Self-correcting Neural Networks for Safe Classification

Klas Leino[1], Aymeric Fromherz[2], Ravi Mangal[1(✉)], Matt Fredrikson[1], Bryan Parno[1], and Corina Păsăreanu[1,3]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
kleino@andrew.cmu.edu; rmangal@andrew.cmu.edu; mfredrik@cmu.edu; parno@cmu.edu; pcorina@andrew.cmu.edu
[2] Inria Paris, 75012 Paris, France
aymeric.fromherz@inria.fr
[3] NASA Ames, Moffett Field, CA 94035, USA

Abstract. Classifiers learnt from data are increasingly being used as components in systems where safety is a critical concern. In this work, we present a formal notion of safety for classifiers via constraints called *safe-ordering constraints*. These constraints relate requirements on the order of the classes output by a classifier to conditions on its input, and are expressive enough to encode various interesting examples of classifier safety specifications from the literature. For classifiers implemented using neural networks, we also present a run-time mechanism for the enforcement of safe-ordering constraints. Our approach is based on a *self-correcting layer*, which provably yields safe outputs regardless of the characteristics of the classifier input. We compose this layer with an existing neural network classifier to construct a *self-correcting network* (SC-Net), and show that in addition to providing safe outputs, the SC-Net is guaranteed to preserve the classification accuracy of the original network whenever possible. Our approach is independent of the size and architecture of the neural network used for classification, depending only on the specified property and the dimension of the network's output; thus it is scalable to large state-of-the-art networks. We show that our approach can be optimized for a GPU, introducing run-time overhead of less than 1 ms on current hardware—even on large, widely-used networks containing hundreds of thousands of neurons and millions of parameters. Code available at github.com/cmu-transparency/self-correcting-networks.

Keywords: Safety · Run-time enforcement · Machine learning · Neural networks · Verification

O. Isac et al. (Eds.): NSV 2022/FoMLAS 2022, LNCS 13466, pp. 96–130, 2022.
https://doi.org/10.1007/978-3-031-21222-2_7

1 Introduction

Classifiers in the form of neural networks are being deployed as components in many safety- and security-critical systems, such as autonomous vehicles, banking systems, and medical diagnostics. A well-studied example is the ACAS Xu networks [20], which provide guidance to an airborne collision avoidance system for commercial aircraft. Unfortunately, standard network training approaches will typically produce models that are accurate but unsafe [29,33]. The ACAS Xu networks, in particular, have been shown [21] to violate safety properties formulated by the developers [20].

What are safety properties for classifiers? Classifiers implemented as neural networks are programs of type $\mathbb{R}^n \to \mathbb{R}^m$, where typically the index of the maximum element of the output m-tuple represents the predicted class. Such classifiers also give an order on the classes, from most likely to least, represented by the order on indices induced by sorting the elements (also referred to as *logits*) of the tuple, and in a variety of domains, systems with classifier components may use this ordering, in addition to the top predicted class, for downstream decision-making.

The ACAS Xu classifiers are an example of a domain where ordering matters. They map sensor readings about the physical state of the aircraft to horizontal maneuver advisories. The sensor readings are imperfect, and the system only has access to a distribution function (or, alternatively, a set of samples) that assigns probability $b(s)$ to being in state s. To issue a maneuver guidance in real time, at each time-step, the system finds the maneuver that maximizes $\sum_s Q(s)_a b(s)$ where $Q(s)_a$ is the value assigned by the neural classifier to maneuver a in state s. As a consequence, the order of the classes, in addition to the top class, are relevant when defining safety properties of ACAS Xu networks.

Another example domain is image classification, where popular datasets, such as CIFAR-100 and ImageNet, have classes with hierarchical structure (e.g., CIFAR-100 has 100 classes with 20 superclasses). Consider a client of an image classifier that averages the logit values over a number of samples for classes that appear in top-k positions and chooses the class with the highest average logit value, due to imperfect sensor information. A reasonable safety property is to require that the chosen class shares its superclass with at least one of the top-1 predictions. This in turn requires reasoning over the order of the classes, and not just the top class.

More generally, the ordering of the logits conveys information about the neural classifier's 'belief' in what the true class is. Under this interpretation, it is natural to express safety constraints on the class order. On the other hand, the exact logit values may be less meaningful, given the approximate nature of neural networks and the fact that logit values are not typically calibrated to any particular value.

Motivated by these observations, we define safety property specifications for classifiers via constraints that we refer to as *safe-ordering constraints*. We argue that these constraints are general enough to encompass the meaningful safety

specifications defined for the ACAS Xu networks [21], as well as those used in other safety verification and repair efforts [29,40]. Formally, safe-ordering constraints can specify non-relational safety properties [8] of the form $P \implies Q$, where P is a precondition, expressed as a decidable formula over the classifier's input, and Q is a postcondition, expressed as a statement over its output in the theory of totally ordered sets.

We note that many safety specifications provided by experts are *underconstrained* [21]; i.e., they say what the classifier *should not do* but not what it *should* do. As a specific example, one of the ACAS Xu safety properties roughly states that if an oncoming aircraft is directly ahead and is moving toward our aircraft, then the clear-of-conflict advisory should not have the maximal output from the network[1]. The decision as to what output should have the maximal value must be determined by learning from the input-output examples in the training data. Thus, even when safety specifications are provided, one still needs to perform training based on labeled data to build the classifier, whose performance is measured by computing its accuracy on a separate test set.

Enforcing Safe-Ordering Constraints. Standard approaches for learning neural classifiers will typically produce models that are accurate but unsafe [29,33]. As a result, considerable work has studied the safety of neural networks in general [3, 11,12,15,19,33,34,39], and the ACAS Xu networks in particular [21,29,33,40, 45].

Some approaches use abstract interpretation [15,39] or SMT solving [21] to verify safety properties of networks trained using standard techniques. Unfortunately, the scalability of these techniques remains a serious challenge for most neural-network applications. Furthermore, post-training verification does not address the problem of constructing safe networks to begin with. Retraining the network when verification fails is prohibitively expensive for modern networks [7,41,42], with no guarantees that the train-verify-train loop will terminate. On the other hand, approaches based on statically repairing the network can damage its accuracy (i.e., frequency of the top predicted class matching the 'true' class) on inputs outside the scope of a given safety specification [40]. An alternate approach is to change the learning algorithm such that it provably produces safe networks [29], but such approaches may not converge during training, thus not being able to provide a safety guarantee for the analyzed networks.

In contrast to these previous works, we propose a lightweight, run-time technique for ensuring that neural classifiers are *guaranteed* to satisfy their safe-ordering specifications and at the same time maintain the network's accuracy. Specifically, we describe a program transformer that, given a neural architecture f_θ (parameterized by θ) and a set of safe-ordering constraints Φ, produces a new architecture f_θ^Φ that satisfies the conjunction of Φ for all parameters θ. Viewing the neural network as a composition of layers, our transformer appends a dif-

[1] While [20] used the convention that the index of the minimal element of ACAS Xu networks is the top predicted advisory, in this paper we will use the more common convention of the maximal value's index.

ferentiable *self-correcting layer* (SC-Layer) to f_θ. This layer encodes a dynamic *check-and-correct* mechanism, so that when $f_\theta(x)$ violates Φ, the SC-Layer modifies the output to ensure safety. Differentiability of the mechanism also opens the possibility for the training procedure to take self-correction into account during training so that safer and more accurate models can be built, reducing the need for the run-time correction.

Consider again the ACAS Xu networks. Ideally, before deploying the system, we would like to certify that the trained neural classifiers meet their safety specifications. Since the training algorithms are not guaranteed to produce safe classifiers [29,33], and the train-verify-train loop may not terminate, one is likely to be forced to deploy uncertified classifiers. A run-time mechanism that flags safety violations can provide some assurance, but for a real-time, unmanned system like ACAS Xu, throwing exceptions during operation and aborting the computation is not acceptable. Instead, to ensure safe operation without interruptions, we propose to correct the outputs of the classifier whenever necessary.

Our approach is similar in spirit to those that dynamically correct errors in long-running programs caused by traditional software issues like division-by-zero, null dereference, and others [5,22,30,36–38], as well as dynamic check-and-correct mechanisms employed by controllers, referred to as shields [2,6,46]. A check-and-correct mechanism may be impractical for arbitrary classifier safety specifications, as they may require solving arbitrarily complex constraint-satisfaction problems. We show that this is not the case for safe-ordering constraints, and that the solver needed for these constraints can be efficiently embedded in the correction layer.

We note that when correcting the neural network output to enforce safety, we still need to preserve its accuracy. To address the issue, we define a property, *transparency*, which ensures that the correction mechanism has no negative impact on the network's accuracy. Transparency requires that the predicted top class of the original network f_θ be retained whenever it is consistent with at least one ordering allowed by Φ. However, if Φ is inconsistent with the "correct" class specified by the data, then it is impossible for the network to be safe without harming accuracy, and the correction prioritizes safety. We prove that our SC-Layer guarantees transparency. More generally, our correction mechanism tries to retain as much of the original class order as possible.

Finally, while the SC-Layer achieves safety without negatively impacting accuracy, it necessarily adds computational overhead each time the network is executed. We design the SC-Layer, including the embedded constraint solver, to be both vectorized and differentiable, allowing the efficient implementation of our approach within popular neural network frameworks. We also present experiments that evaluate how the overhead is impacted by several key factors. We show that the cost of the SC-Layer depends solely on Φ and the length m of the output vector, and thus, is *independent* of the size or complexity of the underlying neural network. On three widely-used benchmark datasets (ACAS Xu [21], Collision Detection [12], and CIFAR-100 [24]), we show that this overhead is small in real terms (0.26–0.82 ms), and does not pose an impediment to

practical adoption. In fact, because the overhead is independent of network size, its impact is less noticeable on larger networks, where the cost of evaluating the original classifier may come to dominate that of the correction. To further characterize the role of Φ and m, we use synthetic data and random safe-ordering constraints to isolate the effects that the postcondition complexity and number of classes have on network run time. While these structural traits of the specified safety constraint can impact run time—the satisfiability of general ordering constraints is NP-complete [16]—our results suggest it will be rare in practice.

Hence, the main contributions of our work are as follows:

- We define *safe-ordering constraints*, as a generic way of writing safety specifications for neural network classifiers.
- We present a method for transforming feed-forward neural network architectures into safe-by-construction versions that are guaranteed to *(i)* satisfy a given set of safe-ordering constraints, and *(ii)* preserve or improve the empirical accuracy of the original model.
- We show that the SC-Layer can be designed to be both fully-vectorized and differentiable, which enables hardware acceleration to reduce run-time overhead, and facilitates its use during training.
- We empirically demonstrate that the overhead introduced by the SC-Layer is small enough for its deployment in practical settings.

2 Problem Setting

In this section, we formalize the concepts of *safe-ordering constraints* and *self-correction*. We begin by presenting background on neural networks and an illustrative application of safe-ordering constraints. We then formally define the problem we aim to solve, and introduce a set of desired properties for our self-correcting transformer.

