Neural Network Verification Using Residual Reasoning

Yizhak Yisrael Elboher(✉), Elazar Cohen, and Guy Katz

The Hebrew University of Jerusalem, Jerusalem, Israel
{yizhak.elboher,elazar.cohen1,g.katz}@mail.huji.ac.il

Abstract. With the increasing integration of neural networks as components in mission-critical systems, there is an increasing need to ensure that they satisfy various safety and liveness requirements. In recent years, numerous sound and complete verification methods have been proposed towards that end, but these typically suffer from severe scalability limitations. Recent work has proposed enhancing such verification techniques with abstraction-refinement capabilities, which have been shown to boost scalability: instead of verifying a large and complex network, the verifier constructs and then verifies a much smaller network, whose correctness implies the correctness of the original network. A shortcoming of such a scheme is that if verifying the smaller network fails, the verifier needs to perform a refinement step that increases the size of the network being verified, and then start verifying the new network from scratch—effectively "wasting" its earlier work on verifying the smaller network. In this paper, we present an enhancement to abstraction-based verification of neural networks, by using *residual reasoning*: the process of utilizing information acquired when verifying an abstract network, in order to expedite the verification of a refined network. In essence, the method allows the verifier to store information about parts of the search space in which the refined network is guaranteed to behave correctly, and allows it to focus on areas where bugs might be discovered. We implemented our approach as an extension to the Marabou verifier, and obtained promising results.

Keywords: Neural networks · Verification · Abstraction refinement · Residual reasoning · Incremental reasoning

1 Introduction

In recent years, the use of deep neural networks (DNNs) [16] in critical components of diverse systems has been gaining momentum. A few notable examples include the fields of speech recognition [10], image recognition [17], autonomous driving [6], and many others. The reason for this unprecedented success is the ability of DNNs to generalize from a small set of training data, and then correctly handle previously unseen inputs.

Still, despite their success, neural networks suffer from various reliability issues. First, they are completely dependent on the training process, which may

B.-H. Schlingloff and M. Chai (Eds.): SEFM 2022, LNCS 13550, pp. 173–189, 2022.
https://doi.org/10.1007/978-3-031-17108-6_11

include data that is anecdotal, partial, noisy, or biased [22,28]; further, the training process has inherent over-fitting limitations [34]; and finally, trained networks suffer from susceptibility to adversarial attacks, as well as from obscurity and lack of explainability [1]. Unless addressed, these concerns, and others, are likely to limit the applicability of DNNs in the coming years.

A promising approach for improving the reliability of DNN models is to apply *formal verification* techniques: automated and rigorous techniques that can ensure that a DNN model adheres to a given specification, in all possible corner cases [15,18,20,30]. While sound and complete formal verification methods can certify that DNNs are reliable, these methods can typically only tackle small or medium-sized DNNs; and despite significant strides in recent years, scalability remains a major issue [4].

In order to improve the scalability of DNN verification, recent studies have demonstrated the great potential of enhancing it with abstraction-refinement techniques [2,8,14,26]. The idea is to use a black-box DNN verifier, and feed it a series of *abstract networks*—i.e., DNNs that are significantly smaller than the original network being verified. Because the complexity of DNN verification is exponential in the size of the DNN in question [20], these queries can be solved relatively quickly; and the abstract networks are constructed so that their correctness implies the correctness of the original, larger network. The downside of abstraction is that sometimes, verifying the smaller network returns an inconclusive result—in which case, the abstract network is *refined* and made slightly larger, and the process is repeated. Is it well known that the heuristics used for performing the abstraction and refinement steps can have a significant impact on performance [8,14], and that poor heuristics can cause the abstraction-refinement sequence of queries to take longer to dispatch than the original query.

In this paper, we propose an extension that can improve the performance of an abstraction-refinement verification scheme. The idea is to use *residual reasoning* [3]: an approach for re-using information obtained in an early verification query, in order to expedite a subsequent query. Presently, a verifier might verify an abstract network N_1, obtain an inconclusive answer, and then verify a refined network, N_2; and it will verify N_2 from scratch, as if it had never verified N_1. Using residual reasoning, we seek to leverage the similarities between N_1 and N_2 in order to identify large portions of the verification search space that need not be explored, because we are guaranteed a-priori that they contain no violations of the property being checked.

More specifically, modern verifiers can be regarded as traversing a large search tree. Each branching in the tree is caused by an *activation function* within the neural network, which can take on multiple linear phases; and each branch corresponds to one of these phases. We show that when a verifier traverses a branch of the search tree and determines that no property violations occur therein, that information can be used to deduce that no violation can exist in some of the branches of the search tree traversed when verifying a refined network. The advantages of this approach are clear: by curtailing the search space, the verification process can be expedited significantly. The disadvantage is that, unlike

in other abstraction-refinement based techniques, the verifier needs to be instrumented, and cannot be used as a black box.

Our contributions in this paper are as follows: (i) we formally define our residual reasoning scheme, in a general way that preserves the soundness and completeness of the underlying verifier; (ii) we specify how our approach can be used to extend the state-of-the-art Marabou DNN verification engine [21]; and (iii) we implement our approach, and evaluate it on the ACAS Xu set of benchmarks [19]. We regard this work as a step towards tapping into the great potential of abstraction-refinement methods in the context of DNN verification.

The rest of the paper is organized as follows. In Sect. 2 we recap the necessary background on DNNs and their verification. Next, in Sect. 3 we describe our general method for residual reasoning; followed by a discussion of how our technique can enhance a specific abstraction-refinement method, in Sect. 4. Sections 5 is then dedicated to explaining how our method can be applied using the Marabou DNN verifier as a backend, followed by our evaluation of the approach in Sect. 6. Related work is covered in Sect. 7, and we conclude in Sect. 8.

