Using Counterexamples to Improve Robustness Verification in Neural Networks

Mohammad Afzal[1,2][0000−0002−6173−3959], Ashutosh Gupta[1][0009−0003−7755−2006], and S. Akshay[1][0000−0002−2471−5997]

[1] Indian Institute of Technology Bombay, Mumbai, India
[2] TCS Research, Pune, India

Abstract Given the pervasive use of neural networks in safety-critical systems it is important to ensure that they are robust. Recent research has focused on the question of verifying whether networks do not alter their behavior under small perturbations in inputs. Most successful methods are based on the paradigm of branch-and-bound, an abstraction-refinement technique. However, despite tremendous improvements in the last five years, there are still several benchmarks where these methods fail. One reason for this is that many methods use off-the-shelf methods to find the cause of imprecisions.
In this paper, our goal is to develop an approach to identify the precise source of imprecision during abstraction. We present a *novel* counterexample guided approach that can be applied alongside many abstraction techniques. As a specific case, we implement our technique on top of a basic abstraction framework provided by the tool DeepPoly and demonstrate how we can remove imprecisions in a targetted manner. This allows us to go past DeepPoly's performance as well as outperform other refinement approaches in literature. Surprisingly, we are also able to verify several benchmark instances on which all leading tools fail.

Keywords: Neural Networks · Abstraction Refinement · Robustness verification · Counterexample guided approaches

1 Introduction

Neural networks are being increasingly used in safety-critical systems such as autonomous vehicles, medical diagnosis, and speech recognition [1,2,3]. It is important not only that such systems behave correctly in theory but also that they are robust in practice. Unfortunately, it is often the case (see e.g., Goodfellow [4]) that a slight change/perturbation in the input can often fool the neural networks into an error. Such errors can be hard to find/analyze/debug as these neural networks contain hundreds of thousands of non-linear nodes.

To address this problem, an entire line of research has emerged focussing on automatically proving (or disproving) the robustness of such networks. Since automatic verification of neural networks is NP-hard [5], researchers use approximations in their methods. Classically, we may divide the methods into two classes,

namely complete and incomplete. The methods [6,7,8,9,10,11,12,13,14,15,16] are complete. Since complete methods explore exact state space, they suffer from scalability issues on large-scale networks. On the other hand, abstraction based methods e.g., [17], [18], [19], [20], [21], [22], [23], [23] are sound and incomplete, because they over-approximate the state space, but they scale extremely well to large benchmarks. A representative method DeepPoly [24] maintains and propagates upper and lower bound constraints using the so-called triangle approximation (also see Section 3.1). This is also sometimes called bound-propagation. Unsurprisingly, DeepPoly and other abstraction based methods suffer from imprecision. Hence, the methods [25,26,27,28,29] refine the over-approximated state space to achieve completeness. In [25,26,29] the authors eliminate the spurious information (i.e., imprecision introduced by abstraction) by bisecting the input space on the guided dimension. In [28], which also works on top of DeepPoly [24], the authors remove the spurious region by conjuncting each neuron's constraints with the negation of the robustness property and using an MILP (mixed integer linear programming) optimizer Gurobi [30] to refine the bounds of neurons. Another work that refines DeepPoly is kPoly [31] which considers a group of neurons at once to generate the constraints and compute the bounds of neurons. One issue with all these approaches is that refinement is not guided by previous information/runs and hence they suffer from scalability issues.

In this paper, we consider the basic abstraction framework provided by DeepPoly and develop a novel refinement technique that is *counterexample guided*, i.e., we use counterexamples generated from imprecisions during abstraction to guide the refinement process. Our main contributions are the following:

- We introduce a new *maxsat-based* technique to find the cause of imprecision and spuriousness. Starting with an input where the abstraction does not get verified (we use a MILP solver to obtain this), we check whether the input generates a real counterexample of falsification of the property or if it is spurious, by executing the neural net. If it is a spurious counterexample, we identify the neuron or the set of neurons that caused it.
- We use these specially identified or *marked* neurons to split and refine. This ensures that, unlike earlier refinement methods, our method progresses at each iteration and eliminates spurious counterexamples.
- We adapt the existing refinement framework built on ideas from MILP-methods and implement this as a counterexample guided abstraction refinement algorithm on top of DeepPoly.
- We show that our technique outperforms to-the-best-of-our-knowledge all existing refinement strategies based on DeepPoly.
- We also identify a class of benchmarks coming from adversarially trained networks, where these state-of-the-art tools do not work well, because of the ineffectiveness of certain preprocessing steps (e.g., PGD attack [32])
- Incorporating such preprocessing techniques in our tool allows us to obtain a significant improvement in the overall performance of our tool. Our implementation is able to verify several benchmarks that are beyond the reach of state-of-the-art tools such as $\alpha\beta$-CROWN [33] and Oval [34].

Related work. A different but very successful line of research has been to revisit the branching heuristics for refinement and use ideas from convex optimization instead of linear or mixed integer linear programming. Starting from a slightly different abstraction/bound propagation method CROWN [23], the work in [14] adopts this approach. This is amenable to parallelizing and hence good for GPU implementations [15]. Recently, techniques based on cutting planes have been used to further improve the refinement analysis, solving more benchmarks at the cost of speed [16]. The success of this line of research can be seen by the fact that the state-of-the-art tool $\alpha\beta$-CROWN [33] (a highly optimized solver that uses a collection of different parametrized algorithms) has won the 2nd and 3rd international Verification of Neural Networks Competition (VNNCOMP'21,'22) in a field of leading tools for robustness verification. OVAL [34], another leading tool, uses multiple optimized techniques, which at its core perform an effective branch and bound on the RELU activation function. They attempt to compute the rough estimate on the improvement on objective function by splitting a particular neuron, and split neurons with the highest estimated improvement. Finally, in MARABOU [11], another leading complete tool, the authors search for an assignment that satisfies the constraints. They treat the non-linear constraints lazily with the hope that some non-linear constraints will not be needed to satisfy. Despite the enormous progress made by these tools in just the last 2-3 years, there still many benchmarks that are out of their reach. Our focus in this paper is orthogonal to these approaches, as we use counterexamples to guide the identification the source of imprecision. In our experiments in Section 5, we show that this allows us to solve many benchmarks which these cannot. Integrating our counterexample guided approach for imprecision-identification with these optimized tools (e.g., $\alpha\beta$-CROWN's branch and bound strategy) would be the next step towards wider coverage and performance. The constraints solved by $\alpha\beta$-CROWN, which also uses branch-and-bound, are from a dual space, and it is a priori unclear how to derive our maxSAT query from the failure of a run.

As mentioned earlier, DEEPSRGR [28] and KPOLY [31] use refinement of DEEPPOLY, but they are not counterexample guided. Elboher et al [27] does perform counterexample guided abstraction refinement, but their abstraction technique is orthogonal to DEEPPOLY. They reduce the network size by merging similar neurons with over-approximation, while DEEPPOLY maintains the linear constraints for each neuron without changing the structure of the network. These approaches also suffer from scalability issues on large-scale networks.

Structure of the paper. We start with a motivating example in the next Section 2. We define the notions and definitions in Section 3. Section 4 contains the algorithmic procedure of our approach as well as proofs of progress and termination. Section 5 contains our experimental results and we conclude in Section 6.