2.1 Background

Neural Networks. A neural network, $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$, is a total function defined by an *architecture*, or composition of linear and non-linear transformations, and a set of *weights*, θ, parameterizing its linear transformations. As neither the details of a network's architecture nor the particular valuation of its weights are relevant to much of this paper, we will by default omit the subscript θ, and treat f as a black-box function. Neural networks are used as classifiers by extracting *class predictions* from the output $f(x) : \mathbb{R}^m$, also called the *logits* of a network. Given a neural network f, we use the upper-case F to refer to the corresponding neural classifier that returns the top class: $F = \lambda x.\, \text{argmax}_i\{f_i(x)\}$. For our purposes, we will assume that argmax returns a single index, $i^* \in [m]$[2]; ties may be broken arbitrarily.

[2] $[m] := \{0, \ldots, m-1\}$.

ACAS Xu: An Illustrative Example. We use ACAS Xu [20] as a running example to illustrate key aspects of the problem that our approach solves. The Airborne Collision Avoidance System X (ACAS X) [23] is a family of collision avoidance systems for both manned and unmanned aircraft. ACAS Xu, the variant for unmanned aircraft, is implemented as a large (2GB) numeric lookup table mapping the physical state of the aircraft and a neighboring object (an *intruder*) to horizontal maneuver advisories. The lookup table is indexed on the distance (ρ) between the aircraft and the intruder, the relative angle (θ) from the aircraft to the intruder, the angle (ψ) from the intruder's heading to the aircraft's heading, the speed of the aircraft (v_{own}), and of the intruder (v_{int}), and the time (τ) until loss of vertical separation. The possible advisories are either that no change is needed (or clear-of-conflict, COC), that the aircraft should steer weakly to the left, weakly to the right, strongly to the left, or strongly to the right.

As the table is too large for many unmanned avionics systems, [20] proposed the use of neural networks as a compressed, functional representation of the lookup table. The networks proposed by [20] are functions $f : \mathbb{R}^5 \to \mathbb{R}^5$; the value τ is discretized and 45 different neural networks are constructed, one for each combination of the previous advisory (a_{prev}) and discretized value of τ. Note that while the neural representation of the lookup table is an effective way to encode it on resource-constrained avionics systems, they are necessarily an approximation of the desired functionality, and may thus introduce unsafe behavior [21,29,33,40,45]. To address this, [21] proposed 10 safety properties, which capture requirements such as, "If the intruder is directly ahead and is moving towards the ownship, the score for *COC* will *not* be maximal." Our goal is to construct networks that are guaranteed to satisfy specifications like these.

2.2 Problem Definition

Definition 1 presents the safe-ordering constraints that we consider throughout the rest of the paper. Intuitively, they correspond to constraints on the relative ordering of a network's output values (a postcondition) with a predicate on the corresponding input (a precondition). As we will see in later sections, the precondition does not need to belong to a particular theory, and need only come with an effective procedure for deciding new instances.

Definition 1 (Safe ordering constraint). *Given a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, a safe-ordering constraint, $\phi = \langle P, Q \rangle$, is a precondition, P, consisting of a decidable proposition over $\mathbb{R}^n$, and a postcondition, Q, given as a Boolean combination of order relations between the real components of $\mathbb{R}^m$.*

$$
\begin{array}{ll}
\textit{precondition} & P := \textit{decidable proposition} \\
\textit{ordering literal} & q := y_i < y_j \ (0 \leq i, j < m) \\
\textit{ordering constraint} & Q := q \mid Q \wedge Q \mid Q \vee Q \\
\textit{safe-ordering constraint} & \phi := \langle P, Q \rangle \\
\textit{set of constraints} & \Phi := \cdot \mid \phi, \Phi
\end{array}
$$

Assuming a function, $\texttt{eval}P\texttt{:}\mathbb{R}^n \to \texttt{bool}$*, that decides* P *given* $x \in \mathbb{R}^n$*, notated as* $P(x)$*, and a similar* $\texttt{eval}$ *function for* Q*, we say* f *satisfies safe-ordering constraint* ϕ *at* x *iff* $P(x) \implies Q(f(x))$*. We use the shorthand* $\phi(x, f(x))$ *to denote this; and given a set of constraints* Φ*, we write* $\Phi(x, f(x))$ *to denote* $\forall \phi \in \Phi \,.\, \phi(x, f(x))$ *and* $\Phi(x)$ *to denote* $\bigwedge_{\langle P_i, Q_i \rangle \in \Phi \mid P_i(x)} Q_i$*.*

Two points about our definition of safe-ordering constraints bear mentioning. First, although postconditions are evaluated using the inequality relation from real arithmetic, we assume that $\forall x \,.\, i \neq j \implies f_i(x) \neq f_j(x)$, and thus specifically exclude equality comparisons between the output components. This is a realistic assumption in nearly all practical settings, and in cases where it does not hold, can be resolved with arbitrary tie-breaking protocols that perturb $f(x)$ to remove any equalities. Second, we omit explicit negation from our syntax, as it can be achieved by swapping the positions of the affected order relations; i.e., $\neg(y_i < y_j)$ is just $y_j < y_i$, as we exclude the possibility that $y_i = y_j$.

Sections 5.3 and 5.4 provide several concrete examples of safe-ordering constraints. Example 1 revisits the safety specification for ACAS Xu that was discussed in the previous section. Notice that this specification is an instance of the situation where *safety need not imply accuracy*, since it does not specify what category *should* be maximal; that choice must be learned from the training data.

Example 1 (Safety need not imply accuracy). Recall the specification described earlier: "If the intruder is directly ahead and is moving towards the ownship, the score for *COC* will not be maximal." This is a safe-ordering constraint $\langle P, Q \rangle$, where the precondition P is captured as a linear real arithmetic formula given by [21]:

$$\begin{aligned} P \equiv\ & 1500 \leq \rho \leq 1800 \ \wedge\ -0.06 \leq \theta \leq 0.06 \ \wedge\ \psi \geq 3.10 \\ & \wedge\ v_{own} \geq 980 \ \wedge\ v_{int} \geq 960 \\ Q \equiv\ & y_0 < y_1 \ \vee\ y_0 < y_2 \ \vee\ y_0 < y_3 \ \vee\ y_0 < y_4 \end{aligned}$$

In fact, nine of the ten specifications proposed by [21] are safe-ordering constraints. The single exception has a postcondition that places a constant lower-bound on y_0, i.e., a constraint on the logit value. We do not consider such constraints because the exact logit values are often less meaningful than the class order, given the approximate nature of neural networks and the fact that logit values are not typically calibrated. Moreover, the logit values of the network can be freely scaled without impacting the network's behavior as a classifier.

Given a set of safe-ordering constraints, Φ, our goal is to obtain a neural network that satisfies Φ everywhere. In later sections, we show how to accomplish this by describing the construction of a *self-correcting transformer* (Definition 2) that takes an existing, possibly unsafe network, and produces a related model that satisfies Φ at all points. While in practice, a meaningful, well-defined specification Φ should be satisfiable for all inputs, our generic formulation of safe-ordering constraints in Definition 1 does not enforce this restriction; we can, for instance, let $\Phi := \langle \top, y_0 < y_1 \rangle, \langle \top, y_1 < y_0 \rangle$. To account for this, we lift predicates ϕ to operate on $\mathbb{R}^m \cup \{\bot\}$, where $\phi(x, \bot)$ is considered valid for all x.

Definition 2 (Self-correcting transformer). *A self-correcting transformer, $SC : \Phi \to (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to (\mathbb{R}^m \cup \{\bot\}))$, is a function that, given a set of safe-ordering constraints, Φ, and a neural network, $f : \mathbb{R}^n \to \mathbb{R}^m$, produces a network, denoted as $f^\Phi : \mathbb{R}^n \to (\mathbb{R}^m \cup \{\bot\})$, that satisfies the following properties:*

(i) Safety*:* $\forall x \,.\, (\, \exists y.\, \Phi(x, y)\,) \implies \Phi(x, f^\Phi(x))$
(ii) Forewarning*:* $\forall x \,.\, (\, f^\Phi(x) = \bot \iff \forall y \,.\, \neg\Phi(x, y)\,)$

In other words, $f^\Phi = SC(\Phi)(f)$ is safe with respect to Φ and produces a non-$\bot$ output wherever $\Phi(x)$ is satisfiable. We refer to the output of SC, f^Φ, as a self-correcting network (SC-Net).

Definition 2(i) captures the essence of the problem that we aim to solve, requiring that the self-correcting network make changes to its output according to Φ. While allowing it to abstain from prediction by outputting $\bot$ may appear to relax the underlying problem, note that this is only allowed in cases where Φ cannot be satisfied on x: Definition 2(ii) is an equivalence that precludes trivial solutions such as $f^\Phi := \lambda x.\bot$. However, it still allows abstention in exactly the cases where it is needed for principled reasons. A set of safe-ordering constraints may be mutually satisfiable almost everywhere, except in some places; for example: $\Phi := \langle x \leq 0.5, y_0 < y_1\rangle, \langle x \geq 0.5, y_1 < y_0\rangle$. In this case, f^Φ can abstain at $x = 0.5$, and everywhere else must produce outputs in $\mathbb{R}^m$ obeying Φ.

While the properties required by Definition 2 are sufficient to ensure a non-trivial, safe-by-construction neural network, in practice, we aim to apply $SC(\Phi)$, which we will write as SC^Φ, to models that *already* perform well on observed test cases, but that still require a safety guarantee. Thus, we wish to correct network outputs without interfering with the existing network behavior when possible, a property we call *transparency* (Property 1).

Property 1 (Transparency). *Let $SC : \Phi \to (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to (\mathbb{R}^m \cup \{\bot\}))$ be a self-correcting transformer. We say that SC satisfies transparency if*

$$\forall \Phi \,.\, \forall f : \mathbb{R}^n \to \mathbb{R}^m \,.\, \forall x \in \mathbb{R}^n \,.$$
$$\left(\exists y.\, \Phi(x, y) \;\wedge\; \operatorname*{argmax}_i\{y_i\} = F(x)\right) \implies F^\Phi(x) = F(x)$$

where $F^\Phi(x) := \bot$ if $f^\Phi(x) = \bot$ else $\operatorname{argmax}_i\{f_i^\Phi(x)\}$. In other words, SC always produces an SC-Net, f^Φ, for which the top class derived from the safe output vectors of f^Φ agrees with the top class of the original model whenever possible.

Property 1 leads to a useful result, namely that whenever Φ is consistent with *accurate* predictions, then the classifier obtained from $SC^\Phi(f)$ is at least as accurate as F (Theorem 2). Formally, we characterize accuracy in terms of agreement with an oracle classifier F^O that "knows" the correct class for each

input, so that F is accurate on x if and only if $F(x) = F^O(x)$. We note that accuracy is often defined with respect to a *distribution* of labeled points rather than an oracle; however our formulation captures the key fact that Theorem 2 holds regardless of how the data are distributed.

Theorem 2 (Accuracy Preservation). *Given a neural network, $f : \mathbb{R}^n \to \mathbb{R}^m$, and set of constraints, Φ, let $f^\Phi := SC^\Phi(f)$ and let $F^O : \mathbb{R}^n \to [m]$ be the oracle classifier. Assume that SC satisfies transparency. Further, assume that accuracy is consistent with safety, i.e.,*

$$\forall x \in \mathbb{R}^n \;.\; \exists y \;.\; \Phi(x, y) \;\wedge\; \operatorname*{argmax}_i \{y_i\} = F^O(x).$$

Then,

$$\forall x \in \mathbb{R}^n \;.\; F(x) = F^O(x) \implies F^\Phi(x) = F^O(x)$$

One subtle point to note is that even when Φ is consistent with accurate predictions, it is possible for a network to be accurate yet unsafe at an input. Example 2 describes such a situation. Our formulation of Property 1 is carefully designed to ensure accuracy preservation even in such scenarios.