2 Background

Deep Neural Networks (DNNs). A neural network [16] $N : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a directed graph, organized into an input layer, multiple hidden layers, and an output layer. Each layer is a set of nodes (neurons), which can take on real values. When an input vector is passed into the input layer, it can be used to iteratively compute the values of neurons in the following layers, all through to neurons in the output layer—which constitute the network's output. We use L_i to denote the i'th layer of the DNN, and $v_{i,j}$ to denote the j'th node in L_i.

Typically, each neuron in the DNN is evaluated by first computing a weighted sum of the values assigned to neurons in the preceding layer, and then applying some activation function to the result. For simplicity, we restrict our attention to the popular ReLU activation function [16], which is a piecewise-linear function defined as $\text{ReLU}(x) = max(x, 0)$. When $x > 0$, we say that the ReLU is active; and otherwise, we say that it is inactive. A simple example appears in Fig. 1, and shows a DNN evaluated on input $\langle 0, 1 \rangle$. The value above each neuron is the weighted sum that it computes, prior to the application of the ReLU activation function. The network's output is 9.

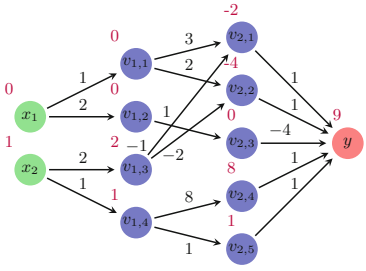


Fig. 1. A DNN with an input layer (green), two hidden layers (blue), and an output layer (red). (Color figure online)

Neural Network Verification. Neural network verification [23] deals with checking whether an input-output relation in a neural network holds. A verification query is a couple $\langle N, \varphi \rangle$, where N is a neural network and φ is a property

of the form: $\vec{x} \in D_I \wedge \vec{y} \in D_O$, meaning that the input $\vec{x}$ is in some input domain D_I and the output $\vec{y}$ is in some output domain D_O. Typically, φ represents *undesirable* behavior; and so the verification problem is to find an input $\vec{x}$ and its matching output $\vec{y}$ that satisfy φ, and so constitute a counter-example (the `SAT` case), or to prove that no such $\vec{x}$ exists (the `UNSAT` case). Without loss of generality, we assume that verification queries only consists of a network N with a single output neuron y, and of a property φ of the form $\vec{x} \in D_I \wedge y > c$; other queries can be reduced to this setting in a straightforward way [14].

As a simple example, consider the DNN in Fig. 1 and the property $\varphi : x_1, x_2 \in [0, 1] \wedge y > 14$. Checking whether input $x_1 = 0, x_2 = 1$ satisfies this property, we get that it does not, since $y = 9 \leq 14$. A sound verifier, therefore, would not return $\langle 0, 1 \rangle$ as a satisfying assignment for this query.

Linear Programming and Case Splitting. A key technique for DNN verification, which is nowadays used by many leading verification tools, is called *case splitting* [21,29,31]. A DNN verification problem can be regarded as a satisfiability problem, where linear constraints and ReLU constraints must be satisfied simultaneously; and while linear constraints are easy to solve [9], the ReLUs render the problem NP-Complete [20]. In case splitting, the verifier sometimes transforms a ReLU constraint into an equivalent disjunction of linear constraints:

$$(y = \mathrm{ReLU}(x)) \equiv ((x \leq 0 \wedge y = 0) \vee (x \geq 0 \wedge y = x))$$

and then each time *guesses* which of the two disjuncts holds, and attempts to satisfy the resulting constraints. This approach gives rise to a search tree, where internal nodes correspond to ReLU constraints, and their outgoing edges to the two linear constraints each ReLU can take. Each leaf of this tree is a problem that can be solved directly, e.g., because all ReLUs have been split upon. These problems are often dispatched using linear programming engines.

Case splitting might produce an exponential number of sub-problems, and so solvers apply a myriad of heuristics to avoid them or prioritize between them. Solvers also use deduction to rule out a-priori case splits that cannot lead to a satisfying assignment. Such techniques are beyond our scope.

Abstraction-Refinement (AR). Abstraction-refinement is a common mechanism for improving the performance of verification tools in various domains [8], including in DNN verification [2,14,26]. A sketch of the basic scheme of AR is illustrated in Fig. 8 in Appendix A of the full version of this paper [13]. The process begins with a DNN N and a property φ to verify, and then *abstracts* N into a different, smaller network N'. A key property is that N' *over-approximates* N: if $\langle N', \varphi \rangle$ is `UNSAT`, then $\langle N, \varphi \rangle$ is also `UNSAT`. Thus, it is usually preferable to verify the smaller N' instead of N.

If a verifier determines that $\langle N', \varphi \rangle$ is `SAT`, it returns a counter-example $\vec{x_0}$. That counter-example is then checked to determine whether it also constitutes a counterexample for $\langle N, \varphi \rangle$. If so, the original query is `SAT`, and we are done; but otherwise, $\vec{x_0}$ is a *spurious* counter-example, indicating that N' is inadequate for

determining the satisfiability of the original query. We then apply *refinement*: we use N', and usually also $\vec{x_0}$, to create a new network N'', which is larger than N' but is still an over-approximation of N. The process is then repeated using N''. Usually, the process is guaranteed to converge: either we are able to determine the satisfiability of the original query using one of the abstract networks, or we end up refining N' all the way back to N, and solve the original query, which, by definition, cannot return a spurious result.

In this paper we focus on a particular abstraction-refinement mechanism for DNN verification [14]. There, abstraction and refinement are performed by merging or splitting (respectively) neurons in the network, and aggregating the weights of their incoming and outgoing edges. This merging and splitting is carried out in a specific way, which guarantees that if N is abstracted into N', then for all input $\vec{x}$ it holds that $N'(\vec{x}) \geq N(\vec{x})$; and thus, if $N'(\vec{x}) \geq c$ is `UNSAT`, then $N(\vec{x}) \geq c$ is also `UNSAT`, as is required of an over-approximation.

