm saves substantial time over formal verification. We explore how to synergistically combine SMT-based formal verification and adversarial attacks in this work.

3 Methodology

In this section, we present our methodology based on counterexample-guided abstraction refinement (CEGAR). We start by presenting the standard CEGAR in literature, and then explain how to integrate it with testing.

3.1 The Standard CEGAR Framework

The standard CEGAR framework based on the abstract-check-refine paradigm is shown in Algorithm 1 (*without* the blue-colored code at lines 3–9). Given a verification query $\langle P, N, Q\rangle$, upon termination Algorithm 1 returns either `UNSAT` indicating that the verification query $\langle P, N, Q\rangle$ holds, or (`SAT`,*cex*) indicating that the query $\langle P, N, Q\rangle$ does not hold where *cex* is a counterexample. In detail, Algorithm 1 first builds an initial abstract model $\widehat{N}$ via invoking the procedure `abstract` (line 1). It then iteratively verifies and refines $\widehat{N}$ until the verification query is proved `UNSAT` or a genuine counterexample *cex* in the target system N is found (lines 10–15). In each iteration, the verification query $\langle P, \widehat{N}, Q\rangle$ with the up-to-date abstract system $\widehat{N}$ is verified by invoking the underlying verification engine `verify` (line 10). If it is proved `UNSAT`, Algorithm 1 returns `UNSAT` and the verification query $\langle P, N, Q\rangle$ holds. In case $\langle P, \widehat{N}, Q\rangle$ is proved `SAT`, a counterexample *cex* is returned by `verify` whose feasibility in the target system N is checked (line 12). If *cex* is a genuine counterexample in the target system N, Algorithm 1 returns (`SAT`,*cex*) (line 13); otherwise the abstract system $\widehat{N}$ is refined via invoking the refinement procedure `refine` (line 14).

Remark that it is implicitly assumed that the abstraction `abstract` and the refinement `refine` procedures only generate over-approximations of the target

Algorithm 1: Our CEGAR framework

```
   Input  : a verification query ⟨P, N, Q⟩
   Output: verification result UNSAT, or SAT with a counterexample cex
1  N̂ ← abstract(P,N,Q);          /* Generate an initial abstract system */
2  while True do
3      cex ←test(P, N̂, Q);                  /* Test the abstract system */
4      if cex ≠ NULL then            /* Find a counterexample by testing */
5          if cex is a counterexample of ⟨P, N, Q⟩ then
6              return (SAT, cex);          /* Find a genuine counterexample */
7          else
8              N̂ ← refine(N̂, cex);          /* Refine the abstract system */
9              continue;                  /* Skip verify and back to test */
10     cex ←verify(P, N̂, Q);                /* Verify the abstract system */
11     if cex ≠ NULL then       /* Find a counterexample by verification */
12         if cex is a counterexample of ⟨P, N, Q⟩ then
13             return (SAT, cex);          /* Find a genuine counterexample */
14         else N̂ ← refine(N̂, cex);         /* Refine the abstract system */
15     else return UNSAT
```

system N and the underlying verification engine `verify` is sound. Otherwise, one cannot conclude that verification query $\langle P, N, Q\rangle$ holds even if Algorithm 1 returns `UNSAT`. Furthermore, the underlying verification engine is often required to be complete and has the capability for producing a counterexample if the verification query is `SAT`, namely, the verification of $\langle P, \widehat{N}, Q\rangle$ returns either `UNSAT` or a counterexample cex if `SAT`.

3.2 Our CEGAR Framework

Our CEGAR framework is based on the key observation that it is fast to find counterexamples in the coarse-grained, abstract systems via testing techniques. As a result, we propose an *abstract-test-refine* paradigm, where check-refine is applied *only* when the testing fails to find a counterexample. Our CEGAR framework is shown in Algorithm 1, where the blue-colored code (lines 3–9) follows the abstract-test-refine paradigm while the other code is the same as in the standard CEGAR framework.

Given a verification query $\langle P, N, Q\rangle$, after building the initial abstract system $\widehat{N}$ (line 1), Algorithm 1 first repeatedly tests and refines the abstract system $\widehat{N}$ until either a counterexample found in the abstract system $\widehat{N}$ is genuine in the target system N; or the procedure `test` fails to find a counterexample in the abstract system $\widehat{N}$ (within a given test budget) (lines 3–9). It is easy to see that the test-refine (lines 3–9) is the same as the original check-refine (lines 10–15), except that the `verify` procedure is replaced by the `test` procedure. When the testing fails to found an adversarial example, check-refine is applied as in the de facto CEGAR scheme except that the refined system $\widehat{N}$ is retested again in

the test-refine loop. At this moment, the abstract system $\widehat{N}$ may have already been significantly refined by the test-refine loop so that computational expensive verification of many coarse-grained abstract systems could be avoided. Ideally, if the testing method is powerful enough, it would be able to find a counterexample in most cases. Consequently, for the verification query that does not hold, the test-refine loop could more likely find a genuine counterexample and avoid calls to verification, thus, the `verify` procedure would be rarely invoked. We note that for the verification query that holds, `verify` would be invoked at least once.

Proposition 1. *If Algorithm 1 returns (*`SAT`*,cex), then cex is a counterexample of the verification query* $\langle P, N, Q\rangle$*. If Algorithm 1 returns* `UNSAT`*, then the verification query* $\langle P, N, Q\rangle$ *holds.* □

Remark that, the new CEGAR scheme may not be effective in verifying general software/hardware systems, as finding counterexamples is still non-trivial via testing. However, for neural networks, counterexamples (adversarial attacks) are pervasive and there have been advanced techniques to find them (cf. Sect. 2).

4 DNN Verification in Our CEGAR Scheme

In this section, we first recall the preprocessing, abstraction and refinement procedures provided in CEGAR-NN based on which we show how to instantiate our CEGAR framework by leveraging the PGD adversarial attack [38] for testing due to its effectiveness and efficiency. We should emphasize that our CEGAR scheme can be used on any de facto CEGAR-based DNN verification approaches and leverage any promising testing methods such as BIM [32], DeepFool [41], C&W [6] and DeepXplore [46].

4.1 CEGAR-NN

CEGAR-NN instantiates the `abstract`, `verify` and `refine` procedures in CEGAR, where `verify` is implemented by the Marabou DNN verification engine [29].

Preprocessing. CEGAR-NN first preprocesses a verification query $\langle P, N, Q\rangle$, by transforming it into an equivalent verification query $\langle P, N', Q'\rangle$ such that the post-condition Q' is a conjunction of linear inequalities of form $y > c$ for some constant c. Furthermore, each hidden neuron should be classified as a `pos`/`neg` neuron, and a `dec`/`inc` neuron. A hidden neuron is `pos` (resp. `neg`) if all the weights on its outgoing edges are positive (resp. negative), while a hidden neuron is `inc` (resp. `dec`) if increasing the value of this neuron while keeping all the inputs unchanged increases (resp. decreases) the values of the output neurons. As stated by Elboher et al. [16], these restrictions are for the sake of simplicity, and can be achieved by adding a few neurons (at most 4× increase in network size) during preprocessing. From now on, we assume the verification query $\langle P, N, Q\rangle$ has already been preprocessed and satisfies the above assumptions.

