Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01138-w

SPECIAL SECTION PAPER

On applying residual reasoning within neural network verification

Yizhak Yisrael Elboher[1] · Elazar Cohen[1] · Guy Katz[1]

Received: 17 February 2023 / Revised: 27 August 2023 / Accepted: 25 October 2023

Abstract
As neural networks are increasingly being integrated into mission-critical systems, it is becoming crucial to ensure that they meet various safety and liveness requirements. Toward, that end, numerous complete and sound verification techniques have been proposed in recent years, but these often suffer from severe scalability issues. One recently proposed approach for improving the scalability of verification techniques is to enhance them with abstraction/refinement capabilities: instead of verifying a complex and large network, abstraction allows the verifier to construct and then verify a much smaller network, and the correctness of the smaller network immediately implies the correctness of the original, larger network. One shortcoming of this scheme is that whenever the smaller network cannot be verified, the verifier must perform a refinement step, in which the size of the network being verified is increased. The verifier then starts verifying the new network from scratch—effectively "forgetting" its earlier work, in which the smaller network was verified. Here, we present an enhancement to abstraction-based neural network verification, which uses *residual reasoning*: a process where information acquired when verifying an abstract network is utilized in order to facilitate the verification of refined networks. At its core, the method enables the verifier to retain information about parts of the search space in which it was determined that the refined network behaves correctly, allowing the verifier to focus on areas of the search space where bugs might yet be discovered. For evaluation, we implemented our approach as an extension to the Marabou verifier and obtained highly promising results.

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

1 Introduction

In the past decade, the use of deep neural networks (DNNs) [26] within diverse and critical systems has been on the rise. A few notable examples include, e.g., the fields of image recognition [27], speech recognition [17], and autonomous driving [11]. This unprecedented success is due in part to the ability of DNNs to generalize well from a small set of training examples and later correctly handle previously unseen inputs.

Still, despite their undeniable success, DNNs suffer from several reliability issues. First, they completely depend on the training process, which may include partial, anecdotal, noisy, or biased data [37, 46]; second, the training process suffers from inherent over-fitting limitations [53]; and third, trained DNNs are susceptible to adversarial attacks and suffer from obscurity and lack of explainability [5]. Unless addressed, these limitations, and others, will likely limit the applicability of DNNs in many domains of interest.

One promising approach aimed at improving the reliability of DNNs is to use *formal verification* techniques: rigorous and automated techniques for ensuring that a DNN model abides by a given specification, in all possible corner cases [23, 28, 33, 49]. Although sound and complete formal verification approaches can certify that DNNs are reliable, they can typically only scale to small- or medium-sized DNNs. Although DNN verification has progressed rapidly in recent years, scalability remains a major issue [8].

In order to render DNN verifiers more scalable, recent work has demonstrated the great potential of enhancing them with abstraction/refinement principles [6, 14, 21, 43]. The core idea is to leverage a DNN verifier as a black box and to use it to dispatch a series of verification queries over

Communicated by Holger Schlingloff and Ming Chai.

Yizhak Yisrael Elboher
yizhak.elboher@mail.huji.ac.il

Elazar Cohen
elazar.cohen1@mail.huji.ac.il

Guy Katz
g.katz@mail.huji.ac.il

1 The Hebrew University of Jerusalem, Jerusalem, Israel

abstract networks—i.e., networks whose correctness immediately implies the correctness of the original DNN. Further, the abstract DNNs are constructed in such a way that they are much smaller than the original network. Due to the fact that DNN verification complexity increases exponentially with the size of the DNN being verified [33, 34], the resulting queries can be solved fairly quickly. The downside of applying abstraction is that oftentimes, verifying the smaller DNNs will return an inconclusive result—and when this happens, the abstract network must be *refined*, making it slightly larger; the process is then repeated. It is widely accepted that the heuristics used for selecting which abstraction and refinement steps to perform have a significant impact on the performance of the overall procedure [14, 21]. Thus, poor heuristics might cause the sequence of abstraction/refinement queries to take longer to solve than simply solving the original query.

Here, we introduce an extension to the abstraction/refine ment verification scheme, aimed at improving the performance of DNN verifiers. The core of the extension is to use *residual reasoning* [7]: an approach for re-using the information gathered when verifying an abstract verification query, in order to expedite the later verification of subsequent, refined queries. Using existing schemes, when a verifier verifies an abstract network N_1 and obtains an inconclusive answer, it will proceed to verify a refined network, N_2, from scratch—as if it had not previously verified N_1. Through residual reasoning, one attempts to leverage the similarities between N_1 and N_2 in identifying large portions of the underlying verification search space where it is a-priori guaranteed that no violations of the property being verified can exist. These portions of the search space then need not be explored, resulting in a much speedier verification process.

In order to realize this concept, we leverage the fact that modern verifiers can typically be regarded as traversing a large search tree. Each *activation function* within the neural network causes the search tree to branch out, and each new branch represents a single linear phase of the activation function. Here, we show that whenever a verifier traverses one of these branches and discovers that no property violations occur within the branch, that information can be stored; and later, when the verifier traverses a search tree corresponding to a refined neural network, the stored information can be used to deduce that no violations exist within specific branches of that search tree. This approach affords clear advantages: by pruning the search space, the verification procedure is expedited significantly. However, the disadvantage is that, unlike in common abstraction/refinement-based techniques, the verifier must be instrumented and can no longer be used as a black box.

In this paper, we make the following contributions: (i) we rigorously define a general residual reasoning scheme for DNNs, which preserves the soundness and completeness of an underlying verifier; (ii) we specify in detail how our approach can extend the state-of-the-art Marabou DNN verifier [35]; and (iii) we present an implementation of our approach and evaluate it using the ACAS Xu set of benchmarks [32]. We view this work as another step toward leveraging abstraction/refinement principles within the broad context of DNN verification.

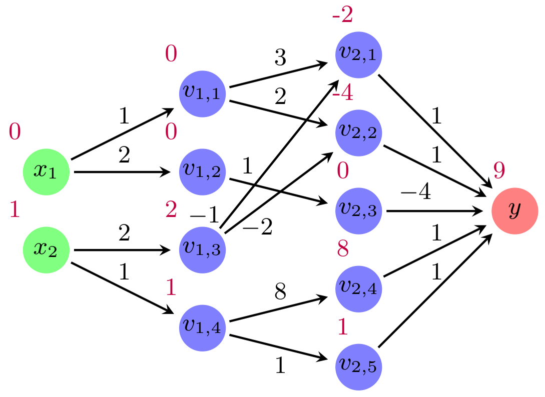

Fig. 1 (From [20]) A DNN with an input layer (green), two hidden layers (blue), and an output layer (red)

The remainder of the paper is organized as follows: In Sect. 2, we present the necessary background on DNNs and DNN verification. This is followed by Sect. 3, where we describe our general residual reasoning method. Next, we discuss how our technique can enhance specific abstraction/refinement methods in Sect. 4. Section 5 is then dedicated to exploring how our method can be integrated with the Marabou DNN verifier backend, followed by an evaluation of the approach in Sect. 6. We cover related work in Sect. 7 and conclude in Sect. 8.

2 Background

Deep Neural Networks (DNNs). A deep neural network $N : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a layered, directed graph [26]. This graph contains an input layer, multiple hidden layers, and finally an output layer. Each layer is comprised of a set of nodes (neurons) of the layer, which take on real values. The network is evaluated by assigning values to its input layer's neurons and then iteratively computing the values of neurons in each successive layer, until the values of the output neurons are computed—and these constitute the network's output.

Each neuron within the DNN is typically evaluated by (a) computing a weighted sum of values assigned to neurons in the preceding layer, and (b) applying an activation function to the weighted sum's result. For simplicity, in this work we restrict our attention to ReLU activations [26], which are defined using the popular, piecewise-linear function $\text{ReLU}(x) = \max(x, 0)$. When a neuron's input is $x \geq 0$, we say that the ReLU is active; whereas for input $x < 0$, we say that it is inactive. The simple example in Fig. 1 depicts a DNN evaluated on the input $\langle 0, 1\rangle$. Listed above each neuron is the weighted sum computed, before the ReLU activation function is applied to it. The network's final output is 9. Although we focus strictly on ReLUs, our approach can be directly applied to any other piecewise-linear activation function and can even support additional, non-piecewise-linear activation functions, through the common approach of piecewise-linear approximations [50, 55].

More formally, we follow common notation in [21] and use L to denote the number of layers of a given DNN N. We let N_i denote the i'th layer of the DNN, and $|N_i|$ the number of nodes of N_i. Layers 1 and L are the input and output layers, respectively. Layers $2, \ldots, L-1$ are the hidden layers. We denote the value of the j'th node of layer i by $v_{i,j}$ and denote the column vector $[v_{i,1}, \ldots, v_{i,|N_i|}]^T$ as V_i. Evaluating N is performed by calculating V_L for a given input assignment V_1. This is done by sequentially computing V_i for $i = 2, 3, \ldots L$, each time using the values of V_{i-1} to compute weighted sums, and then applying the ReLU activation functions. Specifically, layer i (for $i > 1$) is associated with a weight matrix W_i of size $N_i \times N_{i-1}$ and a bias vector B_i of size N_i. If i is a hidden layer, its values are given by $V_i = \text{ReLU}(W_i V_{i-1} + B_i)$, where the ReLUs are applied element-wise, and the output layer is given by $V_L = W_L V_{L-1} + B_L$ (ReLUs are not applied).