An illustrative example appears in Fig. 2. On the left, we have the network from Fig. 1, denoted N. The middle network, denoted N', is obtained by merging together neurons $v_{2,1}$ and $v_{2,2}$ into the single neuron $v_{2,1+2}$; and by merging neurons $v_{2,4}$ and $v_{2,5}$ into the single neuron $v_{2,4+5}$. The weights on the outgoing edges of these neurons are the sums of the outgoing edges of their original neurons; and the weights of the incoming edges are either the min or max or the original weights, depending on various criteria [14]. It can be proven [14] that N' over-approximates N; for example, $N(\langle 3,1\rangle) = -6 < N'(\langle 3,1\rangle) = 6$. Finally, the network on the right, denoted N'', is obtained from N by splitting a previously merged neuron. N'' is larger than N', but it is still an over-approximation of the original N: for example, $N''(\langle 3,1\rangle) = 1 > N(\langle 3,1\rangle) = -6$.

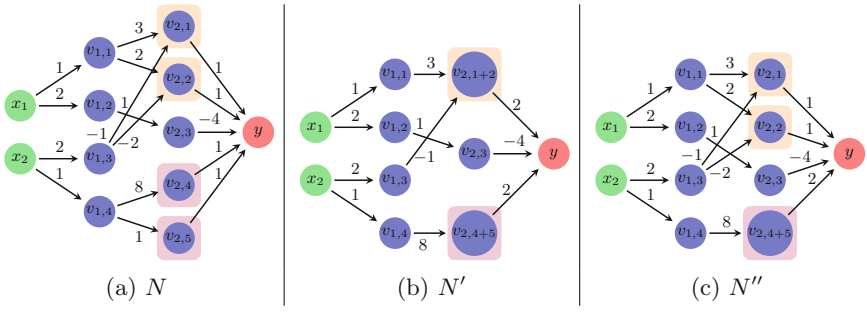


Fig. 2. Neural network abstraction and refinement through the merging and splitting of neurons [14].

3 Residual Reasoning (RR)

Consider again our running example, and observe that property φ is satisfiable for the most abstract network: for $\vec{x_0} = \langle 0,1\rangle$ we have $N'(\vec{x_0}) = 16$. However,

this $\vec{x_0}$ is a spurious counterexample, as $N(\vec{x_0}) = 9$. Consequently, refinement is performed, and the verifier sets out to verify $\langle N'', \varphi \rangle$; and this query is solved from scratch. However, notice that the verification queries of φ in N', N'' are very similar: the networks are almost identical, and the property is the same. The idea is thus to re-use some of the information already discovered when $\langle N', \varphi \rangle$ was solved in order to expedite the solving of $\langle N'', \varphi \rangle$. Intuitively, an abstract network allows the verifier to explore the search space very coarsely, whereas a refined network allows the verifier to explore that space in greater detail. Thus, areas of that space that were determined safe for the abstract network need not be re-explored in the refined network.

In order to enable knowledge retention between subsequent calls to the verifier, we propose to introduce a *context* variable, Γ, that is passed to the verifier along with each verification query. Γ is used in two ways: (i) the verifier can store into Γ information that may be useful if a refined version of the current network is later verified; and (ii) the verifier may use information already in Γ to curtail the search space of the query currently being solved. A scheme of the proposed mechanism appears in Fig. 9 in Appendix A of the full version of this paper [13]. Of course, Γ must be designed carefully in order to maintain soundness.

Avoiding Case-Splits with Γ. In order to expedite subsequent verification queries, we propose to store in Γ information that will allow the verifier to *avoid case splits*. Because case splits are the most significant bottleneck in DNN verification [20,29], using Γ to reduce their number seems like a natural strategy.

Let N' be an abstract network, and N'' its refinement; and observe the queries $\langle N', \varphi \rangle$ and $\langle N'', \varphi \rangle$. Let $R_1, \ldots, R_n$ denote the ReLU constraints in N'. For each ReLU R_i, we use a Boolean variable r_i to indicate whether the constraint is active (r_i is true), or inactive ($\neg r_i$ is true). We then define Γ to be a CNF formula over these Boolean variables:

$$\Gamma : \quad \bigwedge \Big(\bigvee_{l_j \in \bigcup_{i=1}^{n} \{r_i, \neg r_i\}} l_j \Big)$$

In order for our approach to maintain soundness, Γ needs to be a *valid formula* for $\langle N'', \varphi \rangle$; i.e., if there exists an assignment that satisfies $\langle N'', \varphi \rangle$, it must also satisfy Γ. Under this assumption, a verifier can use Γ to avoid case-splitting during the verification of the refined network, using unit-propagation [5]. For example, suppose that one of the clauses in Γ is $(r_1 \vee \neg r_2 \vee \neg r_3)$, and that while verifying the refined network, the verifier has performed two case splits already to the effect that r_1 is false (R_1 is inactive) and r_2 is true (R_2 is active). In this case, the verifier can immediately set r_3 to false, as it is guaranteed that no satisfying assignments exist where r_3 is true, as these would violate the clause above. This guarantees that no future splitting is performed on R_3.

More formally, we state the following Lemma:

Lemma 1 (Soundness of Residual Reasoning). *Let $\langle N', \varphi \rangle$ and $\langle N'', \varphi \rangle$ be verification queries on an abstract network N' and its refinement N'', being solved by a sound verifier; and let Γ be a valid formula as described above. If the verifier uses Γ to deduce the phases of ReLU constraints using unit propagation for the verification of $\langle N'', \varphi \rangle$, soundness is maintained.*

The proof is straightforward, and is omitted. We also note that when multiple consecutive refinement steps are performed, some renaming of variables within Γ is required; we discuss this in later sections.

4 Residual Reasoning and Neuron-Merging Abstraction

Our proposed approach for residual reasoning is quite general; and our definitions do not specify how Γ should be populated. In order to construct in Γ a lemma that will be valid for future refinements of the network, one must take into account the specifics of the abstraction-refinement scheme in use. In this section, we propose one possible integration with a recently proposed abstraction-refinement scheme that merges and splits neurons [14], which was discussed in Sect. 2.

