Title
Effective Formal Verification of Neural Networks using the Geometry of Linear Regions

Authors
Khedr, Haitham
Ferlez, James
Shoukry, Yasser

Publication Date
2020

Peer reviewed

Effective Formal Verification of Neural Networks using the Geometry of Linear Regions

Haitham Khedr, James Ferlez, and Yasser Shoukry
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697, 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 exploit the geometry of neuronal activation regions to further prune the search space of relaxed neuron activations. In particular, conditioning on neurons from input layer to output layer, we can regard each relaxed neuron as having the simplest possible geometry for its activation region: a half-space. 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. For example, our approach completes the standard MNIST verification test bench 2.7-50 times faster than competing algorithms while still proving 14-30% more properties. We also used our framework to verify the safety of a neural network controlled autonomous robot in a structured environment, and observed a 1900 times speed up compared to existing methods.

神经网络（NNs）越来越明显地体现出安全性，这与它们在实际应用中的普及程度相称：意料之外的以及对抗性的错误分类都可能导致致命的后果。结果，形式验证技术已被认为对安全NN的设计和部署至关重要。在本文中，我们介绍了一种新方法来正式验证ReLU NN最常用的安全规范-即网络输入和输出上的多主题规范。像其他方法一样，我们的方法使用轻松的凸程序来减轻问题的组合复杂性。但是，在我们的方法中，独特的是我们利用神经元激活区域的几何形状进一步修剪松弛的神经元激活的搜索空间的方式。特别是，在从输入层到输出层的神经元进行条件调节时，我们可以将每个松弛的神经元视为具有其激活区域最简单的几何形状：半空间。可以利用这种范例来创建一种验证算法，该算法不仅通常比竞争方法要快，而且还可以验证更多的安全性。例如，我们的方法完成标准MNIST验证测试平台的速度是竞争对手算法的2.7至50倍，同时仍证明其性能提高了14至30%。我们还使用我们的框架来验证神经网络控制的自主机器人在结构化环境中的安全性，并观察到其速度是现有方法的1900倍。

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 exceed the ability to reliably verify the safe and correct functioning of these components, especially when they are integrated with other components such as controllers. Thus, there is an increasing need to verify that NNs reliably produce safe outputs, especially in the presence of malicious adversarial inputs [1, 2, 3, 4].

In this paper, we propose PeregriNN, an algorithm for efficiently and formally verifying ReLU NNs. In particular, we consider the problem of whether a particular set of inputs always results in NN outputs within some other (output) set. However, PeregriNN will also verify input and output constraints that are interrelated by convex inequalities: this feature distinguishes PeregriNN from other formal NN verifiers, which verify only static input/output constraints. And in particular, it makes PeregriNN uniquely well suited to the verification of NNs when they are used as state-feedback controllers for dynamical systems: in such cases, static input/output constraints are inadequate to capture the most important safety properties.

神经网络已成为现代机器学习系统中越来越重要的组成部分，包括那些用于安全关键型网络物理系统（如自动驾驶汽车）中的系统。采用率超过了可靠地验证这些组件的安全性和正确功能的能力，尤其是当它们与其他组件（例如控制器）集成在一起时。因此，越来越需要验证NN可靠地产生安全的输出，尤其是在存在恶意对抗输入的情况下[1、2、3、4]。在本文中，我们提出了PeregriNN，这是一种有效且正式地验证ReLU NN的算法。特别地，我们考虑一个特定的输入集是否总是导致其他一些（输出）集中的NN输出的问题。但是，PeregriNN还将验证由凸不等式相互关联的输入和输出约束：此功能将PeregriNN与其他形式的NN验证器区分开，后者仅验证静态输入/输出约束。特别是，当PeregriNN用作动态系统的状态反馈控制器时，它使PeregriNN特别适合用于验证NN：在这种情况下，静态输入/输出约束不足以捕获最重要的安全特性。

Preprint. Under review.

从广义上讲，PeregriNN属于声音以及完整的搜索和优化NN验证程序的类别[5]。像该类别中的其他算法一样，PeregriNN的优化方面是一个宽松的凸程序，其中为每个单个神经元的输出分配了一个松弛变量，该变量与神经元的实际输出分离。凸解算器试图最小化松弛，以驱动每个松弛以匹配其相关神经元的输出，从而获得网络的实际输入/输出响应（另请参见图1）。PeregriNN的搜索方面是以下事实的结果：凸求解器通常无法将所有这些松弛驱动为零：在这种情况下，具有非零松弛的神经元可以被认为是不确定的，必须进行搜索这些神经元的所有可能的激活组合（呈指数增长[6, 7]）。这是通过一次对神经元进行调节直至所有可能的激活组合用尽来实现的，通常是借助一次排除多个组合的方法的好处。

Broadly speaking, PeregriNN falls into the category of sound and complete search and optimization NN verifiers [5]. Like other algorithms in this category, the optimization aspect of PeregriNN is a relaxed convex program where the output of each individual neuron is assigned a slack variable that is decoupled from the actual output of the neuron; the convex solver tries to minimize the slacks in order to drive each slack to match the output of its associated neuron, thereby obtaining an actual input/output response of the network (see also Fig. 1). The search aspect of PeregriNN is a consequence of the fact that the convex solver often cannot drive all of these slacks to zero: in such a case, the neurons with non-zero slacks can be regarded as indeterminate, and a search must be conducted over all the possible combinations of activations (exponentially growing [6, 7]) for these neurons. This is accomplished by means of conditioning on the neurons one at a time until all possible activation combinations are exhausted, usually with the benefit of a methodology for ruling out multiple combinations at once.