Neural Network Verification. The purpose of neural network verification [39] is to determine the validity of a given specification over the neural network's inputs and outputs. A verification query is a couple $\langle N, \varphi\rangle$, where N is a DNN and φ is a specification of the form $\vec{x} \in D_I \wedge \vec{y} \in D_O$. D_I and D_O represent the input and output domains, respectively, implying that the input $\vec{x}$ is in D_I and that the output $\vec{y}$ is in D_O. Typically, φ is used to represent some *undesirable* behavior. Thus, verification amounts to finding an input $\vec{x}$ and a matching output $\vec{y}$ that satisfy φ and consequently constitute a counter-example to φ (the `SAT` case); or, alternatively, to proving that no such $\vec{x}$ exists (the `UNSAT` case). Without loss of generality, we assume here that verification queries consists strictly of a DNN N with a single-output neuron y and that the property φ has the form $\vec{x} \in D_I \wedge y > c$. Other, more complex queries can be reduced to this form, in a straightforward manner [21, 30].

As a toy example, observe the DNN depicted in Fig. 1, and the corresponding property $\varphi : x_1, x_2 \in [0, 1] \wedge y > 14$. We observe that input $x_1 = 0, x_2 = 1$ does not satisfy φ, because $y = 9 \leq 14$. Thus, when presented with this query, a sound verifier would not return $\langle 0, 1\rangle$ as a satisfying assignment.

2.1 Linear programming and case splitting.

One technique that nowadays plays a significant role within many verification tools is called *case splitting* [35, 48, 50]. In case splitting, the DNN verification problem is typically regarded as a satisfiability problem, in which a collection of ReLU constraints and linear constraints and must be simultaneously satisfied. While the linear constraints are fairly easy to solve [16], it is the ReLUs that render the problem NP-Complete [33]. Through case splitting, the verifier will occasionally replace a ReLU constraint with an equivalent disjunction of the following linear constraints:

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

Then, the verifier will each time *guess* that one of the disjuncts must hold and will proceed to attempt to satisfy the resulting constraints. This process results in a search tree, where the internal nodes correspond to the various ReLU constraints, and their outgoing edges correspond to the two linear phases that each ReLU can take. Leaves of this tree are problems that no longer contain any ReLUs and can consequently be solved directly and easily—often, by using linear programming engines.

The disadvantage of case splitting is that it might result in a number of sub-problems that is exponential in the number of ReLU constraints. To mitigate this, solvers apply a variety of heuristics to avoid case splits, or prioritize between them. Solvers often also use deduction steps in order to determine, a priori, that certain case splits cannot lead to a satisfying assignment and consequently need not be considered. Such techniques are beyond the scope of this paper [39].

Abstraction/Refinement (AR). Abstraction/refinement is a method for improving the scalability of verifiers, which has been applied in various domains [14]—such as DNN verification [6, 21, 43]. The basic flow of the AR scheme is depicted in Fig. 2. The process begins with an initial DNN N and some property φ to be verified and then proceeds to *abstract* network N into a different, significantly smaller network N'. A key trait of this procedure is that N' always *over-approximates* N: that is, if $\langle N', \varphi\rangle$ is `UNSAT`, then by construction, $\langle N, \varphi\rangle$ is also `UNSAT`. Consequently, it is usually more efficient to try and verify the smaller N', as opposed to directly verifying N.

Whenever the verifier concludes that $\langle N', \varphi\rangle$ is `SAT`, it produces a counter-example $\vec{x_0}$. This counter-example can then be checked in order to determine whether it constitutes a correct counterexample also for $\langle N, \varphi\rangle$. If that is the case, we can determine that the original query is `SAT`.

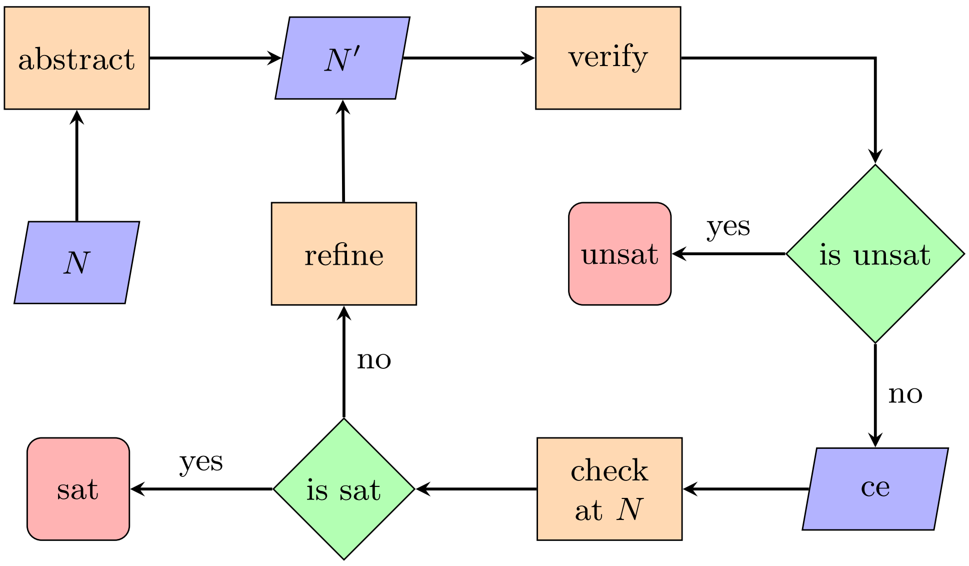

Fig. 2 DNN verification with abstraction/refinement. *ce* stands for counter-example

Otherwise, we say that $\vec{x_0}$ is a *spurious* counter-example, which indicates that N' is inadequate for the purpose of determining the satisfiability of $\langle N, \varphi\rangle$. When this happens, we apply *refinement*: we use N' and $\vec{x_0}$ in order to construct a new network N'', which is an over-approximation of N, although it is larger than N'. This process can then be repeated, using N''. Typically, the sequence of refinement steps is bound to converge: either we can successfully use one of the abstract networks in order to determine the satisfiability of the original query, or we eventually refine N' all the way back to the original N and then solve the original query. By definition, solving the original query cannot return a spurious result, and the process will then terminate.

Here, we target a particular mechanism of abstraction/refinement, used in DNN verification [21]. There, abstraction and refinement steps are carried out by merging, or, respectively, by splitting, neurons in the network. When neurons are merged, the weights of their incoming and outgoing edges are aggregated in a specific way. These manipulation of weights ensure that whenever N is abstracted into N', it will hold that $N'(\vec{x}) \geq N(\vec{x})$ for all input $\vec{x}$. Consequently, if $N'(\vec{x}) \geq c$ is UNSAT, it follows that $N(\vec{x}) \geq c$ is also UNSAT, as is required of an over-approximate verification query.

A simple example is depicted in Fig. 3. On the left, we see network N from Fig. 1. Next, the network on the middle (denoted N') is obtained through the merging of two neurons, $v_{2,1}$ and $v_{2,2}$, into neuron $v_{2,1+2}$, and by the merging of neurons $v_{2,4}$ and $v_{2,5}$ into neuron $v_{2,4+5}$. The weights of the edges outgoing from these neurons are calculated as the sums of the weights listed on the outgoing edges of the original neurons, and the weights of the edges incoming into these neurons are either the max or min of the original weights, as determined according to various criteria [21]. It has been shown [21] that N' is an over-approximation of N, e.g., $N(\langle 3, 1\rangle) = -6 < N'(\langle 3, 1\rangle) = 6$. Last but not least, the network on the right (denoted N'') is obtained from N by refinement—specifically, by splitting a previously merged neuron. We note that N'' is larger than N', but that it still over-approximates the original network N, e.g., $N''(\langle 3, 1\rangle) = 1 > N(\langle 3, 1\rangle) = -6$.

3 Residual reasoning (RR)

Let us return again to our running example. We observe that when we consider the most abstract network, N', property φ is satisfiable, e.g., for $\vec{x_0} = \langle 0, 1\rangle$ we get that $N'(\vec{x_0}) = 16$. Nonetheless, $\vec{x_0}$ is a spurious counterexample, because $N(\vec{x_0}) = 9$. Thus, the verifier performs a refinement step and begins verifying $\langle N'', \varphi\rangle$. This new query is solved from scratch, and without considering the previous query that had already been solved. However, we notice that the queries $\langle N', \varphi\rangle$ and $\langle N'', \varphi\rangle$ are, in fact, quite similar: the two networks are nearly identical, and the property is the same. We thus wish to re-use parts of the information discovered when $\langle N', \varphi\rangle$ was solved in expediting the solving process of $\langle N'', \varphi\rangle$. The intuition is that when verifying the abstract network, the verifier explores the search space at a coarse level, whereas verifying the refined network allows it to explore the search space at a finer granularity. Consequently, parts of that space that were previously determined safe (for the abstract network) need not be re-explored (for the refined network).