The `abstract` and `refine` Procedures. CEGAR-NN has two abstraction strategies, called *abstraction-to-saturation* and *indicator-guided abstraction*. Both strategies are based on the `merge` operator, which merges a pair of hidden neurons in a same layer that share the same `pos`/`neg` and `inc`/`dec` attributes, resulting in an over-approximated DNN. The abstraction-to-saturation strategy iteratively applies the `merge` operator, producing the smallest abstract DNN. However, this strategy may obtain DNNs that are too coarse so that multiple rounds of refinement are required. The indicator-guided abstraction strategy is proposed to address this issue by estimating when the abstraction has become too coarse using a finite set of chosen inputs X_I. After each abstraction step, the post-condition Q is checked in the abstract DNN using the chosen inputs. If the post-condition Q is violated by some input in X_I, the abstraction is then stopped.

Generally speaking, the `refine` procedure is the inverse of `abstract`, which refines an abstract DNN by iteratively recovering two merged neurons from the corresponding abstract neuron until the counterexample is excluded.

4.2 Instantiating Our CEGAR Scheme

To instantiate our CEGAR framework, we show how to disprove a verification query $\langle P, N, Q\rangle$ and improve the `abstract` procedure, both via an adversarial attack based testing.

Disproving Verification Query. Given a verification query $\langle P, N, Q\rangle$, we assume that P is a conjunction of linear constraints $\bigwedge_{i=1}^{m} lp_i \leq x_i \leq up_i$ on the input values and Q is a conjunction of linear inequalities of $\bigwedge_{i=1}^{n} y_i > c_i$, where the variables x_i's and y_i's correspond to the values of input neurons and output neurons respectively, and lp_i's, up_i's and c_i's are constants. Such properties are widely considered in the DNN verification community, e.g., [18,28,58,66]. To leverage an adversarial attack for testing, we encode the pre-condition P by transforming a conjunction of linear constraints $\bigwedge_{i=1}^{m} lp_i \leq x_i \leq up_i$ into a non-standard L_∞ epsilon ball, and encode the post-condition Q in a loss function J which is maximized by the adversarial attack to find a counterexample.

- **Encoding the pre-condition P.** From the pre-condition P, we let $\vec{\hat{x}}$ be an input such that for every $1 \leq i \leq m$, $\vec{\hat{x}}[i] = \frac{up_i + lp_i}{2}$, and $\vec{\epsilon}$ be a vector such that $\vec{\epsilon}[i] = \frac{up_i - lp_i}{2}$ for every $1 \leq i \leq m$. Clearly, for each $\vec{x}' \in X$, $|\vec{\hat{x}} - \vec{x}'| \leq \vec{\epsilon}$ iff $\vec{x}'$ satisfies P. We denote by `encode`(P) the pair $(\vec{\hat{x}}, \vec{\epsilon})$. For example, suppose $m = 2$ and the constraints lp_i and up_i are $[0, 0.5]$ and $[1, 1]$ for $i = 1, 2$, respectively, then we can obtain $\vec{\hat{x}} = [0.5, 0.75]$, $\vec{\epsilon} = [0.5, 0.25]$.
- **Encoding the post-condition Q.** From the post-condition Q, we define the loss function J as

$$J(\vec{x}) := -\prod_{i=1}^{n} \Big(\max(N(\vec{x})[i] - c_i, 0) \Big)$$

  where $N(\vec{x})[i]$ denotes the value of the output neuron y_i. The output property Q in general can be an arbitrary Boolean structure and involve multiple

Algorithm 2: PGD adversarial attack based testing

Input : a verification query $\langle P, N, Q\rangle$, restart times n, number of steps per time m, a small step size α
Output: an adversarial example *cex* or `NULL`

1 $(\vec{\widehat{x}}, \vec{\epsilon}) \leftarrow$ `encode`(P);
2 $J \leftarrow$ `encode`(Q);
3 **for** $i \leftarrow 1$ **to** n **do**
4 Generate a vector of Gaussian noises $\vec{\delta}$ such that $|\vec{\delta}| \leq \vec{\epsilon}$;
5 $\vec{x}' \leftarrow \vec{\widehat{x}} + \vec{\delta}$; /* Create a randomized seed */
6 **for** $j \leftarrow 1$ **to** m **do**
7 $\vec{y} \leftarrow N(\vec{x}')$; /* Get the output */
8 **if** $\vec{y}$ *does not satisfy the post-condition* Q **then**
9 **return** $\vec{x}'$; /* Find a counterexample */
10 **else**
11 $\nabla \leftarrow$ `back_propagate`$(N, J(\vec{x}'))$; /* Get gradient of $J(\vec{x}')$ */
12 $\vec{x}' \leftarrow$ `clip`$_{\vec{\epsilon}, \vec{\widehat{x}}}(\vec{x}' + \alpha \times \text{sign}(\nabla))$; /* Compute a new sample */
13 **return** `NULL`;

neurons which can be transformed into a conjunction of linear inequalities (cf. [16]). Recall that an adversarial attack attempts to maximize $J(\vec{x})$, hence to minimize each term $\max(N(\vec{x})[i] - c_i, 0)$ until it is 0. When $J(\vec{x}')$ is 0 for some input $\vec{x}'$, there exists some i such that $N(\vec{x}')[i] > c_i$ does not hold, hence the output $N(\vec{x}')$ does not satisfy Q. This input $\vec{x}'$ is a counterexample. We denote by `encode`(Q) the loss function J. We should emphasize that our loss function J is different from the cross-entropy loss function used in the PGD adversarial attack [38], which is not applicable when the output property involves lower or upper bounds. Our loss function is constructed for each output property given as a conjunction of linear inequalities, and can be applied in a variety of verification problems [18,28,58,66].

Based on the above encodings, we implement the `test` procedure for Algorithm 1 via a PGD adversarial attack based testing (cf. Algorithm 2). Given a verification query $\langle P, N, Q\rangle$, the number of restart times n, the number of iteration steps per time m, a small step size α, Algorithm 2 returns either a counterexample $\vec{x}'$ that satisfies the pre-condition P but violates the post-condition Q, or `NULL` indicating that no counterexample can be found. Note that the pair of the parameters (n, m) is regarded as the test budget.

In detail, Algorithm 2 first computes the pair $(\vec{\widehat{x}}, \vec{\epsilon})$ that encodes the inputs fulfilling the pre-condition P (line 1) and the loss function J that encodes the post-condition Q (line 2). Then, it iteratively executes the outer for-loop (lines 3–12) up to n times. During each iteration, a randomized seed $\vec{x}'$ is obtained by adding Gaussian noises $\vec{\delta}$ to $\vec{\widehat{x}}$ (lines 4–5) and then the inner for-loop (lines 6–12) is executed, which iteratively computes a series of new samples (up to m samples) starting from the randomized seed $\vec{x}'$.