Example 2 (Accuracy need not imply safety). Consider the property ϕ_2 proposed for ACAS Xu by [21] which says: "If the intruder is distant and is significantly slower than the ownship, the score of the COC advisory should never be minimal." This safe-ordering constraint is applicable for all networks that correspond to $a_{prev} \neq \text{COC}$ and is concretely written as follows:

$$P \equiv \rho \geq 55947.691 \;\wedge\; v_{own} \geq 1145 \;\wedge\; v_{int} \leq 60$$
$$Q \equiv y_1 < y_0 \;\vee\; y_2 < y_0 \;\vee\; y_3 < y_0 \;\vee\; y_4 < y_0$$

For some x such that $P(x)$ is true, let us assume that $F^O(x) = 1$ and for a network f, $f(x) = [100, 900, 300, 140, 500]$, so that $F(x) = 1$. Then, f is accurate at x, but the COC advisory receives the minimal score, meaning f is unsafe at x with respect to ϕ_2. If the transformer SR satisfies Property 1, then by Theorem 2, f^{ϕ_2} is guaranteed to be accurate as well as safe at x, since ϕ_2 is consistent with accuracy here (as ϕ_2 does not preclude class 1 from being maximal).

3 Self-correcting Transformer

We describe our self-correcting transformer, `SC`. We begin with a high-level overview of the approach (Sect. 3.1), and provide algorithmic details in Sect. 3.2. We then provide proofs (Sect. 3.3) and complexity analysis (Sect. 3.4).

3.1 Overview

Our self-correcting transformer, `SC`, leverages the fact that whenever a safe-ordering constraint is satisfiable at a point, it is possible to bring the network into

compliance. Neural networks are typically constructed by composing a sequence of layers; we thus compose an additional *self-correction layer* that operates on the original network's output, and produces a result that will serve as the transformed network's new output. This is reflected in the `SC` routine in Algorithm 3.1. The original network, f, executes normally, and the self-correction layer subsequently takes both the input x (to facilitate checking the preconditions of Φ) and $y := f(x)$, from which it either abstains (outputs $\perp$) or produces an output that is guaranteed to satisfy Φ.

The high-level workflow of the self-correction layer, `SC-Layer`, proceeds as follows. The layer starts by checking the input x against each of the preconditions, and derives an *active postcondition*. This is then passed to a solver, which attempts to find the set of orderings that are consistent with the active postcondition. If no such ordering exists, i.e., if the active postcondition is unsatisfiable, then the layer abstains with $\perp$. Otherwise, the layer minimally permutes the indices of the original output vector in order to satisfy the active postcondition while ensuring transparency (Property 1).

3.2 Algorithmic Details of SC-Layer

The core logic of our approach is handled by a self-correction layer, or `SC-Layer`, that is appended to the original model, and dynamically ensures its outputs satisfy the requisite safety specifications. The procedure followed by this layer, `SC-Layer` (shown in Algorithm 3.1), first checks if the input x and output y of the base network already satisfy Φ (line 5). If they do, no correction is necessary and the repaired network f^{Φ} can safely return y. Otherwise, `SC-Layer` attempts to find a satisfiable ordering constraint that entails the relevant postconditions in Φ (line 8). `FindSatConstraint` either returns such a term q that consists of a conjunction of ordering literals $y_i < y_j$, or returns $\perp$ whenever no such q exists. When `FindSatConstraint` returns $\perp$, then `SC-Layer` does as well (lines 9–10). Otherwise, the constraint identified by `FindSatConstraint` is used to correct the network's output (line 12), where `Correct` permutes the logit values in y to arrive at a vector that satisfies q. Note that because q is satisfiable, it is always possible to find a satisfying solution by simply permuting y, because the specific real values are irrelevant, and only their order matters (see Sect. 3.3).

Finding Satisfiable Constraints. Algorithm 3.2 illustrates the `FindSatConstraint` procedure. Recall that the goal is to identify a conjunction of ordering literals q that implies the *relevant* postconditions in Φ at the given input x. More precisely, this means that for each precondition P_i satisfied by x, the corresponding postcondition Q_i is implied by q. This is sufficient to ensure that any model y' of q will satisfy Φ at x; i.e., $q(y') \implies \Phi(x, y')$.

To accomplish this, `FindSatConstraint` first evaluates each precondition, and obtains (line 9) a disjunctive normal form (DNF), Q_x, of the *active postcondition*, defined by $\texttt{Filter}(\Phi, x) := \bigwedge_{\langle P_i, Q_i \rangle \in \Phi \mid P_i(x)} Q_i$. In practice, we implement a lazy version of `ToDNF` that generates disjuncts as needed (see Sect. 4),

Algorithm 3.1: Self-correcting transformer

Inputs: A set of safety properties, Φ and a network, $f : \mathbb{R}^n \to \mathbb{R}^m$
Output: A network, $f^{\Phi} : \mathbb{R}^n \to \mathbb{R}^m \cup \{\bot\}$

```
1  SC(Φ , f):
2      f^Φ := λ x.SC-Layer(Φ, x, f(x))
3      return f^Φ
4  SC-Layer(Φ , x , y):
5      if Φ(x, y) then
6          return y
7      else
8          q := FindSatConstraint(Φ, x, y)
9          if q = ⊥ then
10             return ⊥
11         else
12             y' := Correct(q, y)
13             return y'
```

as this step may be a bottleneck, and we only need to process each clause individually. At this point, `FindSatConstraint` could proceed directly, checking the satisfiability of each disjunct in Q_x, and returning the first satisfiable one it encounters. This would be correct, but as we wish to satisfy transparency (Property 1), we first construct an ordered list of the terms in Q_x which prioritizes constraints that maintain the maximal position of the original prediction, argmax(y) (`Prioritize`, line 10). Property 3 formalizes the behavior required of `Prioritize`.

Property 3 (Prioritize). *Given $y \in \mathbb{R}^m$ and a list of conjunctive ordering constraints $\overline{Q}$, the result of* `Prioritize`$(\overline{Q}, y)$ *is a reordered list $\overline{Q}' = [\ldots, q_i, \ldots]$ such that:*

$$\forall\ 0 \leq i, j < |\overline{Q}|\ .\ \operatorname*{argmax}_i\{y_i\} \in \mathit{Roots}(\texttt{OrderGraph}(q_i))$$
$$\wedge \operatorname*{argmax}_i\{y_i\} \notin \mathit{Roots}(\texttt{OrderGraph}(q_j)) \implies i < j$$

where $\mathit{Roots}(G)$ denotes the root nodes of the directed graph G.

The `IsSat` procedure (invoked on line 12, also shown in Algorithm 3.2) checks the satisfiability of a conjunctive ordering constraint. It is based on an encoding of q as a directed graph, embodied in `OrderGraph` (lines 1–4), where each component index of y corresponds to a node, and there is a directed edge from i to j if the literal $y_j < y_i$ appears in q. A constraint q is satisfiable if and only if `OrderGraph`(q) contains no cycles (lines 5–7) [35]. Informally, acyclicity is necessary and sufficient for satisfiability because the directed edges encode

Algorithm 3.2: Finding a satisfiable ordering constraint from safe-ordering constraints Φ

Inputs: A set of safe-ordering constraints, Φ, a vector $x : \mathbb{R}^n$, and a vector $y : \mathbb{R}^m$
Output: Satisfiable ordering constraint, q

```
1  OrderGraph(q):
2      V := [m]
3      E := {(i, j) : y_j < y_i ∈ q}
4      return (V, E)

5  IsSat(q):
6      g := OrderGraph(q)
7      return ¬ContainsCycle(V, E)

8  FindSatConstraint(Φ , x , y):
9      Q_x := ToDNF(Filter(Φ, x))
10     Q_p := Prioritize(Q_x, y)
11     foreach q_i ∈ Q_p do
12         if IsSat(q_i) then
13             return q_i
14     return ⊥
```

Algorithm 3.3: Correction procedure for safe-ordering constraints

Inputs: Satisfiable ordering constraint q, a vector $y : \mathbb{R}^m$
Output: A vector $y' : \mathbb{R}^m$

```
1  Correct(q , y):
2      π := TopologicalSort(OrderGraph(q), y)
3      y^s := SortDescending(y)
4      ∀ j ∈ [m] . y'_j := y^s_{π(j)}
5      return y'
```

immediate ordering requirements, and by transitivity, a cycle involving i entails that $y_i < y_i$.

Correcting Violations. Algorithm 3.3 describes the `Correct` procedure, used to ensure the outputs of the `SC-Layer` satisfy safety. The inputs to `Correct` are a satisfiable ordering constraint q, and the output of the original network $y := f(x)$. The goal is to permute y such that the result y' satisfies q, without violating transparency. Our approach is based on `OrderGraph`, the same directed-graph encoding used by `IsSat`. It uses a stable topological sort of the graph encoding of q to construct a total order over the indices of y that is consistent with the partial ordering implied by q (line 2). `TopologicalSort` returns a

permutation π, a function that maps indices in y to their rank (or position) in the total order. Formally, `TopologicalSort` takes as argument a graph $G = (V, E)$, and returns π such that Eq. 1 holds.

$$\forall i, j \in V \;.\; (i, j) \in E \implies \pi(i) < \pi(j) \tag{1}$$

Informally, if the edge (i, j) is in the graph, then i occurs before j in the ordering. In general, many total orderings may be consistent, but in order to guarantee transparency, `TopologicalSort` also needs to ensure the following invariant (Property 4), capturing that the maximal index is listed first in the total order if possible.

Property 4. *Given a graph, $G = (V, E)$, and $y \in \mathbb{R}^m$, the result π of* `TopologicalSort`(G, y) *satisfies*

$$\underset{i}{\text{argmax}}\{y_i\} \in \mathit{Roots}(G) \implies \pi\left(\underset{i}{\text{argmax}}\{y_i\}\right) = 0$$

where $\mathit{Roots}(G)$ denotes the root nodes of the directed graph G.

In other words, the topological sort preserves the network's original prediction when doing so is consistent with q. Then, by sorting y in descending order, the sorted vector y^s can be used to construct the final output of `Correct`, y'. For any index i, we simply set y'_i to the $\pi(i)^{th}$ component of y^s, since $\pi(i)$ gives the desired rank of the i^{th} logit value and components in y^s are sorted according to the component values (line 4). Example 3 shows an example of the complete `Correct` procedure.