We begin by revisiting our example from Fig. 2. Suppose that in order to solve query $\langle N, \varphi \rangle$, we generate the abstract network N' and attempt to verify $\langle N', \varphi \rangle$ instead. During verification, some case splits are performed; and it is discovered that when neuron $v_{2,1+2}$'s ReLU function is active, no satisfying assignment can be found. Later, the verifier discovers a satisfying assignment for which $v_{2,1+2}$ is inactive: $\vec{x} = \langle 0, 1 \rangle \Rightarrow N'(\vec{x}) = 16 > 14$. Unfortunately, this counterexample turns out to be spurious, because $N(\langle 0, 1 \rangle) = 9 \leq 14$, and so the network is refined: node $v_{2,1+2}$ is split into two new nodes, $(v_{2,1}, v_{2,2})$, giving rise to the refined network N''. The verifier then begins solving query $\langle N'', \varphi \rangle$.

We make the following claim: because no satisfying assignment exists for $\langle N', \varphi \rangle$ when $v_{2,1+2}$ is active, and because $v_{2,1+2}$ was refined into $(v_{2,1}, v_{2,2})$, then no satisfying assignment exists for $\langle N'', \varphi \rangle$ when $v_{2,1}$ and $v_{2,2}$ are both active. In other words, it is sound to verify $\langle N'', \varphi \rangle$ given $\Gamma = (\neg r_{2,1} \vee \neg r_{2,2})$, where $r_{2,1}$ and $r_{2,2}$ correspond to the activation phase of $v_{2,1}$ and $v_{2,2}$, respectively. Thus, e.g., if the verifier performs a case split and fixes $v_{2,1}$ to its active phase, it can immediately set $v_{2,2}$ to inactive, without bothering to explore the case where $v_{2,2}$ is also active.

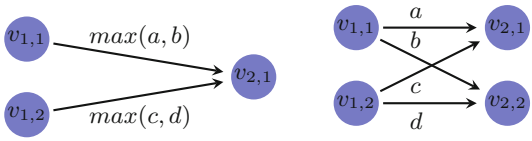
In order to provide intuition as to why this claim holds, we now formally prove it; i.e., we show that if an input $\vec{x}$ satisfies $\langle N'', \varphi \rangle$ when $v_{2,1}$ and $v_{2,2}$ are both active, then it must also satisfy $\langle N'_1, \varphi \rangle$ when $v_{2,1+2}$ is active. First, we observe that because N'' is a refinement of N', it immediately follows that $N''(\vec{x}) \leq N'(\vec{x})$; and because the property φ is of the form $y > c$, if $\langle N'', \varphi \rangle$ is `SAT` then $\langle N', \varphi \rangle$ is also `SAT`. Next, we observe that N'' and N' are identical in all layers preceding $v_{2,1}, v_{2,2}$ and $v_{2,1+2}$, and so all neurons feedings into these three neurons are assigned the same values in both networks. Finally, we assume towards contradiction that $v_{2,1+2}$ is not active; i.e., that $3 \cdot \mathrm{ReLU}(v_{1,1}) -$

$\mathrm{ReLU}(v_{1,3}) < 0$; but because it also holds that $v_{2,1} = 3 \cdot \mathrm{ReLU}(v_{1,1}) - \mathrm{ReLU}(v_{1,3})$, this contradicts the assumption that $v_{2,1}$ and $v_{2,2}$ are both active. This concludes our proof, and shows that $\Gamma = (\neg r_{2,1} \vee \neg r_{2,2})$ is valid.

In the remainder of this section, we formalize the principle demonstrated in the example above. The formalization is complex, and relies on the details of the abstraction mechanism [14]; we give here the gist of the formalization, with additional details appearing in Appendix B of the full version of this paper [13].

Using the terminology of [14], two nodes can be merged as part of the abstraction process if they share a *type*: specifically, if they are both `inc` neurons, or if they are both `dec` neurons. An `inc` neuron has the property that *increasing* its value results in an increase to the network's single output; whereas a `dec` neuron has the property that *decreasing* its value increases the network's single output. In our running example, neuron $v_{2,1+2}$ is an `inc` neuron, whereas neuron $v_{2,3}$ is a `dec` neuron.

We use the term *abstract neuron* to refer to a neuron generated by the merging of two neurons from the same category, and the term *refined neuron* to refer to a neuron that was generated (restored) during a refinement step. An example for the merging of two `inc` neurons appears in Fig. 3.



(a) abstract network (b) refined network

Fig. 3. Abstraction/refinement of two *inc* neurons.

We now state our main theorem, which justifies our method of populating Γ. We then give an outline of the proof, and refer the reader to Theorem 2 in Appendix B of the full version of this paper [13] for additional details.

Theorem 1. *Let $\langle N, \varphi \rangle$ be a verification query, where $N : \vec{x} \rightarrow y$ has a single output node y, and φ is of the form $\varphi = (\vec{l} \leq \vec{x} \leq \vec{u}) \wedge (y > c)$. Let N' be an abstract network obtained from N using neuron merging, and let N'' be a network obtained from N' using a single refinement step in reverse order of abstraction. Specifically, let v be a neuron in N' that was split into two neurons v_1, v_2 in N''. Then, if a certain guard condition $\mathcal{G}$ holds, we have the following:*

1. *If v is `inc` neuron, and during the verification of $\langle N', \varphi \rangle$, the verifier determines that setting v to active leads to an `UNSAT` branch of the search tree, then $\Gamma = (\neg r_1 \vee \neg r_2)$ is a valid formula for $\langle N'', \varphi \rangle$ (where r_1 and r_2 correspond to v_1 and v_2, respectively).*
2. *Symmetrically, if setting a `dec` neuron v to inactive leads to an `UNSAT` branch, then $\Gamma = (r_1 \vee r_2)$ is a valid formula for $\langle N'', \varphi \rangle$.*

The guard condition $\mathcal{G}$ is intuitively defined as the conjunction of the following stipulations, whose goal is to enforce that the branches in both search trees (of $\langle N', \varphi \rangle$ and $\langle N'', \varphi \rangle$) are sufficiently similar:

1. The same case splits have been applied during the verification of N' and N'', for all neurons in the preceding layers of the abstract neuron and for any other neurons in the same layer as the abstract neuron.
2. The same case splits have been applied during the verification of N' and N'' for the abstract neuron and its refined neurons.
3. Every `inc` neuron in layers following the layer of v, v_1, v_2 has been split on and set to active, and every `dec` neuron in these layers has been split on and set to inactive.

