UNIVERSITY OF CALIFORNIA,
IRVINE

Formal Verification of Neural Networks

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Haitham Khedr

Thesis Committee:
Assistant Professor Yasser Shoukry, Chair
Associate Professor Mohammad Al Faruque
Assistant Professor Yanning Shen

2021

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

LIST OF ALGORITHMS **vi**

ACKNOWLEDGMENTS **vii**

ABSTRACT OF THE THESIS **viii**

1 Introduction **1**

2 Background **4**
2.1 Neural networks 4
2.2 ReLU linear relaxation 5
2.3 Interval arithmetic 6
2.4 Symbolic interval analysis 7
2.5 Neural network verification problem 8

3 Related work **10**
3.1 SMT-based methods 10
3.2 MILP-based methods 11
3.3 Reachability based methods 11
3.4 Convex relaxations methods 11

4 Algorithm **13**
4.1 PeregriNN Enhancements 16
4.1.1 Sum-of-Slacks Penalty 16
4.1.2 Max-Slack Conditioning Priority 16
4.1.3 Layer-wise-Weighted Penalty 17
4.1.4 Initial Counterexample Search by Sampling 18

5 Verification Problems **19**
5.1 Adversarial Robustness 19
5.2 Safety of a neural network controlled quadrotor 20
5.2.1 Dynamics and Workspace 20

5.2.2 LiDAR Imaging 21
5.2.3 Imaging-Adapted Workspace Partitioning 23
5.2.4 Finite state abstraction 25
5.2.5 Verification framework 27

6 Experiments **28**
6.1 Adversarial Robustness Verification Task 29
6.1.1 Ablation Experiments 30
6.1.2 Comparison with Other NN Verifiers 33
6.2 Safety of a neural network controlled quadrotor 35

7 Conclusion **37**
7.1 Summary of contributions 38
7.2 Future work 38

Bibliography **40**

LIST OF FIGURES

Page

2.1 Triangular relaxation of the ReLU function 6
2.2 Symbolic interval analysis relaxations 7

4.1 Block Diagram of the PeregriNN Algorithm 14

5.1 Pictorial representation of the problem setup under consideration. 21
5.2 Workspace imaging-adapted partitioning 25
5.3 Safety verificaion on finite state abstraction 25

6.1 Performance of PeregriNN variants with different conditioning priorities . . . 30
6.2 Performance of PeregriNN variants with different objective functions 32
6.3 Cactus plot of various solvers on 300-case testbench 33

LIST OF TABLES

Page

6.1 Architecture of the NN models used in the experiments 29
6.2 The number of safe and unsafe regions for 10 different networks 35
6.3 Scalability results for the safety of a NN controlled Quadrotor 36

LIST OF ALGORITHMS

		Page
1	Verification of ReLU networks	15

ACKNOWLEDGMENTS

I'd like to express my extreme gratitude to my advisor Yasser Shoukry for the useful feedback, guidance throughout the project, and hands-on engagement. I would also like to thank my fellow colleagues for a great work environment. A special thanks goes to James Ferlez and Xiaowu Sun for their collaboration on different projects. Finally, I would also like to thank the National Science Foundation for their funding assistance via awards CNS-2002405 and CNS-2013824. The text of this thesis is a reprint of the material as it appears in the papers; Formal verification of neural network controlled autonomous systems[27] and Effective Formal Verification of Neural Networks using the Geometry of Linear Regions [20]