Example 3 (Self-correct). We refer again to the safety properties introduced for ACAS Xu [21]. The postcondition of property ϕ_2 states that the logit score for class 0 (COC) is not minimal, which can be written as the following ordering constraint:

$$Q \equiv y_1 < y_0 \;\vee\; y_2 < y_0 \;\vee\; y_3 < y_0 \;\vee\; y_4 < y_0$$

Suppose that for some input $x \in \mathbb{R}^n$, the active postcondition is equivalent to Q, and that $y = [100, 900, 300, 140, 500]$. Further, suppose that `FindSatConstraint` has returned $q := y_2 < y_0$, corresponding to the second disjunct of Q (satisfying $q \implies Q$). We then take the following steps according to `Correct`(q, y):

- First we let $\pi :=$ `TopologicalSort`(`OrderGraph`$(q), y)$. We note that all vertices of the graph representation of q are roots except for $j = 2$, which has $j = 0$ as its parent. We observe that $\text{argmax}_i\{y_i\} = 1$, which corresponds to a root node; thus by Property 4, $\pi(1) = 0$. Moreover, by our ordering constraint, we also have that $\pi(0) < \pi(2)$. Thus, the ordering π where $\pi(0) = 2$, $\pi(1) = 0$, $\pi(2) = 3$, $\pi(3) = 4$, and $\pi(4) = 1$ is a possible result of `TopologicalSort`, which we will assume for this example.
- Next we obtain by a descending sort that $y^s = [900, 500, 300, 140, 100]$.
- Finally we obtain y' by indexing y^s by the inverse of π, i.e., $y'_j = y^s_{\pi(j)}$. This gives us $y'_0 = y^s_2 = 300$, $y'_1 = y^s_0 = 900$, $y'_2 = y^s_3 = 140$, $y'_3 = y^s_4 = 100$, and $y'_4 = y^s_1 = 500$, resulting in a final output of $y' = [300, 900, 140, 100, 500]$, which (i) satisfies Q, and (ii) preserves the prediction of class 1.

3.3 Key Properties

We now provide a brief argument that our SC procedure satisfies two key properties; namely (1) SC is a self-correcting transformer (Definition 2)—i.e., it guarantees that the corrected output will always satisfy the requisite safety properties, unless they are *unsatisfiable*, in which case it returns $\perp$—and (2) SC is transparent (Property 1)—i.e., it does not modify the predicted class (the class with the maximal logit value) unless it is absolutely necessary for safety. Full proofs appear in Appendix A.

Theorem 5 (SC is a self-correcting transformer). SC *(Algorithm 3.1) satisfies conditions* (i) *and* (ii) *of Definition 2.*

This follows from the construction of SC, and relies on a few key properties of FindSatConstraint and Correct. First, whenever FindSatConstraint returns $\perp$, the set of safety constraints, Φ, is *unsatisfiable* on the given input. Second, whenever FindSatConstraint returns some $q \neq \perp$, then q is satisfiable on the given input. Finally, when q is satisfiable, Correct always modifies the output such that it satisfies q. Together, these imply Theorem 5.

In addition to ensuring safe-ordering, SC is transparent (Theorem 6), which recall is a precondition for the accuracy preservation property stated in Theorem 2.

Theorem 6 (Transparency of SC). SC*, the self-correcting transformer described in Algorithm 3.1, satisfies Property 1.*

Clearly, on points where the model naturally satisfies the safety properties, no changes to the output are made and SC is transparent. Otherwise, we rely on a few key details of our construction to achieve transparency.

We begin with the observation that whenever the network's predicted top class is a root of the graph encoding of a satisfiable postcondition, q, there exists an output that satisfies q while preserving the predicted top class. Intuitively, this follows because the partial ordering admits *any* of the root nodes to appear first in the total ordering.

With this in mind, we recall that FindSatConstraint searches potential solutions according to Prioritize, which prefers all solutions in which the predicted top class appears as a root node over any in which it does not. Thus, Prioritize will return a solution that is consistent with preserving the network's original predicted top class whenever possible.

Finally, we design our topological sort to be "stable," such that, among other things, the network's original top prediction will appear first in the total ordering whenever it appears as a root node. More details on our topological sort algorithm and the properties it possesses are given in Sect. B.1.

3.4 Complexity

Given a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, we define the input size as n and output size as m. Also, assuming that the postconditions Q_i for all $\langle P_i, Q_i \rangle \in \Phi$

are expressed in DNF, we define the size p_i of a constraint as the number of disjuncts in Q_i and define $\alpha := |\Phi|$, i.e., the number of properties in Φ. Then, the worst-case computational complexity of SC-Layer is given by Eq. 2, where $O(log(m))$ is the complexity of `ContainsCycle`, $O(mlog(m))$ is the complexity of `TopologicalSort`, and $\prod_{i=1}^{\alpha} p_i$ is the maximum number of disjuncts possible in Q_x if the postconditions Q_i are in DNF.

$$O\left(log(m)\prod_{i=1}^{\alpha} p_i + m\,log(m)\right) \tag{2}$$

The complexity given by Eq. 2 is with respect to a cost model that treats matrix operations—e.g., matrix multiplication, associative row/column reductions—as constant-time primitives. Crucially, note that the complexity does not depend on the size of the neural network f.

3.5 Differentiability of SC-Layer

One interesting facet of our approach that remains largely unexplored is the differentiability of the `SC-Layer`. In principle, this opens the door to benefits that could be obtained by training against the corrections made by the `SC-Layer`. Though we found that a "vanilla" attempt to train with the `SC-Layer` did not provide clear advantages over appending it to a model post-learning, we believe this remains an interesting future direction to explore. It is conceivable that a careful approach to training SC-Nets could lead to safer and more accurate models, reducing the need for the run-time correction, as the network could learn to use the modifications made by the `SC-Layer` to its advantage. Furthermore, aspects of the algorithm, including, e.g., the heuristic used to prioritize the search for a satisfiable graph (see **Finding Satisfiable Constraints** in Sect. 3.2), could be parameterized and learned, potentially leading to both accuracy and performance benefits.

4 Vectorizing Self-correction

Widely-used machine learning libraries, such as TensorFlow [1], simplify the implementation of parallelized, hardware-accelerated code by providing a collection of operations on multi-dimensional arrays of uniform type, called *tensors*. One can view such libraries as domain-specific languages that operate primarily over tensors, providing embarrassingly parallel operations like matrix multiplication and associative reduction, as well as non-parallel operations like iterative loops and sorting routines. We use matrix-based algorithms implementing the core procedures used by `SC-Layer` described in Sect. 3. As we will later see in Sect. 5, taking advantage of these frameworks allows our implementation to introduce minimal overhead, typically fractions of milliseconds. Additionally, it means that `SC-Layer` can be automatically differentiated, making it fully compatible with training and fine-tuning. One may use an SMT solver like Z3 to

implement the procedures used by `SC-Layer` but we found that making calls to an SMT solver significantly restricts the efficient use of GPUs. Moreover, it is a useful heuristic for the corrected logits to prioritize the original ordering relationships. Encoding this heuristic would require an optimization variant of SMT (Max-SMT). In contrast, our matrix-based algorithms efficiently calculate the corrected output while prioritizing the original class order. We present the algorithmic details in Appendix B.

5 Evaluation

We have shown that self-correcting networks (SC-Nets) provide safety to an existing network without affecting accuracy, as long as safety and accuracy are mutually consistent. This comes with no additional training cost, suggesting that the only potential downside of SC-Nets is the run-time overhead introduced by the `SC-Layer`. In this section, we present an empirical evaluation of our approach to demonstrate its scalability, and find that the run-time performance is not an issue in practice—overheads range from 0.2–0.8 ms, and scale favorably with the size and complexity of constraints.

We explore the capability of our approach on a variety of domains, demonstrating its ability to solve previously studied safety-verification problems (Sects. 5.1 and 5.2), and its ability to efficiently scale both (i) to large convolutional networks (Sect. 5.3) and (ii) to arbitrary, complex safe-ordering constraints containing disjunctions and overlapping preconditions (Sect. 5.4).

We implemented our approach in Python, using TensorFlow to vectorize our `SC-Layer` (Sect. 4). All experiments were run on an NVIDIA TITAN RTX GPU with 24 GB of RAM, and a 4.2 GHz Intel Core i7-7700K with 32 GB of RAM.

5.1 ACAS Xu

ACAS Xu [23] is a collision avoidance system that has been frequently studied in the context of neural classifier safety verification [20,21,29,39]. Typically considered for this problem is a family of 45 networks proposed by [20]. [21] proposed 10 safety specifications for this family of networks, which have become standard for research on this problem. We consider 9 of these specifications, which can be expressed as *safe-ordering constraints* (Sect. 2.2).

Each of the 45 networks consists of six hidden dense layers of 50 neurons each. Each network needs to satisfy some subset of the 10 safety constraints; that is, more than one safety constraints may apply to each model, but not all safety constraints apply to each model. A network is considered safe if it satisfies *all* of the relevant safety constraints. Among the 45 networks, [21] reported that 9 networks were already safe after standard training, while 36 were unsafe, exhibiting safety constraint violations.

The data used to train the 45 networks is not publicly available; however, [29] provide a synthetic test set for each network, consisting of 5,000 points uniformly sampled from the specified state space and labeled using the respective network

Table 1. Safety certification results on the **(a)** ACAS Xu [20] and **(b)** Collision Detection [12] datasets. We compare the success rate and accuracy to that of ART [29], a recent safe-by-construction approach. The original network is provided as a baseline. Best results are shown in bold. **(c)** Absolute overhead introduced by the SC-Layer per input.

Acas Xu			
	method	*safe networks*	*mean accuracy(%)*
36 unsafe nets	original	0 / 36	100.0
	ART	36 / 36	94.4
	SC-Net	**36 / 36**	**100.0**
9 safe nets	original	**9 / 9**	**100.0**
	ART	9 / 9	94.3
	SC-Net	**9 / 9**	**100.0**

(a)

Collision Detection		
method	*constraints certified*	*accuracy (%)*
original	328 / 500	99.9
ART	481 / 500	96.8
SC-Net	**500 / 500**	**99.9**

(b)

dataset	*overhead (ms)*
ACAS Xu	0.26
Collision Detection	0.58
CIFAR-100 (small CNN)	0.77
CIFAR-100 (ResNet-50)	0.82
Synthetic	0.27

(c)

as an oracle. We note that because this test set is labeled using the original models, the accuracy of each original model on this test set is necessarily 100%.

Table 1a presents the results of applying our `SC` transformer to each of the 45 provided networks. In particular, we consider the number of networks for which safety can be guaranteed, and the accuracy of the resulting SC-Net. We compare our results to those using ART [29], a recent approach to safe-by-construction learning. ART aims to learn neural networks that satisfy safety specifications expressed using linear real arithmetic constraints. It updates the loss function to be minimized during learning by adding a term, referred to as the *correctness loss*, that measures the degree to which a neural network satisfies or violates the safety specification. A value of zero for the correctness loss ensures that the network is safe. However, there is no guarantee that learning will converge to zero correctness loss, and the resulting model may not be as accurate as one trained with conventional methods.

Because the safety constraints for each network are satisfiable on all points, Definition 2 tells us that safety is guaranteed for all 45 SC-Nets. In this case, we see that ART also manages to produce 45 safe networks after training; however we see that it comes at a cost of nearly 6% points in accuracy, *even on the networks that were already safe*. Meanwhile, transparency (Property 1) tells us that SC-Nets will only see a decrease in accuracy relative to the original network when accuracy is in direct conflict with safety. On the 9 original networks that were reported as safe, clearly no such conflict exists, and accordingly, we see that the corresponding SC-Nets achieve the same accuracy as the original networks (100%). On the 36 unsafe networks, we find again that the SC-Nets achieved 100% accuracy. In this case, it would have been possible that the SC-Nets would have achieved lower accuracy than the original networks, as some of the safety properties have the potential to conflict with accuracy. For example, the post-condition of the property ϕ_8 requires that the predicted maneuver advisory is either to continue straight (COC) or to turn weakly to the left. Thus, correcting ϕ_8 on inputs for which it is violated would necessarily change the network's prediction on those inputs; and, since the labels are derived from the original networks' predictions, this would lead to a drop in accuracy. However, we find that none of the test points include violations of such constraints (even though such violations exist in the space generally [21]), as evidenced by the fact that the SC-Net accuracy remained unchanged.

