PEREGRiNN: Penalized-Relaxation Greedy Neural Network Verifier

Haitham Khedr(✉), James Ferlez, and Yasser Shoukry

University of California, Irvine, USA
{hkhedr,jferlez,yshoukry}@uci.edu

Abstract. Neural Networks (NNs) have increasingly apparent safety implications commensurate with their proliferation in real-world applications: both unanticipated as well as adversarial misclassifications can result in fatal outcomes. As a consequence, techniques of formal verification have been recognized as crucial to the design and deployment of safe NNs. In this paper, we introduce a new approach to formally verify the most commonly considered safety specifications for ReLU NNs – i.e. polytopic specifications on the input and output of the network. Like some other approaches, ours uses a relaxed convex program to mitigate the combinatorial complexity of the problem. However, unique in our approach is the way we use a convex solver not only as a linear feasibility checker, but also as a means of penalizing the amount of relaxation allowed in solutions. In particular, we encode each ReLU by means of the usual linear constraints, and combine this with a convex objective function that penalizes the discrepancy between the output of each neuron and its relaxation. This convex function is further structured to force the largest relaxations to appear closest to the input layer; this provides the further benefit that the most "problematic" neurons are conditioned as early as possible, when conditioning layer by layer. This paradigm can be leveraged to create a verification algorithm that is not only faster in general than competing approaches, but is also able to verify considerably more safety properties; we evaluated PEREGRiNN on a standard MNIST robustness verification suite to substantiate these claims.

Keywords: Machine learning/AI · Decision procedures and solvers

1 Introduction

Neural Networks have become an increasingly central component of modern machine learning systems, including those that are used in safety-critical cyber-physical systems such as autonomous vehicles. The rate of this adoption has exceeded the ability to reliably verify the safe and correct functioning of these components, especially when they are integrated with other components such as

This work was sponsored by the NSF awards #CNS-2002405 and #CNS-2013824.

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 287–300, 2021.
https://doi.org/10.1007/978-3-030-81685-8_13

controllers. Thus, there is an increasing need to verify that NNs reliably produce safe outputs, especially subject to malicious adversarial inputs [16,20,27,28].

In this paper, we propose PEREGRiNN, an algorithm for efficiently and formally verifying the input/output behavior of ReLU NNs. In this context, PEREGRiNN falls into the broad category of sound and complete *search and optimization* NN verifiers [22]. The *search* aspect of PEREGRiNN involves iterating over different combinations of neuron activation patterns to verify that each is compatible with the specified safety constraints (on the input and output of the network). Like other algorithms in this category, PEREGRiNN combines this search with *optimization* techniques to make inferences about the feasibility of full-network activation patterns on the basis of activation patterns of only a subset of neurons. The optimization in question reformulates the original NN feasibility problem into a relaxed convex feasibility problem to allow sound inferences: i.e. if the convex relaxation is infeasible, then the original NN problem may soundly be concluded to be infeasible. In this relaxed feasibility problem, the output of each individual neuron is assigned a relaxation variable that is decoupled from the actual output of that neuron. PEREGRiNN also uses a type of reachability analysis (symbolic interval analysis) both to enhance the optimization-based inference described above and as a source of additional sound inference itself. For this reason, PEREGRiNN's search procedure searches neurons in a layer-by-layer fashion, preferring to fix the phases of neurons closest to the input layer first.

In contrast to other search and optimization algorithms, however, PEREGRiNN *augments* each convex feasibility query with a (convex) penalty function in order to obtain better guidance on which activation patterns to search next. In particular, we note that the amount of relaxation needed on a neuron can be regarded as a *quasi-measure* of how close the convex solver came to operating the associated neuron in a valid regime – i.e. at a valid evaluation of that neuron on a particular input. In this sense, the amount of relaxation in aggregate can be regarded as a quasi-measure of how close the solver came to finding a valid evaluation of the network as a whole. Inversely, the largest distance between a relaxation variable and its neuron's closest ReLU constraint intuitively corresponds in some sense to how "problematic" that neuron is with regard to obtaining such a valid evaluation. These distances we refer to as the *"slacks"* for each neuron. Thus, PEREGRiNN may be regarded as *greedily* minimizing a *slack-based penalty*.