In order to allow the verifier to retain information between consecutive calls, we introduce here a *context* variable, Γ. This variable is passed to the verifier as part of each verification query, and it is used in two ways: (i) the verifier can store

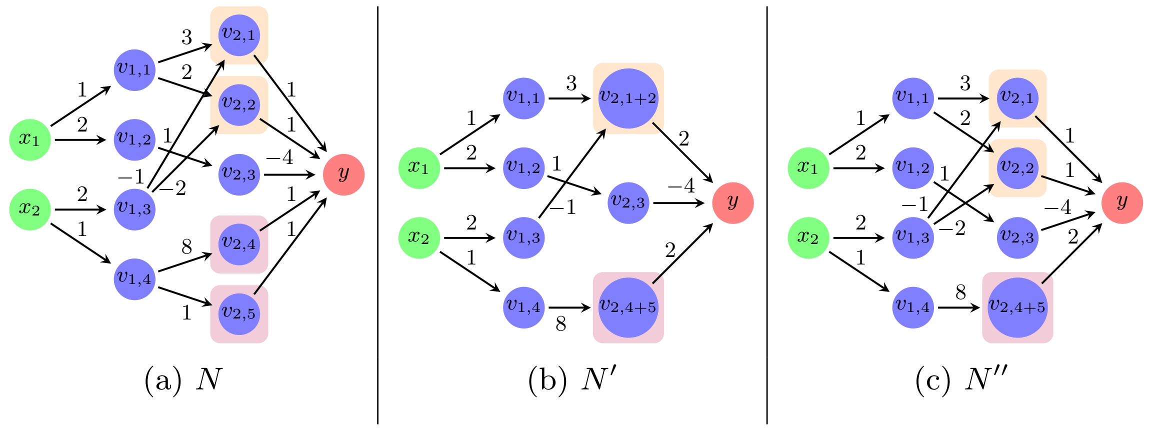

Fig. 3 (From [20]) Abstracting and refining a neural network, through neuron merging and splitting [21]

relevant information into Γ, to be used later, when refined versions of the current network are encountered, and (ii) the verifier may also use the information already in Γ in order to prune the search space of the current query. A high-level scheme of this mechanism appears in Fig. 4. Of course, in order to maintain soundness, the use of Γ must be designed carefully.

3.1 Avoiding case-splits with Γ

Toward the goal of expediting subsequent verification queries, we propose here to use Γ to store information that will allow the DNN verifier to *prune case splits*. Because case splits are a highly significant bottleneck in neural network verification [33, 48], using Γ in this way seems like a natural strategy.

Let N' denote an abstract network, and let N'' denote a refinement of N'. Let us observe the queries $\langle N', \varphi\rangle$ and $\langle N'', \varphi\rangle$, and let $R_1, \ldots, R_n$ denote the ReLU constraints in N'. For each ReLU constraint R_i, we introduce a Boolean variable r_i that indicates whether R_i is active or inactive (r_i is true or $\neg r_i$ is true, respectively). Next, we define Γ as a CNF formula over the Boolean variables we have introduced:

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

In order for our approach to remain sound, we ensure that Γ is a *valid formula* for $\langle N'', \varphi\rangle$; that is, if there exists an assignment that satisfies $\langle N'', \varphi\rangle$, then it must also satisfy Γ. Relying on this assumption, a verifier can use Γ in order to avoid case-splitting while verifying the refined network, by applying unit-propagation [10]. For instance, suppose that Γ contains the clause $(r_1 \vee \neg r_2 \vee \neg r_3)$ and that as part of verifying the refined network, the verifier has already performed two case splits, setting r_1 to false (R_1 is set to inactive) and r_2 to true (R_2 is set to active). Now, the verifier can immediately assign r_3 to false, because it is guaranteed that a satisfying assignment where r_3 is true cannot exist—as this would violate the clause above. This process ensures that no future splitting will be performed on R_3.

Put formally, we have the following Lemma:

Lemma 1 (Soundness of Residual Reasoning) *Let $\langle N', \varphi\rangle$ and $\langle N'', \varphi\rangle$ denote verification queries involving an abstract network N' and its refined network N'', which are being solved by a sound verifier; and let Γ denote a valid formula, as discussed above. If the verifier deduces the phases of ReLU constraints by applying unit propagation on clauses from Γ as it verifies $\langle N'', \varphi\rangle$, then soundness is maintained.*

The proof of this lemma is straightforward and is omitted. We observe that when multiple refinement steps are performed in sequence, variables within Γ may need to be renamed; we discuss this in later sections.

4 Residual reasoning and neuron-merging abstraction

The approach for residual reasoning that we propose here is quite general; indeed, our definitions do not specify precisely how to populate Γ. When constructing in Γ a lemma that is intended to remain valid after future refinements of the network, we must take into account the specifics of the abstraction/refinement scheme in use. In this section, we propose a method for populating Γ designed to work within a recently proposed abstraction/refinement scheme, where abstraction and refinement are performed by merging and splitting neurons [21] (the same abstraction/refinement approach that is discussed in Sect. 2).

Let us consider again our example from Fig. 3. Suppose that, as part of solving the query $\langle N, \varphi\rangle$, we generate an abstract network N' and begin verifying $\langle N', \varphi\rangle$. As the

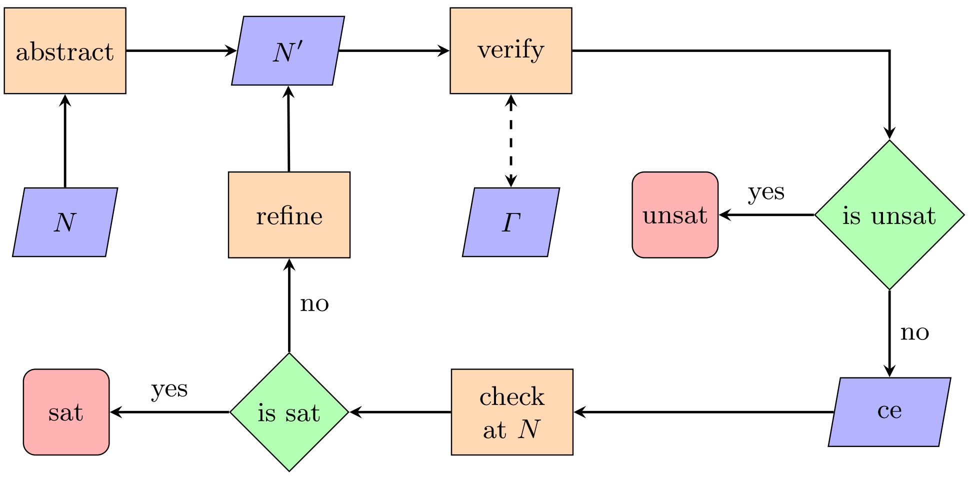

Fig. 4 DNN verification with residual reasoning. *ce* stands for counter-example

verification progresses, case splits are performed, and we then discover that setting neuron $v_{2,1+2}$'s ReLU to active implies that no satisfying assignments can be found. At a later time, we discover a satisfying assignment, for which $v_{2,1+2}$'s ReLU is inactive: $\vec{x} = \langle 0, 1\rangle \Rightarrow N'(\vec{x}) = 16 > 14$. This counterexample is unfortunately discovered to be spurious (because $N(\langle 0, 1\rangle) = 9 \leq 14$), and so we apply refinement: we split node $v_{2,1+2}$ into two new nodes, $(v_{2,1}, v_{2,2})$, which gives rise to a refined network N''. We then begin solving the verification query $\langle N'', \varphi\rangle$.

We claim that the following holds: because (1) no satisfying assignment exists for $\langle N', \varphi\rangle$ when $v_{2,1+2}$ is active, and (2) $v_{2,1+2}$ was refined into $(v_{2,1}, v_{2,2})$, it follows that when $v_{2,1}$ and $v_{2,2}$ are active, no satisfying assignment can exist for $\langle N'', \varphi\rangle$. Differently put, soundness is maintained by verifying $\langle N'', \varphi\rangle$ while setting $\Gamma = (\neg r_{2,1} \vee \neg r_{2,2})$, where $r_{2,1}$ and $r_{2,2}$ correspond to the respective activation phases of $v_{2,1}$ and $v_{2,2}$. Consequently, if the verifier applies a case split that fixes $v_{2,1}$ to active, it can immediately (and soundly) set $v_{2,2}$ to inactive, without exploring the case where $v_{2,2}$ is active.

In order to better justify why this claim holds, we turn to formally proving it, i.e., we now show that any input $\vec{x}$ that satisfies $\langle N'', \varphi\rangle$ with $v_{2,1}$ and $v_{2,2}$ both active, must also satisfy $\langle N'_1, \varphi\rangle$ when $v_{2,1+2}$ is active. We begin by observing that because N'' is a refinement of N', it holds that $N''(\vec{x}) \leq N'(\vec{x})$, and because φ has the form $y > c$, the satisfiability of $\langle N'', \varphi\rangle$ implies the satisfiability of $\langle N', \varphi\rangle$. We also observe that N'' and N' are identical in all layers up to the layers containing $v_{2,1}$, $v_{2,2}$ and $v_{2,1+2}$, and consequently, when the two networks are evaluated on the same input values, all neurons feedings into these three neurons are assigned the exact same values. Now, we assume toward contradiction that $v_{2,1+2}$ is inactive, i.e., that $3\cdot\text{ReLU}(v_{1,1}) - \text{ReLU}(v_{1,3}) < 0$. However, because it holds that $v_{2,1} = 3 \cdot \text{ReLU}(v_{1,1}) - \text{ReLU}(v_{1,3})$, this contradicts our assumption that $v_{2,1}$ and $v_{2,2}$ are both active. Thus, our proof is concluded, implying the validity of $\Gamma = (\neg r_{2,1} \vee \neg r_{2,2})$.