2 A Motivating Example

Consider the neural network depicted in Figure 1, which comprises one input layer, one hidden layer, and one output layer. The hidden layer is divided into

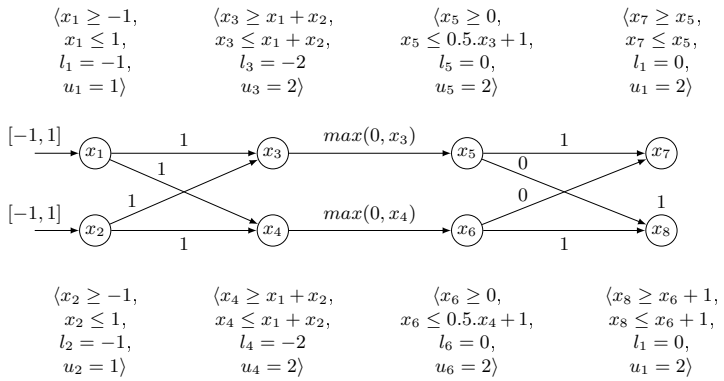


Figure 1: Hypothetical example of neural network

two sub-layers: Affine and ReLU, resulting in a total of four layers shown in Figure 1. Every layer contains two neurons. The neuron x_8 has a bias of 1, and all the other neurons have a bias of 0. Our goal is to verify for all input $x_1, x_2 \in [-1, 1]$ the outputs satisfy $x_7 \leq x_8$. Our approach extends DeepPoly [24]. DeepPoly maintains one upper and one lower constraint and an upper and lower bound for each neuron. For a neuron of the affine layer, the upper and lower constraint is the same, which is the weighted sum of the input neurons i.e. x_3's upper and lower constraint is $x_1 + x_2$. For an activation neuron, the upper and lower expression is computed using triangle approximation [24], which is briefly explained in Section 3.1. To verify the property $x_7 \leq x_8$, DeepPoly creates a new expression $x_9 = x_7 - x_8$ and computes the upper bound of x_9. The upper bound of x_9 should not be greater than 0. DeepPoly computes the upper bound of x_9 by back substituting the expression of x_7 and x_8 from the previous layer. They continue back substituting until only input layer variables are left. The process of back substitution is shown in Equation 1. After back substitution, the upper bound of x_9 is computed as 1, which is greater than 0, hence, the DeepPoly fails to verify the property.

$$\begin{aligned} x_9 &\leq x_7 - x_8 \\ x_9 &\leq x_5 - x_6 - 1 \\ x_9 &\leq 0.5x_3 + 1 - 1 \\ x_9 &\leq 0.5(x_1 + x_2) \\ x_9 &\leq 1 \end{aligned} \tag{1}$$

$$\begin{aligned} -1 \leq x_1 \leq 1 \qquad & -1 \leq x_2 \leq 1 \\ x_1 + x_2 \leq x_3 \leq x_1 + x_2 \qquad & x_1 + x_2 \leq x_4 \leq x_1 + x_2 \\ 0 \leq x_5 \leq 0.5x_3 + 1 \qquad & 0 \leq x_6 \leq 0.5x_4 + 1 \\ x_5 \leq x_7 \leq x_5 \qquad & x_6 + 1 \leq x_8 \leq x_6 + 1 \\ x_7 > x_8 \text{ (negation of property)} & \end{aligned} \tag{2}$$

There are two main reasons for the failure of DeepPoly. First, it cannot maintain the complete correlation between the neurons. In this example, neurons x_3 and x_4 have the same expression $x_1 + x_2$, so they always get the same value. However, in the DeepPoly analysis process, it may fail to get the same value. Second, it uses triangle approximation on ReLU neurons. We take the conjunction of upper and lower expressions of each neuron with the negation of the property as shown in Equation 2, and use the MILP solver to check satisfiability, thus addressing the first issue. The second issue can be resolved either by splitting the bound at zero of the affine node or by using the exact encoding (Equation 6) instead of triangle approximation. But both solutions increase the problem size exponentially in terms of ReLU neurons and this results in a huge blowup if we repair every neuron of the network.

So, the main hurdle toward efficiency is to find the set of important neurons (we call these *marked neurons*), and only repair these. For this, we crucially use the satisfying assignment obtained from the MILP solver. For instance, a possible satisfying assignment of Equation 2 is in Equation 3. We execute the neural network with the inputs $x_1 = 1, x_2 = 1$ and get the values on each neuron as shown in Equation 4. Then we observe that the output values $x'_7 = 2, x'_8 = 3$ satisfy the property, so, the input $x_1 = 1, x_2 = 1$ is a spurious counterexample. The question is to identify the neuron whose abstraction lead to this imprecision.

$$x_1 = 1, x_2 = 1, x_3 = 2, x_4 = 2, x_5 = 2, x_6 = 0, x_7 = 2, x_8 = 1 \tag{3}$$

$$x'_1 = 1, x'_2 = 1, x'_3 = 2, x'_4 = 2, x'_5 = 2, x'_6 = 2, x'_7 = 2, x'_8 = 3 \tag{4}$$

Maxsat based approach to identify marked neurons

To identify the neurons whose abstraction leads to imprecision, let us refer to Figure 2. In the figure, p_i represents the abstract constraint space in layer l_i, while the solid black line denotes the spurious counterexample depicted in Equation 3. On the other hand, the dashed green line represents the exact execution of the input point of the spurious counterexample, as denoted by Equation 4.

The objective is to make the solid black line as close as possible to the dashed green line from the first layer to the last layer while keeping the first and last points the same, i.e., $x_1 = 1$, $x_2 = 1$, and $x_7 = 2$, $x_8 = 1$. The closest line to achieving this goal is represented by the dotted blue line, which is also the abstract execution but exhibits the highest closeness to the exact execution of the spurious counterexample. In this context, v_i refers to the vector of values of neurons in layer l_i of the solid black line, while v'_i and v''_i represent the vectors of the dashed green and dotted blue lines, respectively.

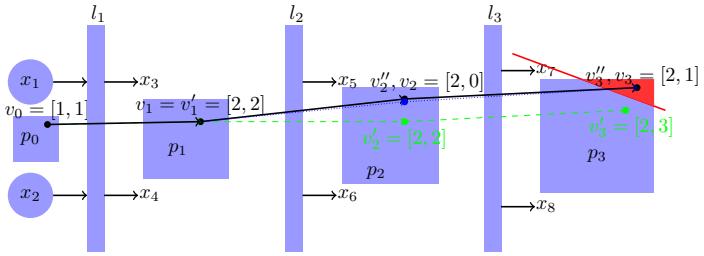


Figure 2: Pictorial representation of our approach on example in figure 1

The green and black points are the same for the input layer, i.e., $[1, 1]$. On the first affine layer, l_1 also, the black point v_1 is the same as the green point v_1' since the affine layer does not introduce any spurious information. For l_2, we try to make v_2'' close to v_2', such that v_2'' reaches to the v_3. We do that by encoding them as soft constraints (i.e., $\{x_5 = 2, x_6 = 2\}$) while maintaining that the rest of the hard constraints are satisfied (see Equation 5) e.g., input points $v_0 = v_0''$ and output points $v_3 = v_3''$ remain same. We mark the neurons of the layer where the dotted blue line starts diverging from the dashed green line, i.e., l_2. The divergence we find by the MaxSat query. If MaxSat returns all the soft constraints as satisfied, it means the blue point becomes equal to the green point. If MaxSat returns partial soft constraints as satisfied, we mark the neurons whose soft constraints are not satisfied. In our example, MaxSat returns soft constraints $\{x_5 = 2\}$ as satisfied, it means soft constraint of x_6 could not satisfied, so, we mark x_6. The dotted blue and solid black lines are the same for our motivating example since it contains only one ReLU layer. However, in general, it may or may not be the same. We optimize the dotted blue line to be close to the dashed green line while also resulting in as few marked neurons as possible.