PeregriNN算法的主要贡献是它的搜索算法（以及使之成为可能的改进的凸问题）。独特地，PeregriNN算法以强调和利用其激活区域的几何形状的方式搜索不确定的神经元激活。尤其是，PeregriNN利用以下几何观察：从网络的输入特征空间来看，输入层1中神经元的激活区域由超平面划分边界，因此比起网络深处的激活区域而言，其几何结构要简单得多。为了重新创建和利用网络中任意不确定神经元的这一优势，PeregriNN在搜索中合并了两个不同级别的优先级。

The main contribution of the PeregriNN algorithm is its search algorithm (and the modified convex problem that makes it possible). Uniquely, the PeregriNN algorithm searches over indeterminate-neuron activations in a way that emphasizes and exploits the geometry of their activation regions. In particular, PeregriNN leverages the following geometric observation: from the input-feature space of a network, the activation regions of neurons in the input layer[1] are demarcated by hyperplanes, and so have a much simpler geometry than activation regions from deeper in the network. To recreate and exploit this advantage for arbitrary indeterminate neurons in the network, PeregriNN incorporates two different levels of prioritization in its search.

1. **Inter-layer prioritization.** PeregriNN always prefers to search (i.e. condition on) neurons closest to the input layer[1], so that the next indeterminate neuron in the search necessarily receives its input from a sub-network operating in a linear region; hence, the next search neuron may itself be regarded as an input neuron of the complementary sub-network. This also depends on a novel convex program.

2. **Intra-layer prioritization.** PeregriNN furthers exploits this exposed hyperplane geometry via a novel search priority within each layer. In particular, PeregriNN prioritizes the activation region with the smallest volume: this is a heuristic that balances the accuracy of over-approximation methods with the number of activation combinations they can prune from the search.

We used PeregriNN to verify the adversarial robustness of networks trained on MNIST [8] and CIFAR-10 [9] datasets, as well as safety properties of a NN-controlled autonomous system. For MNIST, our experiments show that PeregriNN is on average 2.7× faster than Neurify [10] and 50× faster than Marabou [11] which are two state-of-the-art algorithms. It also proves 2 % and 80 % more properties than Neurify and Marabou on CIFAR dataset respectively. PeregriNN is also 1900× faster than SMC [12] for verifying the safety of NN controlled autonomous systems.

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, 13, 14]; (ii) MILP-based solvers, which directly encode the verification problem as a Mixed Integer Linear Program [15, 16, 17, 18, 19, 20, 21]; (iii) Reachability based methods, which perform layer-by-layer reachability analysis to compute the reachable set [22, 23, 24, 25, 26, 27, 28]; and (iv) convex relaxations methods [10, 29, 30]. 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 [31]). These methods are most similar to PeregriNN: for example, [25, 18, 32] recursively refine the problem using input splitting, and [10] does so via neuron splitting. Other search and optimization methods include: Planet [14], which combines a relaxed convex optimization problem with a SAT solver to search over neurons' phases; and Marabou [11], which uses a modified simplex algorithm to handle non-convex ReLU activations.

2 Problem formulation

Neural Networks. In this paper, we will consider Rectified Linear Unit (ReLU) NNs. By a (n-layer) ReLU network, we mean a composition of (n) ReLU layer functions (or just layers): i.e.

[1] We refer to the first hidden layer as the input layer.

$\mathcal{NN} = f_n \circ f_{n-1} \circ \cdots \circ f_1$ where the i^{th} ReLU layer function is parameterized by weights, W_i, and biases, b_i, and is defined as $f_i : y \in \mathbb{R}^{k_{i-1}} \mapsto \max\{W_i y + b_i, 0\} \in \mathbb{R}^{k_i}$.

To simplify future notation, we define the output of each layer in the computation of $\mathcal{NN}(y_0)$ as $y_i = f_i(y_{i-1})$ for $i = 1, \ldots, n$. To refer to individual neurons, we use the notation $(z)_j$ to refer to the j^{th} element of the vector z. As a final notational convenience, we refer to f_1 as the input layer.

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. Then the verification problem is to decide whether

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

Note that the addition of the convex inequality constraints h_j is a unique feature of our problem formulation compared to other NN verifiers, and it significantly broadens the scope of the problem. In particular, other solvers can only verify independent input and output constraints P_{y_0} and P_{y_n}.