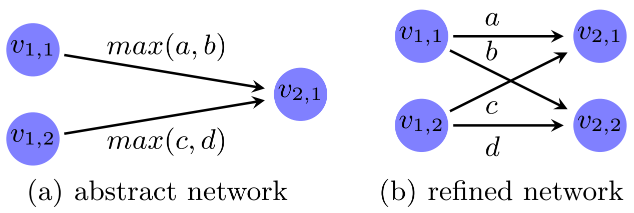

Fig. 5 (From [20]) Abstraction/refinement of two `inc` neurons

The remainder of this section is devoted to formalizing the principle that the previous example demonstrates. This formalization is complex, and it relies significantly on details relating to the underlying abstraction mechanism [21]. We break the proof into theorems: the first theorem deals with applying unit propagation based on valid formulas, whereas the second theorem deals with various guard conditions that are required in order to guarantee the sound application of unit propagation.

Following the terminology used by Elboher et al. [21], two neurons can be merged as part of an abstraction step if they share a *type*: that is, if they are both marked as `inc` neurons, or if they are both marked as `dec` neurons. We say that a neuron is `inc` if it has the property that *increasing* its value always results in an increase to the network's single output neuron. Symmetrically, we say that a neuron is `dec` if *decreasing* its value results in an increase to the network's single output. In our running example, we observe that neuron $v_{2,1+2}$ is an `inc` neuron, while neuron $v_{2,3}$ is `dec`.

We say that a neuron is an *abstract neuron* if it is generated as a result of the merging two neurons that share a category; we say that a neuron is a *refined neuron* if it is generated (restored) as part of a refinement step. A simple example of the merging of two `inc` neurons is depicted in Fig. 5.

Next, we state our main theorem, which serves as the justification for our method for populating Γ.

Theorem 1 *Let $N : \vec{x} \rightarrow y$ denote a DNN with a single output node y, let φ denote a specification of the form $\varphi = (\vec{l} \leq \vec{x} \leq \vec{u}) \wedge (y > c)$, and let $\langle N, \varphi\rangle$ be a verification query. Suppose that N' is an abstract DNN, obtained from N through neuron merging, and suppose that N'' is another DNN, obtained from N' through a single refinement step, in reverse order of abstraction. Let v denote the single neuron in N' that was split into two neurons v_1, v_2 in N'' as a result of the refinement step. Then, assuming that a certain guard condition $\mathcal{G}$ holds, the following also holds:*

1. *If v is* `inc` *neuron, and if while verifying $\langle N', \varphi\rangle$ the verifier concludes that fixing v to its active phase leads to an* `UNSAT` *branch of the search tree, it follows that $\Gamma = (\neg r_1 \vee \neg r_2)$ 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 fixing a* `dec` *neuron v to its inactive phase leads to an* `UNSAT` *branch of the tree, it follows that $\Gamma = (r_1 \vee r_2)$ is a valid formula for $\langle N'', \varphi\rangle$.*

The goal of $\mathcal{G}$ is to ensure that the branches in both search trees (those corresponding to $\langle N', \varphi\rangle$ and $\langle N'', \varphi\rangle$) are sufficiently similar. More concretely, $\mathcal{G}$ is set to be a conjunction of these stipulations:

1. During the verification of N' and N'', the exact same case splits have been applied to all neurons in layers preceding the layer of the abstract neuron, and also to all neurons that share a layer with the abstract neuron.
2. During the verification of N' and N'', the exact same case splits have been applied to the abstract neuron and the refined neurons that were generated from it.
3. In any layer following the layer of neurons v, v_1 and v_2, every `inc` neuron has been split on and was set to active; and every `dec` neuron has been split on and was set to inactive.

We note that the $\mathcal{G}$ condition does not alter the way Γ is populated. However, the verifier must ensure that $\mathcal{G}$ is satisfied before it performs any unit-propagation based on Γ.

In order to formalize the intuitive explanation above, we define the following sets of constraints. Suppose we are given a property of the form $\varphi = (\vec{l} \leq \vec{x} \leq \vec{u}) \wedge (y > c)$ for some vectors $\vec{l}, \vec{u}$ and a constant c, a network N with L layers, and a couple of networks N' and N'' such that N'' is refined from N' through a single refinement step on node u in the r'th layer ($3 \leq r \leq L-1$). For $M \in \{N, N', N''\}$, we denote:

1. $M_{C_{pre}} := \bigcup_{p \in P} \{p = active/inactive\}$ where $P \subseteq \{M_1 \cup \cdots \cup M_r \backslash u\}$, i.e a subset of the preceding layers where abstraction did not occur. $M_{C_{pre}}$ includes case splits of neurons in preceding layers (including the abstraction layer, except for the abstract neuron). Each neuron $p \in P$ can be active/inactive.
2. $M_{C^{inc}_{abs}} := \{u = active\}$ if u is an increasing node; else it is $\emptyset$.
3. $M_{C^{dec}_{abs}} := \{u = inactive\}$ if u is a decreasing node; else it is $\emptyset$.
4. $M_{C^{inc}_{ref}} := \{u_1 = active, u_2 = active\}$ if u is an increasing node; else it is $\emptyset$.
5. $M_{C^{dec}_{ref}} := \{u_1 = inactive, u_2 = inactive\}$ if u is a decreasing node; else it is $\emptyset$.
6. $M_{C^{inc}_{post}} := \bigcup_{q \in Q^{inc}} \{q = active\}$ where $Q^{inc} \subseteq \{M_{r+1} \cup \cdots \cup M_L\}$ is a set of all increasing nodes in layers $M_{r+1}, \ldots, M_L$. $M_{C^{inc}_{post}}$ requires that every `inc` neuron in the consecutive layers after the abstract node is *active*.
7. $M_{C^{dec}_{post}} := \bigcup_{q \in Q^{dec}} \{q = inactive\}$ where $Q^{dec} \subseteq \{M_{r+1} \cup \cdots \cup M_L\}$ is a set of all decreasing nodes in layers $M_{r+1}, \ldots, M_L$. $M_{C^{dec}_{post}}$ requires that every `dec` neuron in the consecutive layers after the abstract node is *inactive*.

where M_C (for $C \in \{C_{pre}, C^{inc}_{abs}, C^{dec}_{abs}, C^{inc}_{ref}, C^{dec}_{ref}, C^{inc}_{post}, C^{dec}_{post}\}$) is a partial set C of the case splits that were applied during the verification of the property in M.

Using these definitions, the three guard stipulations above are defined as follows:

- $N'_{C_{pre}} = N''_{C_{pre}}$ realizes the first stipulation.
- $(N'_{C^{inc}_{abs}} = \emptyset \Leftrightarrow N''_{C^{inc}_{ref}} = \emptyset) \wedge (N'_{C^{dec}_{abs}} = \emptyset \Leftrightarrow N''_{C^{dec}_{ref}} = \emptyset)$ realizes the second stipulation.
- $(|N'_{C^{inc}_{post}} \cup N'_{C^{dec}_{post}}| = |N'_{r+1} \cup \cdots \cup N'_L|) \wedge (|N''_{C^{inc}_{post}} \cup N''_{C^{dec}_{post}}| = |N''_{r+1} \cup \cdots \cup N''_L|)$ realizes the third stipulation.

Next, we need to prove that the three conditions above imply the correctness of the implication of unsatisfiability, when a refinement step is performed. We prove this by finding the violated case split in the refinement.

Observe that a possible solution from the verifier is an input $\vec{x}$ that, when propagated in the abstract network, induces a set of case splits $\{s_i\}_{i=1}^n$. Hence, the solution can be treated as a couple $(\vec{x}, \{s_i\}_{i=1}^n)$.

Theorem 2 *If all three guard conditions hold, then for any solution $(\vec{x}, \{s_i\}_{i=1}^n)$, if any constraint $s \in \{s_i\}_{i=1}^n$ is violated in N', then there is a corresponding constraint which is violated in N''.*

Proof The violation can occur in any neuron in the abstract network, and we handle all options by considering the cases of input layer neurons, preceding layer neurons, abstract neurons, consecutive layer neurons, or output layer neurons.