ABSTRACT OF THE THESIS

Formal Verification of Neural Networks

By

Haitham Khedr

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2021

Assistant Professor Yasser Shoukry, Chair

Neural networks(NNs) have been widely used over the past decade at the core of intelligent systems from sensing modules to learning-based controllers. They've also been deployed in different safety-critical domains including healthcare and transportation. However, recent work has shown that NNs are fragile and can make dangerous mistakes that are either unintentional or adversarial. As a consequence, formal verification of NNs holds the promise of providing safety guarantees on the behaviour of such systems. We focus our work on ReLU networks as it is the most widely used activation function. Exact formal verification of ReLU NNs was proved to be NP-hard due to the combinatorial nature of the problem, therefore all of the current verification methods use some relaxation of the problem. We propose a novel framework for formal verification of ReLU neural networks that can ensure that they satisfy some polytopic specifications on the input and the output of the network. Our approach uses a relaxed convex program to mitigate the combinatorial complexity of the problem together with some optimization heuristics to efficiently verify the satisfaction of the specification on a given network. We have implemented our algorithm in a toolkit, PeregriNN. To test PeregriNN, we run it on two test benches in different domains. First, we achieve SOTA results on verifying the adversarial robustness of different networks on the MNIST dataset. Second, we verify the safety of a neural network controlled autonomous robot in a structured environment.

Chapter 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. This has been well studied in the domain of image classification. Most state-of-the-art object detectors are sensitive to small adversarial perturbation[28] to their inputs that are not noticeable to humans. Moreover, these adversarial attacks were shown to be able to fool systems operating in the physical world [26, 21] which could lead to fatalities. For example, a change in lighting condition led a Tesla autopilot to crash into a truck[1]. Similarly, Uber's autonomous vehicle collided[2] with a woman on a bicycle because it kept fluctuating between identifying her as an unknown object, a vehicle, or a bicycle. Such examples significantly limit NNs usefulness, especially in safety-critical applications.

We propose a novel framework, PeregriNN, for formal verification of neural networks that can ensure that they satisfy some specifications. Formal verification algorithms can be

classified according to soundness and completeness. The former means that the algorithm can verify that a system actually satisfy the specification, but can't decide if the system violates the specification. While the latter means that the algorithm can verify that a system actually violates the specification. Sound algorithms usually scale better, but they might not be able to solve many problems given a finite amount of time. Complete algorithms are guaranteed to solve the verification problem, however, they suffer from scalability due to their computational complexity. We only consider complete methods in this thesis.

PeregriNN is a sound and complete 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 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.

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.*

We compared the performance of our proposed algorithm against present state-of-the-art verification algorithms for proving adversarial robustness of networks trained on the MNIST dataset. On average, PeregriNN verified more properties than all SOTA methods. We also used PeregriNN to verify safety of transitions of a ground robot in a structured environment with dynamical constraints. These experiments are explained in details in chapter 6

Chapter 2

Background

In this chapter, we introduce all the notation and concepts needed to define our problem, as well as the formal definiton of the NN verification problem. We also discuss briefly the state of the art algorithms of the field and categorize them based on the underlying methods used.

2.1 Neural networks

We consider an n-layer feedforward neural network $\mathcal{NN} : \mathbb{R}^{k_0} \to \mathbb{R}^{k_n}$ with input $x \in \mathbb{R}^{k_0}$ and ouput $z \in \mathbb{R}^{k_n}$, where $\mathbb{R}$ is the set of real numbers, k_0 is the dimension of the input, and k_n is the dimension of the output of the network. The i-th layer in $\mathcal{NN}$ corresponds to a function $f_i : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$. Hence, the neural network can be represented by $\mathcal{NN} = f_n \circ f_{n-1} \circ ... \circ f_1$, where $\circ$ is function composition operator. For the i-th hidden layer, we denote the layer inputs (pre-activations) by $\hat{y}_i \in \mathbb{R}^{k_i}$ and the layer outputs (post-activations) by $y_i \in \mathbb{R}^{k_i}$. Specifically, we consider networks with ReLU activation layer which is parameterized by *weights*, W_i, and *biases*, b_i, and is defined as $f_i : y \in \mathbb{R}^{k_{i-1}} \mapsto \max\{W_i y + b_i, 0\} \in \mathbb{R}^{k_i}$.