3 Approach

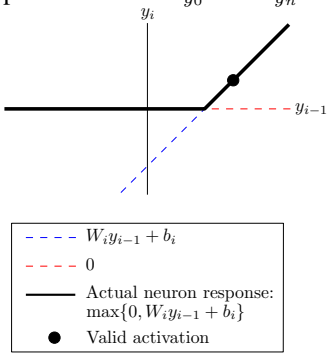

Figure 1: Neuron activation by minimizing a slack variable, y_i.

As in some other algorithms, we convert the verification problem (Section 2) into a relaxed convex program (e.g. [10, 25, 18]). Convex programs of this type assign a slack variable to the output of each neuron that is purposely allowed to differ from the actual output of the neuron (as calculated from its inputs). Each slack variable is then constrained to lie above the response of its associated neuron, and the input and output sets from the verification problem are incorporated as further constraints on the relevant slack variables. The objective, then, is to minimize the sum of total slacks in the hope that each slack variable will be driven to lie exactly on the response of its respective neuron. In such a case, the solver will have found a solution that corresponds to an actual input/output pair for the network, and thus decide the verification problem. Mathematically, relaxed convex programs of this type can be written as:

$$\min_{y_0,..,y_n} \sum_{i=0}^{n} \sum_{j=1}^{k_i} y_{ij} \quad \text{s.t.} \quad \begin{cases} y_i \geq 0,\ y_i \geq W_i y_{i-1} + b_i & \forall i = 1, \ldots, n \\ y_0 \in P_{y_0},\ y_n \in P_{y_n},\ \bigwedge_{\ell=1}^{m} h_\ell(y_0, y_n) \leq 0 \end{cases} \tag{2}$$

A valid input/output pair for the network is obtained when either $y_i = 0$ or $y_i = W_i y_{i-1} + b_i$ for all $i = 1, \ldots, n$. An example of this situation is depicted in Fig. 1, where the black dot illustrates a choice of the slack variable y_i that corresponds to a valid input/output pair for the illustrated neuron.

Given the efficiency of modern convex solvers, this is an extremely attractive approach when it is successful. However, all formulations of this type suffer from the same drawback: the solver may return a solution in which some of the slack variables are indeterminate – i.e. they do not lie on the response of their respective neurons. In a case like this, the solution doesn't correspond to an actual evaluation of the network, and more work is needed to decide the verification problem. In particular, the indeterminate neurons must be conditioned – i.e. constrained to be either active or inactive – and then the convex program re-solved with these new constraints. This process is repeated as a search over all combinations of different conditionings until either a valid solution is found or the problem is shown to be infeasible.

The primary novelty in our approach is thus the way in which we search over the different possible conditionings of indeterminate neurons. Most approaches treat all indeterminate neurons as roughly equivalent for the purpose of search, and use ReLU over-approximation methods to guide which neuron to consider next (and to exclude combinations when possible). Instead, we propose a new, more efficient heuristic that prioritizes neurons in a way that emphasizes the geometry of their activation regions: our approach uses this information to more effectively exclude combinations of conditionings from the search space. Our approach has two specific levels of prioritization that we described subsequently.

3.1 Inter-Layer Prioritization

Consider neurons' activation regions as expressed in the input-feature space of a network: the activation regions of input-layer neurons are always hyperplanes, whereas the activation regions for deeper neurons are more complicated regions – see the top pane of Fig. 2 for an example. Thus, input-neuron activation regions are specifically subject to all of the geometric properties of hyperplane arrangements [33]. In particular, the geometric properties of hyperplane arrangements govern which input-layer-neuron activation regions have non-empty intersections, and hence which combinations of activations are simultaneously possible. In the extreme case when there are more neurons in an input layer than there are inputs, the reduction in possible activation combinations can be considerable. It is known that in such cases, the number of non-empty regions formed by an arrangement of hyperplanes scales sub-exponentially in the number of hyperplanes – i.e. the number of input-layer neurons. In particular, the number of regions formed by N such hyperplanes each of dimension n is at most $\sum_{i=0}^{n} \binom{N}{i}$.

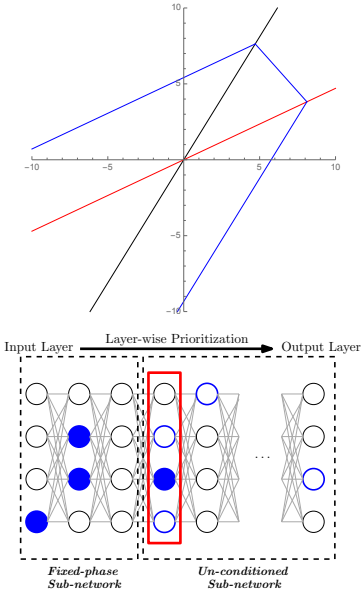

Figure 2: **(Top)** Activation regions for a three-neuron NN with a 2-d input-feature space. The NN has two input neurons and one output neuron: black and red lines show activation boundaries for input neurons; the blue line shows the activation region for the output neuron. **(Bottom)** Illustration of the conditioning order. Indeterminate neurons are shown in blue, with those already conditioned also filled-in; all other neurons are active or inactive as specified by the convex solver. The next neuron is selected from the red rectangle.