1. If there was a violation in the input layer: the violated constraint s is an input constraint (denoted as $N'_{C_{in}}$). s is also violated in N'', since $N'_{C_{in}} = N''_{C_{in}}$ and N'' receives the same input.
2. If the violation is in the output layer: s is an output constraint (denoted as $N'_{C_{out}}$), and s is also violated in N'', because $N''_{C_{out}} = N'_{C_{out}}$ and the output of a refined network is smaller than that of the original (and $y'' < y' < c$).
3. If there is a violation in the preceding layers: $s \in N'_{C_{pre}}$, then from the first condition we get that $s \in N''_{C_{pre}}$. Hence, there is a violation also in N'', because the values in its preceding layers are equal.
4. If there is a violation in the abstract neuron: the violated constraint is $s \in N'_{C^{inc}_{abs}} \cup N'_{C^{dec}_{abs}}$. There are 2 cases for the neuron where the violation occurs, denoted as u (with refined neurons u_1, u_2):
   - if the type of neuron u is `inc`, then by definition $N'_{C^{dec}_{abs}} = \emptyset$. In this case, $s \in N'_{C^{inc}_{abs}}$ and $N'_{C^{inc}_{abs}} \neq \emptyset$, so from the second condition we get $N''_{C^{inc}_{abs}} \neq \emptyset$, and by definition $N''_{C^{inc}_{ref}} = \{u_1 = active, u_2 = active\}$. The violation of s means that $u < 0$, so from Lemma 2 we get that $u_1 < 0 \vee u_2 < 0$, and at least one of s_1, s_2 is violated in N''.
   - if the type of neuron u is `dec`, then by definition $N'_{C^{inc}_{abs}} = \emptyset$. In this case $s \in N'_{C^{dec}_{abs}}$ and $N'_{C^{dec}_{abs}} \neq \emptyset$, so from the second condition we get $N''_{C^{dec}_{abs}} \neq \emptyset$, and by definition $N''_{C^{dec}_{ref}} = \{u_1 = inactive, u_2 = inactive\}$. The violation of s means that $u > 0$, so from Lemma 3 we get that $u_1 > 0 \vee u_2 > 0$, and at least one of s_1, s_2 is violated in N''.
5. If the violation occurs in a consecutive layer: $s \in N'_{C^{inc}_{post}} \bigcup N'_{C^{dec}_{post}}$, then from the third condition we get that $s \in N''_{C^{inc}_{post}} \bigcup N''_{C^{dec}_{post}}$ (the condition implies that $N'_{C^{inc}_{post}} \bigcup N_{C^{dec}_{post}} = N''_{C^{inc}_{post}} \bigcup N''_{C^{dec}_{post}}$ because the 2 sets of neurons are equal and the constraints for each neuron are equal).
   - if any constraint in $N'_{C_{in}}, N'_{C_{out}}, N'_{C_{pre}}, N'_{C_{abs}}$ is violated, we have already shown above that a corresponding constraint in $N''_{C_{in}}, N''_{C_{out}}, N''_{C_{pre}}, N''_{C_{abs}}$ is violated.
   - otherwise, denote the violated constraint's neuron with p
   - if p is an `inc` neuron, then the violation is that $p < 0$. After refinement, p decreases, so still $p < 0$ and s is violated again.
   - if p is a `dec` neuron, then the violation is that $p > 0$. After refinement, p increases, so still $p > 0$ and s is violated again.

□

When all the conditions are met and Theorem 1 is applicable, it implies that the existence of a satisfying assignment within the specific branch of the search tree corresponding to the refined network must entail the existence of a satisfying assigning within the matching branch of the abstract network's search tree. However, we already know that this is impossible; and so it follows that that branch can soundly be skipped. In order to prove the theorem, we first require the two following lemmas—each of which corresponds to one of the two cases handled by the theorem.

Lemma 2 *Let v be an abstract `inc` node, let v_1 and v_2 be refined nodes that correspond to v, and let $\vec{x}$ be an input to the DNN. If v takes a negative value when the network is evaluated on $\vec{x}$, then v_1 or v_2 (or both) must also be assigned a negative value when the refined network is evaluated on $\vec{x}$.*

Proof Outline We give an overview of the proof for this lemma, using the network snippet from Fig. 5. Generalizing the proof for an arbitrary network can be achieved in a straightforward way.

Let us observe nodes $v_{2,1}$ and $v_{2,2}$ in Fig. 5b. These nodes are refined nodes, corresponding to node $v_{2,1}$ in Fig 5a. We need to show that the following implication holds:

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

Because the values x_1, x_2 are the results of ReLUs, they are nonnegative by definition. Thus, we can consider 4 cases:

1. If $x_1 = 0, x_2 = 0$, the implication trivially holds.
2. If $x_1 = 0, x_2 > 0$, then $x_2 \cdot \max(c, d) < 0$, and so $c, d < 0$. It follows 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 again 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

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

   We denote $a' = x_1 \cdot a,\ b' = x_1 \cdot b$ and $c' = x_2 \cdot c,\ d' = x_2 \cdot d$. The lemma transforms into:

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

- If $a' \geq b'$, then $a' = \max(a', b')$ and $a' + \max(c', d') < 0$. It follows that

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

  as needed.
- If $a' < b'$, then $b' = \max(a', b')$ and $b' + \max(c', d') < 0$. It follows that

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

  again as needed.

□

For `inc` neurons, the correctness of Theorem 1 follows from Lemma 2. For the `dec` case, we require also the following, symmetrical lemma:

Lemma 3 *Let v be an abstract* `dec` *node, let v_1 and v_2 be refined nodes that correspond to v, and let $\vec{x}$ be an input to the DNN. If v takes a positive value when the network is evaluated on $\vec{x}$, then v_1 or v_2 (or both) must also be assigned a positive value when the refined network is evaluated on $\vec{x}$.*

Proof Again, we explain how to prove the lemma using the network snippet from Fig. 5; this proof can be generalized to any network in a straightforward way. Observe nodes $v_{2,1}$ and $v_{2,2}$ in Fig. 5b, which are nodes refined from node $v_{2,1}$ in Fig. 5a. We need to prove that the following implication holds:

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

The values of x_1, x_2 are the outputs of ReLUs, and so are nonnegative. 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 $x_2 \cdot \min(c, d) > 0$, 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

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

   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:

   $$\min(a', b') + \min(c', d') > 0 \Rightarrow a' + c' > 0 \vee b' + d' > 0$$

- If $a' \leq b'$, then $a' = \min(a', b')$ and $a' + \min(c', d') > 0$. We then get that

  $$b' + d' \geq a' + d' \geq a' + \min(c', d') > 0$$

  as needed.
- If $a' > b'$, then $b' = \min(a', b')$ and $b' + \min(c', d') > 0$. We then get that

  $$a' + c' \geq b' + c' \geq b' + \min(c', d') > 0$$

  again as needed.

□

Figure 3 illustrates the results of applying Theorem 1, as part of verifying of our running example (Fig. 6). Each rectangle in the figure represents a single verification query, whereas blue lines indicate abstraction steps. The interior of each rectangle depicts the verifier's search tree. Triangles represent sub-trees—and red triangles represent sub-trees in which the verifier is able to deduce that a satisfying assignment does not exist. As the figure shows, when the verifier solves the query in the bottom rectangle it discovers an `UNSAT` sub-tree, and this sub-tree meets the conditions of the Theorem. This fact allows the verifier to conclude that another sub-tree, which is part of another rectangle/query, is also `UNSAT`. This is indicated by a green arrow. Specifically, the verifier discovers that setting $v_{2,1+2}$ to *active* would result in `UNSAT`, and then, it deduces that setting both $v_{2,1}$ and $v_{2,2}$ to *active* is bound to also produce an `UNSAT` result.

Multiple Refinement Steps. So far, we have focused solely on populating Γ when performing a single refinement step. However, a need often arises to adjust Γ across multiple refinement steps. When this happens, each invocation of Theorem 1 adds yet another CNF clause to the formula being constructed in Γ. In addition, some renaming and bookkeeping are required, because neuron identifiers change as refinement is performed: intuitively, whenever abstract neuron v is split into neurons v_1 and v_2, the literal representing v must be replaced with the clause $v_1 \vee v_2$. We formalize these notions in Sect. 5; the procedure's soundness can be proven by repeatedly invoking Theorem 1.

5 Adding residual reasoning to reluplex

In contrast to previous attempts at applying abstraction/refi nement within DNN verification [6, 21, 43], residual reasoning requires that the DNN verifier be instrumented, in order to populate, and later use, Γ. Next, we describe such an instrumentation for Reluplex [33], which is the core verification algorithm used in the state-of-the-art Marabou verifier [35].

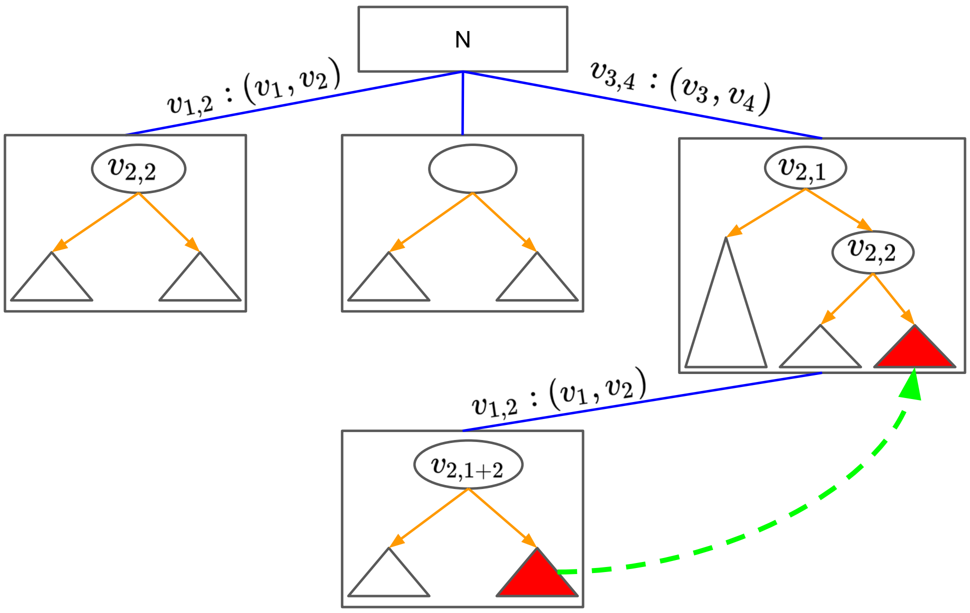

Fig. 6 (From [20]) Applying Theorem 1 while solving the query from Fig. 3

Reluplex is a sound and complete verification algorithm, which employs case-splitting—as discussed in Sect. 2. It also employs various heuristics for pruning the search space and reducing the number of splits performed [51, 52], and it has also been used within abstraction/refinement based-schemes [21], rendering it a natural candidate for using residual reasoning. We termed our enhanced version of Reluplex AR^4, for abstraction/refinement with residual reasoning for Reluplex.