Finally, we evaluated the performance of PEREGRiNN by using it to verify the adversarial robustness of networks trained on the MNIST [21] dataset. Our experiments show that PEREGRiNN is on average 1.27× faster than Neurify [31], 1.24× faster than Venus [6], 1.15× faster than nnenum [4], and 1.65× faster than Marabou [19]. It also proves 27%, 19%, 10%, and 51% more properties than the other solvers, respectively. PEREGRiNN's unique convex penalty augmentations are also considered in ablation experiments to validate their benefits.

Related Work. Since PEREGRiNN is a sound and complete verification algorithm, we restrict our comparison to other sound and complete algorithms. NN verifiers can be grouped into roughly three categories: (i) SMT-based methods, which encode the problem into a Satisfiability Modulo Theory problem [11,18,19]; (ii) MILP-based solvers, which directly encode the verification

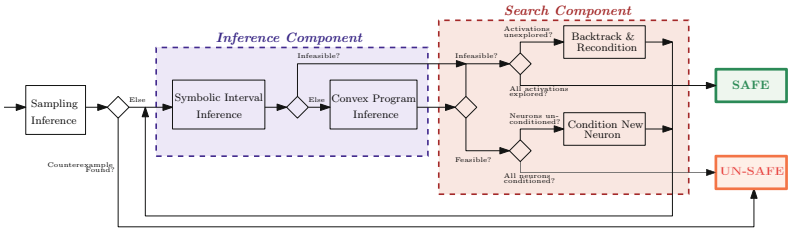


Fig. 1. Block diagram of the PEREGRiNN algorithm

problem as a Mixed Integer Linear Program [3,5–8,14,23,29]; (iii) Reachability based methods, which perform layer-by-layer reachability analysis to compute the reachable set [4,13,15,17,30,32,34,35]; and (iv) convex relaxations methods [10,31,33]. In general, (i), (ii) and (iii) suffer from poor scalability. On the other hand, convex relaxation methods depend heavily on pruning the search space of indeterminate neuron activations; thus, they generally depend on obtaining good approximate bounds for each of the neurons in order to reduce the search space (the exact bounds are computationally intensive to compute [9]). These methods are most similar to PEREGRiNN: for example, [7,25,32] recursively refine the problem using input splitting, and [31] does so via neuron splitting. Other search and optimization methods include: Planet [11], which combines a relaxed convex optimization problem with a SAT solver to search over neurons' phases; and Marabou [19], which uses a modified simplex algorithm.

2 Problem Formulation

In this paper, we will consider Rectified Linear Unit (ReLU) NNs. An n-layer ReLU network, is a composition of n ReLU layer functions: i.e. $\mathcal{NN} = f_n \circ f_{n-1} \circ \cdots \circ f_1$ where the i^{th} ReLU layer function is defined as $f_i : y \in \mathbb{R}^{k_{i-1}} \mapsto \max\{W_i y + b_i, 0\} \in \mathbb{R}^{k_i}$. We refer to f_1 as the *input layer*. Finally, to refer to individual neurons, we use the notation $(z)_j$ to indicate the j^{th} element of z.

Verification Problem. Let $\mathcal{NN}$ be an n-layer NN as defined above. Furthermore, let $P_{y_0} \subset \mathbb{R}^{k_0}$ be a convex polytope in the input space of $\mathcal{NN}$, and let $P_{y_n} \subset \mathbb{R}^{k_n}$ be a convex polytope in the output space of $\mathcal{NN}$. Finally, let $h_\ell : \mathbb{R}^{k_0} \times \mathbb{R}^{k_n} \to \mathbb{R}$, $\ell = 1, \ldots, m$ be convex functions defining joint input/output constraints on $\mathcal{NN}$. Then the verification problem is to decide whether

$$\left\{ x \in \mathbb{R}^{k_0} \;\middle|\; x \in P_{y_0} \bigwedge \mathcal{NN}(x) \in P_{y_n} \bigwedge \left(\bigwedge_{\ell=1}^{m} h_\ell(x, \mathcal{NN}(x)) \leq 0 \right) \right\} = \emptyset. \quad (1)$$