During each iteration of the inner for-loop (lines 6–12), Algorithm 2 first computes the output $\vec{y} = N(\vec{x}')$ of the DNN N by forward propagating the input $\vec{x}'$. If $\vec{y}$ does not satisfy the post-condition Q, then $\vec{x}'$ is a counterexample and Algorithm 2 returns $\vec{x}'$. Otherwise, Algorithm 2 performs a backward propagation to get the gradient ∇ of J using $J(\vec{x}')$ (line 11) from which a new sample $\vec{x}'$ is created (line 12), where the clip function $\texttt{clip}_{\vec{\epsilon},\widehat{\vec{x}}}$ ensures that $|\vec{\widehat{x}} - \vec{x}'| \leq \vec{\epsilon}$ after updating, hence the new sample $\vec{x}'$ still satisfies the pre-condition P.

Lemma 1. *If Algorithm 2 returns cex for the verification query $\langle P, N, Q\rangle$, then cex is a counterexample of the verification query $\langle P, N, Q\rangle$.*

One may be wondering how to choose hyper-parameters such as restart times n, number of steps per time m, and step size α, and how to handle non-differentiable layers when leveraging adversarial attacks. According to our experiments, the time consumed by attacks is marginal compared to that used in the complete verification method, and the parameters can be selected as in the prior work [32,38]. For non-differentiable layers, gradient estimation methods (e.g. [9]) can be used to approximate the gradient of J.

Improving `abstract` via Attacks. We exploit the adversarial attack based testing in the building of the initial abstract system, i.e., the `abstract` procedure in Algorithm 1.

Recall that Elboher et al. [16] proposed two abstraction strategies in CEGAR-NN: abstraction-to-saturation and indicator-guided abstraction, where the former may produce DNNs that are too coarse so that multiple rounds of refinement are required, while the latter is proposed to address this issue by checking if the abstraction has become too coarse using a finite set of chosen inputs X_I, all of which satisfy the pre-condition P. It was mentioned that the set X_I can be generated randomly (adopted in their tool), or according to some coverage criterion of the input region. In this work, we present a more effective way, i.e., *attack-guided abstraction*, to generate the set X_I via an adversarial attack based testing which are more likely to be counterexamples in abstract systems.

We first adjust Algorithm 2 to return all generated $n \times m$ samples, named Algorithm 2*. Our attack-guided abstraction is formalized in Algorithm 3. Given a verification query $\langle P, N, Q\rangle$, and the parameters (restart times n, number of steps per time m, step size α) for the adversarial attack (cf. Algorithm 2*), Algorithm 3 returns an abstract DNN $\widehat{N}$.

In detail, the abstract DNN $\widehat{N}$ is initialized with the given DNN N (line 1) and a set X_I of samples is created by applying Algorithm 2* to $\langle P, \widehat{N}, Q\rangle$ (line 2). After that, we check if the post-condition Q holds using the samples from X_I (line 3). If Q is violated by some sample $\vec{x} \in X_I$, Algorithm 3 returns $\widehat{N}$ (line 4). Otherwise, it iteratively performs the `merge` operation to compute a less accurate abstract DNN $\widehat{N}'$ and tests $\widehat{N}'$ against Q until either no neurons that can be merged or $\langle P, \widehat{N}', Q\rangle$ becomes `SAT`, i.e., a counterexample is found (lines 5–15).

During each iteration of the while-loop (lines 5–15), Algorithm 3 first chooses a mergeable pair (v_i, v_j) of neurons (line 6), for which we adopt an approach

Algorithm 3: Attack-guided abstraction

Input : a verification query $\langle P, N, Q\rangle$, restart times n, number of steps per time m, step size α
Output: an abstract DNN $\widehat{N}$

1 $\widehat{N} \leftarrow N$;
2 $X_I \leftarrow$ the set of samples created by applying Algorithm 2* to $\langle P, \widehat{N}, Q\rangle$;
3 **if** $\exists \vec{x} \in X_I.\widehat{N}(\vec{x})$ *does not satisfy* Q **then**
4 | **return** $\widehat{N}$;
5 **while** $\exists$ *a pair of neurons that can be merged* **do**
6 | $(v_i, v_j) \leftarrow$`ChooseBestMergeablePair`$(\widehat{N})$;
7 | $\widehat{N}' \leftarrow$ `merge`$(\widehat{N}, v_i, v_j)$;
8 | **if** $\exists \vec{x} \in X_I.\widehat{N}'(\vec{x})$ *does not satisfy* Q **then**
9 | | **return** $\widehat{N}$;
10 | **else**
11 | | $X' \leftarrow$ the set of samples created by applying Algorithm 2* to $\langle P, \widehat{N}', Q\rangle$;
12 | | **if** $\exists \vec{x} \in X'.\widehat{N}'(\vec{x})$ *does not satisfy* Q **then**
13 | | | **return** $\widehat{N}$;
14 | | **else** $X_I \leftarrow X_I \cup X'$
15 | $\widehat{N} \leftarrow \widehat{N}'$;
16 **return** $\widehat{N}$;

by Elboher et al. [16]. Next, we build a less accurate abstract DNN $\widehat{N}'$ by merging (v_i, v_j) in $\widehat{N}$ (line 7) and test if there exists some counterexample $\vec{x} \in X_I$ for $\langle P, \widehat{N}', Q\rangle$ (line 8). If so, we return the previous abstract DNN $\widehat{N}$ (line 9). Otherwise, we create a new set X' of samples by applying Algorithm 2* to the query $\langle P, \widehat{N}', Q\rangle$ (line 11). After that, the query $\langle P, \widehat{N}', Q\rangle$ is tested again using the new samples from X' (line 12). If a counterexample $\vec{x} \in X'$ for $\langle P, \widehat{N}', Q\rangle$ exists, we return the previous abstract DNN $\widehat{N}$ (line 13); otherwise the set X_I and the abstract DNN $\widehat{N}$ are updated accordingly for the next iteration.

Lemma 2. *For any verification query $\langle P, N, Q\rangle$, if Algorithm 3 returns an abstract DNN $\widehat{N}$, then either $\langle P, \widehat{N}, Q\rangle$ or $\langle P, \widehat{N}', Q\rangle$ has a counterexample, where $\widehat{N}'$ is the abstract DNN obtained from $\widehat{N}$ by merging a pair of neurons in the while-loop of Algorithm 3. Furthermore, $\widehat{N}$ is an over-approximation of the DNN N according to soundness of the* `merge` *operator [16].*

CLEVEREST-NN. By instantiating the `abstract` and `test` procedures in Algorithm 3 and Algorithm 2 respectively, as well as the `refine` and `verify` procedures implemented as in CEGAR-NN, we obtain a concrete CEGAR-based neural network verification algorithm, named CLEVEREST-NN. Thanks to the completeness of `verify` in CEGAR-NN and the termination guarantee of the refinement, CLEVEREST-NN is both sound and complete.