For our ends, it is convenient to consider Reluplex as a collection of derivation rules, which are applied according to some implementation-specific strategy. The most relevant parts of this calculus of derivation rules, borrowed from Katz et al. [33], appear (in simplified form) in Fig. 7. Other rules, most notably those that pertain to the technical aspects of solving linear programs, were omitted for brevity.

Internally, the Reluplex algorithm represents a verification query as a collection of linear equalities, as well as lower and upper bounds, over a set of variables. Separately, it maintains a set of ReLU constraints. A Reluplex *configuration* over variable set $\mathcal{X}$ is either the distinguished symbols SAT or UNSAT, or a tuple $\langle T, l, u, \alpha, R\rangle$, where T, the *tableau*, contains the collection of linear equations; l, u are mappings from each variable $x \in \mathcal{X}$ to its lower and upper bound, respectively; α, the *assignment*, maps each variable $x \in \mathcal{X}$ to some real value; and R is the collection of ReLU constraints, i.e., $\langle x, y\rangle \in R$ implies that $y = \text{ReLU}(x)$. As it solves a query, Reluplex often derives *tighter bounds*, by discovering smaller upper bounds or greater lower bounds for some of the variables.

Through these definitions, the rules depicted in Fig. 7 are interpreted as follows: the Failure rule is applicable whenever Reluplex discovers inconsistent bounds for a variable, which indicates that the query is UNSAT. The ReluSplit rule is applicable for any ReLU constraint with a yet unknown linear phase; and using it allows Reluplex to “guess” a linear phase for that ReLU constraint, by setting its input x’s upper bound to 0 (inactive), or x’s lower bound to 0 (active). The Success rule, which returns SAT, is applicable whenever Reluplex’s current configuration simultaneously satisfies all constraints.

In order to allow support for AR^4, we propose to extend the Reluplex calculus with new rules, as depicted in Fig. 8. We introduce the context variable Γ and use it to store a valid CNF formula in order to assist the verifier. We also introduce Γ_A and Γ_B, which are two additional context variables, used for book-keeping. Specifically, we use Γ_A to store a mapping between abstract neurons and their matching refined neurons, i.e., Γ_A is comprised of triples $\langle v, v_1, v_2\rangle$, each of which indicates that an abstract neuron v has been refined into neurons v_1 and v_2. We use Γ_B for storing past case splits, already performed by the verifier, which can be used in populating Γ when the verifier encounters an UNSAT branch. Given variable x of neuron v, we use $\mathcal{G}^{\text{inc}}(\Gamma_A, \Gamma_B, x)$ and $\mathcal{G}^{\text{dec}}(\Gamma_A, \Gamma_B, x)$ to denote the Boolean function that returns true if and only if the guard conditions needed for applying Theorem 1 hold (either the inc or dec conditions, depending on neuron v’s type).

The rules depicted in Fig. 8 are to be interpreted as follows. The AbstractionStep rule is used for merging pairs of neurons, and for creating the initial, abstract network. The RefinementStep rule is applicable when dealing with an already abstract network (as indicated by $\Gamma_A \neq \emptyset$), and initiates a refinement step by undoing the previous abstraction step. The ApplyAbstraction rule is applicable anytime; it generates an abstract network, in accordance with the infor-

Fig. 7 The derivation rules of Reluplex calculus (partial, simplified)

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

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

Fig. 8 Derivation rules for the AR^4 calculus

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

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

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

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

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

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

mation in Γ_A, and updates the relevant contexts to reflect this. The Success rule, taken from the original Reluplex calculus, is included also in the AR^4 calculus; however, we note that the distinguished SAT state that it reaches is applicable only to the current network, which can be an abstract network, and could thus signify that a spurious satisfying assignment has been reached. To circumvent this issue, we add a new rule, RealSuccess, used for checking whether a SAT result actually holds also for the original network. Consequently, in addition to SAT or UNSAT, the distinguished RealSAT state is also defined to be a terminal state for our calculus.

The Failure rule is new, and replaces the Reluplex rule with the same name. This rule is applicable whenever contradictory variable bounds are discovered. Apart from declaring that the UNSAT state has been reached, this rule also populates the Γ context variable with the current case-split history (stored in Γ_B), to be used in future pruning of the search space. The ReluSplit rule, which is similar to the Reluplex version, guesses a linear phase for one the ReLU constraints and also records that guess in the Γ_B context variable. Finally, the $\text{Prune}_{1/2}$ rules become applicable when the conditions required for applying Theorem 1 hold (distinguishing between the inc and dec cases). These rules trim the search tree and update the Γ context variable accordingly.
Side Procedures. We proceed to describe possible implementations for the side condition functions *CanAbstract*, *Abstract*, *UpdateContext* and *isRealSat* that appeared in the aforementioned calculus. Apart from the symbols $\mathcal{X}, T, l, u, \alpha$ and R, which were introduced earlier, we introduce one additional symbol, B, which signifies the set of basic variables [33]. Intuitively, basic variables are variables expressed as linear combinations of other, non-basic variables, used in solving linear programs [16]. We also use the symbols pos(v), neg(v), inc(v) and dec(v) to indicate that a neuron v has a particular type, as part of the abstraction process [21].

– CanAbstract (Algorithm 1) checks whether a pair of neurons can be merged, according to their assigned types. It also ensures that these variables' assignment has not yet changed as part of the verification process.

Algorithm 1 CanAbstract(v_1, v_2)
```
1: if $layer(v_1) = layer(v_2)$
     and $pos(v_1) \leftrightarrow pos(v_2)$
     and $inc(v_1) \leftrightarrow inc(v_2)$
     and $\{v_1^b, v_1^f, v_2^b, v_2^f\} \subseteq B$
     and $\exists \Gamma_A, \Gamma_B : X, T, \alpha, l, u, B =$ ApplyAbstraction
   $(X_0, T_0, \alpha_0, l_0, u_0, B_0, \Gamma_A, \Gamma_B)$ then
2:   return True
3: end if
4: return False
```

– Abstract (Algorithm 2) performs a sequence of abstraction steps. Each single abstraction step (Algorithm 3) is comprised of the merging of a pair neurons that are known to be part of the basis and share a layer and a type, provided that the current state is reproducible from the initial state by neuron merging only. For the merging operation, we use Algorithm 4, which unifies two variables in the basis, by replacing their corresponding rows in the tableau by a single row, representing the unified neuron variable. More specifically, the new row includes 0's in the cells of any existing slack variable (line 5), 1 in the cell of the new variable (line 7) and the cell of the new slack variable (line 23), and a relevant aggregated value (max / min according to inc/dec type, replaced for bias variables) for every non-basic variable (lines 8-19).

Algorithm 2 Abstract($X_0, T_0, \alpha_0, l_0, u_0, B_0, \Gamma_A$)

```
1: $X, T, \alpha, l, u, B = X_0, T_0, \alpha_0, l_0, u_0, B_0$
2: for $v_{1,2}, v_1, v_2 \in \Gamma_A$ do
3:     $X := X \cup \{v_{1,2}^b, v_{1,2}^f\} \setminus \{v_1^b, v_1^f, v_2^b, v_2^f\}$
4:     $B := B \cup \{v_{1,2}^b, v_{1,2}^f\} \setminus \{v_1^b, v_1^f, v_2^b, v_2^f\}$
5:     $\alpha := \alpha \cup \{v_{1,2}^b = 0, v_{1,2}^f = 0\} \setminus \{v_1^b = 0, v_1^f = 0, v_2^b = 0, v_2^f = 0\}$
6:     $l := l \cup \{v_{1,2} > -\infty\} \setminus \{v_1 > -\infty, v_2 > -\infty\}$
7:     $u := u \cup \{v_{1,2} < \infty\} \setminus \{v_1 < \infty, v_2 < \infty\}$
8:     $T := UnifyBasic(v_{1,2}, (v_1, v_2), T)$
9: end for
10: return $X, T, \alpha, l, u, B$
```

Algorithm 3 AbstractStep($X, T, \alpha, l, u, B, v_1, v_2$)

```
1: $X := X \cup \{v_{1,2}^b, v_{1,2}^f\} \setminus \{v_1^b, v_1^f, v_2^b, v_2^f\}$
2: $B := B \cup \{v_{1,2}^b, v_{1,2}^f\} \setminus \{v_1^b, v_1^f, v_2^b, v_2^f\}$
3: $\alpha := \alpha \cup \{v_{1,2}^b = 0, v_{1,2}^f = 0\} \setminus \{v_1^b = 0, v_1^f = 0, v_2^b = 0, v_2^f = 0\}$
4: $l := l \cup \{v_{1,2} > -\infty\} \setminus \{v_1 > -\infty, v_2 > -\infty\}$
5: $u := u \cup \{v_{1,2} < \infty\} \setminus \{v_1 < \infty, v_2 < \infty\}$
6: $T := UnifyBasic(v_{1,2}, (v_1, v_2), T)$
7: return $X, T, \alpha, l, u, B$
```

– The UpdateContext procedure sets ($\Gamma_B = \emptyset$) in order to clear the case-splitting context. It also updates the clauses within Γ to use new variables: any variable that represents an inc node, $\neg r$, is replaced with the clause $\neg r_1 \vee \neg r_2$; and any variable that represents a dec node, r, is replaced with $r_1 \vee r_2$.
– The $isRealSat$ procedure (Algorithm 5) checks whether a counterexample, represented by an assignment α, is a true counterexample in the original network.