Thus, we propose to always condition on those indeterminate neurons closest to the input layer, since this will recreate these geometric properties for all indeterminate neurons within the same layer – even if that layer is deeper in the network. Spurious conditionings can thus be eliminated by evaluating the compatibility of these indeterminate neurons according to their activation-region hyperplanes before conditioning them. This amounts to a direct pruning of conditionings from the search space. Note that the sub-exponential savings is particularly salient in shallow networks where the number of neurons typically exceeds the number of inputs by a significant factor.

This prioritization works because it effectively partitions the network along a layer boundary into two sub-networks: see the bottom pane of Fig. 2. The suggestively named fixed-phase sub-network necessarily operates in a (known) linear region because the phases of all of its neurons are fixed (either by the convex solver or already taken conditioning decisions). Consequently, the relevant portion of the input constraint set can be propagated through this fixed-phase sub-network to obtain an exact polytope representation of its outputs. But the next indeterminate neuron to be conditioned can be regarded as belonging to the input layer of the un-conditioned sub-network, whose input is of course supplied by the output of the fixed-phase sub-network. Thus, we can effectively reconsider the original verification problem in terms of a new verification problem on the un-conditioned sub-network, and one where next indeterminate neuron to be conditioned is in the input layer.

There is one important implementation detail that is necessary to actually implement this layer-wise prioritization, but which we have not yet mentioned. Notably, the search procedure involves re-solving the convex program with neuron conditionings as we decide them, and in general, this may result in new indeterminate neurons in layers closer to the input layer than the last neuron we conditioned. In order to prevent this, and thus ensure that re-solving the convex program forces new indeterminate neurons to appear deeper in the network, we modify the objective function in (2) with layer-wise weights that strongly penalize slacks in earlier layers. That is we use an objective function like:

$$\min_{y_0,..,y_n} \sum_{i=0}^{n} \sum_{j=1}^{k_i} q_i y_{ij} \quad \text{s.t.} \quad \begin{cases} y_i \geq 0,\ y_i \geq W_i y_{i-1} + b_i & \forall i = 1,\dots,n \\ y_0 \in P_{y_0},\ y_n \in P^{\mathsf{c}}_{y_n},\ \bigwedge_{\ell=1}^{m} h_\ell(y_0, y_n) \leq 0 \end{cases} \tag{3}$$

where the coefficients $q_1 \ll q_2 \ll \cdots \ll q_n$ are chosen so that non-zero slack variables are guaranteed to be driven as close to the input layer as possible using the notion of prefix-ordered monotone formulas shown in [12].

3.2 Intra-Layer Prioritization

One important technique to improve the conditioning search is to use ReLU over-approximations (e.g. symbolic interval analysis [34, 25]) to show that a particular conditioning cannot possibly satisfy the constraints of the verification problem. That is if the over-approximation can't satisfy the output constraints, then the particular choice of conditioning can't either. Over-approximations of this type are best be employed when not all of the neurons have yet been conditioned: if a particular neuron conditioning is impossible, then there is no need to check the activation combinations of the remaining indeterminate neurons. The opposite conditioning can thus be incorporated into the convex program as a constraint to aid the convex solver the next time it is run. In particular, when a specific indeterminate neuron can be shown to necessarily be on or off, the inequality constraints for that neuron in (2) can be replaced by the combined constraints $(y_i)_j = (W_i y_{i-1} + b_i)_j \wedge (y_i)_j \geq 0$ or $(y_i)_j = 0 \wedge (W_i y_{i-1} + b_i)_j \leq 0$, respectively.

However, over-approximations like this come with a drawback: the more neurons that can be both active and inactive in a region, the worse the approximation error is – and hence the less likely that that particular combination is invalidated by the over-approximation. On the other hand, the more neurons that have both phases active in a region, the more possible combinations that are eliminated if the activation region is infeasible. Thus, there would seem to be an intuitive trade-off between the number of further conditionings that are possible within an activation region, and the likelihood that those combinations can be excluded by over-approximating the network.

Given the need to use over-parameterized deep networks for effective training, it is plausible to suppose that activation regions are roughly uniformly distributed within any un-conditioned network (as defined in Fig. 2). According to this heuristic, then, the volume of an activation region is a useful proxy for the number of indeterminate neurons that can be excluded by over-approximating the network on that region. This volume heuristic is an especially convenient one given the discussion above: after all, it is considerably easier to compute the volume of activation regions when those activation regions are defined by hyperplanes, as they are for neurons in the input layer; again, refer the top pane of Fig. 2.