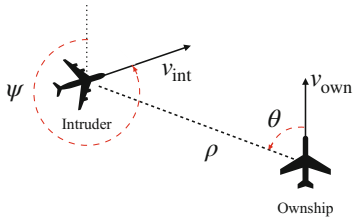

Fig. 2. An illustrating scenario of the ACAS Xu system

In addition to solving verification queries, CLEVEREST-NN also features a binary-search based approach to approximate the maximum robustness radius ϵ such that $\langle(\vec{x}, \epsilon), N, Q\rangle$ holds for a given DNN N, an input $\vec{x} \in X$ and a post-condition Q. To this end, for each candidate ϵ, we leverage the CEGAR-based approach to verify $\langle(\vec{x}, \epsilon), N, Q\rangle$.

5 Implementation and Evaluation

We have implemented our method in the tool CLEVEREST-NN, where the `verify` and `refine` modules are the same as CEGAR-NN [16]. (Indeed, `verify` is the Marabou DNN verification engine [29].) The input of CLEVEREST-NN is a DNN in the NNet format, pre- and post-conditions, forming a verification query, and the parameters (n, m, α) for adversarial attack based testing. When computing a maximal robustness radius, the pre-condition should be an input sample, the lower bound and upper bound of the radius, instead of a linear constraint.

We conduct experiments on 45 ACAS Xu DNNs for airborne collision avoidance [27,28]. ACAS Xu is a system (cf. Figure 2) designed for an unmanned aircraft (called `Ownship`) to produce horizontal turning advisories in order to prevent a collision with another nearby aircraft (called `Intruder`). Each ACAS Xu DNN has 310 neurons, 5 inputs, 6 hidden layers and 5 outputs. The five inputs are normalized data from airborne sensors, indicating the distance ρ between `Intruder` and `Ownship`, the relative angles θ, ψ between `Ownship` and `Intruder`, the speeds v_{own} and v_{int} of `Ownship` and `Intruder`. The five outputs represent turning advisories: strong left, weak left, strong right, weak right, or clear-of-conflict (i.e., safe to continue along the current trajectory). The ACAS Xu system selects one of 45 DNNs according to the data reading from the airborne sensors and the turning advisory of the selected DNN with the lowest score is the final turning advisory of the system.

In our experiments, we consider two groups of verification queries and one group of queries for computing maximal robustness radii, where the former two groups are provided by CEGAR-NN and the latter one is obtained from Reluplex [28]. The first group, called `COC-queries`, consists of 2 verification queries for each ACAS Xu DNN, which ensure that the DNN always advises clear-of-conflict for distant intruders, i.e., the output of clear-of-conflict is always smaller than the other labels (e.g., the previous runner-up operation). The second group,

called `ROB-queries`, consists of 20 robustness properties with $\epsilon = 0.1$ which ensure that the DNN is robust against small input perturbations.

The third group, called `MR-queries`, consists of 5 queries for one chosen ACAS Xu DNN, which are used to compute maximal robustness radii, where the five inputs are Points 1–5 of Reluplex [28].

We first evaluate the effectiveness of our attack-guided abstraction strategy (i.e., Algorithm 3), and then evaluate the effectiveness of our CEGAR scheme (cf. Sect. 3.2) using the PGD adversarial attack based testing (i.e., Algorithm 2) both for solving verification queries, and finally evaluate the performance of the overall framework for computing maximal robustness radii. For the sake of presentation, we refer to the different CEGAR schemes with two different abstraction strategies as follows.

- SAT-CLEVEREST-NN = CLEVEREST-NN + abstraction-to-saturation,
- SAT-CEGAR-NN = CEGAR-NN + abstraction-to-saturation,
- ATT-CLEVEREST-NN = CLEVEREST-NN + attack-guided abstraction,
- ATT-CEGAR-NN = CEGAR-NN + attack-guided abstraction.

The experiments were conducted on a machine with Intel Xeon CPU E5-2690 2.60GHz CPU, 64-bit Ubuntu 18.04 LTS operating systems, 256G RAM, with a 3 h timeout per query unless stated explicitly. Note that all experiments were performed on the CPU only for a fair comparison with CEGAR-NN. We remark that the adversarial attack in our CLEVEREST-NN could be accelerated using GPU. The restart times n, number of steps per time m and step size α of the PGD adversarial attack based testing are 10, 10 and $\frac{up-lp}{4}$, respectively, where up and lp are the upper bound and lower bound of the inputs. Note that we compare with [16] only because it is the only publicly available CEGAR-based tool that abstracts/refines network structures. We expect to verify more properties and datasets in the future, with the development and implementation of the CEGAR-based neural network verification framework.

5.1 Performance of Our Attack-guided Abstraction

To evaluate the effectiveness of our attack-guided abstraction strategy, we compare it with the abstraction-to-saturation strategy in both the CEGAR-NN and CLEVEREST-NN frameworks (i.e., ATT-CEGAR-NN vs. SAT-CEGAR-NN and ATT-CLEVEREST-NN vs. SAT-CLEVEREST-NN) for solving 90 (2×45) `COC-queries`. We exclude the indicator-guided abstraction strategy, as it was shown in [16] that the indicator-guided abstraction strategy is significantly worse than the abstraction-to-saturation strategy in the CEGAR-NN framework.

SAT-CEGAR-NN vs. ATT-CEGAR-NN. Figure 3(a) depicts a comparison between ATT-CEGAR-NN and SAT-CEGAR-NN. The blue marks above the red dashed line are the verification queries where ATT-CEGAR-NN (i.e., the attack-guided abstraction strategy) is faster. The red marks on the top are the verification queries where SAT-CEGAR-NN time-outs, while those on the right

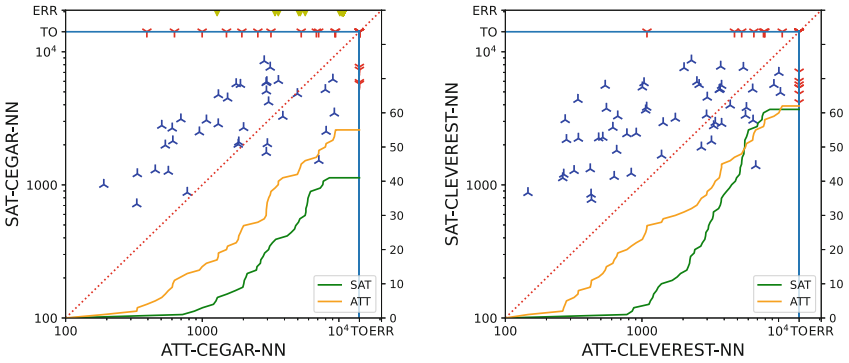

Fig. 3. (a) Comparison between SAT-CEGAR-NN and ATT-CEGAR-NN and (b) comparison between SAT-CLEVEREST-NN and ATT-CLEVEREST-NN, for solving the 90 `COC-queries`, where the scatter plots compare execution time (log-scale, in seconds); TO denotes timeout; ERR denotes erroneous results on abstract DNNs; the curve plots the number of solved queries with the increased time limit per query.

are where ATT-CEGAR-NN time-outs. The yellow marks on the top are verification queries where SAT-CEGAR-NN reported incorrect results on abstract DNNs. SAT-CEGAR-NN reported `UNSAT` on 28 abstract DNNs that are indeed `SAT`.[1]

In summary, ATT-CEGAR-NN solved 55 out of 90 verification queries while SAT-CEGAR-NN solved 41. On those solved by both tools, ATT-CEGAR-NN is faster than SAT-CEGAR-NN on 75.68% verification queries and the average speed-up is 2.23×. From the curve plot in Fig. 3(a), we can observe that ATT-CEGAR-NN constantly solve more verification queries than SAT-CEGAR-NN with the increased time limit per query. We conclude that our attack-guided abstraction strategy outperforms the abstraction-to-saturation strategy in CEGAR-NN.