Implementation Strategy. The derivation rules that appear in Fig. 8 specify a set of "legal moves" that the AR^4 can perform. It is guaranteed that by applying these moves, the verification process will be sound. Still, when implementing this framework, the derivation rules much be applied according to some *strategy*. We next describe the strategy used in our proof-of-concept implementation.

At first, we apply the AbstractionStep rule to saturation, with the goal of reaching as small an abstract network as possible. Next, we apply the ApplyAbstraction rule once, in order to initialize the context variables. Next, we enter the main loop of abstraction-based verification: we repeatedly apply the Reluplex core rules, according to existing strategies [35], with the modification that whenever the ReluSplit rule is applied, it is immediately followed by an application of Prune_1 and Prune_2, if possible. The Success and Failure rules are applied as in the original Reluplex, with RealSuccess being applied immediately after Success, if this is possible; if not, we apply the RefinementStep rule, and repeat the process. Finally, we also attempt to apply Prune_1 and Prune_2 after each application of Failure, in order to update Γ.

Algorithm 4 UnifyBasic($v_{1,2}, (v_1, v_2), T$)

```
1: $row_{v_1}, row_{v_2}$ = the rows of $T$ where $v_1, v_2$ are basic variables
2: $row_{v_{1,2}} = [0, \ldots, 0]$
3: for $i \in 0, \ldots, len(X)$ do
4:     if the $i$'th variable is a slack variable then
5:         $func = 0$
6:     else if the $i$'th variable corresponds to $v_{1,2}$ then
7:         $func = 1$
8:     else if $i$ is a bias variable then
9:         if the $i$'th variable corresponds to an increasing neuron then
10:            $func = min$
11:        else
12:            $func = max$
13:        end if
14:    else
15:        if the $i$'th variable corresponds to an increasing neuron then
16:            $func = max$
17:        else
18:            $func = min$
19:        end if
20:    end if
21:    $row_{v_{1,2}}[i] = func(row_{v_1}[i], row_{v_2}[i])$
22: end for
23: add cells for $v_{1,2}$ and its slack variable (1 in $row_{v_{1,2}}$, 0 in other rows)
24: remove cells of $v_1, v_2$ and their slack variables from all rows
25: remove $row_{v_1}, row_{v_2}$ from $T$, append $row_{v_{1,2}}$ to $T$
```

Algorithm 5 IsRealSAT(α)

```
1: extract input layer values from $\alpha$
2: evaluate the original network on these input values
3: compute the original network's outputs
4: return whether the computed output values satisfy the property
```

6 Experiments and evaluation

In order to evaluate our approach, we used a proof-of-concept implementation of AR^4. As a baseline for comparison, we

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	–

used the only framework currently available that supports CEGAR-based verification of neural networks—namely, the implementation of [21], which extends Marabou. We then applied both tools to verify a set of benchmarks over the family of 45 ACAS Xu DNNs for airborne collision avoidance [32] (each of which contained 310 neurons, spread across 8 layers). In our experiments, we verified a set of 4 safety properties, and also 20 adversarial robustness properties over the ACAS Xu networks, yielding a total of 1080 benchmarks. From these experiments we recorded, for each tool, the runtime (including instrumentation time), the number of properties verified successfully within the allowed two-hour timeout, and also the number of case splits performed by each tool. We conducted our experiments on x86-64 Gnu/Linux-based machines, using a single Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz core. The code that we used is publicly available online.[1]

The results of our experiments appear in Table 1, with Fig. 9 used for visualizing the advantages afforded by AR^4 when compared to AR. The AR^4 tool timed out on 18.4% fewer benchmarks and successfully solved 188 benchmarks more quickly than its AR counterpart, whereas AR proved faster than AR^4 in only 119 instances. We observe that in these comparisons, we considered experiments in which both tools completed their execution within 5 s of each other as ties. We also observe that the residual reasoning mechanism was able to curtail the search space significantly. On average, AR^4 had to traverse 5.634 search-tree states per experiment, whereas AR traversed 7.178 states on average—a 21.5% decrease.

In spite of the advantages that AR^4 affords, it does not always outperform AR—due to the cost of instrumenting the verifier, which is sometimes significant. In our evaluation, the verifier spent 82 s on average in order to execute the instrumentation code, out of an average total runtime of 885 s—nearly a 10% portion, which is significant. In order to reduce instrumentation costs, moving forward we intend to improve the engineering of our tool, e.g., by improving the implementation of its internal unit-propagation mechanism, through the use of watch literals [10].

7 Related work

Many modern DNN verification engines leverage key principles from SMT and SAT solving [19, 28, 33, 35, 41], abstract interpretation [23, 40, 45, 50], mixed integer linear programming [12, 18, 19, 48], and other domains. Many of these approaches make use of case-splitting and could potentially benefit from residual reasoning.

In our evaluation, we used the Marabou DNN verifier [35, 51]. Marabou is a state-of-the-art DNN verifier, which uses a native Simplex solver, combined with abstraction and abstract interpretation techniques [15, 21, 42, 45, 49, 54], proof-production capabilities [29], advanced splitting heuristics [52], DNN optimization [47], and support for varied activation functions [3]. Additionally, Marabou has been applied to a variety of verification-based tasks, such as verifying recurrent networks [31] and DRL-based systems [1, 2, 22, 36], network repair [25, 44], network simplification [24, 38], ensemble selection [4], and explainable AI [9]. Integrating our approach with additional DNN verifiers is left for future work.

Abstraction/refinement schemes are known to be highly successful in verifying various hand-crafted systems [14]. More recently, they have also been promising attempts to apply them within the context of DNN verification by merging neurons [21], deleting neurons [6] and generating interval neural networks [43]. Of these, the framework of Ashok et al. [6] is potentially unsound, whereas the work of Prabhakar et al. [43] generates abstract artifacts that are not standard neural networks, and thus, the work closest to ours is that of Elboher et al. [21]. To the best of our knowledge, our approach is the first that applies residual reasoning to verify DNNs.

There are various optimizations that can enhance CEGAR-based approaches. These include "tightening" the examined property while abstracting the network [15]; more sophisticated neuron deletion schemes that leverage linear programming [13]; and incorporating testing-based methods into the CEGAR-based verification process [56]. Combining these enhancements with our scheme is left for future work.

[1] https://zenodo.org/record/8224307.

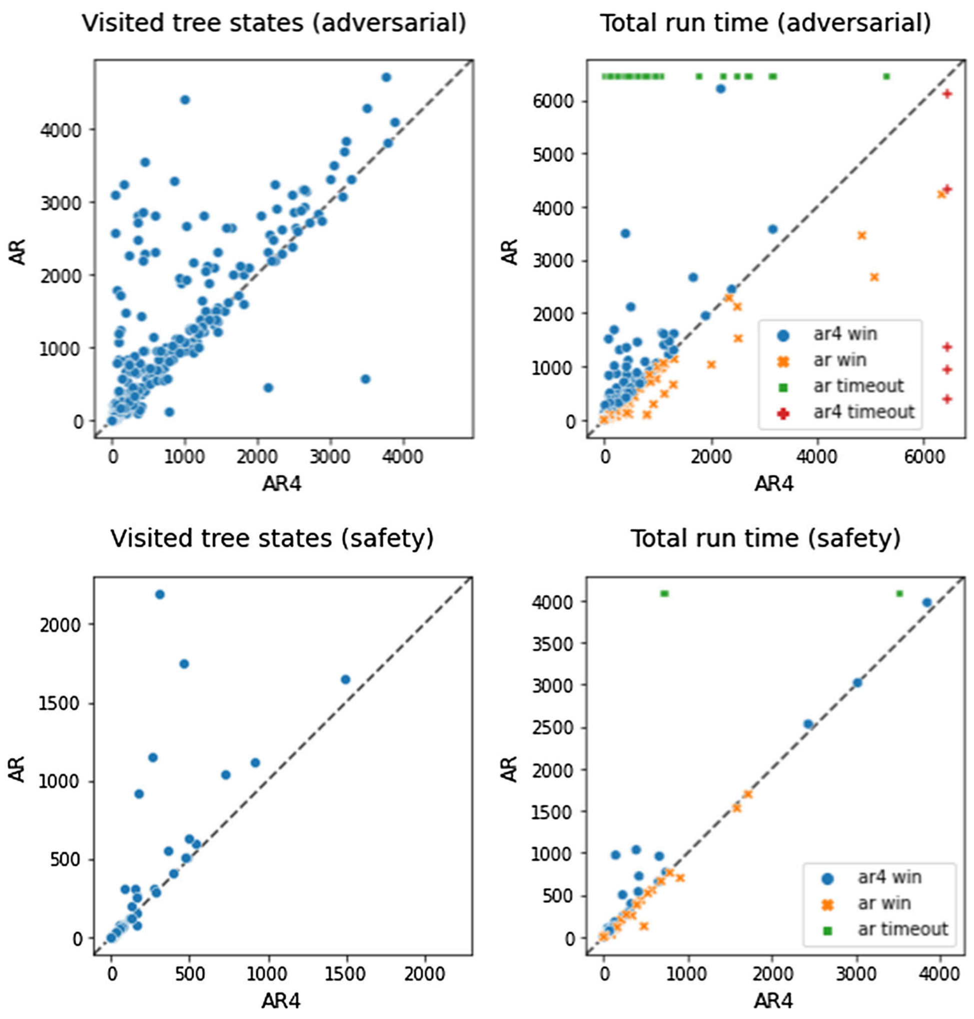

Fig. 9 (From [20]) Comparing AR^4 and AR

8 Conclusion

With the increasing integration of DNNs into safety-critical software systems, it is becoming crucial to improve the scalability of DNN verification. Abstraction/refinement schemes could potentially play a significant role in this endeavor, but in some cases they result in a significant amount of redundant work for the underlying verifier. The residual-reasoning-based approach that we propose advocate here can eliminate some of this redundancy, resulting in speedier verification. We consider our work here as a step toward tapping the significant potential of abstraction/refinement schemes within the broad context of DNN verification.