Hence, the neurons' input and output can be represented by

$$\begin{aligned} \hat{y}_i &= W_i y_{i-1} + b_i, \qquad && i = 1, ..., n \\ y_i &= \max(\mathbf{0}, \hat{y}_i), && i = 1, ..., n-1 \\ y_0 &= x,\ z = y_n = \hat{y}_n, \end{aligned} \tag{2.1}$$

where $W_i \in \mathbb{R}^{k_i \times k_{i-1}}$, $b_i \in \mathbb{R}^{k_i}$ are the weights and bias of the i-th layer respectively, and the max function is taken element-wise. Moreover, we denote the j-th neuron in the i-th layer by n_{ij}, the lower bound of $\hat{y}_i$ by $\hat{l}_i$, the upper bound of $\hat{y}_i$ by by $\hat{u}_i$. Similarly, the lower and upper bounds of y_i are denoted by l_i and u_i.

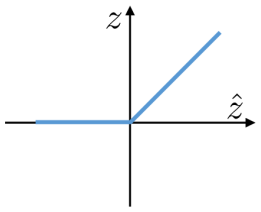
2.2 ReLU linear relaxation

Due to the non-linearity and non-convexity of the ReLU function, many of the algorithms that will be discussed use a convex relaxation to approximate the ReLU neurons. Ehlers [12] introduced linear (triangular) relaxation to approximate the ReLU non-linear function. The relaxation replaces each ReLU constraint by a set of three linear constraints on the input and the output of the neuron. This convex relaxation leads to an overestimation of the actual output set of the ReLU function.

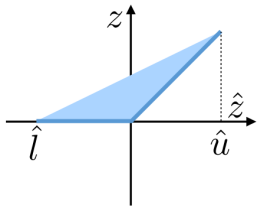
Formally, for a ReLU with input $\hat{z}$, the output set is given by $\{z \mid z = max(0, \hat{z})\}$. Given the lower and upper bounds $\hat{l}$ and $\hat{u}$ on the ReLU input, the relaxed convex set will be

$$\{z \mid z \geq 0, \quad z > \hat{z}, \quad z \leq \frac{\hat{u}(\hat{z} - \hat{l})}{\hat{u} - \hat{l}}\}.$$

Figure 2.1 shows a pictorial representation of the ReLU function (figure 2.1a) and its relaxation (figure 2.1b).



(a) ReLU activation function



(b) ReLU triangular relaxation

Figure 2.1: Triangular relaxation of the ReLU function

2.3 Interval arithmetic

Calculating the lower and upper bounds of each neuron is used in many of the tools discussed in this thesis. In this section we discuss interval arithmetic and how it can be used to compute the neuron lower and upper bounds.

By using interval arithmetic, given the bounds of layer $i-1$, the bounds at layer i is given by

$$\hat{l}_i = W_i^+ l_{i-1} + W_i^- u_{i-1} + b_i, \tag{2.2}$$

$$\hat{u_i} = W_i^+ u_{i-1} + W_i^- l_{i-1} + b_i, \tag{2.3}$$

$$l_i = \max(0, \hat{l}_i), \tag{2.4}$$

$$u_i = \max(0, \hat{u_i}), \tag{2.5}$$

where $W^+ = \max(0, W^+)$ and $W^- = \min(0, W^-)$ element wise. The interval arithmetic bounds are usually very loose, which is a major drawback of this method. This is due to the fact that the neurons cannot reach their maximum and minimum at the same time. This is not taken into account in equations (2.2)-(2.5)

2.4 Symbolic interval analysis

To tighten the bounds computed by interval arithmetic, [31] introduced symbolic interval analysis. Symbolic interval analysis propagates linear symbolic equations instead of concrete bounds. Concrete neuron bounds can then be computed by maximizing and minimizing the symbolic equation.