3 PEREGRiNN Overview

The general structure of PEREGRiNN is depicted in Fig. 1. Like other search and optimization based NN verifiers it has two main components: a *search component* and an *inference component*, and PEREGRiNN iterates back and forth

between these these two components until termination. In particular, the search and inference components interact in the following way. The search component successively iterates over all possible on/off activations for each neuron; this is done by fixing these activations one neuron at a time, starting from the input layer and working towards the output layer. The process of fixing a neuron's activation is referred to as *conditioning its phase*: each neuron can be in either its active phase (operating linearly) or inactive phase (outputting zero). Thus, the search component provides the inference component a subset of neurons, each of which has been conditioned; the inference component then attempts to soundly reason about whether the remaining, unconditioned neurons can be operated in such a way as to violate the safety constraint. If the inference component soundly concludes safety for all possible activations of the remaining unconditioned neurons, then the search component backtracks, oppositely reconditioning one of the neurons that was already conditioned. Otherwise, if a sound safe conclusion is not made, then the search component uses information from the inference component to decide on a new neuron to condition, and the process repeats. The algorithm terminates if either a counterexample to safety is found, or else all possible neuron activations are considered without finding such a counterexample.

The convex program inference block is at the heart of the inference component and PEREGRiNN itself. In this block, PEREGRiNN, like other search and optimization solvers, uses a relaxed linear feasibility program where the output of each individual neuron is assigned a relaxation variable that is decoupled from the actual output of that neuron. In the notation of Sect. 2, such a linear feasibility program can be written as follows, where the vector variables $y_i, i \neq 0$ are the relaxation variables.

$$\begin{cases} y_i \geq 0,\ y_i \geq W_i y_{i-1} + b_i & \forall i = 1, \ldots, n \\ y_0 \in P_{y_0},\ y_n \in P^{\mathsf{c}}_{y_n},\ \bigwedge_{\ell=1}^{m} h_\ell(y_0, y_n) \leq 0 \end{cases} \tag{2}$$

Importantly, if (2) is infeasible, then the original NN problem in (1) may be soundly concluded to be infeasible as well – and hence, safe. However, as described above, the primary function of the convex feasibility program is to use a set of conditioned neurons supplied by the search component in order to soundly reason about the remaining neurons. To do this, the conditioned neurons supplied by the search component are incorporated into the feasibility program (2) as *equality* constraints in the following way:

$$\text{Neuron } (y_i)_j \text{ ON:} \quad (y_i)_j = (W_i y_{i-1} + b_i)_j \wedge (y_i)_j \geq 0 \tag{3}$$

$$\text{Neuron } (y_i)_j \text{ OFF:} \quad (y_i)_j = 0 \wedge (W_i y_{i-1} + b_i)_j \leq 0. \tag{4}$$

Inferences created by the symbolic interval inference block using Symbolic Interval Analysis [32] are also incorporated using equality constraints like (3) and (4).

Of the remaining blocks, the "Backtracking & Reconditioning" block is essentially described above. The "Condition New Neuron" and "Sampling Inference" blocks have features unique to PEREGRiNN that are described in Sect. 4; the

former implements a novel neuron prioritization, and the latter is a unique approach to quickly obtaining initial safety counterexamples.

4 PEREGRiNN Enhancements

4.1 Sum-of-Slacks Penalty

The core enhancement in PEREGRiNN is the inclusion of a specific objective function in the convex program used by the inference component. As per the discussion above, this objective function is interpreted as a *penalty* on how far away a particular solution is from a valid input/output response of the network (and activation pattern on all hidden neurons). Specifically, this penalty function penalizes the sum of all of the "slack" variables for the entire network, where each neuron's slack variable is defined as $s_i \triangleq y_i - (W_i \cdot y_{i-1} + b_i)$. That is the distance between a relaxation variable y_i and the linear response of its associated neuron. During each feasibility/inference call, this has the obvious effect of incentivizing the convex solver to choose an actual input/output response of the network.

In addition, this penalty is effectively the L_1-norm of the *vector* of all the slack variables, since the slack variables are non-negative. The L_1-norm of a vector, used as a penalty function, is well known to effectively encourage *sparsity* on the resulting optimal solution. Thus, the sum-of-slacks effectively incentivizes the convex solver to leave as *few* neurons as possible indeterminate in the solution. That is a sum-of-slacks penalty effectively encourages the convex solver to fix the phases of as many neurons as possible.