$$
\begin{gathered}
x_1 = 1 \wedge x_2 = 1 \\
x_3 = x_1 + x_2 \wedge x_4 = x_1 + x_2 \\
0 \leq x_5 = 0.5x_3 + 1 \wedge 0 \leq x_6 \leq 0.5x_4 + 1 \\
x_7 = x_5 \wedge x_8 = x_6 + 1 \\
x_7 = 2 \wedge x_8 = 1
\end{gathered} \tag{5}
$$

Once we have x_6 as the marked neuron, we use an *MILP based approach*, and add the exact encoding of the marked neuron (x_6) in addition to the constraints in Equation (2) and check the satisfiability, now it becomes UNSAT, hence, the property verified (see Equation 6 for more details).

3 Preliminaries

In this section, we present some basic definitions, starting with a neural network.

Definition 1 *A neural network* $N = (Neurons, Layers, Edges, W, B, Type)$ *is a 6-tuple, where*

- $Neurons$ *is the set of neurons in* N,
- $Layers = \{l_0, ..., l_k\}$ *is an indexed partition of* $Neurons$,
- $Edges \subseteq \bigcup_{i=1}^{k} l_{i-1} \times l_i$ *is a set of edges linking neurons on consecutive layers,*
- $W : Edges \mapsto \mathbb{R}$ *is a weight function on edges,*
- $B : Neurons \mapsto \mathbb{R}$ *is a bias function on neurons,*
- $Type : Layers \mapsto \{\text{AFFINE}, \text{RELU}\}$ *defines type of neurons on each layer.*

A neural network is a collection of layers $l_0, l_1, l_2, ...l_k$, where k represents the number of layers. Each layer contains neurons that are also indexed, with n_{ij} denoting the jth neuron of layer l_i. We call l_0 and l_k the *input* and *output layers* respectively, and all other layers as *hidden layers*. In our presentation, we assume separate layers for the activation functions. Though there are different kinds of activations, we focus only on RELU, hence each layer can either be AFFINE or RELU layer. The definition of W and B applies only to the AFFINE layer. Without loss of generality, we assume that the output layer is an AFFINE layer (we can always append an identity AFFINE layer), and layers $l_1, l_3, l_5, ..., l_k$ to be the AFFINE layers, layers $l_2, l_4, l_6, ..., l_{k-1}$ to be the RELU layers. If $Type_i =$ RELU, then $|l_{i-1}| = |l_i|$. We extend the weight function from edges to layers using matrix $W_i \in \mathbb{R}^{|l_i| \times |l_{i-1}|}$ that represents the weight for layer l_i, s.t.,

$$W_i[t_1, t_2] = \begin{cases} W(e) & e = (n_{(i-1)t_2}, n_{it_1}) \in Edges, \\ 0 & \text{otherwise.} \end{cases}$$

We also write matrix $B_i \in \mathbb{R}^{|l_i| \times 1}$ to denote the bias matrix for layer l_i. The entry $B_i[t, 0] = B(n_{it})$, where $n_{it} \in Neurons$.

To define the semantics of N, we will use vectors $val_i = [val_{i1}, val_{i2}, ...val_{i|l_i|}]$ that represent the values of each neuron in the layer l_i. Let f_i be a function that computes the output vector of values at layer i using the values at layer $i-1$ as $val_i = f_i(val_{i-1})$. For each type the layer the functions are defined as follows: if $Type_i =$ AFFINE, then $f_i(val_{i-1}) = W_i * val_{i-1} + B_i$; if $Type_i =$ RELU, then $f_i(val_{i-1})_j = max(val_{(i-1)j}, 0)$. Then, the semantics of a neural network N is a function (we abuse notation and also denote this function as N) which takes an input, an $|l_0|$-dimensional vector of reals and gives as output an $|l_k|$-dimensional vector of reals, as a composition of functions $f_k \circ ... \circ f_1$. Thus, for an input $v \in \mathbb{R}^{|l_0|}$, we write its value computed by N at layer i as $val_i^v = f_i \circ ... \circ f_1(v)$.

Let us define $LinExpr = \{w_0 + \sum_i w_i x_i | w_i \in \mathbb{R} \text{ and } x_i \text{ is a real variable}\}$ and $LinConstr = \{expr \text{ op } 0 | expr \in LinExpr \wedge op \in \{\leq, =\}\}$. A *predicate* is a Boolean combination of $LinConstr$. We use real variable x_{ij} to represent values of n_{ij} in the predicates. Let P and Q be predicates over input and output layers respectively. A *verification query* is a triple $\langle N, P, Q \rangle$. We need to prove that for each input v, if $v \models P$, $N(v) \models Q$. We assume P has the form $\bigwedge_{i=1}^{|l_0|} lb_{0i} \leq x_{0i} \leq ub_{0i}$, where lb_{0i}, ub_{0i} are lower and upper bounds respectively for a neuron n_{0i}.

3.1 DeepPoly

We develop our abstract refinement approaches on top of abstraction based method DEEPPOLY [24], which uses a combination of well-understood polyhe-

dra [35] and box [36] abstract domain. The abstraction maintains upper and lower linear expressions as well as upper and lower bounds for each neuron. The variables appearing in upper and lower expressions are only from the predecessor layer. Formally, we define the abstraction as follows.

Definition 2 *For a neuron n, an abstract constraint $A(n) = (lb, ub, lexpr, uexpr)$ is a tuple, where $lb \in \mathbb{R}$ is lower bound on the value of n, $ub \in \mathbb{R}$ is the upper bound on the value of n, $lexpr \in LinExpr$ is the expression for the lower bound, and $uexpr \in LinExpr$ is the expression for the upper bound.*

In DeepPoly, we compute the abstraction A as follows.