Table 1c shows the average overhead introduced by applying `SC` to each of the ACAS Xu networks. We see that the absolute overhead is only $\sim$0.25 ms per instance on average, accounting for less than an 8$\times$ increase in prediction time. Note, however, that while our implementation of `SC-Layer` is optimized for and evaluated using a GPU, the Acas Xu system is expected to be deployed on resource-constrained hardware without access to GPUs [20] which is likely to result in larger overheads.

5.2 Collision Detection

The Collision Detection dataset [12] provides another instance of a safety verification task that has been studied in the prior literature. In this setting, a neural network controller is trained to predict whether two vehicles following curved paths at different speeds will collide. As this is a binary decision task, the network contains two outputs, corresponding to the case of a collision and the case of no collision. [12] proposes 500 safety properties for this task, corresponding to ℓ_∞ *robustness regions* around 500 particular inputs; i.e., property ϕ_i for $i \in \{1, \ldots, 500\}$ corresponds to a point, x_i, and radius, ϵ_i, and is defined according to Eq. 3.

$$\phi_i(x, y) \;:=\; ||x_i - x||_\infty \leq \epsilon_i \implies y = F(x_i) \tag{3}$$

Such specifications of *local robustness* at fixed inputs can be represented as safe-ordering constraints, where the postcondition of ϕ_i is defined to be $y_0 > y_1$ if $F(x_i) = 0$ and $y_0 < y_1$ if $F(x_i) = 1$.

Table 1b presents the results of applying our `SC` transformer to the original network provided by [12]. Similarly to before, we consider the number of constraints with respect to which safety can be guaranteed, and the accuracy of the resulting SC-Net, comparing our results to those of ART.

We see in this case that ART was unable to guarantee safety for all 500 specifications. Meanwhile, it resulted in a drop in accuracy of approximately 3% points. On the other hand, it is simple to check that the conjunction of all 500 safety constraints is satisfiable for all inputs; thus, Definition 2 tells us that safety is guaranteed with respect to all properties. Meanwhile SC-Nets impose no penalty on accuracy, as none of the test points violate the constraints.

Table 1c shows the overhead introduced by applying `SC` to the collision detection model. In absolute terms, we see the overhead is approximately half a millisecond per instance, accounting for under a 3× increase in prediction time.

5.3 Scaling to Larger Domains

One major challenge for many approaches that attempt to verify network safety—particularly post-learning methods—is scalability to very large neural networks. Such networks pose a problem for several reasons. Many approaches analyze the parameters or intermediate neuron activations using algorithms that do not scale polynomially with the network size. This is a practical problem, as large networks in use today contain hundreds of millions of parameters. Furthermore, abstractions of the behavior of large networks may see compounding imprecision in large, deep networks.

Our approach, on the other hand, treats the network as a black-box and is therefore not sensitive to its specifics. In this section we demonstrate that this is borne out in practice; namely the absolute overhead introduced by our `SC-Layer` remains relatively stable even on very large networks.

For this, we consider a novel set of safety specifications for the CIFAR-100 image dataset [24], a standard benchmark for object recognition tasks. The CIFAR-100 dataset is comprised of 60,000 32×32 RGB images categorized into 100 different classes of objects, which are grouped into 20 superclasses of 5 classes each. We propose a set of safe-ordering constraints that are reminiscent of a variant of *top-k accuracy* restricted to members of the same superclass, which has been studied recently in the context of certifying relational safety properties of neural networks [25]. More specifically, we require that if the network's prediction belongs to superclass C_k then the top 5 logit outputs of the network must all belong to C_k. Formally, there are 20 constraints, one for each superclass, where the constraint, ϕ_k for superclass C_k, for $k \in \{1, \ldots, 20\}$, is defined according to Eq. 4. Notice that with respect to these constraints, a standard trained network can be accurate, yet unsafe, even without accuracy and safety being mutually inconsistent.

$$\phi_k(x, y) \;:=\; F(x) \in C_k \implies \bigwedge_{i,j \;\mid\; i \in C_k,\; j \notin C_k} y_j < y_i \tag{4}$$

As an example application requiring this specification, consider a client of the classifier that averages the logit values over a number of samples for classes appearing in top-5 positions and chooses the class with the highest average logit value (due to imperfect sensor information - a scenario similar to the ACAS Xu example). A reasonable specification is to require that the chosen class shares its superclass with at least one of the top-1 predictions. We can ensure that this specification is satisfied by enforcing Eq. 4.

Table 1c shows the overhead introduced by applying `SC`, with respect to these properties, to two different networks trained on CIFAR-100. The first is a convolutional neural network (CNN) that is much smaller than is typically used for vision tasks, containing approximately 1 million parameters. The second is a standard residual network architecture, ResNet-50 [18], with approximately 24 million parameters.

In absolute terms, we see that both networks incur less than 1 ms of overhead per instance relative to the original model (0.77 ms and 0.82 ms, respectively), making `SC` a practical option for providing safety in both networks. Moreover, the absolute overhead varies only by about 5% between the two networks, suggesting that the overhead is not sensitive to the size of the network. This overhead accounts for approximately a 12× increase in prediction time on the CNN. Meanwhile, the overhead on the ResNet-50 accounts for only a 6× increase in prediction time relative to the original model. The ResNet-50 is a much larger and deeper network; thus its baseline prediction time is longer, so the overhead introduced by the `SC-Layer` accounts for a smaller fraction of the total computation time. In this sense, our `SC` transformer becomes relatively *less* expensive on larger networks.

Interestingly, we found that the original network violated the safety constraints on approximately *98% of its inputs*, suggesting that obtaining a violation-free network without `SC` might prove particularly challenging. Meanwhile, the SC-Net eliminated *all* violations, with *no cost to accuracy*, and less than 1 ms in overhead per instance.

5.4 Handling Arbitrary, Complex Constraints

Safe ordering constraints are capable of expressing a wide range of compelling safety specifications. Moreover, our `SC` transformer is a powerful, general tool for ensuring safety with respect to arbitrarily complex safe-ordering constraints, comprised of many conjunctive and disjunctive clauses. Notwithstanding, the properties presented in our evaluation thus far have been relatively simple. In this section we explore more complex safe-ordering constraints, and describe experiments that lend insight as to which factors most impact the scalability of our approach.

To this end, we designed a family of synthetic datasets with associated safety constraints that are randomly generated according to several specified parameters, allowing us to assess how aspects such as the number of properties (α), the number of disjunctions per property (β), and the dimension of the output vector (m) impact the run-time overhead. In our experiments, we fix the input

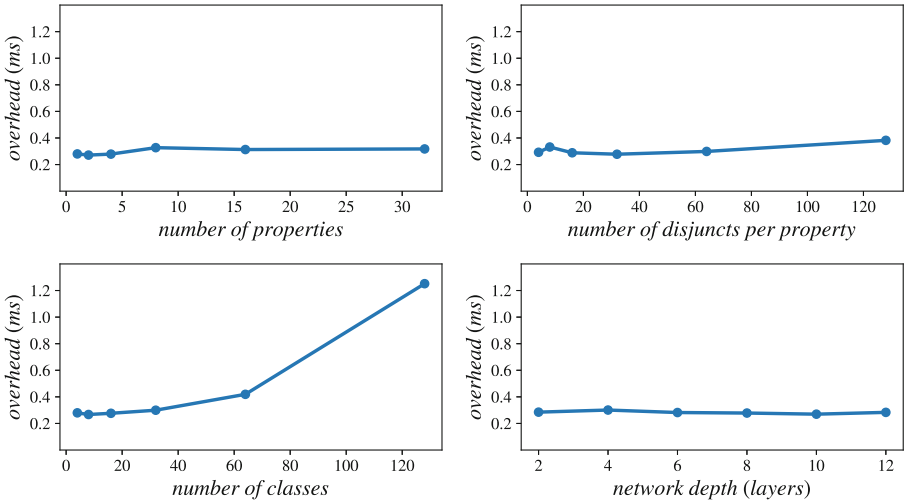

Fig. 1. Absolute overhead in milliseconds introduced by the SC-Layer as either the number of properties, i.e., safe-ordering constraints (α), the number of disjuncts per property (β), the number of classes (m), or the network depth (δ) are varied. In each plot, the respective parameter varies according to the values on the x-axis, and all other parameters take a default value of $\alpha = 4$, $\beta = 4$, $m = 8$, and $\delta = 6$. As the depth of the network varies, the number of neurons in each layer remains fixed at 1,000 neurons. Reported overheads are averaged over 5 trials.

dimension, n, to be 10. Each dataset is parameterized by α, β, and m, and denoted by $\mathcal{D}(\alpha, \beta, m)$; the procedure for generating these datasets is provided in Appendix C.

We use a dense network with six hidden layers of 1,000 neurons each as a baseline, trained on $\mathcal{D}(4, 4, 8)$. Table 1c shows the overhead introduced by applying `SC` to our baseline network. We see that the average overhead is approximately a quarter of a millisecond per instance, accounting for a 10× increase in prediction time. Figure 1 provides a more complete picture of the overhead as we vary the number of safe-ordering constraints (α), the number of disjuncts per constraint (β), the number of classes (m), or the depth of the network (δ).

We observe that among these parameters, the overhead is sensitive only to the number of classes. This is to be expected, as the complexity of the `SC-Layer` scales directly with m (see Sect. 3.4), requiring a topological sort of the m elements of the network's output vector. On the other hand, perhaps surprisingly, increasing the complexity of the safety constraints through either additional safe-ordering constraints or larger disjunctive clauses in the ordering constraints had little effect on the overhead. While in the *worst case* the complexity of the `SC-Layer` is also dependent on these parameters (Sect. 3.4), if `FindSatConstraint` finds a satisfiable disjunct quickly, it will short-circuit. The average-case complexity of `FindSatConstraint` is therefore more nuanced, depending to a greater extent on the specifics of the constraints rather than

simply their size. Altogether, these observations suggest that the topological sort in `SC-Layer` tends to account for the majority of the overhead.

Finally, the results in Fig. 1 concur with what we observed in Sect. 5.3; namely that the overhead is independent of the size of the network.

6 Related Work

Static Verification and Repair of Neural Networks. A number of approaches for verification of already-trained neural networks have been presented in recent years. They have focused on verifying safety properties similar to our safe-ordering constraints. Abstract interpretation approaches [15,39] verify properties that associate polyhedra with pre- and postconditions. Reluplex [21] encodes a network's semantics as a system of constraints, and poses verification as constraint satisfiability. These approaches can encode safe-ordering constraints, which are a special case of polyhedral postconditions, but they do not provide an effective means to construct safe networks. Other verification approaches [19,44] do not address safe ordering.