4.2 Max-Slack Conditioning Priority

As noted above, the search component of PEREGRiNN operates layer-wise from input layer to output layer in order to leverage Symbolic Interval Analysis for additional inference. Hence, the search component always chooses the next neuron to be searched (i.e. conditioned) from among those as-yet-unconditioned neurons that are closest to the input layer. It further makes sense to only consider conditioning neurons that the convex solver was unable to operate at valid inputs/output. However, the convex solver typically returns several neurons to choose from with this property, and it is necessary to choose which of them to search next. Given the interpretation of a neuron's "slack" variable as a measure of how "problematic" that neuron was for the solver to obtain a valid evaluation of the network, PEREGRiNN's search component chooses the next neuron to condition based on slack-order ranking of those neurons that are not being operated at valid input/output points. This "max-slack" heuristic choice is unique to PEREGRiNN; compare to the output gradient heuristic employed in [31].

4.3 Layer-wise-Weighted Penalty

PEREGRiNN takes the "max-slack" neuron search priority one step further, though. Using techniques similar to those in [26], it is possible to show that

there exists weights $q_1, \ldots, q_n$ such that solving (2) with the penalty

$$\min_{y_0,..,y_n} \sum_{i=0}^{n} \sum_{j=1}^{k_i} q_i s_{ij} \tag{5}$$

will result in a solution that is guaranteed to concentrate the most total slack in the earliest (unconditioned) layer. Thus, by using the layer-wise weighted sum-of-slacks penalty in (5), PEREGRiNN is uniquely able to force the (unconditioned) layer closest to the input layer to have the *largest* total slack among all the layers. As a consequence, PEREGRiNN effectively concentrates the most "problematic" neurons in the layer where the next conditioning choice will be made. This scheme makes it much more likely that the neuron with the highest slack among *all* of the neurons will be among the next neurons considered for conditioning – in effect, often guiding the search component to condition on the most problematic neuron in the whole network (although this is not guaranteed).

As noted above, SMC [26] can be used to obtain layer-wise weights that guarantee concentration of slack in the earliest (shallowest) layer. However, these weights are often very large, since they depend on bounding the slack variables (most readily by over-approximation); the effect of this is possible computational instability in the convex program. Thus, as an *implementation* matter, we instead select these weights using a heuristic scheme characterized by two real-valued hyperparameters, λ_0 and γ. In particular, the weight of the i^{th} layer, q_i, is selected as $q_i = \lambda_0 \cdot \gamma^i$. In our experiments, we found the values $\lambda_0 = 10^{-7}$ and $\gamma = 10^3$ to effectively achieve the maximum slack concentration in the earliest layers.

4.4 Initial Counterexample Search by Sampling

Finally, PEREGRiNN extends a simple idea first introduced in [32] to rapidly identify counterexamples by means of sampling. The basic idea is to sample within a known region of the input to the NN (or the input to some deeper layer), and evaluate the NN (sub-NN) exactly on those samples in order to rapidly identify a counterexample; this approach help identify un-safe networks/properties early on. However, whereas [32] samples from within hyper-rectangle sets derived by symbolic interval analysis, PEREGRiNN uses the Volesti [12] Python library to uniformly sample points within the *polytopic* input constraint set, P_{y_0}, and thus applies to be more general input constraint sets in (1).

5 Experiments

We evaluated the performance and effectiveness of PEREGRiNN at verifying the adversarial robustness of NNs trained to recognize digits using the standard MNIST dataset. This verification problem fits into the general NN verification problem described in Sect. 2, and it is described subsequently in detail. In this context, we evaluated PEREGRiNN with two objectives described as follows.

Table 1. Architecture of the NN models used in the experiments

Models	# ReLUs	Architecture
MNIST_FC1	512	$< 784, 256, 256, 10 >$
MNIST_FC2	1024	$< 784, 256, 256, 256, 256, 10 >$
MNIST_FC3	1536	$< 784, 256, 256, 256, 256, 256, 256, 10 >$

1. We conducted ablation experiments for all of PEREGRiNN's novel features as described in Sect. 4. In particular, we compared the performance of a full implementation of PEREGRiNN – i.e. *exactly* as described in Sect. 4 – with implementations that are otherwise the same except for changing one and only one of the following: the penalty function used in the convex program inference block; the neuron prioritization used by the search component.
2. We compared PEREGRiNN against other state-of-the-art NN verifiers, both in terms of the time required to verify individual networks and properties and in terms of the number of properties proved with a common, fixed timeout.

Implementation. We implemented PEREGRiNN in Python, and used an off-the-shelf Gurobi 9.1 [1] convex optimizer for solving linear programs; the Volesti [12] Python interface was used to sample from the input polytope for the sampling inference block. For the other NN verifiers, we used publicly available implementations that were published by their creators (citations are included below). Each instance of of any verifier was run within its own single-core Virtual Box VM with 30 GB of memory; no more than 4 VMs were run concurrently on a host machine with 48 hyperthreaded cores and 256 GB of memory.