- If $Type_i = \text{Affine}$, we set $A(x_{ij}).lexpr := A(x_{ij}).uexpr := \sum_{t=1}^{|l_{i-1}|} W_i[j,t] * x_{(i-1)t} + B_i[j,0]$. We compute $A(x_{ij}).lb$ and $A(x_{ij}).ub$ by back substituting the variables in $A(x_{ij}).lexpr$ and $A(x_{ij}).uexpr$ respectively up to input layer. Since P of the verification query has lower and upper bounds of the input layer, we can compute the bounds for x_{ij}. Consider the neuron x_3 in Figure 1. Both its upper and lower constraints are same, represented by the expression $x_1 + x_2$. To compute the upper bound of x_3, we substitute the upper bounds of x_1 and x_2, which are both 1. Consequently, the upper bound is calculated as $2(1 + 1)$. Similarly, for the lower bound of x_3, we substitute the lower bounds of x_1 and x_2, which are both -1. Thus, the lower bound is computed as $-2(-1 + -1)$.
- If $Type_i = \text{ReLU}$ and $y = \text{ReLU}(x)$, where x is a neuron in l_{i-1} and y is a neuron in l_i, we consider the following three cases:
  1. If $A(x).lb \geq 0$ then ReLU is in active phase and $A(y).lexpr := A(y).uexpr := x$, and $A(y).lb := A(x).lb$ and $A(y).ub := A(x).ub$
  2. If $A(x).ub \leq 0$ then ReLU is in passive phase and $A(y).lexpr := A(y).uexpr := 0$, and $A(y).lb := A(y).ub := 0$.
  3. If $A(x).lb < 0$ and $A(x).ub > 0$, the behavior of ReLU is uncertain, and we need to apply over-approximation. We set $A(y).uexpr := u(x - l)/(u - l)$, where $u = A(x).ub$ and $l = A(x).lb$. And $A(y).lexpr := \lambda.x$, where $\lambda \in \{0, 1\}$. We can choose any value of λ dynamically. We compute $A(y).lb$ and $A(y).ub$ by doing the back-subtitution similar to the Affine layer's neuron. Consider the neuron x_5 in Figure 1, whose input is x_3. Since x_3's upper bound is positive and lower bound is negative, the behavior of x_5 becomes uncertain. The upper expression of x_5 is computed using the above method as $0.5 * x_3 + 1$. By backsubstituting the upper expression of x_3, it becomes $0.5(x_1 + x_2) + 1$. Using the upper bounds of x_1 and x_2, x_5's upper bound is computed as 2. On the other hand, the lower expression of x_5 remains 0, by taking the value of λ as 0.

The constraints for an Affine neuron are exact because it is just an Affine transformation of input neurons. The constraints for a ReLU neuron are also exact if the ReLU is either in the active or passive phase. The constraints for ReLU are over-approximated if the behavior of ReLU is uncertain. Although we

may compute exact constraints for this case, but the constraints will be arbitrary polyhedron, which are expensive to compute. The DeepPoly abstraction finds a balance between precision and efficiency.

For the verification query $\langle N, P, Q\rangle$, we check if $\neg Q \wedge \bigwedge_{j=1}^{|l_k|} lb_{kj} \leq x_{kj} \leq ub_{kj}$ are satisfiable. If the formula is unsatisfied then we have proven the query successfully. Otherwise, DeepPoly fails to prove the query.

3.2 Solver

In our algorithms, we use two major calls checkSAT and maxSat. The function checkSAT in Algorithm 2, takes a quantifier-free formula as input and returns SAT or UNSAT. The function maxSat in the Algorithm 3 takes two arguments as input hardConstr and softConstr. The hardConstr is a Boolean formula of constraints, and softConstr is a set of constraints. The function maxSat satisfies the maximum number of constraints in softConstr while satisfying the hardConstr. The function maxSat returns SAT with the set of constraints satisfied in softConstr, or returns UNSAT if hardConstr fails to satisfy. We are using Gurobi(v9.1) [30] to implement both checkSAT and maxSat functions. Furthermore, Algorithm 2 includes a function called getModel, which serves as a mapping from variables to satisfying values. The getModel function is utilized in cases where checkSAT returns SAT to retrieve the satisfying assignment for the variables.

4 Algorithm

In this section, we present our method to refine DeepPoly. DeepPoly is a sound and incomplete technique because it does over-approximations. If DeepPoly verifies the property then the property is guaranteed to be verified, otherwise, its result is unknown. We overcome this limitation by using a CEGAR-like technique, which is complete. In our refinement approach, we mark some ReLU neurons to have exact behavior on top of DeepPoly constraints, similar to the strategy of refinement in the most complete state-of-the-art techniques [14,15]. We add the encoding of the exact behavior to the DeepPoly constraints and use an MILP solver on the extended constraints to check if the extra constraints rule out all spurious counterexamples. The calls to MILP solvers are expensive, therefore we use the spurious counterexamples discovered to identify as small as possible set of marked neurons which suffice to be repaired.

4.1 The top level algorithm

We start by describing Algorithm 1, where we present the top-level flow of our approach. The algorithm takes a verification query $\langle N, P, Q\rangle$ as input, where N is a neural network and P, Q are predicates over input and output layers respectively, and returns success if the verification is successful. Otherwise it returns a counterexample to the query. The algorithm uses supporting algorithms

Algorithm 1 A CEGAR based approach of neural network verification

Input: A verification problem $\langle N, P, Q\rangle$
Output: Verified or Counterexample

1: cex, $bounds = preprocessing(N, P, Q)$
2: **if** cex is not None **then**
3: **return** Failed(cex) ▷ cex is a counter example
4: $A :=$ DeepPoly$(N, P, bounds)$ ▷ use DeepPoly to generate abstract constraints.
5: $marked := \{\}$
6: **while** True **do**
7: result = isVerified$(\langle N, P, Q\rangle, A, marked)$
8: **if** result = CEX$(v_0, v_1 ... v_k)$ **then**
9: **if** $N(v_0) \models \neg Q$ **then**
10: **return** Failed(v_0) ▷ v_0 is a counter example
11: **else**
12: $markedNt :=$ getMarkedNeurons$(N, A, marked, v_0, v_1 ... v_k)$
13: $marked := marked \cup markedNt$
14: **else**
15: **return** verified

getMarkedNeurons and isVerified (described subsequently) to get more marked neurons to refine and check the validity of the verification query after refinement.

The first line of the algorithm performs preprocessing steps similar to state-of-the-art tools (e.g. $\alpha\beta$-CROWN). These preprocessing steps are optional and are explained in more detail in Section 5, where they are used to compare our results with those of state-of-the-art tools. The fourth line of Algorithm 1 generates all the abstract constraints by using DeepPoly, as described in Section 3.1. For a node $n_{ij} \in N.neurons$, the abstract constraints consist of the lower and upper constraints as well as the lower and upper bounds. Let $A.lc_i = \bigwedge_{j=1}^{|l_i|} A(n_{ij}).lexpr \leq x_{ij} \leq A(n_{ij}).uexpr$, which is a conjunction of upper and lower constraints of each neuron of layer l_i with respect to abstract constraint A. The $lexpr$ and $uexpr$ for any neuron of a layer contain variables only from the previous layer's neurons, hence $A.lc_i$ contains the variables from layers l_{i-1} and l_i. If the preprocessing steps in line 1 are applied, then DeepPoly generates the $lexpr$ and $uexpr$ for ReLU neurons as per the triangle approximation. In this case, we may return a counter-example and stop or use these bounds without performing any back-substitution.

At line 5, we initialize the variable $marked$ to the empty set of neurons. At the next line, we iterate in a while loop until either we verify the query or find a counterexample. At line 7, we call isVerified with the verification query, abstraction A, and the set of marked neurons. In this verification step, the behavior of the marked neurons is encoded exactly, as detailed in Section 4.2. The call either returns that the query is verified or returns an abstract counterexample, which is defined as follows.