Let $eq_{low}^{i-1}(x)$ and $eq_{up}^{i-1}(x)$ be the lower and upper bound equations of layer $i-1$. Then the preactivation lower and upper bound equations of layer i can be computed using the formulas:

$$\hat{eq}_{low}^{i}(x) = W_i^+ eq_{low}^{i-1}(x) + W_i^- eq_{up}^{i-1}(x) + b_i$$

$$\hat{eq}_{up}^{i}(x) = W_i^+ eq_{up}^{i-1}(x) + W_i^- eq_{low}^{i-1}(x) + b_i$$

The challenge in symbolic propagation is how to maintain linear bound equations. However, propagating the preactivation bounds equations through a ReLU will yield a nonlinear function. Instead of propagating the symbolic bounds through the ReLU function, Wang et al.[31] addressed this issue by introducing two linear relaxations, the upper bound equation is propagated through the upper linear relaxation, while the lower bound equation is propagated through the lower linear relaxation. Figure 2.2 shows the upper and lower linear relaxations used.

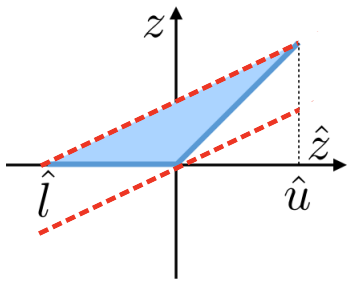


Figure 2.2: Symbolic interval analysis relaxations, the upper dotted line is the upper relaxation, and the lower line is the lower bound relaxation

Formally, the post-activation lower and upper bound linear equations for neuron n_{ij} can be computed using the formulas:

$$eq_{low}^{ij}(x) = \frac{u_{low}^{ij}}{u_{low}^{ij} - l_{low}^{ij}} eq_{low}^{ij}(x),$$

$$eq_{up}^{ij}(x) = \frac{u_{up}^{ij}}{u_{up}^{ij} - l_{up}^{ij}} (eq_{low}^{ij}(x) - l_{up}^{ij}),$$

where u_{low}^{ij} and l_{low}^{ij} are the pre-activation concrete upper and lower bounds for the lower bound equation $\hat{eq}_{low}^{ij}$. Similarly, u_{up}^{ij} and l_{up}^{ij} are the pre-activation concrete upper and lower bounds for the upper bound equation. These bounds can be simply be computed as follows. For a linear equation $eq(x) = \sum_i c_i x_i$, the upper and lower bounds are given by:

$$u = \sum_{i|c_i>0} c_i x_i^u + \sum_{i|c_i\leq 0} c_i x_i^l \tag{2.6}$$

$$l = \sum_{i|c_i>0} c_i x_i^l + \sum_{i|c_i\leq 0} c_i x_i^u, \tag{2.7}$$

where x_i^u and x_i^l are the upper and lower bounds on the input to the neural network.

2.5 Neural network 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 for all inputs $x \in P_{y_0}$, the output is $\mathcal{NN}(x) \in P_{y_n}$ given the constraints h_ℓ for $\ell = 1, \ldots, m$. Formally, this is equivalent to deciding whether

$$\left\{ x \in \mathbb{R}^{k_0} \;\middle|\; x \in P_{y_0} \bigwedge \mathcal{NN}(x) \in P_{y_n} \bigwedge \left(\bigwedge_{\ell=1}^{m} h_\ell(x, \mathcal{NN}(x)) \leq 0 \right) \right\} = \emptyset. \tag{2.8}$$

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}.

So, given the input and output constraints, we either prove that no valid input that violates those constraints and deem the property provably satisfied (SAT) or find an input that does and return a counter example which shows that the property is unsatisfied (UNSAT).

We can verify Neural networks against different specifications in different domains using the same formulation explained above. We will discuss two different problems in chapter 6 where we are able to use this formulation to verify neural networks with different types of specifications.

Chapter 3

Related work

In this chapter, we review current state of the art verification algorithms, discuss them briefly, and categorize them into different categories based on their underlying methodologies. We only consider to sound and complete verification algorithms. Most of the SOTA algorithms can be grouped in four categories as follows.