5.1 Adversarial Robustness Verification Task

Subsequent experiments used the testbench we describe in this section; it is largely identical to the PAT-FCN test in the VNN-COMP 2020 competition [2].

Neural Networks. We used three ReLU NNs to recognize digits using the standard MNIST training database; these NNs are exactly as in the PAT-FCN portion of [2]. The sizes of these fully-connected networks are described in Table 1. Each entry in the "Architecture" column of Table 1 is the number of number of neurons in a layer, from input layer on the left to output layer on the right.

Verification Properties. We created a number of NN verification tasks based on proving whether the above described networks were robust against max-norm perturbations of their inputs. In particular, each verification task involves proving whether a particular input image, x', always results in the same classification when it is subjected to a max-norm perturbation of at most some fixed size, $\epsilon > 0$. Thus, each such verification problem is parameterized by both the specified input image, x', and the maximum amount of perturbation, ϵ.

Formally, let x' be a given image in category $t \in \{1, \ldots, M\}$, and let $\epsilon > 0$ be a specified maximum amount of max-norm perturbation of x'. Then we say that a NN with M classification outputs, $\mathcal{NN}$, is robust if for each classification category $m \in \{1, \ldots, M\} \setminus \{t\}$ the set of inputs yielding classification of x' as m

$$\phi_m \triangleq \{x \mid x \in \mathbb{R}^{k_0},\ \|x - x'\|_\infty \leq \epsilon,\ z \in \mathbb{R}^{k_n},\ \max_{i=1,\ldots,n} \mathcal{NN}(x)_i = \mathcal{NN}(x)_m\} \quad (6)$$

is empty. Note that each instance of (6) is compatible with the problem in (1).

Adversarial Robustness Verifier Testbench. Our verification testbench was then constructed by selecting 50 test images from the MNIST test dataset; this set of test images includes the 25 used in the PAT-FCN portion of [2]. Each test instance was then a combination of one of those images, one of the networks from Table 1 and one the following two max-norm perturbations, $\epsilon = 0.02$ or $\epsilon = 0.05$; these perturbations are same ones used in PAT-FCN [2]. Thus, each verification test in our testbench can be identified by one of 300 tuples of the form: $(net, image, perturb.) \in \mathscr{TB} \triangleq \{\texttt{FC1}, \texttt{FC2}, \texttt{FC2}\} \times \{1, \ldots, 50\} \times \{0.02, 0.05\}$.

5.2 Ablation Experiments

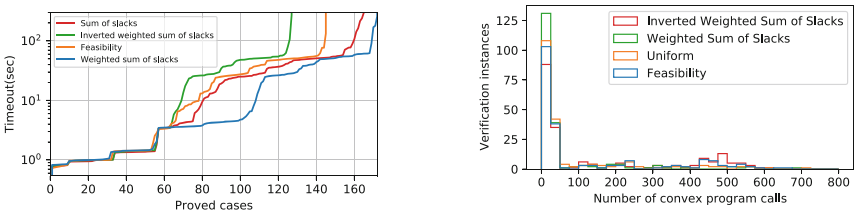
In this series of experiments we evaluated the contribution that each of the primary PEREGRiNN enhancements made to its overall performance. This was done by comparing the full PEREGRiNN algorithm – as described in Sect. 4 – with altered versions that replace exactly one of those enhancements at a time. ***Note:*** removing core features of PEREGRiNN often resulted in much longer run times, so the experiments in this section use a testbench $\mathscr{TB}' \subset \mathscr{TB}$ that excludes all tests with one of the larger networks `FC2` or `FC3` *and* $\epsilon = 0.05$.

Penalty Function Ablation. Our first ablation experiment evaluated the contribution of PEREGRiNN's unique penalty function features; see Sect. 4.1 and Sect. 4.3. In particular, we ran different variants of PEREGRiNN with the following penalty functions used inside the convex program inference block:

1. *"Weighted sum of slacks"*: PEREGRiNN's own weighted sum of slacks penalty;
2. *"Sum of slacks"*: A sum-of-slacks penalty with equal weighting on all layers;
3. *"Feasibility"*: A feasibility-only convex program such as the one used in other tools, e.g. [31] (i.e. simply using a constant penalty function of 1);
4. *"Inverted weighted sum of slacks"*: PEREGRiNN's own weighted sum of slacks penalty, except with the layer-wise weights applied in reverse order to force slack towards deeper layers rather than shallower ones (see also Sect. 4.3).