Algorithm 2 Verify $\langle N, P, Q\rangle$ with abstraction A

Name: isVerified
Input: Verification query $\langle N, P, Q\rangle$, abstract constraints A, $marked \subseteq N.neurons$
Output: verified or an abstract counterexample.

1: $constr := P \wedge (\bigwedge_{i=1}^{k} A.lc_i) \wedge \neg Q$
2: $constr := constr \wedge (\bigwedge_{n \in marked} exactConstr(n))$ ▷ as in Equation 6
3: isSat = checkSat(constr)
4: **if** isSat **then**
5: m := getModel(constr)
6: **return** CEX$(m(x_0), ..., m(x_k))$ ▷ Abstract counter example where x_i is a vector of variables in layer l_i
7: **else**
8: **return** verified

Definition 3 *A sequence of value vectors $v_0, v_1, ..., v_k$ is an* abstract execution *of abstract constraint A if $v_0 \models lc_0$ and $v_{i-1}, v_i \models A.lc_i$ for each $i \in [1, k]$. An abstract execution $v_0, ..., v_k$ is an* abstract counterexample *if $v_k \models \neg Q$.*

If these algorithms return verified, we are done, otherwise we analyze the abstract counterexample $CEX(v_0, ..., v_k)$. The abstract counterexample $CEX(v_0, ..., v_k)$ may or may not be a real counterexample, so, we first check at line 8, if executing the neural network N on input v_0 violates the predicate Q. If yes, we report input v_0 as a counterexample, for which the verification query is not true. Otherwise, we declare the abstract counterexample to be spurious. We call getMarkedNeurons to analyze the counterexample and return the cause of spuriousness, which is a set of neurons *markedNt*. We add the new set *markedNt* to the old set *marked* and iterate our loop with the new set of marked neurons. Now let us present isVerified and getMarkedNeurons in detail.

4.2 Verifying query under marked neurons

In Algorithm 2, we present the implementation of isVerified, which takes the verification query, the DeepPoly abstraction A, and a set of marked neurons as input. At line 1, we construct constraints *contr* that encodes the executions that satisfy abstraction A at every step. At line 2, we also include constraints in *constr* that encodes the exact behavior of the marked neurons. The following is the encoding of the exact behavior [37] of neuron n_{ij}.

$$\begin{aligned} exactConstr(n_{ij}) := x_{(i-1)j} \leq x_{ij} \leq x_{(i-1)j} - A(n_{(i-1)j}).lb * (1-a) \ \wedge \\ 0 \leq x_{ij} \leq A(n_{(i-1)j}).ub * a \wedge a \in \{0, 1\} \end{aligned} \tag{6}$$

where a is a fresh variable for each neuron.

At line 3, we call a solver to find a satisfying assignment of the constraints. If *constr* is satisfiable, we get a model m. From the model m, we extract an abstract counterexample and return it. If *constr* is unsatisfiable, we return that the query is verified.

Algorithm 3 Marked neurons from counterexample

Name: getMarkedNeurons
Input: Neural network N, DeepPoly abstraction A, $marked \subseteq N.neurons$, and abstract counterexample $(v_0, v_1 ... v_k)$
Output: New marked neurons.

Let $val_{ij}^{v_0}$ be the value of n_{ij}, when v_0 is input of N.
for $i = 1$ to k **do** ▷ inputLayer excluded
  if l_i is ReLU layer **then**
    $constr := \bigwedge_{t=i}^{k} A.lc_t$
    $constr := constr \wedge (\bigwedge_{n \in marked} exactConstr(n))$ ▷ as in Equation 6
    $constr := constr \wedge \bigwedge_{j=1}^{|l_{i-1}|} (x_{(i-1)j} = val_{(i-1)j}^{v_0})$
    $constr := constr \wedge \bigwedge_{j=1}^{|l_k|} (x_{kj} = v_{kj})$
    $softConstrs := \cup_{j=1}^{|l_i|} (x_{ij} = val_{ij}^{v_0})$
    $res, softsatSet = \text{maxSat}(constr, softConstrs)$ ▷ res always SAT
    $newMarked := \{n_{ij} | 1 \leq j \leq |l_i| \wedge (x_{ij} = val_i(j)) \notin softsatSet\}$
    if $newMarked$ is empty **then**
      continue
    else
      return $newMarked$

4.3 Maxsat based approach to find the marked neurons

In Algorithm 3, we present getMarkedNeurons which analyzes an abstract spurious counterexample. In our abstract constraints, we encode Affine neurons exactly, but over-approximate ReLU neurons. We identify a set of marked neurons whose exact encoding will eliminate the counterexample in the future analysis. As we defined earlier, let $val_i^{v_0}$ represent the value vector on layer l_i, if we execute the neural network on input v_0. Let us say $v_0, v_1, ..., v_k$ is an abstract spurious counterexample. We iteratively modify the counterexample such that its values coincides with $val_i^{v_0}$. Initially, $val_0^{v_0}$ is equal to v_0. Since we encode the affine layer exactly in $A.lc_i$, the following theorem follows.

Theorem 1. *Let $v_0, v_1, ...v_k$ be an abstract execution. For all $1 \leq i \leq k$, if $Type_i =$ Affine and $val_{i-1}^{v_0} = v_{i-1}$, then $val_i^{v_0} = v_i$.*

By the above theorem, v_1 and $val_1^{v_1}$ are also equal. The core idea of our algorithm is to find v_2' as close as possible to $val_2^{v_0}$, such that $v_0, v_1, v_2', ...v_{k-1}', v_k$ becomes an abstract spurious counterexample. We measure closeness by the number of elements of v_2' are equal to the corresponding element of vector $val_2^{v_0}$.

1. If v_2' is equal to $val_2^{v_0}$ then v_3' will also become equal to $val_3^{v_0}$ due to Theorem 1. Now we move on to the next ReLU layer l_4 and try to find the similar point v_4', such that $v_0, v_1, v_2, v_3, v_4''...v_{k-1}'', v_k$ is an abstract spurious counterexample. We repeat this process until the following case occurs.
2. If at some i, we can not make v_i' equal to $val_i^{v_0}$ then we collect the neurons whose values are different in v_i' and $val_i^{v_0}$. We call them marked neurons.

In the algorithm, the above description is implemented using MAXSAT solver. The loop at line 2 iterates over RELU layers. Line 4 builds the abstract constraints generated by DEEPPOLY from layer i onwards. Line 5 encodes the exact encoding of marked neurons, i.e. x_6 is identified as a marked neuron in our motivating example. Line 6 and 7 ensure layer l_{i-1}'s neurons have value equal to $val_{i-1}^{v_0}$, and the execution finishes at v_k. The first and last line of Equation 5 represents the constrains of line 6 and line 7, in our motivating example. At Line 8, we construct soft constraints, which encodes x_{ij} is equal to $val_i^{v_0}$. In our motivating example, the set $\{x_5 = 2, x_6 = 2\}$ represents the soft constraints. At line 9, we call MAXSAT solver. This call to MAXSAT solver will always find a satisfying assignment because our hard constraints are always satisfiable. The solver will also return a subset $softsatSet$ of the soft constraints. At line 10, we check which soft constraints are missing in $softsatSet$. The corresponding neurons are added in $newMarked$. If $newMarked$ is empty, we have managed to find a spurious abstract counterexample from $val_i^{v_0}$ and we go to the next layer. Otherwise, we return the new set of marked neurons.