SAT-CLEVEREST-NN vs. ATT-CLEVEREST-NN. Figure 3(b) depicts a comparison between ATT-CLEVEREST-NN and SAT-CLEVEREST-NN on solving the 90 `COC-queries`.

In summary, ATT-CLEVEREST-NN solved 62 out of 90 verification queries while SAT-CLEVEREST-NN solved 61. On those solved by both tools, ATT-CLEVEREST-NN is faster than SAT-CLEVEREST-NN on 76.92% of the verification queries and the average speed-up is 2.17×. From the curve plot in Fig. 3(b), we can observe that ATT-CLEVEREST-NN can solve more verification queries

[1] This issue has been reported to and confirmed by some authors of Marabou and CEGAR-NN; they replied that this problem is triggered by networks having both very small and very large weights. ATT-CEGAR-NN avoided these errors because these abstract DNNs were proved `SAT` via our adversarial attack based testing. We have performed differential verification using another sound and complete tool on all intermediate abstract DNNs to confirm our findings.

than SAT-CLEVEREST-NN with the increased time limit per query up to 5,500 s, while SAT-CLEVEREST-NN becomes slightly better than ATT-CLEVEREST-NN when the time limit per query is greater than 5,500 s.

5.2 Performance of Our CEGAR Framework CLEVEREST-NN

To evaluate the effectiveness of CLEVEREST-NN, we compare CLEVEREST-NN and CEGAR-NN configured with the same abstraction strategy, i.e., SAT-CEGAR-NN vs. SAT-CLEVEREST-NN and ATT-CEGAR-NN vs. ATT-CLEVEREST-NN. We use the 90 (2×45) `COC-queries` and 900 (20×45) `ROB-queries`.

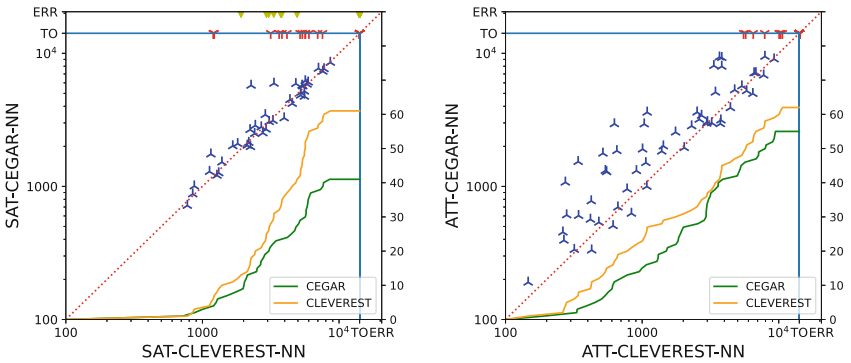

Fig. 4. (a) Comparison between SAT-CEGAR-NN and SAT-CLEVEREST-NN and (b) comparison between ATT-CEGAR-NN and ATT-CLEVEREST-NN, for solving the 90 `COC-queries`.

SAT-CEGAR-NN vs. SAT-CLEVEREST-NN on COC-queries. Figure 4(a) depicts a comparison between SAT-CEGAR-NN and SAT-CLEVEREST-NN for solving the 90 `COC-queries`.

In summary, SAT-CLEVEREST-NN solved 61 out of 90 verification queries, while SAT-CEGAR-NN solved 41 and reported erroneous results on abstract DNNs for 28 verification queries. On those solved by both tools, SAT-CLEVEREST-NN is faster than SAT-CEGAR-NN on 53.66% verification queries and the average speed-up is 1.09×.

ATT-CEGAR-NN vs. ATT-CLEVEREST-NN on COC-queries. Figure 4(b) depicts a comparison between ATT-CEGAR-NN and ATT-CLEVEREST-NN for solving the 90 `COC-queries`. ATT-CLEVEREST-NN solved 62 out of 90 verification queries, while ATT-CEGAR-NN solved only 55 verification queries without reporting erroneous results on abstract DNNs. On those solved by both tools, ATT-CLEVEREST-NN is faster than SAT-CEGAR-NN on 74.54% verification queries and the average speed-up is 1.64×. From the curve

plot in Fig. 4(a) (resp. Figure 4(b)), we can observe that SAT-CLEVEREST-NN (resp. ATT-CLEVEREST-NN) constantly solved more verification queries than SAT-CEGAR-NN (resp. ATT-CEGAR-NN) with the increased the time limit per query with just a handful of exceptions. These results suggest that our CEGAR framework CLEVEREST-NN is more effective than CEGAR-NN for both the abstraction-to-saturation and attack-guided-saturation strategies.

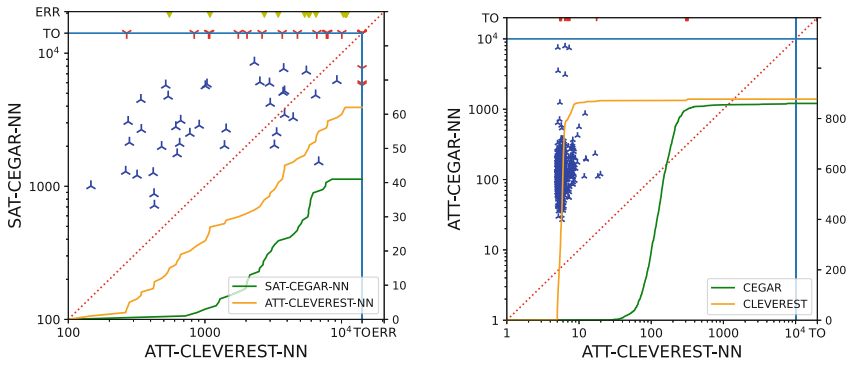

Fig. 5. (a) Comparison between SAT-CEGAR-NN and ATT-CLEVEREST-NN for solving the 90 `COC-queries`, and (b) Comparison between ATT-CEGAR-NN and ATT-CLEVEREST-NN, for solving the 900 (20×45) `ROB-queries`.

SAT-CEGAR-NN vs. ATT-CLEVEREST-NN on COC-queries. Figure 5(a) depicts a comparison between SAT-CEGAR-NN and ATT-CLEVEREST-NN for solving the 90 `COC-queries`. ATT-CLEVEREST-NN solved 62 out of 90 verification queries, while SAT-CEGAR-NN solved only 41 verification queries and reported erroneous results on abstract DNNs for 28 verification queries. On those solved by both tools, ATT-CLEVEREST-NN is faster than SAT-CEGAR-NN on 81.58% verification queries and the average speed-up is 3.75×. From the curve plot in Fig. 5(a), we can observe that ATT-CLEVEREST-NN solved more verification queries than SAT-CEGAR-NN with the increased time limit per query. These results reveal the improvement brought by this work over CEGAR-NN.