Figure 2a shows a cactus plot of the number of proved cases vs. the timeout permitted to the algorithm: i.e. to prove at least a specified number of the test cases, each algorithm must have its timeout set at to the value of its curve in

Fig. 2a. Figure 2b shows a histogram of the number of times each of the algorithm variants needed to call the convex solver in order to terminate; this quantifies each algorithm's cost in a well-known unit of computation, also the single most computationally costly part of PEREGRiNN. Figure 2b plots the number of convex solver calls required for evenly spaced bins of convex solver calls.



(a) Cactus plot; proved cases vs. timeout (b) Histogram; number convex calls used

Fig. 2. Performance of PEREGRiNN variants with different objective functions

Conclusions: Figure 2a demonstrates that PEREGRiNN's weighted sum of slacks has a clear benefit over both a uniformly weighted sum-of-slacks penalty and a plain feasibility convex program. For timeouts of longer than ≈ 1.2 seconds, PEREGRiNN overtakes the other two in terms of number of properties proved; even the uniform sum-of-slacks penalty considerably outperforms the feasibility convex program at similar timeouts. Note that *reversing* the layer-wise weights of PEREGRiNN's penalty function incurs a *performance hit*, especially for timeouts >1.2 s. This suggests that driving slacks toward shallower layers, where the next neuron is conditioned, is the correct heuristic to apply. Figure 2b also shows that going from feasibility to sum-of-slacks to weighted sum-of-slacks significantly reduces the number of test cases that require between 425 and 525 calls to the convex solver. This order of comparison shows a concomitant net influx of tests into the lowest bin of < 25 convex calls; PEREGRiNN has the most test cases in this category, with $\approx$130 test cases proved in < 25 convex solver calls.

Neuron Conditioning Priority Ablation. In the second ablation experiment, we evaluated the contribution of PEREGRiNN's maximum-slack neuron conditioning priority (see Sect. 4.2). To that end, we ran variants of PEREGRiNN with three different neuron conditioning priorities for the search component:

1. *"Maximum slack"*: PEREGRiNN's max-slack neuron conditioning priority;
2. *"Minimum slack"*: This variant conditions the neuron with the smallest slack;
3. *"Random choice"*: This variant conditions on a random indeterminate neuron.

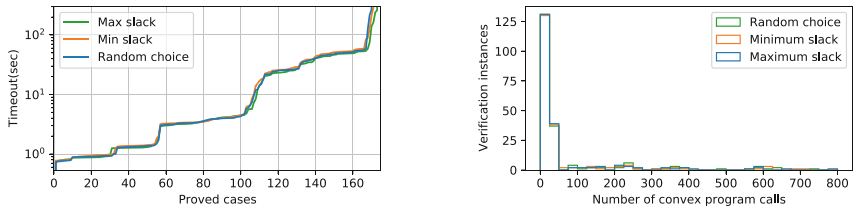
The performance of these algorithm variants is shown in Fig. 3a and Fig. 3b. As in the previous ablation experiment, Fig. 3a shows a cactus plot of the number

of proved cases vs. the timeout, and Fig. 3b shows a histogram of the number of calls to the convex solver required under each of the conditioning priorities.

Conclusions: Figure 3a shows that PEREGRiNN's max-slack neuron priority allows it to prove slightly more properties than either a random neuron choice priority or the minimum-slack priority. The maximum slack priority also required the fewest total convex calls across all instances: it used 178 fewer than minimum slack and 686 fewer than a random choice. Thus, we conclude PEREGRiNN's max-slack heuristic slightly improves performance on this testbench.

5.3 Comparison with Other NN Verifiers

In this experiment, we evaluated PEREGRiNN with respect to a number of state-of-the-art NN verifiers on our adversarial robustness testbench, $\mathscr{TB}$. In particular, we ran the following tools on $\mathscr{TB}$: Venus [6]; Marabou [19]; Neurify [31]; and nnenum [4]. Venus was run with `st_ratio`=0.4, `depth_power`=4, `offline_deps` = `True`, `online_deps` = `True`, and `ideal_cuts` = `True`; Marabou and Neurify were used with default parameters but `THREADS` = 1; and nnenum had `ADVERSARIAL_SEARCH` turned off. Each algorithm had its own one-core VM.