3.1 SMT-based methods

SMT-based algorithms [19, 18, 12] rely on a Linear Programming (LP) solver to check the feasibility of a relaxed version of the verification problem. If the algorithm cannot conclude the satisfiability of the property, they refine the problem by splitting the nonlinear neurons, the neuron splitting is done by the aid of a SAT solver to assign different phases to the nonlinear neurons. Even though these algorithm use a similar strategy, their approach is different. Katz et al.[18] extend the simplex algorithm for solving LPs to be able to support the non-linear constraints of the ReLU neurons. Ehlers[12] used linear relaxation to relax the nonlinear ReLUs and then use an LP solver to check the feasibility of the problem. It

uses a SAT solver to explore all possible combinations of neurons' splits.

3.2 MILP-based methods

Algorithms in this category [23, 29, 6, 8, 15, 3, 9, 7] directly encode the verification problem in (2.8) into a Mixed Integer Linear Program (MILP) using five constraints per neuron. Using MILP requires computing the upper and lower bounds of each neurons, which can be computed by interval arithmetic or linear programming.

3.3 Reachability based methods

Reachability based methods [4, 34, 35, 16, 32, 30, 17, 14, 5] perform layer-by-layer reachability analysis to compute the reachable set and compare it with the property output set to conclude satisfiability of the propert. Wang et al. [32] uses interval arithmetic to compute an overapproximation of the output reachable set. If the algorithm can't conclude the satisfiability of the property, they refine the problem by splitting the input domain of the neural network which results in two refined problems that are verified independantly. Another class of algorithms [4, 30] use star sets to compute an overapproximation of the output reachable set, they refine the problem by splitting one of the hidden neurons if the initial problem was inconclusive.

3.4 Convex relaxations methods

Convex relaxation methods[31, 11, 33] use linear relaxation to the ReLU constraints and rely on solving a convex program to determine the feasibility of the problem. Similar to

reachability, if the solution of the convex program is not conclusive, these methods refine the problem by splitting one of the neurons and solving two convex problems, one for each phase.

In general, SMT, MILP and reachability methods 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 [10]). The convex relaxation methods are the most similar to PeregriNN. The full details of the algorithm is explained in chapter 4.

Chapter 4

Algorithm

The general structure of PeregriNN is depicted in Figure 4.1. 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 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

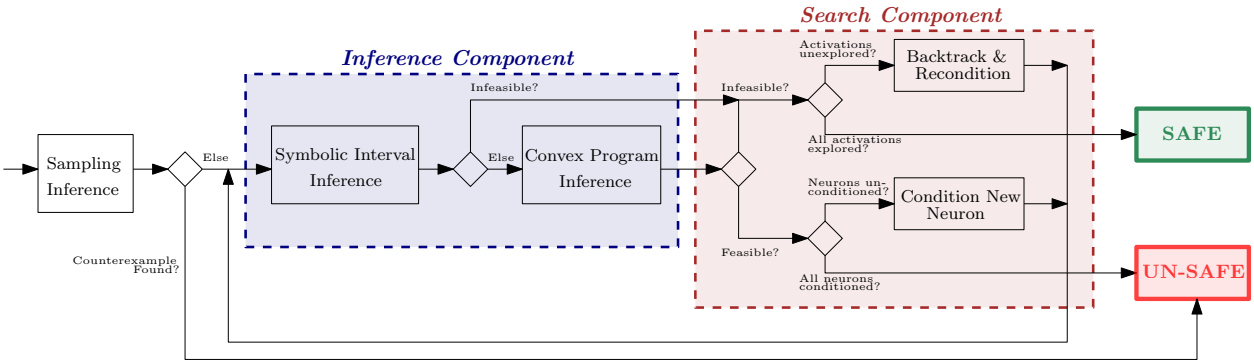


Figure 4.1: Block Diagram of the PeregriNN Algorithm

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 Equation 2.1, such a linear feasibility program can be written as follows, where the vector variables $y_i, i \neq 0$ are the relaxation variables.