We stress that the guard condition $\mathcal{G}$ does not change the way Γ is populated; but that the verifier must ensure that $\mathcal{G}$ holds before it applies unit-propagation based on Γ. The precise definitions and proof appear in Appendix B of the full version of this paper [13].

When the conditions of the theorem are met, a satisfying assignment within the specific branch of the search tree of the refined network would indicate that the corresponding branch in the abstract network is also `SAT`, which we already know is untrue; and consequently, that branch can be soundly skipped. To prove the theorem, we require the two following lemmas, each corresponding to one of the two cases of the theorem.

Lemma 2. *Given an input $\vec{x}$, if the value of an abstract `inc` node v is negative, then at least one of the values of the refined nodes v_1 and v_2 is negative for the same $\vec{x}$.*

Proof Outline. We explain how to prove the lemma using the general network from Fig. 3; and this proof can be generalized to any network in a straightforward way. Observe nodes $v_{2,1}$ and $v_{2,2}$ in Fig. 3(b), which are nodes refined from node $v_{2,1}$ in Fig 3(a). We need to prove that the following implication holds:

$$x_1 \cdot max(a, b) + x_2 \cdot max(c, d) < 0 \Rightarrow (x_1 \cdot a + x_2 \cdot c < 0 \vee x_1 \cdot b + x_2 \cdot d < 0)$$

The values of x_1, x_2 are the outputs of ReLUs, and so are non-negative. We can thus split into 4 cases:

1. If $x_1 = 0, x_2 = 0$, the implication holds trivially.
2. If $x_1 = 0, x_2 > 0$, then $x_2 \cdot max(c, d) < 0$, and so $c, d < 0$. We get that $x_1 \cdot a + x_2 \cdot c = x_2 \cdot c < 0$ and $x_1 \cdot b + x_2 \cdot d = x_2 \cdot d < 0$, and so the implication holds.
3. The case where $x_1 > 0, x_2 = 0$ is symmetrical to the previous case.
4. If $x_1 > 0, x_2 > 0$, the implication becomes

   $$max(x_1 \cdot a, x_1 \cdot b) + max(x_2 \cdot c, x_2 \cdot d) < 0 \Rightarrow (x_1 \cdot a + x_2 \cdot c < 0 \vee x_1 \cdot b + x_2 \cdot d < 0)$$

   Let us denote $a' = x_1 \cdot a,\ b' = x_1 \cdot b$ and $c' = x_2 \cdot c,\ d' = x_2 \cdot d$. The lemma then becomes:

   $$max(a', b') + max(c', d') < 0 \Rightarrow a' + c' < 0 \vee b' + d' < 0$$

- If $a' \geq b'$, then $a' = max(a', b')$ and $a' + max(c', d') < 0$. We then get that
$$b' + d' \leq a' + d' \leq a' + max(c', d') < 0$$
as needed.
- If $a' < b'$, then $b' = max(a', b')$ and $b' + max(c', d') < 0$. We then get that
$$a' + c' \leq b' + max(c', d') < 0$$
again as needed.

Lemma 2 establishes the correctness of Theorem 1 for `inc` neurons. We also have the following, symmetrical lemma for `dec` neurons:

Lemma 3. *Given an input $\vec{x}$, if the value of an abstract* `dec` *node v is positive, then at least one of the values of the refined nodes v_1 and v_2 is positive for the same $\vec{x}$.*

The proof outline is similar to that of Lemma 2, and appears in Appendix B of the full version of this paper [13].

The result of applying Theorem 1 as part of the verification of our running example from Fig. 2 is illustrated in Fig. 4. There, each rectangle represents a single verification query, and blue lines indicate abstraction steps. Within each rectangle, we see the verifier's search tree, where triangles signify sub-trees—and red triangles are sub-trees where the verifier was able to deduce that no satisfying assignment exists. The figure shows that, when solving the query in the bottom rectangle, the verifier discovered an `UNSAT` sub-tree that meets the conditions of the Theorem. This allows the verifier to deduce that another sub-tree, in another rectangle/query, is also `UNSAT`, as indicated by a green arrow. Specifically, by discovering that setting $v_{2,1+2}$ to *active* results in `UNSAT`, the verifier can deduce that setting $v_{2,1}$ to *active* and then $v_{2,2}$ to *active* must also result in `UNSAT`.

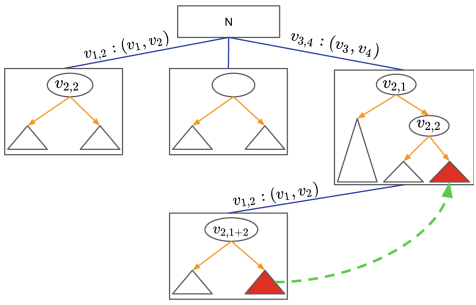


Fig. 4. Applying Theorem 1 while solving the query from Fig. 2.

Multiple Refinement Steps. So far, we have only discussed populating Γ for a single refinement step. However, Γ can be adjusted as multiple refinement steps are performed. In that case, each invocation of Theorem 1 adds another CNF clause to the formula already stored in Γ. Further, some book-keeping and renaming is required, as neuron identifiers change across the different networks: intuitively, whenever an abstract neuron v is split into neurons v_1 and v_2, the literal v must be replaced with $v_1 \vee v_2$. These notions are formalized in Sec. 5; and the soundness of this procedure can be proven using repeated invocations of Theorem 1.