Thus, we start from the observations in Section 3.1 to propose the following intra-layer conditioning priority: we condition the next indeterminate neuron (within a layer) according to the (valid) activation region that has the smallest volume. We reiterate that this heuristic is only reasonable to implement because of the geometric basis we established above. Moreover, we have found that minimum-volume priority is an effective heuristic across all of our experiments, which suggests that the aforementioned trade-off skews heavily toward the accuracy of the over-approximation, not the number of neurons that can be excluded. Nevertheless, this analysis suggests that there is a rich opportunity for future work here. In particular, it is natural to consider the possibility of a volume-based prioritization that is adaptive according to the properties of the individual neuron under consideration.

4 Algorithm

We now describe how our novel search procedure integrates with other techniques to form the full PeregriNN algorithm. The state of our algorithm is captured by a list of conditioning choices already made. The main loop of the algorithm updates this state once per iteration in two sequential stages: a convex query stage and a state update stage. The convex query stage entails solving a single convex problem derived from the current list of conditioning choices. Based on the result of the convex query, the state update stage proceeds in one of two ways: it conditions if the convex solver returns indeterminate neurons (i.e. descends deeper, adding conditionings to the state), or it backtracks if the convex solver returns infeasible (i.e. ascends shallower, removing conditionings from the state). Before executing the convex solver, we also include an inference step, which further improves efficiency by providing the solver additional constraints that are inferred by symbolic interval analysis.

Convex Query. The convex query step takes the current list of conditioning choices, and translates them into constraints on the slack variables as follows:

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

$$\text{Neuron } (y_i)_j \text{ OFF:} \quad (y_i)_j = 0 \wedge (W_i y_{i-1} + b_i)_j \leq 0. \tag{5}$$

These constraints are then added of the convex program (3), and the program is passed to the convex solver to solve. The output of the convex solver (for our purposes) is a either a list of

Algorithm 1 Verification of ReLU networks

```
1: procedure NN_VERIFY(nn, problem)
2:     inferred = φ; decided = φ
3:     while True do
4:         inferred, undecided ← SYMINTERVALANALYSIS(nn, problem.input_bounds)
5:         sol, realxed_neurons←CHECKFEAS(nn, problem, undecided)        ▷ Inter layer prioritization
6:         if sol.status == INFEASIBLE then
7:             if decided == φ then return SAFE
8:             else
9:                 decided ← BACKTRACK(decided, problem)
10:        else if |relaxed_neurons| == 0 then return UNSAFE, sol
11:        else
12:            neuron ←PICK_ONE(relaxed_neurons)                      ▷ Intra layer prioritization
13:            decided ← decided ⋃ neuron
```

indeterminate neurons or the conclusion that the program was infeasible. Any list of indeterminate neurons returned by the solver does not include any of the neurons we have conditioned on already by definition. Moreover, any returned indeterminate neurons are at least as deep in the network as the last conditioned neuron: this is because of the choice of the q_i in (3) (see Section 3.2).

State Update. The state update stage proceeds to update the algorithm's state by one of two mechanisms, depending on the output of the convex solver.

Conditioning. If the convex solver returns a list of indeterminate neurons, then PeregriNN's unique prioritizations are used to update the state with a new conditioning choice (see Section 3). First, the inter-layer prioritization is applied: the list of indeterminate neurons is sorted by depth, and only those neurons in the layer closest to the input are eligible for conditioning. From these neurons, the next conditioning choice is made using the intra-layer prioritization scheme: the activation region with the smallest volume (that is also compatible with the other conditionings in the layer) is selected, and added to the state. After the state update, a new iteration of the main loop starts.

Backtracking. If the convex solver returns infeasible, then we have explored as far as possible into the network given the current conditioning choices. Thus, the algorithm must backtrack, and undo some of those choices to explore other possible activation combinations. Since each conditioning choice is in direct correspondence with a pair of convex constraints – i.e. (4) and (5) – and the convex solver returned infeasible, we know that these constraints are mutually incompatible. Thus, we can use an Irreducible Inconsistent Subsystem (IIS) of these constraints to find compatible constraints, and roll back some of the conditioning choices in the current state. Moreover, since an IIS is irreducible, this method often rolls back a considerable number of conditionings at once. After the state update, a new iteration of the main loop starts.

As an additional optimization, we perform an inference step right before executing the convex query.

Inference. Given a set of conditioning choices, it is often possible to rule out some other neuron activations directly by over-approximation methods. Thus, we use Symbolic Interval Analysis[25] to find other neuron activations that must follow from the current state of conditioning choices. These inferred conditioning choices are converted to convex constraints as in (4) and (5), and added to the convex program right before the convex query stage is executed.

5 Experiments

Implementation. We implemented PeregriNN in Python, but used off-the-shelf Gurobi [35] convex optimizer. We ran our experiments on a 24-core machine with 256 GB of memory. For fairness of comparison across implementations as well as single-threaded algorithms, we limited all algorithms(including PeregriNN) to run on a single core.

In order to evaluate the performance and effectiveness of PeregriNN, we conducted two different experiments, which can be summarized as follows.