$$\begin{cases} y_i \geq 0,\ y_i \geq W_i y_{i-1} + b_i & \forall i = 1, \ldots, n \\ y_0 \in P_{y_0},\ y_n \in P_{y_n}^{\mathsf{c}},\ \bigwedge_{\ell=1}^{m} h_\ell(y_0, y_n) \leq 0 \end{cases} \tag{4.1}$$

Importantly, if (4.1) is infeasible, then the original NN problem in (2.8) may be soundly concluded to be infeasible as well – and hence, the property is SAT. 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

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
```

incorporated into the feasibility program (4.1) as *equality* constraints in the following way:

$$\text{Neuron } (y_i)_j \text{ ON:} \quad (y_i)_j = (W_i y_{i-1} + b_i)_j \wedge (y_i)_j \geq 0 \tag{4.2}$$

$$\text{Neuron } (y_i)_j \text{ OFF:} \quad (y_i)_j = 0 \wedge (W_i y_{i-1} + b_i)_j \leq 0. \tag{4.3}$$

Inferences created by the symbolic interval inference block using Symbolic Interval Analysis [32] are also incorporated using equality constraints like (4.2) and (4.3). Algorithm 1 briefly summarizes the verification steps.

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 section 4.1; the former implements a novel neuron prioritization, and the latter is a unique approach to quickly obtaining initial safety counterexamples.

4.1 PeregriNN Enhancements

4.1.1 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" variable 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.

4.1.2 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].

4.1.3 Layer-wise-Weighted Penalty

PeregriNN takes the "max-slack" neuron search priority one step further, though. Using techniques similar to those in [25], it is possible to show that there exists weights $q_1, \ldots, q_n$ such that solving (4.1) with the penalty

$$\min_{y_0,..,y_n} \sum_{i=0}^{n} \sum_{j=1}^{k_i} q_i s_{ij} \tag{4.4}$$

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 (4.4), 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).

4.1.4 Initial Counterexample Search by Sampling

Finally, PeregriNN incorporates a simple, yet novel, approach to help identify un-safe networks early on. In particular, PeregriNN uses the Volesti [13] Python library to uniformly sample points within the input constraint set, P_x, and evaluates the network exactly for those inputs. Unsafe outputs thus constitute counterexamples to the safety of the network, and PeregriNN can terminate.

The impact of these enhancements are tested by running ablation tests. These tests are reported in chapter 6

Chapter 5

Verification Problems

In this chapter we discuss two different applications where verification of NNs is needed. We briefly discuss both tasks, and how can each of the verification problems be formulated as discussed in chapter 2.

5.1 Adversarial Robustness

A Neural network is said to be adversarially robust, if small perturbations in the input doesn't lead to misclassification of the target. We consider max-norm perturbations in this work. 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

$$\phi_m \triangleq \{x \mid x \in \mathbb{R}^{k_0},\, \|x - x'\|_\infty \leq \epsilon,\, z \in \mathbb{R}^{k_n},\, \max_{i=1,\ldots,n} \mathcal{NN}(x)_i = \mathcal{NN}(x)_m\} \tag{5.1}$$