Many of the above approaches can provide counterexamples when the network is unsafe, but none of them are capable of repairing the network. A recent repair approach [40] can provably repair neural networks that have piecewise-linear activations with respect to safety specifications expressed using polyhedral pre- and postconditions. In contrast to our transparency guarantee, they rely on heuristics to favor accuracy preservation.

Safe-by-Construction Learning. Recent efforts seek to learn neural networks that are correct by construction. Some approaches [14,28,31] modify the learning objective by adding a penalty for unsafe or incorrect behavior, but they do not provide a safety guarantee for the learned network. Balancing accuracy against the modified learning objective is also a concern. In our work we focus on techniques that provide guarantees without requiring external verifiers.

As discussed in Sect. 5.1, ART [29] aims to learn networks that satisfy safety specifications by updating the loss function used in training. Learning is not guaranteed to converge to zero correctness loss, and the resulting model may not be as accurate as one trained with conventional methods. In contrast, our program transformer is guaranteed to produce a safe network that preserves accuracy.

A similar approach is presented in [33] to enforce local robustness for all input samples in the training dataset. This technique also updates the learning objective and uses a differentiable abstract interpreter for over-approximating the set of reachable outputs. For both this approach and that of [29], the run time of the differentiable abstract interpreter depends heavily on the size and complexity of the network, and it may be difficult or expensive to scale them to realistic architectures.

An alternative way to achieve correct-by-construction learning is to modify the architecture of the neural model. This approach has been employed to

construct networks that have a fixed Lipschitz constant [4,27,43], a relational property that is useful for certifying local robustness and ensuring good training behavior. Recent work [25,26] shows how to construct models that achieve relaxed notions of global robustness, where the network is allowed to selectively abstain from prediction at inputs where local robustness cannot be certified. [10] use optimization layers to enforce stability properties of neural network controllers. These techniques are closest to ours in spirit, although we focus on safety specifications, and more specifically safe-ordering constraints, which have not been addressed previously in the literature.

Shielding Control Systems. Recent approaches have proposed ensuring safety of control systems by constructing run-time check-and-correct mechanisms, also referred to as *shields* [2,6,46]. Shields check at run time if the system is headed towards an unsafe state and provide corrections for potentially unsafe actions when necessary. To conduct these run-time checks, shields need access to a model of the environment that describes the environment dynamics, i.e., the effect of controller actions on environment states. Though shields and our proposed `SC-Layer` share the run-time check-and-correct philosophy, they are designed for different problem settings.

Recovering from Program Errors. Embedding run-time checks into a program to ensure safety is a familiar technique in the program verification literature. Contract checking [13,32], run-time verification [17], and dynamic type checking are all instances of such run-time checks. If a run-time check fails, the program terminates before violating the property. A large body of work also exists on gracefully recovering from errors caused by software issues such as divide-by-zero, null-dereference, memory corruption, and divergent loops [5,22,30,36–38]. These approaches are particularly relevant in the context of long-running programs, when aiming to repair state just enough so that computation can continue.

7 Conclusion and Future Directions

We presented a method for transforming a neural network into a safe-by-construction *self-correcting network*, termed SC-Net, without harming the accuracy of the original network. This serves as a practical tool for providing safety with respect to a broad class of safety specifications, namely, *safe-ordering constraints*, that we characterize in this work.

Unlike prior approaches, our technique guarantees safety without further training or modifications to the network's parameters. Furthermore, the scalability of our approach is not limited by the size or architecture of the model being repaired. This allows it to be applied to large, state-of-the-art models, which is impractical for most other existing approaches.

A potential downside to our approach is the run-time overhead introduced by the `SC-Layer`. We demonstrate in our evaluation that our approach maintains small overheads (less than one millisecond per instance), due to our vectorized implementation, which leverages GPUs for large-scale parallelism.

In future work, we plan to leverage the differentiability of the `SC-Layer` to further explore training against the repairs made by the `SC-Layer`, as this can potentially lead to both accuracy and safety improvements.

Acknowledgment. This material is based upon work supported by the Software Engineering Institute under its FFRDC Contract No. FA8702-15-D-0002 with the U.S. Department of Defense, the National Science Foundation under Grant No. CNS-1943016, DARPA GARD Contract HR00112020006, and the Alfred P. Sloan Foundation.

A Appendix 1: Proofs

Theorem 2 (Accuracy Preservation)**.** *Given a neural network, $f : \mathbb{R}^n \to \mathbb{R}^m$, and set of constraints, Φ, let $f^\Phi := SC^\Phi(f)$ and let $F^O : \mathbb{R}^n \to [m]$ be the oracle classifier. Assume that SC satisfies transparency. Further, assume that accuracy is consistent with safety, i.e.,*

$$\forall x \in \mathbb{R}^n \;.\; \exists y \;.\; \Phi(x, y) \;\wedge\; \operatorname*{argmax}_i\{y_i\} = F^O(x).$$

Then,

$$\forall x \in \mathbb{R}^n \;.\; F(x) = F^O(x) \implies F^\Phi(x) = F^O(x)$$

Proof. Let $x \in \mathbb{R}^n$ such that $F(x) = F^O(x)$. By hypothesis, we have that $\exists y \;.\; \Phi(x, y) \wedge \operatorname{argmax}_i\{y_i\} = F^O(x)$, hence we can apply Property 1 to conclude that $F^\Phi(x) = F(x) = F^O(x)$.

We now prove that the transformer presented in Algorithm 3.1, `SC`, is indeed self-correcting; i.e., it satisfies Properties 2(i) and 2(ii). Recall that this means that f^Φ will either return safe outputs vectors, or in the event that Φ is inconsistent at a point, and *only* in that event, return $\perp$.

Let $x : \mathbb{R}^n$ be an arbitrary vector. If $\Phi(x, f(x))$ is initially satisfied, the SC-Layer does not modify the original output $y = f(x)$, and Properties 2(i) and 2(ii) are trivially satisfied. If $\Phi(x, f(x))$ does not hold, we will rely on two key properties of `FindSatConstraint` and `Correct` to establish that `SC` is self-correcting. The first, Property 7, requires that `FindSatConstraint` either return $\perp$, or else return ordering constraints that are sufficient to establish Φ.

Property 7 (`FindSatConstraint`). *Let Φ be a set of safe-ordering constraints, $x : \mathbb{R}^n$ and $y : \mathbb{R}^m$ two vectors.*
Then $q =$ `FindSatConstraint`(Φ, x, y) satisfies the following properties:

(i) $q = \perp \iff \forall y' \;.\; \neg\Phi(x, y')$
(ii) $q \neq \perp \implies (\; \forall y' \;.\; q(y') \implies \Phi(x, y') \;)$

Proof. The first observation is that the list of ordering constraints in $Q_p :=$ `Prioritize`(Q_x, y) accurately models the initial set of safety constraints Φ, i.e.,

$$\forall y' \;.\; \Phi(x, y') \iff (\; \exists q \in Q_p \;.\; q(y') \;) \tag{5}$$

This stems from the definition of the disjunctive normal form, and from the fact that `Prioritize` only performs a permutation of the disjuncts.

We also rely on the following loop invariant, stating that all disjuncts considered so far, when iterating over $\texttt{Prioritize}(Q_x, y)$, were unsatisfiable:

$$\forall q \in Q_p \;.\; \texttt{idx}(q, Q_p) < \texttt{idx}(q_i, Q_p) \implies (\; \forall y \;.\; \neg q(y) \;) \tag{6}$$

Here, $\texttt{idx}(q, Q_p)$ returns the index of constraint q in the list Q_p. This invariant is trivially true when entering the loop, since the current q_i is the first element of the list. Its preservation relies on $\texttt{IsSat}(q)$ correctly determining whether q is satisfiable, i.e., $\texttt{IsSat}(q) \iff \exists y \;.\; q(y)$ [35].

Combining these two facts, we can now establish that `FindSatConstraint` satisfies 7(i) and 7(ii). By definition, $\texttt{FindSatConstraint}(\Phi, x, y)$ outputs $\bot$ if and only if it traverses the entire list Q_p, never returning a q_i. From loop invariant 6, this is equivalent to $\forall q \in Q_p.\; \forall y'.\; \neg q(y')$, which finally yields Property 7(i) from Eq. 5. Conversely, if $\texttt{FindSatConstraint}(\Phi, x, y)$ outputs $q \neq \bot$, then $q \in Q_p$. We directly obtain Property 7(ii) as, for any $y' : \mathbb{R}^m$, $q(y')$ implies that $\Phi(x, y')$ by application of Eq. 5

Next, Property 8 states that `Correct` correctly permutes the output of the network to satisfy the constraint that it is given. Combined with Property 7, this is sufficient to show that `SC` is a self-correcting transformer (Theorem 5).

Property 8 (`Correct`). *Let q be a satisfiable ordering constraint, and $y : \mathbb{R}^m$ a vector. Then $\texttt{Correct}(q, y)$ satisfies q.*

Proof. Let $y_i < y_j$ be an atom in q. Reusing notation from Algorithm 3.3, let $y' = \texttt{Correct}(q, y)$, $y^s := \texttt{SortDescending}(y)$, and $\pi := \texttt{TopologicalSort}(\texttt{OrderGraph}(q), y)$. We have that (j, i) is an edge in $\texttt{OrderGraph}(q)$, which implies that $\pi(j) < \pi(i)$ by Eq. 1. Because the elements of y are sorted in descending order, and assumed to be distinct (Definition 1), we obtain that $y^s_{\pi(i)} < y^s_{\pi(j)}$, i.e., that $y'_i < y'_j$.

Theorem 5 (`SC` is a self-correcting transformer)**.** `SC` *(Algorithm 3.1) satisfies conditions* (i) *and* (ii) *of Definition 2.*

Proof. By definition of Algorithm 3.1, $\texttt{FindSatConstraint}(\Phi, x, y) = \bot$ if and only if $f^\Phi(x) = \texttt{SC}(\Phi)(f)(x)$ outputs $\bot$. We derive from Property 7(i) that this is equivalent to $\forall y'.\; \neg\Phi(x, y')$, which corresponds exactly to Property 2(ii). Conversely, if Φ is satisfiable for input x, i.e., $\exists y'.\; \Phi(x, y')$, then $\texttt{FindSatConstraint}(\Phi, x, y)$ outputs $q \neq \bot$. By definition, we have $f^\Phi(x) = \texttt{Correct}(q, y)$, which satisfies q by application of Property 8, which in turn implies that $\Phi(x, f^\Phi(x))$ by application of Property 7(ii).

Now that we have demonstrated that our approach produces safe-by-construction networks, we next prove that it also preserves the top predicted class when possible, i.e., that `SC` satisfies *transparency*, as formalized in Property 1.

Let $x : \mathbb{R}^n$ be an arbitrary vector. As in the previous section, if $\Phi(x, f(x))$ is initially satisfied, transparency trivially holds, as the correction layer does not modify the original output $f(x)$. When $\Phi(x, f(x))$ does not hold, we will rely on several additional properties about `FindSatConstraint`, `Correct`, and `OrderGraph`. The first, Property 9, states that whenever the index of the network's top prediction is a root of the graph encoding of q used by `FindSatConstraint` and `Correct`, then there exists an output which satisfies q that preserves that top prediction.