4.4 Proofs of progress and termination

Our refinement strategy ensures progress, i.e., the spurious counterexample does not repeat in the future iterations of Algorithm 1. Let us suppose the algorithm GETMARKEDNEURONS gets the abstract spurious counterexample $v_0, v_1, ...v_k$ and returns marked neurons in some iteration of the while loop, say i^{th}-iteration. The call to MAXSAT at line 10 declares that $constr \wedge softsatSet$ is satisfiable. We can extract an abstract spurious counterexample from a model of $constr \wedge softsatSet$. Let m be the model. Let the abstract spurious counterexample be $cex = val_0^{v_0}, ..., val_{i-1}^{v_0}, v_i', ..., v_{k-1}', v_k$. Before the iteration i, cex follows the execution of N on input v_0. After the iteration i, we use the model to construct cex, i.e., $m(x_i) = v_i'$.

Lemma 1. *In the rest of run of Algorithm 1, i.e., future iterations of the while loop,* ISVERIFIED *will not return the abstract spurious counterexample cex again.*

Proof. For $n_{ij} \in newMarked$, the MAXSAT query ensures that $val_{ij}^{v_0} \neq v_{ij}'$. If we have the same counterexample again in the future then input of n_{ij} will be $val_{(i-1)j}^{v_0}$. Since we will have exact encoding for n_{ij}, the output will be $val_{ij}^{v_0}$, which contradicts the earlier inequality.

Next, we turn to termination of the algorithm. We have two lemmmas.

Lemma 2. *In every refinement iteration* GETMARKEDNEURONS *returns a non-empty set of marked neurons.*

Proof. By the definition of abstract spurious counterexample, $v_k \models \neg Q$. By the check at line 6 of Algorithm 1 $val_k^{v_0} \models Q$. If the set of returned new marked neurons is empty, $newMarked = \emptyset$ for each layer. Therefore, all the neurons in any layer l_i become equal to $val_i^{v_0}$, which implies v_k equals to $val_k^{v_0}$, but $v_k \models \neg Q$ and $val_k^{v_0} \models Q$, which is a contradiction.

Lemma 3. *In every refinement iteration* GETMARKEDNEURONS *returns marked neurons, which were not marked in previous iterations.*

Proof. We will show that if a neuron n_{ij} got marked in t^{th} iteration then n_{ij} will not be marked again in any iteration greater than t. Consider an iteration $t' > t$, if we get the marked neurons from layer other than l_i then n_{ij} can not be part of it because n_{ij} is in layer l_i. Consider the case where marked neurons are from the layer l_i in t'^{th} iteration. Since we have made n_{ij} exact in line 6 of Algorithm 3, its behavior while optimizing constraints will be same as the exact RELU. Moreover, $v'_{ij} = val_{ij}^{v_0}$, which implies the soft constraint for neuron n_{ij} will always be satisfied. Hence it will not occur as a marked neuron as per the criteria of new marked neurons in line 10 of Algorithm 3.

Lemmas 2 and 3 imply that in every iteration GETMARKEDNEURONS returns a nonempty set of unmarked neurons, which will now be marked. In worst case, the algorithm will mark all the neurons of the network, and encode them in the exact behavior. Thus, we conclude,

Theorem 2. *Algorithm 1 always terminates.*

5 Experiments

We have implemented our approach in a prototype and compared it to three types of approaches (i) DEEPPOLY [24] and its refinements KPOLY [31], DEEPSRGR [28], (ii) other cegar based approaches, and (iii) state-of-the-art tools $\alpha\beta$-CROWN [23,14,15,38,37], OVAL [39,34,40,41,41,42,43], and MARABOU [11]. Furthermore, we conducted a comparison of performance across different epsilon values for all the tools employed. We extended this analysis to focus specifically on adversarially trained networks and observed a significant improvement in performance. Moreover, we conducted a detailed comparison with $\alpha\beta$-CROWN, utilizing the same preprocessing steps employed by the $\alpha\beta$-CROWN tool.

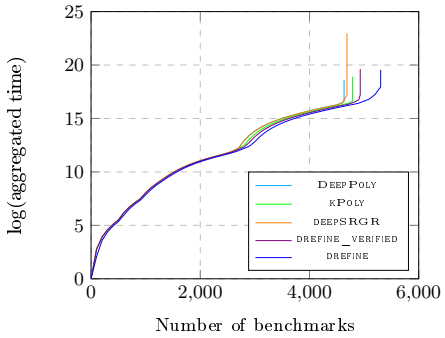
The tools $\alpha\beta$-CROWN and OVAL use a set/portfolio of different algorithms and optimizations. $\alpha\beta$-CROWN achieved the first rank consistently in both VNN-COMP'21 and VNN-COMP'22 (International Verification of Neural Networks Competitions) [3].We use the same configuration to run $\alpha\beta$-CROWN and OVAL, which these tools used in VNN-COMP.

Implementation. We have implemented our techniques in a tool, which we call DREFINE, in C++ programming language. The tool DREFINE is available at `https://github.com/afzalmohd/VeriNN/tree/atva2023`. Our approach relies on DEEPPOLY, so we also have implemented DEEPPOLY in C++. We are using

[3] We could not compare with VERINET, which is the 2nd and 3rd of VNNCOMP 2021 and VNNCOMP 2022 respectively, as we had difficulties with its external solver's (XPRESS) license. We also could not compare with MN-BaB since it required GPU to run. Also, since we are comparing with DEEPPOLY and KPOLY, and ERAN uses these techniques internally, we skipped a direct comparison with ERAN.

Neural Network	#hidden layers	#activation units	Defensive training
3×50	2	110	None
3×100	2	210	None
5×100	4	410	None
6×100	5	510	DiffAI
9×100	8	810	None
6×200	5	1010	None
9×200	8	1610	None
6×500	6	3000	None
6×500	6	3000	PGD, $\epsilon = 0.1$
6×500	6	3000	PGD, $\epsilon = 0.3$
4×1024	3	3072	None

(a)



(b)

Figure 3: (a) Neural networks details (b) Cactus plot [46] with related techniques: The x-axis represents the number of benchmarks solved, sorted in increasing order based on the time taken to solve them. The y-axis represents the cumulative sum of the time taken to solve the benchmarks up to a certain point on the x-axis.

a C++ interface of the tool Gurobi [30] to check the satisfiability as well as solve maxSat queries.

Benchmarks. We use the MNIST [44] dataset to check the effectiveness of our tool and comparisons. We use 11 different fully connected feedforward neural networks with ReLU activation, as shown in Figure 3(a). These benchmarks are taken from the DeepPoly's paper [24]. The input and output dimensions of each network are 784 and 10, respectively. The authors of DeepPoly used projected gradient descent (PGD) [32] and DiffAI [45] for adversarial training. Figure 3(a) contains the defended network i.e. trained with adversarial training, as well as the undefended network. The last column of Figure 3(a) shows how the defended networks were trained.