1. We used PeregriNN to verify the adversarial robustness of NNs trained on standard datasets (MNIST and CIFAR); this experiment allowed us to compare PeregriNN against state-of-the-art NN verifiers both in terms of execution time and in terms of effectiveness at proving properties.

2. We used PeregriNN to evaluate the safety of NN controllers for a LiDAR equipped quadrotor; this experiment exercised PeregriNN's unique ability to verify properties specified with interrelated input/output constraints (see (1)) in a practical safety verification problem.

5.1 Adversarial robustness

In this experiment, our objective is to compare PeregriNN with other NN verifiers both in terms of performance and in terms of effectiveness at proving properties. To this end, we evaluated adversarial robustness of NN classifiers that we trained on the MNIST and CIFAR datasets. Each property we considered characterized whether a $\max$-norm perturbation of the input could lead to a misclassification, and we parameterized these properties by the size of the input perturbation, ϵ. Specifically, let z_t be the classifier output indicating the belief that the input belongs to the t^{th} category out of M categories; then checking the robustness of the NN around x' is equivalent to checking the truth of:

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

Problem (6) can proved by checking $M-1$ instances of (1). Specifically, for each instance $m \in \{1,\ldots,M\} \setminus \{t\}$ we check if:

$$\{x \mid x \in \mathbb{R}^{k_0}, \|x - x'\|_\infty \leq \epsilon, z \in \mathbb{R}^{k_n}, \max_{i=1,\ldots,n}(z)_i = (z)_m\} = \phi, \tag{7}$$

where $z = \mathcal{NN}(x)$. If any of those instances is unsatisfied (nonempty set), then the property is violated. Otherwise, the property holds.

Table 1: Models used in the experiments

Models	# ReLUs	Architecture	Accuracy
MNIST_FC1	48	<784,24,24,10>	96.5%
MNIST_FC2	100	<784,50,50,10>	97.5%
MNIST_FC3	1024	<784,512,512,10>	98.2%
CIFAR_FC1	2048	<3072,1024,512,512,10>	98%

MNIST Results. We evaluated the robustness of three different classifiers for the MNIST dataset using four different magnitudes of input perturbation. The architectures of these classifiers are shown in Table 1 together with the accuracy and the number of ReLUs for each of these networks. We tested our framework with 100 randomly selected images and compared it with Marabou and Neurify. We chose to compare with Neurify because to our knowledge, Neurify is the best performing NN verifier on this dataset (and similar ones) [10]; we chose to compare with Marabou, since it is one of the newest verifiers. Each query consists of an input image, and the property to be checked is whether the network is robust against an ϵ $\max$-norm perturbation. The timeout for checking each property is 20 minutes. Table 2 summarizes the performance of the three solvers on the three networks. PregeriNN outperforms Marabou and Neurify by 50$\times$ and 2$\times$ respectively on average execution time, and it proves 14-30% more properties than Marabou and Neurify respectively. The results also show that for the cases that Neurify performs faster, PeregeriNN can still prove more properties in almost the same amount of time. We note here that when running Neurify on MNIST_FC3 with $\epsilon = 15$, it gave segmentation faults due to huge memory consumption, and we counted these cases as timeouts. As shown in the table, the number of unproved cases generally increases with ϵ and with the size of the network; this is due to looser bounds estimates on the neurons and larger search space induced by larger networks.

Table 2: Performance (execution time, number of proved cases) of the three solvers using three different networks. For each cell, the total execution time for the 100 queries is reported together with the number of successfully proved properties. The last column shows the number of properties proved to be safe by PeregeriNN.

ϵ	Architecture	Marabou Time(s) / proved cases	Neurify Time(s) / proved cases	PeregeriNN Time(s) / proved cases	Safe
1	MNIST_FC1	308.8 / **100**	**7.1 / 100**	10 / **100**	97
	MNIST_FC2	682.2 / **100**	**6.37 /100**	7.806 / **100**	99
	MNIST_FC3	38778.5 / 95	384 / **100**	**91.6 / 100**	100
5	MNIST_FC1	2198.6 / **100**	4058.6 / 99	**595.726 / 100**	77
	MNIST_FC2	41812.13 / 81	45482.1 / 68	**5921 / 99**	86
	MNIST_FC3	1200000 / 0	105088.2 / 13	**39477 / 71**	69
10	MNIST_FC1	23329.0 / 94	24725.5 / 86	**7580 / 99**	29
	MNIST_FC2	107815.86 / 16	**48650.74** / 61	80844 / **83**	26
	MNIST_FC3	120000 / 0	**100306.86 / 17**	118478.55 / 4	1
15	MNIST_FC1	25039.82 / 89	8056.5 / 94	**6330 / 98**	3
	MNIST_FC2	84655.4 / 32	**25211.23 / 79**	31384.321 / 78	2
	MNIST_FC3	120000 / 0	120000 / 0	**118356.89 / 3**	0

CIFAR Results. We ran the same experiments on different networks trained on CIFAR dataset. We chose the CIFAR dataset to evaluate the performance of PeregeriNN on networks with large input spaces (3072 features). Table 3 shows the number of proved properties out of 100 random queries for each of the solvers. The results shows that PeregeriNN can prove more properties than the other solvers on networks with large input spaces and ReLU counts.