5 Adding Residual Reasoning to Reluplex

Unlike in previous abstraction-refinement approaches for DNN verification [2, 14,26], residual reasoning requires instrumenting the DNN verifier in question, for the purpose of populating, and using, Γ. We next describe such an instrumentation for the Reluplex algorithm [20], which is the core algorithm used in the state-of-the-art verifier Marabou [21]. Reluplex, a sound and complete DNN verification algorithm, employs case-splitting as discussed in Sect. 2, along with various heuristics for curtailing the search space and reducing the number of splits [32,33]; and it has been integrated with abstraction-refinement techniques before [14], rendering it a prime candidate for residual reasoning. We term our enhanced version of Reluplex AR^4, which stands for Abstraction-Refinement with Residual Reasoning for Reluplex.

For our purposes, it is convenient to think of Reluplex as a set of derivation rules, applied according to an implementation-specific strategy. The most relevant parts of this calculus, borrowed from Katz et al. [20] and simplified, appear in Fig. 5; other rules, specifically those that deal with the technical aspects of solving linear problems, are omitted for brevity.

$$\textsf{Failure}\ \frac{\exists x \in \mathcal{X}.\ l(x) > u(x)}{\texttt{UNSAT}} \qquad \textsf{ReluSplit}\ \frac{\langle x_i, x_j \rangle \in R,\quad l(x_i) < 0,\quad u(x_i) > 0}{u(x_i) := 0 \qquad l(x_i) := 0}$$

$$\textsf{Success}\ \frac{\forall x \in \mathcal{X}.\ l(x) \leq \alpha(x) \leq u(x),\quad \forall \langle x, y \rangle \in R.\ \alpha(y) = \max(0, \alpha(x))}{\texttt{SAT}}$$

Fig. 5. Derivation rules of Reluplex calculus (partial, simplified).

Internally, Reluplex represents the verification query as a set of linear equalities and lower/upper bounds over a set of variables, and a separate set of ReLU constraints. A *configuration* of Reluplex over a set of variables $\mathcal{X}$ is either a distinguished symbol from the set $\{\texttt{SAT}, \texttt{UNSAT}\}$, or a tuple $\langle T, l, u, \alpha, R \rangle$, where: T, the *tableau*, contains the set of linear equations; l, u are mappings that assign each variable $x \in \mathcal{X}$ a lower and an upper bound, respectively; α, the *assignment*, maps each variable $x \in \mathcal{X}$ to a real value; and R is the set of ReLU constraints, i.e. $\langle x, y \rangle \in R$ indicates that $y = \text{ReLU}(x)$. Reluplex will often derive *tighter*

bounds as it solves a query; i.e., will discover greater lower bounds or smaller upper bounds for some of the variables.

Using these definitions, the rules in Fig. 5 can be interpreted follows: Failure is applicable when Reluplex discovers inconsistent bounds for a variable, indicating that the query is UNSAT. ReluSplit is applicable for any ReLU constraint whose linear phase is unknown; and it allows Reluplex to "guess" a linear phase for that ReLU, by either setting the upper bound of its input to 0 (the inactive case), or the lower bound of its input to 0 (the active case). Success is applicable when the current configuration satisfies every constraint, and returns SAT.

In order to support AR^4, we extend the Reluplex calculus with additional rules, depicted in Fig. 6. We use the context variable Γ, as before, to store a valid CNF formula to assist the verifier; and we also introduce two additional context variables, Γ_A and Γ_B, for book-keeping purposes. Specifically, Γ_A stores a mapping between abstract neurons and their refined neurons; i.e., it is comprised of triples $\langle v, v_1, v_2 \rangle$, indicating that abstract neuron v has been refined into neurons v_1 and v_2. Γ_B is used for storing past case splits performed by the verifier, to be used in populating Γ when the verifier finds an UNSAT branch. Given variable x of neuron v, we use $\mathcal{G}^{\texttt{inc}}(\Gamma_A, \Gamma_B, x)$ and $\mathcal{G}^{\texttt{dec}}(\Gamma_A, \Gamma_B, x)$ to denote a Boolean function that returns true if and only if the guard conditions required for applying Theorem 1 hold, for an inc or dec neuron v, respectively.

$$\textsf{ReluSplit}\ \frac{\langle x_i, x_j \rangle \in R, \quad l(x_i) < 0, \quad u(x_i) > 0}{\begin{array}{ll} u(x_i) := 0 & l(x_i) := 0 \\ \Gamma_B := \Gamma_B \vee r_i & \Gamma_B := \Gamma_B \vee \neg r_i \end{array}}$$

$$\textsf{Failure}\ \frac{\exists x_i \in \mathcal{X}.\ l(x_i) > u(x_i)}{\texttt{UNSAT}, \Gamma := \Gamma \wedge \Gamma_B} \qquad \textsf{AbstractionStep}\ \frac{CanAbstract(x_1, x_2)}{\Gamma_A := \Gamma_A \cup \langle x_{1,2}, x_1, x_2 \rangle}$$

$$\textsf{RefinementStep}\ \frac{\Gamma_A \neq \emptyset}{\Gamma_A := \Gamma_A[:-1]} \qquad \textsf{RealSuccess}\ \frac{\texttt{SAT} \wedge isRealSAT(\Gamma_A)}{\texttt{RealSAT}}$$

$$\textsf{ApplyAbstraction}\ \frac{true}{Abstract(\Gamma_A), UpdateContext(\Gamma, \Gamma_A, \Gamma_B)}$$

$$\textsf{Prune}_1\ \frac{\langle x, x_i, x_j \rangle \in \Gamma_A \wedge \neg r_i, \neg r_j \in \Gamma_B \wedge \mathcal{G}^{\texttt{inc}}(\Gamma_A, \Gamma_B, x) \wedge l(x_i) = 0}{u(x_j) = 0, \Gamma_B := \Gamma_B \vee r_j}$$

$$\textsf{Prune}_2\ \frac{\langle x, x_i, x_j \rangle \in \Gamma_A \wedge r_i, r_j \in \Gamma_B \wedge \mathcal{G}^{\texttt{dec}}(\Gamma_A, \Gamma_B, x) \wedge u(x_i) = 0}{l(x_j) = 0, \Gamma_B := \Gamma_B \vee \neg r_j}$$

Fig. 6. Derivation rules for the AR^4 calculus.