The predicate P on the input layer is created using the input image $\boldsymbol{im}$ and user-defined parameter ϵ. We first normalize each pixel of $\boldsymbol{im}$ between 0 and 1, then create $P = \bigwedge_{i=1}^{|l_0|} im(i) - \epsilon \leq x_{0i} \leq im(i) + \epsilon$, such that the lower and upper bound of each pixel should not exceed 0 and 1, respectively. The predicate Q on the output layer is created using the network's output. Suppose the predicted label of $\boldsymbol{im}$ on network N is y, then $Q = \bigwedge_{i=1}^{|l_k|} x_{ki} < y$, where $i \neq y$. One query instance $\langle N, P, Q \rangle$ is created for one network, one image, and one epsilon value. In our evaluation, we took 11 different networks, 8 different epsilons, and 100 different images. The total number of instances is 8800. However, there are 304 instances for which the network's predicted label differs from the image's actual label. We avoided such instances and consider a total of 8496 benchmark instances.

5.1 Results

We conducted the experiments on a machine with 64GB RAM, 2.20 GHz INTEL(R) XEON(R) CPU E5-2660 V2 processor with CentOS Linux 7 operating system. To make a fair comparison between the tools, we provide only a single CPU, and 2000 seconds timeout for each instance for each tool. We use vanilla DEEPPOLY (i.e., DEEPPOLY without preprocessing in line 1 of Algorithm 1) to generate the abstract constraints of benchmarks instances. Figure 3(b) represents the cactus plot of (log of) time taken vs the number of benchmarks solved for the most related techniques. Table 1 and 2 represent a pairwise comparison of the number of instances that a tool could solve which another couldn't (more precisely, the (i, j)-entry of the table is the number of instances which could be verified by tool i but not by tool j), and Figure 4, compares wrt epsilon, the robustness parameter.

Comparison with the most related techniques: In this subsection, we consider the techniques DEEPPOLY, KPOLY, and DEEPSRGR to compare with ours. We consider DEEPPOLY because it is at the base of our technique, and the techniques KPOLY and DEEPSRGR refine DEEPPOLY just as we do. These tools only report VERIFIED instances, while our tool can report VERIFIED and counter-example. Hence, we compare these techniques with only VERIFIED instances of our technique in the line of DREFINE_VERIFIED in cactus plot 3(b).

Our technique outperforms the others in terms of the verified number of instances. One can also see that when they do verify, DEEPPOLY and KPOLY are often more efficient, which is not surprising, while our tool is more efficient than DEEPSRGR. From Table 1, we also see that our tool solves all the benchmark instances which are solved by these three techniques (and in fact around ~ 700 more), except 14 instances where KPOLY succeeds and our tool times out.

Comparison with cegar based techniques: CEGAR_NN [27] is a tool that also uses counter example guided refinement. But the abstraction used is quite different from DEEPPOLY. This tool reduces the size of the network by merging similar neurons, such that they maintain the overapproximation and split back in the refinement process. We can conclude from Table 1 that CEGAR_NN verified only 18.88%, while our tool verified 61.42% of the total number of benchmark instances. Although, in total CEGAR_NN solves significantly fewer benchmarks, it is pertinent to note that this technique solves many unique benchmark instances as can be inferred from Table 1.

Comparison with state-of-the-art solvers: The tools $\alpha\beta$-CROWN and OVAL use several algorithms that are highly optimized and use several techniques. The authors of $\alpha\beta$-CROWN implement the techniques [23,14,15,38,37], and the authors of OVAL implement [39,34,40,41,41,42,43]. The authors of MARABOU implement the technique [11]. Table 1 shows that all three tools indeed solve

Verified \ Unverified	DeepPoly	kPoly	deepSRGR	cegar_nn	$\alpha\beta$-CROWN	Oval	marabou	drefine	total
DeepPoly	**0**	0	0	3708	63	66	383	0	4633
kPoly	156	**0**	114	3760	63	66	389	14	4789
deepSRGR	54	2	**0**	3736	63	66	401	0	4687
cegar_nn	713	609	687	**0**	217	238	87	417	1624
$\alpha\beta$-CROWN	1672	1516	1618	4821	**0**	84	944	1095	6242
Oval	1622	1466	1568	4789	31	**0**	902	1052	6189
marabou	2042	1892	2006	4741	994	1005	**0**	1624	6292
drefine	694	552	640	4106	180	190	655	**0**	5327

Table 1: Pairwise comparison of tools, e.g. entry on row kPoly and column DeepPoly represents 156 benchmark instances on which kPoly verified but DeepPoly fails. The green row highlights the number of solved benchmark instances by drefine and not others while the red column is the opposite.

about 1000 (out of 8496) more than we do[4]. However, we found around 180 benchmarks instances where $\alpha\beta$-CROWN fails, and our tool works, and around 190 benchmarks on which Oval fails and our tool works. Also, around 655 benchmarks where marabou fails and our tool works; see Table 1 for more details. In total, we are solving 59 unique benchmarks where all three tools fail to solve. Thus we believe that these tools are truly orthogonal in their strengths and could potentially be combined.

Epsilon vs. performance: As a sanity check, we analyzed the effect of perturbation size and the performance of the tools. In Figure 4, we present the comparison of fractional success rate of tools as epsilon grows from 0.005 to 0.05. At $\epsilon = 0.005$, the performance of all the tools is almost the same except cegar_nn and marabou. As epsilon increases, the success rate of tools drops consistently except cegar_nn and marabou. Here also, we perform better than DeepPoly, kPoly, deepSRGR, and cegar_nn, while $\alpha\beta$-CROWN and Oval perform better. We are performing better compared to marabou only when ϵ is less than 0.015.

Comparison with adversarially trained networks: The networks considered for evaluation in this study are the ones corresponding to the 4st, 9th, and 10th rows of Figure 3(a). These networks have been trained using adversarial techniques, where adversarial examples were generated using standard methods such as PGD/DiffAI, and the network was subsequently trained on these adversarial examples to enhance its robustness. Table 2 presents a pairwise comparison of verifiers on these adversarially trained networks, encompassing a total of 2223 benchmark instances. Our approach demonstrates significant superiority over DeepPoly, kPoly, and deepSRGR in terms of performance on these benchmarks. While $\alpha\beta$-CROWN and Oval outperform our approach by approximately 135 benchmarks, we are still able to solve 75 unique benchmarks that

[4] $\alpha\beta$-CROWN outperforms marabou in VNN-COMP'22, while in our experiments marabou performs better; potential reasons could be difference in benchmarks and that $\alpha\beta$-CROWN uses GPU, while marabou uses CPU in VNN-COMP'22.