(a) Cactus plot; proved cases vs. timeout (b) Histogram; number convex calls used

Fig. 3. Performance of PEREGRiNN variants with different conditioning priorities

Figure 4 contains a cactus plot showing the results for each of these algorithms, including PEREGRiNN. For a given number of test cases to be proved, Fig. 4 depicts the corresponding timeout required for each of the algorithm to prove that many cases. Of all the algorithms, PEREGRiNN was able to prove the most properties within the timeout limit of 600 s: PEREGRiNN was able to prove 190 properties; it was followed by nnenum, which proved 172; Venus, which proved 159; Neurify, which proved 149; and Marabou, which proved 125. Marabou consistently performed the worst, proving fewer cases than any other algorithm at every timeout. By contrast, Neurify was able to prove significantly more test cases than any other algorithm for extremely short timeouts, but it failed to prove more than 150 out of 300 test cases across the whole experiment. nnenum performed worse than Neurify on the way to proving 150 test cases, but it fared significantly better than either PEREGRiNN or Venus, which had more or less similar performance below this threshold. However, after ≈150 test cases,

PEREGRiNN significantly outperformed all other algorithms: as the timeout was increased, PEREGRiNN proved additional properties at a rate significantly outpacing its closest competitor in this regime, nnenum. We further note that all algorithms proved a mixture of SAT and UNSAT properties.

This data, taken as a whole, suggests that PEREGRiNN suffers from a worse "best-case" performance than several other algorithms, especially nnenum and Neurify. However, PEREGRiNN's performance seems to be much more consistent across different test cases. This allows it to prove more properties in aggregate at the expense of being slower on a smaller subset of them. This further suggests that PEREGRiNN is significantly less sensitive to peculiarities of particular test cases on the $\mathscr{TB}$ testbench. This will likely be a considerable advantage, on average, when faced with verifying unknown networks and properties of this type.

6 Discussion: Analogy to SAT Solvers

It is possible to draw a loose analogy between SAT solvers and search-and-optimization NN verifiers such as PEREGRiNN. Indeed, since each neuron has two phases, the operational phase of each neuron can be captured by a binary variable; then any valuation of *all* these variables can be interpreted as SAT or UNSAT based on the Input/Output properties to be verified on the network (subject to that conditioning). Thus, the neuron conditioning step in PEREGRiNN is analogous to variable splitting in a SAT solver, and the *backtrack and re-condition* block (see Fig. 1) functions analogously to backtracking. In this analogy, infeasibility of the convex program and symbolic interval analysis function roughly like unit resolution in a SAT solver: they soundly reason about the overall property before all neurons have been conditioned (i.e. variables split).

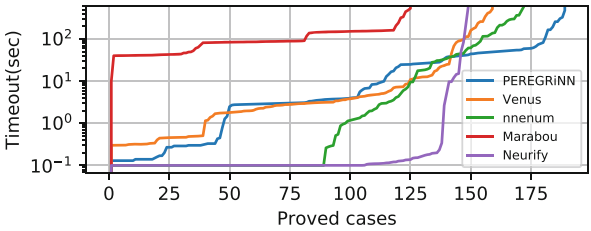


Fig. 4. Cactus plot of various solvers on 300-case testbench, $\mathscr{TB}$

However, the main contribution of PEREGRiNN is a heuristic for deciding which neuron to condition next: it is thus analogous to a heuristic for choosing the next variable to split in a SAT solver. Specifically, PEREGRiNN's heuristic provides a numerical ranking of the as-yet-unconditioned neurons, and therefore has a functional similarity to variable-ranking heuristics in SAT solvers (e.g. VSIDS [24]). On the other hand, PEREGRiNN's neuron ranking comes directly

from the output of the convex solver, which we argued reveals some information about the underlying verification problem – this has no direct SAT-solver analog.

7 Conclusion

In this paper, we introduced PEREGRiNN, a new tool for formally verifying input/output properties for ReLU NNs. PEREGRiNN compares favorably with other state-of-the-art NN verifiers, thanks to a number of unique algorithmic features. The benefits of these features were established with ablation experiments.