ATT-CEGAR-NN vs. ATT-CLEVEREST-NN on ROB-queries. Figure 5(b) depicts a comparison of between ATT-CEGAR-NN and ATT-CLEVEREST-NN for solving the 900 (20×45) `ROB-queries`.

ATT-CLEVEREST-NN solved 877 out of 900 verification queries, while ATT-CEGAR-NN solved 860 verification queries without reporting any incorrect results on the abstract DNNs. On those solved by both tools, ATT-CLEVEREST-NN is faster than SAT-CEGAR-NN on all the verification queries and the average speed-up is 29.52×. These results indicate that our CEGAR framework is significantly more efficient in verifying robustness properties. We found that almost all

Table 1. #Call per query to the verification engine

Tool	#Call
SAT-CEGAR-NN	2.24
SAT-CLEVEREST-NN	1.09
ATT-CEGAR-NN	2.49
ATT-CLEVEREST-NN	1.00

Table 2. #Binary search step

	Index 1		Index 2		Index 3		Index 4		Average	
	No_c	No_a	No_c	No_a	No_c	No_a	No_c	No_a	No_c	No_a
Point 1	1	3	1	8	4	9	2	2	2.0	5.5
Point 2	1	2	1	3	1	3	1	3	1.0	2.75
Point 3	1	3	1	2	1	1	1	1	1.0	1.75
Point 4	1	5	1	6	1	6	1	6	1.0	5.75
Point 5	1	1	1	1	1	1	1	1	1.0	1.0

`ROB-queries` are non-robust on which ATT-CLEVEREST-NN is able to disprove most of the verification queries without invoking the verification engine.

Understanding the Improvements. To understand why ours can improve the performance, we analyze the usage of the underlying verification engine and compare the execution time of `test` and `verify` operations on abstract DNNs, for verifying the 90 `COC-queries`.

Table 1 reports the number of average calls to the verification engine per verification query (where the verification queries on which CEGAR-NN reported erroneous results on abstract DNNs are excluded). We can observe that both our attack-guided abstraction and abstract-test-refine paradigm can reduce the usage of formal verification (except for SAT-CEGAR-NN vs. ATT-CEGAR-NN), which play a major role in improving the efficiency. When ATT-CLEVEREST-NN is used, the verification engine is invoked only once per verification query. (Note that the only verification is unavoidable, because all the 90 `COC-queries` are `UNSAT`, so the verification engine has to be used to prove `UNSAT`.) The number of calls to the verification engine for SAT-CEGAR-NN and ATT-CEGAR-NN is somehow counter-intuitive. We found it is because SAT-CEGAR-NN often performs a large number of `merge` operations in one refinement step to exclude a counterexample, while ATT-CEGAR-NN only performs few `merge` operations in one refinement step to exclude a counterexample. The execution time is improved by reducing the number `merge` operations.

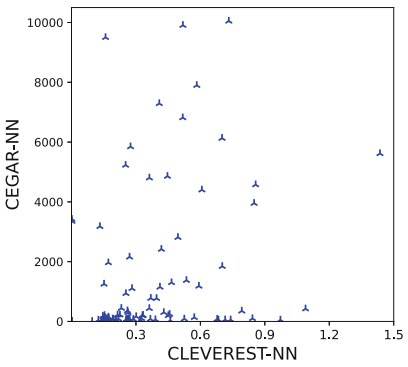

Fig. 6. Comparison of `test` and `verify` operations on abstract DNNs between CLEVEREST-NN and CEGAR-NN.

Figure 6 depicts the execution time of `test` and `verify` operations on abstract DNNs between CLEVEREST-NN and CEGAR-NN using both abstraction strategies, where the last calls to the verification engine are executed. We can observe that testing is significantly faster than formal verification. Indeed, the average testing time used by CLEVEREST-NN is 0.26s while the average verification time used by CEGAR-NN is 1006.94s, with average speed-up 31,513× per verification query.

5.3 Approximating Maximum Robustness Radii

To evaluate CLEVEREST-NN for approximating maximum robustness radii, we compare the number of binary-search steps of ATT-CLEVEREST-NN and ATT-CEGAR-NN within 6 h on the 20 `MR-queries`, where the larger number indicates better capability for approximating maximum robustness radii. The 20 `MR-queries` are obtained from the Points 1–5 of Reluplex [28] each of which has four queries (named Index 1–4) for approximating maximum robustness radii without changing the *clear-of-conflict* output advisory.

The results are shown in Table 2, where columns (No_c) and (No_a) give the number of binary search steps of ATT-CEGAR-NN and ATT-CLEVEREST-NN respectively. We can observe that ATT-CLEVEREST-NN excels in this case.

6 Related Work

CLEVEREST proposes a synergy between testing and CEGAR-based formal verification for neural networks. As there is a vast amount of literature regarding these topics, we discuss here the most relevant ones.

Robustness Testing. The robustness of neural networks have received extensive attention over the past few years. Many adversarial attacks under the white-box setting have been proposed [6,12,20,32,38,45], where white-box means that all information about the network is available. White-box adversarial attacks often find counterexamples by leverage gradient information, therefore are highly efficient. There also exist black-box adversarial attacks [4,9] that use only the inputs and outputs of the network to find counterexamples. We instantiate our methodology by leveraging the PGD adversarial attack which is a white-box one, as it is generally assumed that all network details are known during network verification. Remarkably, both black-box and white-box adversarial attacks could be leveraged in our CEGAR scheme thanks to the generality of our methodology.

Neural networks have also received attention from the perspective of traditional software testing. For example, DeepXplore [46] proposes the notion of neuron coverage to guide the testing process. Following their idea, a series of coverage criteria have been suggested for neural network testing [30,36]. Conventional testing techniques have also been adapted to test neural networks, such as concolic testing [53] and mutation testing [37]. We did not use neuron coverage to guide testing in this work, as several coverage metrics are not related

to robustness [13] and coverage-guided testing is mainly used to improve the coverage, instead of quickly finding counterexamples.

Neural Network Verification. Various formal verification techniques have been proposed to verify neural networks including robustness and fairness properties, based on abstract interpretation [18,21,33,40,42,49,50,56,57,59,60,62,63,65], and constraint solving (e.g., SMT [15,24,28,29,48], LP and MILP [5,14,34,55,61]). Although these approaches feature theoretical guarantees, they usually suffer limitations in either scalability or efficiency, hence are difficult to be applied to precisely verify large models in practice. To address the issue, different approaches have emerged. A few approaches, such as proof reuse [17], input quantization [26], divide-and-conquer [5], eager falsification [23] and network abstraction [2,16,19,47,52], have been proposed to accelerate the verification while some others were proposed to refine either the estimated output ranges [50,60,63,65] or the abstract network [16,44]. We instead offer an alternative solution by integrating the efficient yet inaccurate testing techniques into the CEGAR-based verification framework. As mentioned before, we did not compare with these approaches, as our main goal is to push the frontier of CEGAR-based verification approaches towards which this work makes a significant step.