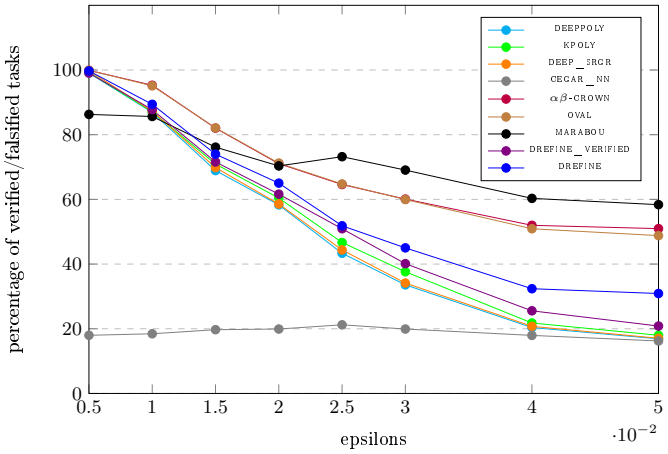


Figure 4: Size of input perturbation (epsilon) vs. percentage of solved instances

Verified \ Unverified	DeepPoly	kPoly	deepSRGR	$\alpha\beta$-CROWN	Oval	dRefine	total
DeepPoly	**0**	0	0	3	3	0	1626
kPoly	9	**0**	9	5	4	2	1635
deepSRGR	2	2	**0**	3	3	0	1628
$\alpha\beta$-CROWN	302	295	302	**0**	1	205	1925
Oval	312	304	310	11	**0**	214	1935
dRefine	170	163	168	76	75	**0**	1796

Table 2: Pairwise comparison of tools on adversarially trained networks

remain unsolved by both of these tools. Considering the total number of benchmarks is 2223, this indicates a notable number of benchmarks that our approach successfully addresses.

Detailed comparison with $\alpha\beta$-CROWN with same preprocessing: To evaluate the effectiveness of our approach, we conducted a benchmark analysis on the instances where refinement was applied. We applied the same preprocessing steps used by $\alpha\beta$-CROWN to filter the benchmarks.

The preprocessing steps include the so-called PGD (Projected Gradient Descent) attack, followed by CROWN [23] which is an incomplete technique. The PGD attack is a method that can generate counter examples. It works by iteratively updating the perturbation in the direction of the gradient of the loss function with respect to the input data, while constraining the magnitude of the perturbation to be within a predefined limit. If PGD fails, then CROWN runs to generate the over-approximated bounds.

After preprocessing, the total number of benchmarks was reduced to 2362, which were the benchmarks that were not solved by preprocessing steps. *Out of these benchmarks, $\alpha\beta$-CROWN was able to verify* 626*, while our approach verified* 570 *benchmarks. Notably, our approach was able to solve* 311 *benchmarks*

that were not solved by $\alpha\beta$-CROWN, while $\alpha\beta$-CROWN solved 366 *benchmarks that were not solved by our approach. All* 311 *benchmarks solved by our approach were from adversarially trained networks, while not a single benchmark out of* 366 *solved by $\alpha\beta$-CROWN were from adversarially trained networks*, suggesting that our approach performs well for such networks, as PGD attack is not very effective on these benchmarks. These results indicate that the majority of the benchmarks (1095) solved by $\alpha\beta$-CROWN and not by our approach, as shown in Table 1, are likely solved by preprocessing steps rather than the refinement procedure. *Further, the union of benchmarks solved by both tools results in a total of 937 benchmarks, demonstrating a clear improvement over the number of benchmarks solved by $\alpha\beta$-CROWN alone.* This highlights the significant contribution of our technique in the context of portfolio verifiers, as it complements and enhances the overall performance when integrated with other verification tools.

Subroutine time: We also conducted measurements to determine the average time required by the GETMARKNEURON subroutine and the REFINEMENT subroutine. The GETMARKNEURON subroutine exhibited an average execution time of 15.66 seconds, whereas the REFINEMENT subroutine took 89.69 seconds on average. In comparison, the GETMARKNEURON subroutine accounted for only 14.86% of the total time, which represents a small proportion. Furthermore, we measured the average number of marked neurons, which amounted to 14.81, and the average refinement iteration count, which stood at 3.19. These values also indicate a relatively small magnitude. These findings suggest that integrating our GETMARKNEURON method with an efficient refinement procedure could yield further improvements in overall performance.

6 Conclusion

We have presented a novel cegar-based approach. Our approach comprises two parts. One part finds the causes of spuriousness, while the other part refine the information found in the first part. Experimental evaluation shows that we outperform related refinement techniques, in terms of efficiency and effectivity. We also are able to verify several benchmarks that are beyond state of the art solvers, highly optimized solvers. Our experiments indicate when our technique can be useful and valuable as part of the portfolio of techniques for scalability of robustness verification. As futurework, we plan to extend our technique/tool to make it independent of DEEPPOLY and applicable with other abstraction based techniques and tools.

References

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

2. Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis. volume 11, pages 47–58. Elsevier, 2013.
3. Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.
4. Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
5. Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, pages 1–30, 2021.
6. Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward relu neural networks. *CoRR*, abs/1706.07351, 2017.
7. Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints An Int. J.*, 23(3):296–309, 2018.
8. Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.
9. Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017.
10. 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*, pages 97–117. Springer, 2017.
11. Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.
12. Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.
13. Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.
14. Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.
15. Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *CoRR*, abs/2011.13824, 2020.
16. Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. General cutting planes for bound-propagation-based neural network verification. *CoRR*, abs/2208.05740, 2022.

17. Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *UAI*, volume 1, page 3, 2018.
18. Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 3–18. IEEE Computer Society, 2018.
19. Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. Fast and effective robustness certification. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10825–10836, 2018.
20. Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. Boosting robustness certification of neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
21. Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.
22. Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5286–5295. PMLR, 2018.
23. Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in neural information processing systems*, 31, 2018.
24. Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
25. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1599–1614, 2018.
26. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6367–6377, 2018.
27. Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer, 2020.
28. Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. Improving neural network verification through spurious region guided refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–408. Springer, 2021.
29. Xuankang Lin, He Zhu, Roopsha Samanta, and Suresh Jagannathan. Art: Abstraction refinement-guided training for provably correct neural networks. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 148–157. IEEE, 2020.
30. Bob Bixby. The gurobi optimizer. *Transp. Re-search Part B*, 41(2):159–178, 2007.

31. Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32, 2019.
32. Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9185–9193, 2018.
33. $\alpha\beta$-crown. `https://github.com/huanzhang12/alpha-beta-CROWN`, 2021.
34. Rudy Bunel, P Mudigonda, Ilker Turkaslan, P Torr, Jingyue Lu, and Pushmeet Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.
35. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.
36. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.
37. Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
38. Huan Zhang, Shiqi Wang, Kaidi Xu, Yihan Wang, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. A branch and bound framework for stronger adversarial attacks of relu networks. In *International Conference on Machine Learning*, pages 26591–26604. PMLR, 2022.
39. Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.
40. Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip Torr, and M Pawan Kumar. Lagrangian decomposition for neural network verification. In *Conference on Uncertainty in Artificial Intelligence*, pages 370–379. PMLR, 2020.
41. Alessandro De Palma, Harkirat S Behl, Rudy Bunel, Philip Torr, and M Pawan Kumar. Scaling the convex barrier with active sets. In *Proceedings of the ICLR 2021 Conference*. Open Review, 2021.
42. Alessandro De Palma, Harkirat Singh Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. Scaling the convex barrier with sparse dual algorithms. *CoRR*, abs/2101.05844, 2021.
43. Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *CoRR*, abs/2104.06718, 2021.
44. Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.
45. Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.
46. Martin Brain, James H Davenport, and Alberto Griggio. Benchmarking solvers, sat-style. In *SC^2 @ ISSAC*, 2017.