References

1. Gurobi optimizer 9.1. http://www.gurobi.com
2. International Verification of Neural Networks Competition 2020 (VNN-COMP 2020). https://sites.google.com/view/vnn20
3. Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., Vielma, J.P.: Strong mixed-integer programming formulations for trained neural networks. Math. Program. **183**(1), 3–39 (2020). https://doi.org/10.1007/s10107-020-01474-5
4. Bak, S., Tran, H.-D., Hobbs, K., Johnson, T.T.: Improved geometric path enumeration for verifying ReLU neural networks. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 66–96. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_4
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. Adv. Neural Inf. Process. Syst. **29**, 2613–2621 (2016)
6. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient verification of ReLU-based neural networks via dependency analysis. Proc. AAAI Conf. Artif. Intell. **34**, 3291–3299 (2020). https://doi.org/10.1609/aaai.v34i04.5729
7. Bunel, R., Lu, J., Turkaslan, I., Kohli, P., Torr, P., Mudigonda, P.: Branch and bound for piecewise linear neural network verification. J. Mach. Learn. Res. **21**(42), 1–39 (2020)
8. Cheng, C.-H., Nührenberg, G., Ruess, H.: Maximum resilience of artificial neural networks. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 251–268. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_18
9. Dutta, S., Jha, S., Sanakaranarayanan, S., Tiwari, A.: Output range analysis for deep neural networks (2017). https://arxiv.org/abs/1709.09130
10. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: Globerson, A., Silva, R. (eds.) Uncertainty in Artificial Intelligence, vol. 1, pp. 550–559 (2018)
11. 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
12. Emiris, I.Z., Fisikopoulos, V.: Practical Polytope Volume Approximation. ACM Trans. Math. Softw. **44**(4), 38:1–38:21 (2018). https://doi.org/10.1145/3194656

13. Fazlyab, M., Robey, A., Hassani, H., Morari, M., Pappas, G.: Efficient and accurate estimation of lipschitz constants for deep neural networks. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32, pp. 11423–11434. Curran Associates, Inc. (2019)
14. Fischetti, M., Jo, J.: Deep neural networks and mixed integer linear optimization. Constraints **23**(3), 296–309 (2018). https://doi.org/10.1007/s10601-018-9285-6
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. IEEE (2018). https://doi.org/10.1109/SP.2018.00058
16. Goodfellow, I.J., Shlens, J., Szegedy, C.S.: Explaining and harnessing adversarial examples (2014). https://arxiv.org/abs/1412.6572
17. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, pp. 169–178. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3302504.3311806
18. 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
19. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 443–452. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_26
20. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world (2016). https://arxiv.org/abs/1607.02533
21. LeCun, Y.: The MNIST database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/
22. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for Verifying Deep Neural Networks (2019). http://arxiv.org/abs/1903.06758
23. Lomuscio, A., Maganti, L.: An approach to reachability analysis for feed-forward relu neural networks (2017). https://arxiv.org/abs/1706.07351
24. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, pp. 530–535 (2001). https://doi.org/10.1145/378239.379017
25. Royo, V.R., Calandra, R., Stipanovic, D.M., Tomlin, C.: Fast neural network verification via shadow prices (2019). https://arxiv.org/abs/1902.07247
26. Shoukry, Y., Nuzzo, P., Sangiovanni-Vincentelli, A.L., Seshia, S.A., Pappas, G.J., Tabuada, P.: SMC: satisfiability modulo convex programming. Proc. IEEE **106**(9), 1655–1679 (2018). https://doi.org/10.1109/JPROC.2018.2849003
27. Song, D., et al.: Physical adversarial examples for object detectors. In: Proceedings of the 12th USENIX Conference on Offensive Technologies. WOOT 2018, USENIX Association (2018)
28. Szegedy, C., et al.: Intriguing properties of neural networks (2013). https://arxiv.org/abs/1312.6199
29. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming (2017). https://arxiv.org/abs/1711.07356

30. Tran, H.-D., et al.: NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 3–17. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_1
31. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 31, pp. 6367–6377 (2018)
32. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of the 27th USENIX Conference on Security Symposium, SEC 2018, pp. 1599–1614. USENIX Association (2018). https://doi.org/10.5555/3277203.3277323
33. Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope (2017). https://arxiv.org/abs/1711.00851
34. Xiang, W., Tran, H.D., Johnson, T.T.: Reachable set computation and safety verification for neural networks with relu activations (2017). https://arxiv.org/abs/1712.08163
35. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5777–5783 (2018). https://doi.org/10.1109/TNNLS.2018.2808470