Our methodology is general and can leverage any testing methods, iterative abstractions, CEGAR-based schemes and back-end verification engines. As these continue to improve, it is expected that our method will become more scalable.

Combination of Testing and Verification. There also exist techniques in the conventional software verification field combining testing and verification to mitigate the high complexity of verification. For instance, the authors in [11,25] combine both techniques together but the techniques do not assist each other. Instead, they test and verify different subprograms separately by program partitioning or constructing residual programs. The approaches proposed in [43,64] leverage testing techniques to choose a good abstraction for verification, whilst the authors utilize the information from testing to refine the abstraction in the case spurious counterexamples are found [3,22]. Our methodology CLEVEREST presents the first attempt to synergistically combine these two complementary techniques under the neural network setting, specifically, accelerating the de facto CEGAR framework by integrating the abstract-test-refine paradigm.

7 Conclusion

In this paper, we have proposed a new CEGAR-based framework CLEVEREST for DNN verification by synergistically combining testing and CEGAR-based verification techniques, which brings the best of both worlds. We have instantiated and implemented our methodology by leveraging the CEGAR-NN verification approach and the PGD adversarial attack, giving rise to the tool CLEVEREST-NN. Extensive experiments on the ACAS Xu DNN benchmark demonstrated the efficacy of our methodology.

A Appendix

A.1 PGD Adversarial Attack Algorithm

Algorithm 4 describes the process of the PGD adversarial attack. Given a DNN N, an input $\vec{x} \in X$, the number of restart times n, the number of iteration steps per time m, a step size α, a L_∞ norm distance threshold ϵ, Algorithm 4 returns either an adversarial example $\vec{x}'$ such that $N_c(\vec{x}) \neq N_c(\vec{x}')$ and $\|\vec{x} - \vec{x}'\|_\infty \leq \epsilon$, or `NULL` indicating that no adversarial example can be found.

In detail, the outer for-loop (lines 1–11) performs up to n times of iterations, each of which has a randomized seed $\vec{x}'$ obtained by adding a Gaussian noise δ onto the input $\vec{x}$ (lines 2–3). During each iteration of the outer for-loop, the inner for-loop (lines 4–10) iteratively computes a seises of new samples (up to m samples) starting from the randomized seed $\vec{x}'$.

During each iteration of the inner for-loop (lines 4–10), Algorithm 4 first computes the classification result $y = N_c(\vec{x}')$ of the DNN N by forward propagating the input $\vec{x}'$ and then compares the result with the ground-truth class $N_c(\vec{x})$. If they are different, $\vec{x}'$ is an adversarial example and Algorithm 4 returns $\vec{x}'$. If they are the same, then $\vec{x}'$ not is an adversarial example. Algorithm 4 performs a backward propagation to get the gradient $\nabla_{\vec{x}}$ (line 9) from which a new sample $\vec{x}'$ is created (line 10).

Algorithm 4: PGD adversarial attack

input : a DNN N, an input $\vec{x}$, restart times n, number of steps per time m, step size α, L_∞ norm distance threshold ϵ
output: adversarial example $\vec{x}'$ or `NULL`

1 **for** $i \leftarrow 1$ **to** n **do**
2 Generate a Gaussian noise $\vec{\delta}$ such that $\|\vec{\delta}\|_\infty \leq \epsilon$;
3 $\vec{x}' \leftarrow \vec{x} + \vec{\delta}$; /* Create a randomized seed */
4 **for** $j \leftarrow 1$ **to** m **do**
5 $y \leftarrow N_c(\vec{x}')$; /* Get the model output */
6 **if** $y \neq N_c(\vec{x})$ **then**
7 **return** $\vec{x}'$; /* Find an adversarial example */
8 **else**
9 $\nabla_{\vec{x}} \leftarrow$ `back_propagate`$(N, J(\vec{x}', N_c(\vec{x})))$; /* Get gradient */
10 $\vec{x}' \leftarrow \texttt{clip}_{\epsilon,\vec{x}}(\vec{x}' + \alpha \times \text{sign}(\nabla_{\vec{x}}))$; /* Compute a new sample */
11 **return** `NULL`;

References

1. Apollo: an open, reliable and secure software platform for autonomous driving systems. http://apollo.auto (2018)
2. Ashok, P., Hashemi, V., Kretínský, J., Mohr, S.: Deepabstract: neural network abstraction for accelerating verification. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis, pp. 92–107 (2020)

3. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S.D., Thakur, A.V.: Proofs from tests. IEEE Trans. Softw. Eng. **36**, 495–508 (2010)
4. Bu, L., Zhao, Z., Duan, Y., Song, F.: Taking care of the discretization problem: a comprehensive study of the discretization problem and a black-box adversarial attack in discrete integer domain. IEEE Trans. Dependable Secur. Comput. **19**(5), 3200–3217 (2022)
5. Bunel, R., Lu, J., Turkaslan, I., Torr, P.H.S., Kohli, P., Kumar, M.P.: Branch and bound for piecewise linear neural network verification. J. Mach. Learn. Res. **21**, 1–39 (2020)
6. Carlini, N., Wagner, D.A.: Towards evaluating the robustness of neural networks. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 39–57 (2017)
7. Chen, G., et al.: Who is real Bob? adversarial attacks on speaker recognition systems. In: Proceedings of the 42nd IEEE Symposium on Security and Privacy, pp. 694–711 (2021)
8. Chen, G., Zhao, Z., Song, F., Chen, S., Fan, L., Liu, Y.: AS2T: Arbitrary source-to-target adversarial attack on speaker recognition systems. IEEE Trans. Dependable Secur. Comput., 1–17 (2022)
9. Chen, P., Zhang, H., Sharma, Y., Yi, J., Hsieh, C.: ZOO: zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security. pp. 15–26 (2017)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
11. Czech, M., Jakobs, M.C., Wehrheim, H.: Just test what you cannot verify. In: Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering, pp. 100–114 (2015)
12. Dimitrov, D.I., Singh, G., Gehr, T., Vechev, M.: Provably robust adversarial examples. In: Proceedings of the International Conference on Learning Representations (2021)
13. Dong, Y., et al.: An empirical study on correlation between coverage and robustness for deep neural networks. In: Proceedings of the 25th International Conference on Engineering of Complex Computer Systems, pp. 73–82 (2020)
14. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: Proceedings of the 10th International Symposium NASA Formal Methods, pp. 121–138 (2018)
15. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis, pp. 269–286 (2017)
16. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Proceedings of the 32nd International Conference on Computer Aided Verification (2020)
17. Fischer, M., Sprecher, C., Dimitrov, D.I., Singh, G., Vechev, M.T.: Shared certificates for neural network verification. In: Proceedings of the 34th International Conference on Computer Aided Verification, pp. 127–148 (2022)
18. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy, pp. 3–18 (2018)
19. Gokulanathan, S., Feldsher, A., Malca, A., Barrett, C.W., Katz, G.: Simplifying neural networks using formal verification. In: Proceedings of the 12th International Symposium NASA Formal Methods, pp. 85–93 (2020)