Property 9 (`OrderGraph`). *Let q be a satisfiable, disjunction-free ordering constraint, and $y : \mathbb{R}^m$ a vector. Then,*

$$\begin{aligned} &\operatorname*{argmax}_i\{y_i\} \in \mathit{Roots}(\texttt{OrderGraph}(q)) \iff \\ &\quad \exists y'.\ q(y') \ \wedge \ \operatorname*{argmax}_i\{y_i\} = \operatorname*{argmax}_i\{y'_i\} \end{aligned}$$

The intuition behind this property is that $i^* := \operatorname{argmax}_i\{y_i\}$ belongs to the roots of $\texttt{OrderGraph}(q)$ if and only if there is no $y_{i^*} < y_j$ constraint in q; hence since q is satisfiable, we can always permute indices in a solution y' to have $\operatorname{argmax}_i\{y'_i\} = i^*$. Formally, Lemma 1 in Sect. B.1 entails this property, as it shows that the permutation returned by `TopologicalSort` satisfies it.

Next, Property 10 formalizes the requirement that whenever `FindSatConstraint` returns a constraint (rather than $\bot$), then that constraint will not eliminate any top-prediction-preserving solutions that would otherwise have been compatible with the full set of safe-ordering constraints Φ.

Property 10 (`FindSatConstraint`). *Let Φ be a set of safe-ordering constraints, $x : \mathbb{R}^n$ and $y : \mathbb{R}^m$ two vectors, and $q = \texttt{FindSatConstraint}(\Phi, x, y)$. Then,*

$$\begin{aligned} &q \neq \bot \ \wedge \ \left(\exists y'.\ \Phi(x, y') \ \wedge \ \operatorname*{argmax}_i\{y_i\} = \operatorname*{argmax}_i\{y'_i\}\right) \implies \\ &\exists y'.\ q(y') \ \wedge \ \operatorname*{argmax}_i\{y_i\} = \operatorname*{argmax}_i\{y'_i\} \end{aligned}$$

Proof. Let us assume that $q \neq \bot$, and that $\exists y'.\ \Phi(x, y') \wedge \operatorname{argmax}_i\{y_i\} = \operatorname{argmax}_i\{y'_i\}$. We will proceed by contradiction, assuming that there does not exist y'' such that $q(y'')$ and $\operatorname{argmax}_i\{y_i\} = \operatorname{argmax}_i\{y''_i\}$, which entails that $\operatorname{argmax}_i\{y_i\} \notin \text{Roots}(\texttt{OrderGraph}(q))$ by application of Property 9. In combination with the specification of `Prioritize` (Property 3), this implies that any

$q' \in Q_p$ such that $\exists y'.\ q'(y') \wedge \text{argmax}_i\{y_i\} = \text{argmax}_i\{y'_i\}$ occurs before q in $\texttt{Prioritize}(Q_x, y)$, i.e., $\texttt{idx}(q', Q_p) < \texttt{idx}(q, Q_p)$. From loop invariant 6, we therefore conclude that there does not exist such a $q' \in Q_p$, which contradicts the hypothesis $\Phi(x, y')$ by application of Eq. 5.

Lastly, Property 11 states that `Correct` (Algorithm 3.3) will always find an output that preserves the original top prediction, whenever the constraint returned by `FindSatConstraint` allows it. This is the final piece needed to prove Theorem 6, the desired result about the self-correcting transformer.

Property 11 (`Correct`). *Let q be a satisfiable term, and $y : \mathbb{R}^m$ a vector. Then,*

$$\begin{aligned}&(\ \exists y'.\ q(y') \wedge \underset{i}{\text{argmax}}\{y_i\} = \underset{i}{\text{argmax}}\{y'_i\}\ )\\ &\implies \underset{i}{\text{argmax}}\{\texttt{Correct}(q, y)_i\} = \underset{i}{\text{argmax}}\{y_i\}\end{aligned}$$

Proof. Assume that there exists y' such that $q(y')$ and $\text{argmax}_i\{y_i\} = \text{argmax}_i\{y'_i\}$. This entails that $\text{argmax}_i(y_i) \in \text{Roots}(\texttt{OrderGraph}(q))$ (Property 9), which in turn implies that $\pi(\text{argmax}_i\{y_i\})$ is 0 (Property 4). By definition of a descending sort, we have that $\text{argmax}_i\{\texttt{Correct}(q, y)_i\} = j$, such that $\pi(j) = 0$, hence concluding that $j = \text{argmax}_i\{y_i\}$ by injectivity of π.

Theorem 6 (Transparency of `SC`)**.** `SC`*, the self-correcting transformer described in Algorithm 3.1 satisfies Property 1.*

Proof. That the `SC` transformer satisfies transparency is straightforward given Properties 9–11. Let us assume that there exists y' such that $\Phi(x, y')$ and $\text{argmax}_i\{y'_i\} = F(x)$. By application of Property 7(i), this implies that $\texttt{FindSatConstraint}(\Phi, x, f(x))$ outputs $q \neq \bot$, and therefore that there exists y' such that $q(y')$ and $\text{argmax}\{y'_i\} = F(x)$ by application of Property 10, since $F(x)$ is defined as $\text{argmax}_i\{f_i(x)\}$. Composing this fact with Property 11, we obtain that $F^{\Phi}(x) = F(x)$, since $F^{\Phi}(x) = \text{argmax}_i\{f_i^{\Phi}(x)\}$ by definition.

B Appendix 2: Vectorizing Self-Correction

Several of the subroutines of `FindSatConstraint` and `Correct` (Algorithms 3.2 and 3.3 presented in Sect. 3) operate on an `OrderGraph`, which represents a conjunction of ordering literals, q. An `OrderGraph` contains a vertex set, V, and edge set, E, where V contains a vertex, i, for each class in $\{0, \ldots, m-1\}$, and E

Algorithm B.1: Stable Topological Sort

Inputs: A graph, G, represented as an $m \times m$ adjacency matrix, and a vector, $y : \mathbb{R}^m$

Result: A permutation, $\pi : [m] \to [m]$

```
TopologicalSort(G , y):
    P := all_pairs_longest_paths(G)
    ∀ i, j ∈ [m] . P'_ij := { y_i  if P_ij ≥ 0
                            { ∞    otherwise
    ∀ j ∈ [m] . v_j := min_i { P'_ij }
     // set the value of each vertex to the
                                  // smallest value among its ancestors
    ∀ j ∈ [m] . d_j := max_i { P_ij }
     // calculate the depth of each vertex
    return argsort([ ∀j ∈ [m] . (−v_j, d_j) ])
     // break ties in favor of minimum depth
```

contains an edge, (i, j), from vertex i to vertex j if the literal $y_j < y_i$ is in q. We represent an `OrderGraph` as an $m \times m$ adjacency matrix, M, defined according to Eq. 7.

$$M_{ij} := \begin{cases} 1 & \text{if } (i,j) \in E; \text{ i.e., } y_j < y_i \in q \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Section B.1 describes the matrix-based algorithm that we use to conduct the stable topological sort that `Correct` (Algorithm 3.3) depends on. It is based on a classic parallel algorithm due to [9], which we modify to ensure that `SC` satisfies transparency (Property 1). Section B.2 describes our approach to cycle detection, which is able to share much of its work with the topological sort. Finally, Sect. B.3 discusses efficiently prioritizing ordering constraints, needed to ensure that `SC` satisfies transparency.

B.1 Stable Topological Sort

Our approach builds on a parallel topological sort algorithm given by [9], which is based on constructing an *all pairs longest paths* (APLP) matrix. However, this algorithm is not *stable* in the sense that the resulting order depends only on the graph, and not on the original order of the sequence, even when multiple orderings are possible. While for our purposes this is sufficient for ensuring safety, it is not for transparency. We begin with background on constructing the APLP matrix, showing that it is compatible with a vectorized implementation, and then describe how it is used to perform a stable topological sort.

All Pairs Longest Paths. The primary foundation underpinning many of the graph algorithms in this section is the *all pairs longest paths* (APLP) matrix, which we will denote by P. On acyclic graphs, P_{ij} for $i, j \in [m]$ is defined to be the length of the *longest* path from vertex i to vertex j. Absent the presence of cycles, the distance from a vertex to itself, P_{ii}, is defined to be 0. For vertices i and j for which there is no path from i to j, we let $P_{ij} = -\infty$.

We compute P from M using a matrix-based algorithm from [9], which requires taking $O(\log m)$ matrix *max-distance products*, where the max-distance product is equivalent to a matrix multiplication where element-wise multiplications have been replaced by additions and element-wise additions have been replaced by the pairwise maximum. That is, a matrix product can be abstractly written with respect to operations $\otimes$ and $\oplus$ according to Eq. 8, and the max-distance product corresponds to the case where $x \otimes y := x + y$ and $x \oplus y := \max\{x, y\}$.

$$(AB)_{ij} := (A_{i1} \otimes B_{1j}) \oplus \ldots \oplus (A_{ik} \otimes B_{kj}) \tag{8}$$

Using this matrix product, $P = P^{2^{\lceil \log_2(m) \rceil}}$ can be computed recursively from M by performing a fast matrix exponentiation, as described in Eq. 9.

$$P^k = P^{k/2} P^{k/2} \qquad P^1_{ij} = \begin{cases} 1 & \text{if } M_{ij} = 1 \\ 0 & \text{if } M_{ij} = 0 \ \wedge \ i = j \\ -\infty & \text{otherwise} \end{cases} \tag{9}$$

Stable Sort. We propose a stable variant of the [9] topological sort, shown in Algorithm B.1. Crucially, this variant satisfies Property 4 (Lemma 1), which Sect. 3.2 identifies as sufficient for ensuring transparency. Essentially, the value of each logit y_j is adjusted so that it is at least as small as the smallest logit value corresponding to vertices that are parents of vertex j, including j itself. A vertex, i, is a parent of vertex j if $P_{ij} \geq 0$, meaning that there is some path from vertex i to vertex j or $i = j$. The logits are then sorted in descending order, with ties being broken in favor of minimum depth in the dependency graph. The depth of vertex j is the maximum of the j^{th} column of P_{ij}, i.e., the length of the longest path from any vertex to j. An example trace of Algorithm B.1 is given in Fig. 2. By adjusting y_j into v_j such that for all ancestors, i, of j, $v_i \geq v_j$, we ensure each child vertex appears after each of its parents in the returned ordering–once ties have been broken by depth—as the child's depth will always be strictly larger than that of any of its parents since a path of length d to an immediate parent of vertex j implies the existence of a path of length $d + 1$ to vertex j.