The rules in Fig. 6 are interpreted as follows. AbstractionStep is used for merging neurons and creating the initial, abstract network. RefinementStep is applicable when dealing with an abstract network (indicated by $\Gamma_A \neq \emptyset$), and performs a refinement step by canceling the last abstraction step. ApplyAbstraction is applicable anytime, and generates an abstract network according to the information in Γ_A, updating the relevant contexts correspondingly. The Success rule from the

original Reluplex calculus in included, as is, in the AR^4 calculus; but we note that a `SAT` conclusion that it reaches is applicable only to the current, potentially abstract network, and could thus be spurious. To solve this issue, we add the RealSuccess rule, which checks whether a `SAT` result is true for the original network as well. Thus, in addition to `SAT` or `UNSAT`, the `RealSAT` state is also a terminal state for our calculus.

The Failure rule replaces the Reluplex rule with the same name, and is applicable when contradictory bounds are discovered; but apart from declaring `UNSAT`, it also populates Γ with the current case-split history in Γ_B, for future pruning of the search space. The ReluSplit rule, similarly to the Reluplex version, guesses a linear phase for one the ReLUs, but now also records that action in Γ_B. Finally, the $\text{Prune}_{1/2}$ rules are applicable when all the conditions of Theorem 1 (for the `inc`/`dec` cases, respectively) are met, and they trim the search tree and update Γ accordingly.

Side Procedures. We intuitively describe the four functions, *CanAbstract*, *Abstract*, *UpdateContext* and *isRealSat*, which appear in the calculus; additional details can be found in Appendix C of the full version of this paper [13].

- CanAbstract (Algorithm 1, Appendix C [13]) checks whether two neurons can be merged according to their types; and also checks whether the assignment to the variables did not change yet during the verification process.
- Abstract (Algorithm 3, Appendix C [13]) performs multiple abstraction steps. A single abstraction step (Algorithm 2, Appendix C [13]) is defined as the merging of two neurons of the same type, given that the assignments to their variables were not yet changed during verification.
- UpdateContext clears the case-splitting context by setting ($\Gamma_B = \emptyset$), and also updates clauses in Γ to use new variables: for variables representing `inc` nodes, $\neg r$ is replaced with $\neg r_1 \vee \neg r_2$; and for variables representing `dec` nodes, r is replaced with $r_1 \vee r_2$.
- *isRealSat* (Algorithm 5, Appendix C [13]) checks whether a counterexample holds in the original network.

Implementation Strategy. The derivation rules in Fig. 6 define the "legal moves" of AR^4—i.e., we are guaranteed that by applying them, the resulting verifier will be sound. We now discuss one possible *strategy* for applying them, which we used in our proof-of-concept implementation.

We begin by applying AbstractionStep to saturation, in order to reach a small abstract network; and then apply once the ApplyAbstraction rule, to properly initialize the context variables. Then, we enter into the loop of abstraction-based verification: we apply the Reluplex core rules using existing strategies [21], but every time the ReluSplit rule is applied we immediately apply Prune_1 and Prune_2, if they are applicable. The Failure and Success rules are applied as in Reluplex, and RealSuccess is applied immediately after Success if it is applicable; otherwise, we apply RefinementStep, and repeat the process. We also attempt to apply Prune_1 and Prune_2 after each application of Failure, since it updates Γ.

Table 1. Comparing AR^4 and AR.

	Adversarial		Safety		Total (weighted)	
	AR^4	AR	AR^4	AR	AR^4	AR
Timeouts	95/900	116/900	7/180	9/180	102/1080	125/1080
Instances solved more quickly	160	95	28	24	188	119
Uniquely solved	26	5	2	0	28	5
Visited tree states	6.078	7.65	3.569	4.98	5.634	7.178
Avg. instrumentation time	91.54	–	36.5	–	82.367	–

6 Experiments and Evaluation

For evaluation purposes, we created a proof-of-concept implementation of AR^4, and compared it to the only tool currently available that supports CEGAR-based DNN verification—namely, the extension of Marabou proposed in [14]. We used both tools to verify a collection of properties of the ACAS Xu family of 45 DNNs (each with 310 neurons, spread across 8 layers) for airborne collision avoidance [19]. Specifically, we verified a collection of 4 safety properties and 20 adversarial robustness properties for these networks, giving a total of 1080 benchmarks; and from these experiments we collected, for each tool, the runtime (including instrumentation time), the number of properties successfully verified within the allotted timeout of two hours, and the number of case splits performed. The experiments were conducted on x86-64 Gnu/Linux based machines using a single Intel(R) Xeon(R) Gold 6130 CPU @ 2.10 GHz core. Our code is publicly available online.[1]

The results of our experiments appear in Table 1 and Fig. 7, and demonstrate the advantages of AR^4 compared to AR. AR^4 timed out on 18.4% fewer benchmarks, and solved 188 benchmarks more quickly than AR, compared to 119 where AR was faster. We note that in these comparisons, we treated experiments in which both tools finished within 5 s of each other as ties. Next, we observe that residual reasoning successfully curtailed the search space: on average, AR^4 traversed 5.634 states of the search tree per experiment, compared to 7.178 states traversed by AR—a 21.5% decrease.

Despite the advantages it often affords, AR^4 is not always superior to AR—because the cost of instrumenting the verifier is not always negligible. In our experimenters, the verifier spent an average of 82 s executing our instrumentation code out of an average total runtime of 885 seconds—nearly 10%, which is quite significant. In order to mitigate this issue, moving forward we plan to strengthen the engineering of our tool, e.g., by improve its implementation of unit-propagation through the use of watch literals [5].