Table 3: Number of proved properties out of 100 queries using CIFAR_FC1

ϵ	Marabou	Neurify	PeregeriNN
0.005	0	89	**96**
0.0075	0	88	**92**
0.01	0	76	**78**
0.02	0	**57**	**57**
0.05	0	**75**	74

5.2 Safety of Neural Network Controlled Physical systems

In this experiment, our objective is to study two properties: (i) safety verification of a NN-controlled autonomous system and (ii) how our framework scales with the size of the trained NN to be verified. To this end, we consider the problem of verifying the safety of an autonomous quadrotor equipped with a LiDAR sensor and controlled by a NN that processes LiDAR measurements to produce control actions [36]. One way to verify such systems is to discretize the workspace into discrete partitions $S_1, \ldots, S_w$ and check the feasibility of transition between these partitions to the unsafe set (obstacles) $O_1, \ldots, O_o$. Let $x \in \mathbb{R}^2$ be the position of the quadrotor. As shown in [36], the next position of the quadrotor is then given by $Ax + B\,\mathcal{NN}(Hx + d)$ where the matrices A and B describes the physics of the robot (e.g., mass, friction, .. etc) while the affine term $Hx + d$ captures the relation between the quadrotor position and the LiDAR image. Therefore, checking the safety of the NN controller is then written as:

$$\left\{ x \middle| x \in \bigcup_{m=1}^{w} S_m,\ Ax + B\,\mathcal{NN}(Hx + d) \in \bigcup_{t=1}^{o} O_t \right\} = \emptyset. \tag{8}$$

Indeed, the system safety property (8) can be checked by solving $w \times o$ formulas of the form (1).

We use PeregriNN to verify (8) by varying the workspace discretization parameter ϵ and recording the execution time for 10 different NN that have the same exact architecture and are all trained using imitation learning with 1143 episodes. Table 4 shows how the safe regions of the workspace varies with the discretization parameter ϵ. PeregriNN is able to verify the safety properties for all the networks and exactly identify the safe regions in the workspace. Next, we evaluate the scalability of PeregriNN by verifying the property (8) for NNs with different architectures and recording the verification time. Table 5 shows the scalability of our framework with different architectures of NNs. PeregeriNN can verify networks with 100,000 ReLUs in just a few seconds. However, increasing the depth of the network increases the difficulty of the verification problem. Note that the results reported in [36], which uses SMC solvers [12], are capable of handling at most networks with 1000 ReLUs. Comparing PeregeriNN to SMC solver in [36], we conclude that PeregeriNN can verify networks that are 2 orders of magnitude larger than SMC with 1900 times less execution time.

Table 4: Shows the number of safe and unsafe regions for 10 different networks

Epsilon	Number of safe/unsafe regions									
	1	2	3	4	5	6	7	8	9	10
0.25	46/52	33/65	49/49	45/53	46/52	53/45	51/47	63/35	74/24	51/47
0.5	27/38	22/43	30/35	27/38	27/38	29/36	31/34	39/26	49/16	36/29
0.75	20/34	17/37	24/30	21/33	21/33	23/31	26/28	31/23	43/11	32/22

Table 5: **(Left)**Shows the execution time in seconds for checking the feasibility of transition between a pair of regions in the workspace. We test the scalability of the solver by solving the verification problem for different architectures by varying the number of neurons per layer and the depth of the network. **(Right)** shows the verification time for single layer networks with different width.

# of neurons per layer	# of layers					
	1	2	3	4	5	6
20	0.025	0.0479	0.1184	0.4767	26.76	0.257
128	0.267	1.57	243.8	3394.18	2740.341	1368.55
256	0.31	0.92	6956.69	136.44	4.4352	1471.29
512	0.679	19.83	5.43	10058.13	9649.55	35783.58

# of neurons	time(s)
1024	3.374
4096	7.2517
20000	7.458
50000	30.189
100000	68.8614

Broader Impact

New advances in AI systems have created an urgency to study safety, reliability, and potential problems that can rise and impact the society by the deployment of AI-enabled systems in the real world. Mathematically based techniques for the specification, development, and verification of complex systems, also known as formal methods, hold the promise to provide appropriate rigorous analysis of the reliability and safety of such AI-enabled systems.