Lemma 1. `TopologicalSort` *satisfies Property 4.*

Proof. Note that the adjusted logit values, v, are chosen according to Eq. 10.

$$v_j := \min_{i \ \mid \ i \text{ is an ancestor of } j \ \vee \ i=j} \left\{ \ y_i \ \right\} \tag{10}$$

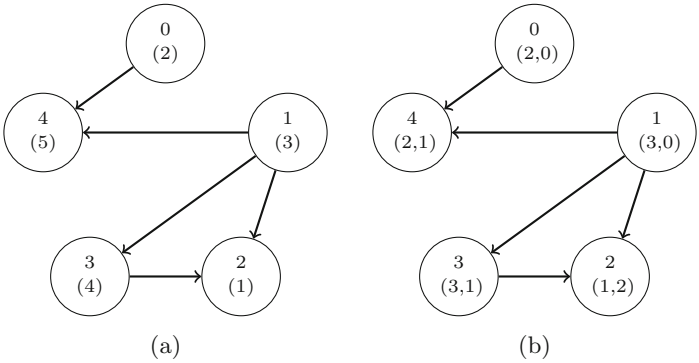

Fig. 2. Example trace of Algorithm B.1. **(a)**: The dependency graph and original logit values, y. The values of each logit are provided; the non-bracketed number indicates the logit index and the number in brackets is the logit value, e.g., $y_0 = 2$. Arrows indicate a directed edge in the dependency graph; e.g., we require $y_4 < y_0$. **(b)**: updated values passed into `argsort` as a tuple. For example, y_4 is assigned $(2, 1)$, as its smallest ancestor (y_0) has logit value 2 in (a) and its depth is 1; and y_2 is assigned value $(1, 2)$ because its logit value in (a), 1, is already smaller than that any of its parents, and its depth is 2. The values are sorted by *decreasing* value and *increasing* depth, thus the final order is $\langle y_1, y_3, y_0, y_4, y_2 \rangle$, corresponding to the permutation π, where $\pi(0) = 2$, $\pi(1) = 0$, $\pi(2) = 4$, $\pi(3) = 1$, and $\pi(4) = 3$.

We observe that *(i)* for all root vertices, i, $v_i = y_i$, and *(ii)* the root vertex with the highest original logit value will appear first in the topological ordering. The former follows from the fact that the root vertices have no ancestors. The latter subsequently follows from the fact that the first element in a valid topological ordering must correspond to a root vertex. Thus if $\text{argmax}_i\{y_i\} = i^* \in \text{Roots}(g)$, then i^* is the vertex with the highest logit value, and so by *(ii)*, it will appear first in the topological ordering produced by `TopologicalSort`, establishing Property 4.

B.2 Cycle Detection

`IsSat`, a subroutine of `FindSatConstraint` (Algorithm 3.2) checks to see if an ordering constraint, q, is satisfiable by looking for any cycles in the corresponding dependency graph, `OrderGraph`(q). Here we observe that the existence of a cycle can easily be decided from examining P, by checking if $P_{ii} > 0$ for some $i \in [m]$; i.e., if there exists a non-zero-length path from any vertex to itself. Since $P_{ii} \geq 0$, this is equivalent to $\text{Trace}(P) > 0$. While strictly speaking, P_{ij}, as constructed by [9], only reflects the longest path from i to j in *acyclic* graphs, it can nonetheless be used to detect cycles in this way, as for any $k \leq m$, P_{ij} is guaranteed to be at least k if there exists a path of length k from i to j, and any cycle will have length at most m.

B.3 Prioritizing Root Vertices

As specified in Property 3, in order to satisfy transparency, the search for a satisfiable ordering constraint performed by `FindSatConstraint` must prioritize constraints, q, in which the original predicted class, $F(x)$, is a root vertex in q's corresponding dependency graph. We observe that root vertices can be easily identified using the dependency matrix M. The in-degree, d_j^{in}, of vertex j is simply the sum of the j^{th} column of M, given by Eq. 11. Meanwhile, the root vertices are precisely those vertices with no ancestors, that is, those vertices j satisfying Eq. 11.

$$d_j^{in} = \sum_{i \in [m]} M_{ij} = 0 \tag{11}$$

In the context of `FindSatConstraint`, the subroutine `Prioritize` lists ordering constraints q for which $d_{F(x)}^{in} = 0$ in `OrderGraph`(q) before any other ordering constraints. To save memory, we do not explicitly list and sort all the disjuncts of Q_x (the DNF form of the active postconditions for x); rather we iterate through them one at a time. This can be done by, e.g., iterating through each disjunct twice, initially skipping any disjunct in which $F(x)$ is not a root vertex, and subsequently skipping those in which $F(x)$ is a root vertex.

C Appendix 3: Generation of Synthetic Data

In Sect. 5.4, we utilize a family of synthetic datasets with associated safe-ordering constraints that are randomly generated according to several specified parameters, allowing us to assess how aspects such as the number of constraints (α), the number of disjunctions per constraint (β), and the dimension of the output vector (m) impact the run-time overhead. In our experiments, we fix the input dimension, n, to be 10. The synthetic data, which we will denote by $\mathcal{D}(\alpha, \beta, m)$, are generated according to the following procedure.

(i) First, we generate α random safe-ordering constraints. The preconditions take the form $b_\ell \leq x \leq b_u$, where b_ℓ is drawn uniformly at random from $[0.0, 1.0 - \epsilon]$ and $b_u := b_\ell + \epsilon$. We choose $\epsilon = 0.4$ in our experiments; as a result, the probability that any two preconditions overlap is approximately 30%. The ordering constraints are disjunctions of β randomly-generated cycle-free ordering graphs of m vertices, i.e., β disjuncts. Specifically, in each graph, we include each edge, (i, j), for $i \neq j$ with equal probability, and require further that at least one edge is included, and the expected number of edges is γ (we use $\gamma = 3$ in all of our experiments). Graphs with cycles are resampled until a graph with no cycles is drawn.

(ii) Next, for each safe-ordering constraint, ϕ, we sample N/α random inputs, x, uniformly from the range specified by the precondition of ϕ. In all of our experiments we let $N = 2{,}000$. For each x, we select a random disjunct from the postcondition of ϕ, and find the roots of the corresponding ordering graph. We select a label, y^* for x uniformly at random from this set of

roots, i.e., we pick a random label for each point that is consistent with the property for that point.

(iii) Finally, we generate N random points that do not satisfy any of the preconditions of the α safe-ordering constraints. We label these points via a classifier trained on the N labeled points already generated in (ii). This results in a dataset of $2N$ labeled points, where 50% of the points are captured by at least one safe-ordering constraint.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 265–283 (2016)
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
3. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 731–744. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3314221.3314614
4. Anil, C., Lucas, J., Grosse, R.: Sorting out Lipschitz function approximation. In: International Conference on Machine Learning, pp. 291–301. PMLR (2019)
5. Berger, E.D., Zorn, B.G.: DieHard: probabilistic memory safety for unsafe languages. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 158–168. Association for Computing Machinery, New York (2006)
6. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_51
7. Brown, T.B., et al.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020)
8. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium, pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7
9. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAM J. Comput. **10**, 657–675 (1981)
10. Donti, P.L., Roderick, M., Fazlyab, M., Kolter, J.Z.: Enforcing robust control guarantees within neural network policies. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=5lhWG3Hj2By
11. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T., Kohli, P.: A dual approach to scalable verification of deep networks. In: Proceedings of the Thirty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI 2018), Corvallis, Oregon, pp. 162–171. AUAI Press (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. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002 (2002)
14. Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.: DL2: training and querying neural networks with logic. In: International Conference on Machine Learning, pp. 1931–1941. PMLR (2019)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18 (2018)
16. Guttmann, W., Maucher, M.: Variations on an ordering theme with constraints. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) TCS 2006. IIFIP, vol. 209, pp. 77–90. Springer, Boston, MA (2006). https://doi.org/10.1007/978-0-387-34735-6_10
17. Havelund, K., Rosu, G.: Monitoring programs using rewriting. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pp. 135–143 (2001)
18. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016)
19. 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
20. Julian, K.D., Kochenderfer, M.J., Owen, M.P.: Deep neural network compression for aircraft collision avoidance systems. J. Guid. Control Dyn. **42**(3), 598–608 (2019)
21. 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
22. Kling, M., Misailovic, S., Carbin, M., Rinard, M.: Bolt: on-demand infinite loop escape in unmodified binaries. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 431–450. Association for Computing Machinery, New York (2012)
23. Kochenderfer, M.J., et al.: Optimized airborne collision avoidance, pp. 249–276 (2015)
24. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Technical report 0, University of Toronto, Toronto, Ontario (2009)
25. Leino, K., Fredrikson, M.: Relaxing local robustness. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
26. Leino, K., Wang, Z., Fredrikson, M.: Globally-robust neural networks. In: International Conference on Machine Learning (ICML) (2021)
27. Li, Q., Haque, S., Anil, C., Lucas, J., Grosse, R.B., Jacobsen, J.H.: Preventing gradient attenuation in lipschitz constrained convolutional networks. In: Advances in Neural Information Processing Systems 32, pp. 15390–15402 (2019)
28. Li, T., Gupta, V., Mehta, M., Srikumar, V.: A logic-driven framework for consistency of neural models. In: Inui, K., Jiang, J., Ng, V., Wan, X. (eds.) Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, 3–7 November 2019, pp. 3922–3933. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/D19-1405

29. Lin, X., Zhu, H., Samanta, R., Jagannathan, S.: ART: abstraction refinement-guided training for provably correct neural networks. In: 2020 Formal Methods in Computer Aided Design (FMCAD), pp. 148–157 (2020)
30. Long, F., Sidiroglou-Douskos, S., Rinard, M.: Automatic runtime error repair and containment via recovery shepherding. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 227–238. Association for Computing Machinery, New York (2014)
31. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: International Conference on Learning Representations (2018). https://openreview.net/forum?id=rJzIBfZAb
32. Meyer, B.: Eiffel: The Language (1992)
33. Mirman, M., Gehr, T., Vechev, M.: Differentiable abstract interpretation for provably robust neural networks. In: International Conference on Machine Learning, pp. 3578–3586. PMLR (2018)
34. Müller, C., Serre, F., Singh, G., Püschel, M., Vechev, M.: Scaling polyhedral neural network verification on GPUS. In: Proceedings of Machine Learning and Systems 3 (2021)
35. Nieuwenhuis, R., Rivero, J.M.: Practical algorithms for deciding path ordering constraint satisfaction. Inf. Comput. **178**(2), 422–440 (2002). https://doi.org/10.1006/inco.2002.3146
36. Perkins, J.H., et al.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 87–102. Association for Computing Machinery, New York (2009)
37. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies–a safe method to survive software failures. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 235–248. Association for Computing Machinery, New York (2005)
38. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebee, W.S.: Enhancing server availability and security through failure-oblivious computing. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6, OSDI 2004, p. 21. USENIX Association, Berkeley (2004)
39. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: Proceedings of the ACM on Programming Languages, 3(POPL), January 2019
40. Sotoudeh, M., Thakur, A.V.: Provable repair of deep neural networks. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 588–603 (2021)
41. Tan, M., Le, Q.: EfficientNet: rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning, pp. 6105–6114. PMLR (2019)
42. Tan, M., Le, Q.V.: EfficientNetV2: smaller models and faster training. arXiv preprint arXiv:2104.00298 (2021)
43. Trockman, A., Kolter, J.Z.: Orthogonalizing convolutional layers with the Cayley transform. In: International Conference on Learning Representations (2021)
44. Urban, C., Christakis, M., Wüstholz, V., Zhang, F.: Perfectly parallel fairness certification of neural networks. In: Proceedings of the ACM on Programming Languages, 4(OOPSLA), November 2020. https://doi.org/10.1145/3428253

45. Wu, H., et al.: Parallelization techniques for verifying neural networks. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, 21–24 September 2020, pp. 128–137. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_20
46. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 686–701. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3314221.3314638