[1] https://drive.google.com/file/d/1onk3dW3yJeyXw8_rcL6wUsVFC1bMYvjL.

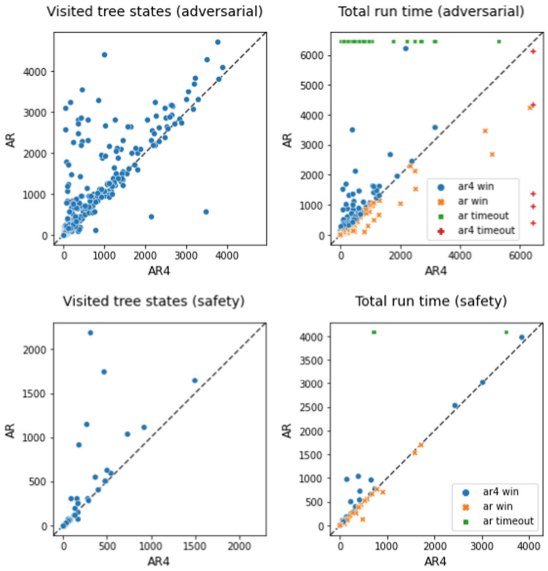


Fig. 7. Comparing AR^4 and AR.

7 Related Work

Modern DNN verification schemes leverage principles from SAT and SMT solving [12,18,20,21,25], mixed integer linear programming [7,11,12,29], abstract interpretation [15,24,27,31], and others. Many of these approaches apply case-splitting, and could benefit from residual reasoning.

Abstraction-refinement techniques are known to be highly beneficial in verifying hand-crafted systems [8], and recently there have been promising attempts to apply them to DNN verification as well [2,14,26]. As far as we know, ours is the first attempt to apply residual reasoning in this context.

8 Conclusion

As DNNs are becoming increasingly integrated into safety-critical systems, improving the scalability of DNN verification is crucial. Abstraction-refinement techniques could play a significant part in this effort, but they can sometimes create redundant work for the verifier. The residual reasoning technique that we propose can eliminate some of this redundancy, resulting in a speedier verification procedure. We regard our work here as another step towards tapping the potential of abstraction-refinement methods in DNN verification.

Moving forward, we plan to improve the engineering of our AR^4 tool; and to integrate it with other abstraction-refinement DNN verification techniques [2].

Acknowledgments. This work was supported by ISF grant 683/18. We thank Jiaxiang Liu and Yunhan Xing for their insightful comments about this work.

References

1. Angelov, P., Soares, E.: Towards explainable deep neural networks (xDNN). Neural Netw. **130**, 185–194 (2020)
2. Ashok, P., Hashemi, V., Kretinsky, J., Mühlberger, S.: DeepAbstract: neural network abstraction for accelerating verification. In: Proceedings of 18th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 92–107 (2020)
3. Azzopardi, S., Colombo, C., Pace, G.: A technique for automata-based verification with residual reasoning. In: Proceedings of 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 237–248 (2020)
4. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (VNN-COMP 2021): summary and results. Technical report (2021). http://arxiv.org/abs/2109.00498
5. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. IOS Press (2009)
6. Bojarski, M., et al.: End to end learning for self-driving cars. Technical report (2016). http://arxiv.org/abs/1604.07316
7. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Kumar, M.: Piecewise linear neural network verification: a comparative study. Technical report (2017). http://arxiv.org/abs/1711.00455
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
9. Dantzig, G.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
10. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. Technical report (2018). http://arxiv.org/abs/1810.04805
11. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks. In: Proceedings of 10th NASA Formal Methods Symposium (NFM), pp. 121–138 (2018)
12. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 269–286. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_19
13. Elboher, Y., Cohen, E., Katz, G.: Neural network verification using residual reasoning. Technical report (2022). http://arxiv.org/abs/2208.03083
14. Elboher, Y.Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 43–65. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_3
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, E., Chaudhuri, S., Vechev, M.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of 39th IEEE Symposium on Security and Privacy (S&P) (2018)
16. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)

18. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 3–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_1
19. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings of 35th Digital Avionics Systems Conference (DASC), pp. 1–10 (2016)
20. 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
21. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
22. Kim, B., Kim, H., Kim, K., Kim, S., Kim, J.: Learning not to learn: training deep neural networks with biased data. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 9004–9012 (2019)
23. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.: Algorithms for verifying deep neural networks. Technical report (2020). http://arxiv.org/abs/1903.06758
24. Müller, M., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.: PRIMA: general and precise neural network certification via scalable convex hull approximations. In: Proceedings of 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2022)
25. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks. Technical report (2017). http://arxiv.org/abs/1709.06662
26. Prabhakar, P., Afzal, Z.: Abstraction based output range analysis for neural networks. Technical report (2020). http://arxiv.org/abs/2007.09527
27. Singh, G., Gehr, T., Puschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings of 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2019)
28. Song, H., Kim, M., Park, D., Shin, Y., Lee, J.G.: End to end learning for self-driving cars. Technical report (2020). http://arxiv.org/abs/2007.08199
29. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. Technical report (2017). http://arxiv.org/abs/1711.07356
30. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of 27th USENIX Security Symposium (2018)
31. Wang, S., et al.: Beta-CROWN: efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. In: Proceedings of 35th Conference on Neural Information Processing Systems (NeurIPS) (2021)
32. Wu, H., et al.: Parallelization techniques for verifying neural networks. In: Proceedings of 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 128–137 (2020)
33. Wu, H., Zeljić, A., Katz, G., Barrett, C.: Efficient neural network analysis with sum-of-infeasibilities. In: TACAS 2022. LNCS, vol. 13243, pp. 143–163. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_8
34. Ying, X.: An overview of overfitting and its solutions. J. Phys: Conf. Ser. **1168**, 022022 (2019)