This work provides a new solver to formally verify whether a NN satisfies specified formal properties in a bounded model checking scheme. The basic idea of bounded model checking is to search for a counterexample that violate the formal property. Such counterexamples can be then used by NN developers to better understand the limitations of the trained NN in terms of safety, robustness, and hopefully bias. This in turn can enable the use of AI in safety critical cyber-physical applications that are generally regarded to have positive societal influences: autonomous cars and aircraft collision avoidance systems, for example. This work can also be used to identify performance and robustness problems in NNs that are used in non-cyber-physical applications: for example, NNs that are used in criminal justice contexts or to decide creditworthiness. On the negative side, formally verified AI systems may result into a false sense of safety since such technologies do not reason about un-modeled behaviors and side-effects. Another negative effect stems from the proliferation of the technologies that it enables: for example, increased deployment of autonomous vehicles has the potential to cause job loss.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," arXiv preprint arXiv:1312.6199, 2013.

[2] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," arXiv preprint arXiv:1412.6572, 2014.

[3] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," arXiv preprint arXiv:1607.02533, 2016.

[4] D. Song, K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, F. Tramer, A. Prakash, and T. Kohno, "Physical adversarial examples for object detectors," in 12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18), 2018.

[5] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for Verifying Deep Neural Networks," 2019.

[6] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in Advances in neural information processing systems, pp. 2924–2932, 2014.

[7] R. Pascanu, G. Montufar, and Y. Bengio, "On the number of response regions of deep feed forward networks with piece-wise linear activations," arXiv preprint arXiv:1312.6098, 2013.

[8] Y. LeCun, "The mnist database of handwritten digits.." `http://yann.lecun.com/exdb/mnist/`, 1998.

[9] V. Krizhevsky, A.; Nair and G. Hinton, "The cifar-10 dataset.." `http://www.cs.toronto.edu/kriz/cifar.html`, 2014.

[10] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," in Advances in Neural Information Processing Systems, pp. 6367–6377, 2018.

[11] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, et al., "The marabou framework for verification and analysis of deep neural networks," in International Conference on Computer Aided Verification, pp. 443–452, Springer, 2019.

[12] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "SMC: Satisfiability Modulo Convex Programming," Proceedings of the IEEE, vol. 106, no. 9, pp. 1655–1679, 2018.

[13] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in Computer Aided Verification (R. Majumdar and V. Kunčak, eds.), Lecture Notes in Computer Science, pp. 97–117, Springer International Publishing, 2017.

[14] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in International Symposium on Automated Technology for Verification and Analysis, pp. 269–286, Springer, 2017.

[15] A. Lomuscio and L. Maganti, "An approach to reachability analysis for feed-forward relu neural networks," arXiv preprint arXiv:1706.07351, 2017.

[16] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," arXiv preprint arXiv:1711.07356, 2017.

[17] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in Advances in neural information processing systems, pp. 2613–2621, 2016.

[18] R. Bunel, J. Lu, I. Turkaslan, P. Kohli, P. Torr, and P. Mudigonda, "Branch and bound for piecewise linear neural network verification," Journal of Machine Learning Research, vol. 21, no. 2020, 2020.

[19] M. Fischetti and J. Jo, "Deep neural networks and mixed integer linear optimization," Constraints, vol. 23, no. 3, pp. 296–309, 2018.

[20] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma, "Strong mixed-integer programming formulations for trained neural networks," Mathematical Programming, pp. 1–37, 2020.

[21] C.-H. Cheng, G. Nührenberg, and H. Ruess, "Maximum resilience of artificial neural networks," in International Symposium on Automated Technology for Verification and Analysis, pp. 251–268, Springer, 2017.

[22] W. Xiang, H.-D. Tran, and T. T. Johnson, "Reachable set computation and safety verification for neural networks with relu activations," arXiv preprint arXiv:1712.08163, 2017.

[23] W. Xiang, H.-D. Tran, and T. T. Johnson, "Output reachable set estimation and verification for multilayer neural networks," IEEE transactions on neural networks and learning systems, vol. 29, no. 11, pp. 5777–5783, 2018.

[24] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "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.

[25] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 1599–1614, 2018.

[26] H.-D. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems," arXiv preprint arXiv:2004.05519, 2020.

[27] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "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, pp. 169–178, 2019.

[28] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. Pappas, "Efficient and accurate estimation of lipschitz constants for deep neural networks," in Advances in Neural Information Processing Systems, pp. 11423–11434, 2019.

[29] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli, "A dual approach to scalable verification of deep networks.," in UAI, vol. 1, p. 2, 2018.

[30] E. Wong and J. Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," arXiv preprint arXiv:1711.00851, 2017.

[31] S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari, "Output range analysis for deep neural networks," arXiv preprint arXiv:1709.09130, 2017.

[32] V. R. Royo, R. Calandra, D. M. Stipanovic, and C. Tomlin, "Fast neural network verification via shadow prices," arXiv preprint arXiv:1902.07247, 2019.

[33] R. P. Stanley, "An Introduction to Hyperplane Arrangements," p. 90.

[34] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," in Advances in neural information processing systems, pp. 4939–4948, 2018.

[35] G. Optimization, "Gurobi optimizer 5.0," Gurobi: http://www. gurobi. com, 2013.

[36] X. Sun, H. Khedr, and Y. Shoukry, "Formal verification of neural network controlled autonomous systems," in Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 147–156, 2019.