Moving forward, our plan is to improve the engineering of our AR^4 tool; and also to integrate it with additional abstraction/refinement approaches for DNN verification [6].

Acknowledgements 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. Amir, G., Corsi, D., Yerushalmi, R., Marzari, L., Harel, D., Farinelli, A., Katz, G.: Verifying learning-based robotic navigation systems. In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 607–627 (2023)
2. Amir, G., Schapira, M., Katz, G.: Towards scalable verification of deep reinforcement learning. In: Proceedings of the 21st International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 193–203 (2021)
3. Amir, G., Wu, H., Barrett, C., Katz, G.: An SMT-based approach for verifying binarized neural networks. In: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 203–222 (2021)
4. Amir, G., Zelazny, T., Katz, G., Schapira, M.: Verification-aided deep ensemble selection. In: Proceedings of the 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 27–37 (2022)
5. Angelov, P., Soares, E.: Towards explainable deep neural networks (xDNN). Neural Netw. **130**, 185–194 (2020)
6. Ashok, P., Hashemi, V., Kretinsky, J., Mühlberger, S.: DeepAbstract: neural network abstraction for accelerating verification. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 92–107 (2020)

7. Azzopardi, S., Colombo, C., Pace, G.: A technique for automata-based verification with residual reasoning. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 237–248 (2020)
8. Bak, S., Liu, C., Johnson, T.: The second international verification of neural networks competition (VNN-COMP 2021): summary and results (2021). Technical Report. arXiv:2109.00498
9. Bassan, S., Katz, G.: Towards formal XAI: formally approximate minimal explanations of neural networks. In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 187–207 (2023)
10. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. IOS Press, Amsterdam (2009)
11. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., Zieba, K.: End to end learning for self-driving cars (2016). Technical Report. arXiv:1604.07316
12. Bunel, R., Turkaslan, I., Torr, P., Kohli, P., Kumar, M.: Piecewise linear neural network verification: a comparative study (2017). Technical Report. arXiv:1711.00455
13. Chau, C., Kretinsky, J., Mohr, S.: Syntactic vs semantic linear abstraction and refinement of neural networks (2023). Technical Report. arXiv:2307.10891
14. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceeding of the 12th International Conference on Computer Aided Verification (CAV), pp. 154–169 (2000)
15. Cohen, E., Elboher, Y.Y., Barrett, C., Katz, G.: Tighter abstract queries in neural network verification. In: Proceedings of the 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp. 124–143 (2023)
16. Dantzig, G.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
17. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding (2018). Technical Report. arXiv:1810.04805
18. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks. In: Proceedings of the 10th NASA Formal Methods Symposium (NFM), pp. 121–138 (2018)
19. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 269–286 (2017)
20. Elboher, Y., Cohen, E., Katz, G.: Neural network verification using residual reasoning. In: Proceedings of the 20th International Conference on Software Engineering and Formal Methods (SEFM), pp. 173–189 (2022)
21. Elboher, Y., Gottschlich, J., Katz, G.: An abstraction-based framework for neural network verification. In: Proceedings of the 32nd International Conference on Computer Aided Verification (CAV), pp. 43–65 (2020)
22. Eliyahu, T., Kazak, Y., Katz, G., Schapira, M.: Verifying learning-augmented systems. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp. 305–318 (2021)
23. 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 the 39th IEEE Symposium on Security and Privacy (S&P) (2018)
24. Gokulanathan, S., Feldsher, A., Malca, A., Barrett, C., Katz, G.: Simplifying neural networks using formal verification. In: Proceedings of the 12th NASA Formal Methods Symposium (NFM), pp. 85–93 (2020)
25. Goldberger, B., Adi, Y., Keshet, J., Katz, G.: Minimal modifications of deep neural networks using verification. In: Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp. 260–278 (2020)
26. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
27. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
28. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification (CAV), pp. 3–29 (2017)
29. Isac, O., Barrett, C., Zhang, M., Katz, G.: Neural network verification with proof production. In: Proceedings 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 38–48 (2022)
30. Isac, O., Zohar, Y., Barrett, C., Katz, G.: DNN verification, reachability, and the exponential function problem. In: Proceedings of the 34th International Conference on Concurrency Theory (CONCUR) (2023)
31. Jacoby, Y., Barrett, C., Katz, G.: Verifying recurrent neural networks using invariant inference. In: Proceedings of the 18th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 57–74 (2020)
32. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proceedings of the 35th Digital Avionics Systems Conference (DASC), pp. 1–10 (2016)
33. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Proceedings of the 29th International Conference on Computer Aided Verification (CAV), pp. 97–117 (2017)
34. Katz, G., Barrett, C., Dill, D., Julian, K., Kochenderfer, M.: Reluplex: a calculus for reasoning about deep neural networks. Formal Methods in System Design (FMSD), (2021)
35. Katz, G., Huang, D., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D., Kochenderfer, M., Barrett, C.: The Marabou framework for verification and analysis of deep neural networks. In: Proceedings of the 31st International Conference on Computer Aided Verification (CAV), pp. 443–452 (2019)
36. Kazak, Y., Barrett, C., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: Proceedings of the 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI), pp. 83–89 (2019)
37. Kim, B., Kim, H., Kim, K., Kim, S., Kim, J.: Learning not to learn: training deep neural networks with biased data. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 9004–9012 (2019)
38. Lahav, O., Katz, G.: Pruning and slicing neural networks using formal verification. In: Proceedings of the 21st International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 183–192 (2021)
39. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.: Algorithms for verifying deep neural networks (2020). Technical Report. arXiv:1903.06758
40. 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 the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2022)
41. Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., Walsh, T.: Verifying properties of binarized deep neural networks (2017). Technical Report. arXiv:1709.06662
42. Ostrovsky, M., Barrett, C., Katz, G.: An abstraction-refinement approach to verifying convolutional neural networks. In: Pro-

ceedings of the 20th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 391–396 (2022)
43. Prabhakar, P., Afzal, Z.: Abstraction based output range analysis for neural networks (2020). Technical Report. arXiv:2007.09527
44. Refaeli, I., Katz, G.: Minimal multi-layer modifications of deep neural networks. In: Proceedings of the 5th Workshop on Formal Methods for ML-Enabled Autonomous Systems (FoMLAS) (2022)
45. Singh, G., Gehr, T., Puschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2019)
46. Song, H., Kim, M., Park, D., Shin, Y., Lee, J.-G.: End to end learning for self-driving cars (2020). Technical Report. arXiv:2007.08199
47. Strong, C., Wu, H., Zeljic', A., Julian, K., Katz, G., Barrett, C., Kochenderfer, M.: Global optimization of objective functions represented by ReLU networks. Mach. Learn. **12**, 3685–3712 (2023)
48. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming (2017). Technical Report. arXiv:1711.07356
49. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Security Symposium (2018)
50. Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., Kolter, Z.: Beta-CROWN: efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. In: Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS), (2021)
51. Wu, H., Ozdemir, A., Zeljić, A., Irfan, A., Julian, K., Gopinath, D., Fouladi, S., Katz, G., Păsăreanu, C., Barrett, C.: Parallelization techniques for verifying neural networks. In: Proceedings of the 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 128–137 (2020)
52. Wu, H., Zeljić, A., Katz, K., Barrett, C.: Efficient neural network analysis with sum-of-infeasibilities. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 143–163 (2022)
53. Ying, X.: An overview of overfitting and its solutions. J. Phys. Conf. Ser. **1168**, 022022 (2019)
54. Zelazny, T., Wu, H., Barrett, C., Katz, G.: On reducing over-approximation errors for neural network verification. In: Proceedings of the 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 17–26 (2022)
55. Zhang, H.,Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, pp. 4944–4953 (2018)
56. Zhao, Z., Zhang, Y., Chen, G., Song, F., Chen, T., Liu, J.: CLEVEREST: accelerating CEGAR-based neural network verification via adversarial attacks. In: Proceedings of the 29th Static Analysis Symposium (SAS), (2022)

Yizhak Yisrael Elboher is a PhD student at The Hebrew University of Jerusalem, Israel. He received his M.Sc. from Tel-Aviv University and his B.Sc. from The Hebrew University of Jerusalem. His resea rch deals with formal verification of neural networks at scale, by utilizing abstraction and refinement techniques in order to optimize the verification process.

Elazar Cohen has a Master's degree in Computer Science from the Hebrew University of Jerusalem. His research deals with verification of neural networks, and in particular optimizing abstraction-based verification. He lives with his wife and 2-year-old daughter in Jerusalem.

Guy Katz is an associate professor at the Hebrew University of Jerusalem, Israel. He received his PhD at the Weizmann Institute of Science in 2016. His research interests lie at the intersection between Formal Methods and Software Engineering and in particular in the application of formal methods to software systems with components generated via machine learning.