20. Goodfellow, I., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: Proceedings of the 3th International Conference on Learning Representations (2015)
21. Goubault, E., Palumby, S., Putot, S., Rustenholz, L., Sankaranarayanan, S.: Static analysis of ReLU neural networks with tropical polyhedra. In: Proceedings of the 28th International Symposium Static Analysis, pp. 166–190 (2021)
22. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 117–127 (2006)
23. Guo, X., Wan, W., Zhang, Z., Zhang, M., Song, F., Wen, X.: Eager falsification for accelerating robustness verification of deep neural networks. In: Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering, pp. 345–356 (2021)
24. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification, pp. 3–29 (2017)
25. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: a framework for combining static and dynamic analysis. In: Proceedings of the International Workshop on Dynamic Analysis (2006)
26. Jia, K., Rinard, M.C.: Verifying low-dimensional input neural networks via input quantization. In: Proceedings of the 28th International Symposium Static Analysis, pp. 206–214 (2021)
27. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J.: Policy compression for aircraft collision avoidance systems. In: IEEE/AIAA Digital Avionics Systems Conference (2016)
28. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient smt solver for verifying deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
29. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Proceedings of the International Conference on Computer Aided Verification, pp. 443–452 (2019)
30. Kim, J., Feldt, R., Yoo, S.: Guiding deep learning system testing using surprise adequacy. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering, pp. 1039–1049 (2019)
31. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
32. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. In: Proceedings of International Conference on Learning Representations (2017)
33. Li, J., Liu, J., Yang, P., Chen, L., Huang, X., Zhang, L.: Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In: Proceedings of the 26th International Symposium Static Analysis, pp. 296–319 (2019)
34. Lin, W., Yang, Z., Chen, X., Zhao, Q., Li, X., Liu, Z., He, J.: Robustness verification of classification deep neural networks via linear programming. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 11418–11427 (2019)
35. Liu, W., Song, F., Zhang, T., Wang, J.: Verifying ReLU neural networks from a model checking perspective. J. Comput. Sci. Technol. **35**(6), 1365–1381 (2020)

36. Ma, L., et al.: DeepGauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 120–131 (2018)
37. Ma, L., et al.: DeepMutation: mutation testing of deep learning systems. In: Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering, pp. 100–111 (2018)
38. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: Proceedings of the International Conference on Learning Representations (2018)
39. Mangal, R., Sarangmath, K., Nori, A.V., Orso, A.: Probabilistic Lipschitz analysis of neural networks. In: Proceedings of the 27th International Symposium Static Analysis, pp. 274–309 (2020)
40. Mazzucato, D., Urban, C.: Reduced products of abstract domains for fairness certification of neural networks. In: Proceedings of the 28th International Symposium Static Analysis, pp. 308–322 (2021)
41. Moosavi-Dezfooli, S., Fawzi, A., Frossard, P.: DeepFool: a simple and accurate method to fool deep neural networks. In: Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition, pp. 2574–2582 (2016)
42. Müller, M.N., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.T.: PRIMA: general and precise neural network certification via scalable convex hull approximations. Proc. ACM Program. Lang. 6(POPL), 1–33 (2022)
43. Naik, M., Yang, H., Castelnuovo, G., Sagiv, M.: Abstractions from tests. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 373–386 (2012)
44. Ostrovsky, M., Barrett, C.W., Katz, G.: An abstraction-refinement approach to verifying convolutional neural networks. CoRR abs/2201.01978 (2022)
45. Papernot, N., McDaniel, P.D., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: Proceedings of IEEE European Symposium on Security and Privacy, pp. 372–387 (2016)
46. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 1–18 (2017)
47. Prabhakar, P., Afzal, Z.R.: Abstraction based output range analysis for neural networks. In: Proceedings of the Annual Conference on Neural Information Processing Systems (2019)
48. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: Proceedings of the 22nd International Conference on Computer Aided Verification (2010)
49. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: Proceedings of the Annual Conference on Neural Information Processing Systems, pp. 10825–10836 (2018)
50. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. 3(POPL), 41:1–41:30 (2019)
51. Song, F., Lei, Y., Chen, S., Fan, L., Liu, Y.: Advanced evasion attacks and mitigations on practical ml-based phishing website classifiers. Int. J. Intell. Syst. **36**(9), 5210–5240 (2021)
52. Sotoudeh, M., Thakur, A.V.: Abstract neural networks. In: Proceedings of the 27th International Symposium Static Analysis, pp. 65–88 (2020)
53. Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineerin, pp. 109–119 (2018)

54. Szegedy, C., et al.: Intriguing properties of neural networks. In: Proceedings of the 2nd International Conference on Learning Representations (2014)
55. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. In: Proceedings of the 7th International Conference on Learning Representations (2019)
56. Tran, H., et al.: Star-based reachability analysis of deep neural networks. In: Proceedings of the 3rd World Congress on Formal Methods, pp. 670–686 (2019)
57. Urban, C., Christakis, M., Wüstholz, V., Zhang, F.: Perfectly parallel fairness certification of neural networks. Proc. ACM Program. Lang. 4(OOPSLA), 185:1–185:30 (2020)
58. VNN-COMP: 2nd international verification of neural networks competition. https://sites.google.com/view/vnn2021 (2021)
59. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Proceedings of Annual Conference on Neural Information Processing Systems (2018)
60. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Security Symposium on Security, pp. 1599–1614 (2018)
61. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: Proceedings of the 35th International Conference on Machine Learning, pp. 5283–5292 (2018)
62. Yang, P., Li, J., Liu, J., Huang, C., Li, R., Chen, L., Huang, X., Zhang, L.: Enhancing robustness verification for deep neural networks via symbolic propagation. Formal Aspects Comput. **33**(3), 407–435 (2021)
63. Yang, P., et al.: Improving neural network verification through spurious region guided refinement. In: Proceedings of 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 389–408 (2021)
64. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 145–156 (2006)
65. Zhang, H., Shinn, M., Gupta, A., Gurfinkel, A., Le, N., Narodytska, N.: Verification of recurrent neural networks for cognitive tasks via reachability analysis. In: Proceedings of 24th European Conference on Artificial Intelligence, pp. 1690–1697 (2020)
66. Zhang, H., et al.: Alpha-Beta-CROWN: a fast and scalable neural network verifier with efficient bound propagation (2021). https://github.com/huanzhang12/alpha-beta-CROWN
67. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: BDD4BNN: a BDD-based quantitative analysis framework for binarized neural networks. In: Proceedings of the 33rd International Conference on Computer Aided Verification, pp. 175–200 (2021)
68. Zhang, Y., Zhao, Z., Chen, G., Song, F., Zhang, M., Chen, T.: QVIP: an ILP-based formal verification approach for quantized neural networks. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (2022)
69. Zhao, Z., Chen, G., Wang, J., Yang, Y., Song, F., Sun, J.: Attack as defense: characterizing adversarial examples using robustness. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 42–55 (2021)