is empty. Note that each instance of (5.1) is compatible with the problem in (2.8).

5.2 Safety of a neural network controlled quadrotor

We consider the problem of formally verifying the safety of an autonomous robot equipped with a Neural Network (NN) controller that processes LiDAR images to produce control actions. Given a workspace that is characterized by a set of polytopic obstacles, our objective is to compute the set of safe initial states such that a robot trajectory starting from these initial states is guaranteed to avoid the obstacles. Our approach is to construct a finite state abstraction of the system and use standard reachability analysis over the finite state abstraction to compute the set of safe initial states. To mathematically model the imaging function, that maps the robot position to the LiDAR image, we introduce the notion of imaging-adapted partitions of the workspace in which the imaging function is guaranteed to be affine. Given this workspace partitioning, a discrete-time linear dynamics of the robot, and a pre-trained NN controller with Rectified Linear Unit (ReLU) nonlinearity, we use PeregriNN to verify the safety of transitions between workspace partitions.

5.2.1 Dynamics and Workspace

We consider an autonomous robot moving in a 2-dimensional polytopic (compact and convex) workspace $\mathcal{W} \subset \mathbb{R}^2$. We assume that the robot must avoid the workspace boundaries ∂W along with a set of obstacles $\{\mathcal{O}_1, \ldots, \mathcal{O}_o\}$, with $\mathcal{O}_i \subset \mathcal{W}$ which is assumed to be polytopic. We denote by $\mathcal{O}$ the set of the obstacles and the workspace boundaries which needs to be avoided, i.e., $\mathcal{O} = \{\partial W, \mathcal{O}_1, \ldots, \mathcal{O}_o\}$. The dynamics of the robot is described by a discrete-time linear system of the form:

$$x^{(t+1)} = Ax^{(t)} + Bu^{(t)}, \tag{5.2}$$

where $x^{(t)} \in \mathcal{X} \subseteq \mathbb{R}^n$ is the state of robot at time $t \in \mathbb{N}$ and $u^{(t)} \subseteq \mathbb{R}^m$ is the robot input.

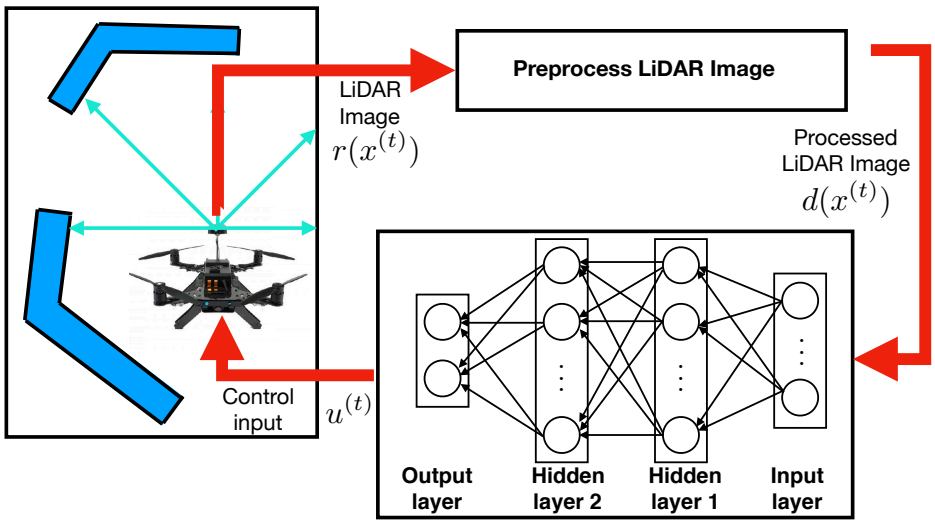


Figure 5.1: Pictorial representation of the problem setup under consideration.

The matrices A and B represent the robot dynamics and have appropriate dimensions. For a robot with nonlinear dynamics that is either differentially flat or feedback linearizable, the state space model (5.2) corresponds to its feedback linearized dynamics. We denote by $\zeta(x) \in \mathbb{R}^2$ the natural projection of x onto the workspace $\mathcal{W}$, i.e., $\zeta(x^{(t)})$ is the position of the robot at time t.

5.2.2 LiDAR Imaging

We consider the case when the autonomous robot uses a LiDAR scanner to sense its environment. The LiDAR scanner emits a set of N lasers evenly distributed in a 2π degree fan. We denote by $\theta^{(t)}_{\text{lidar}} \in \mathbb{R}$ the heading angle of the LiDAR at time t. Similarly, we denote by $\theta^{(t)}_i = \theta^{(t)}_{\text{lidar}} + (i-1)\frac{2\pi}{N}$, with $i \in \{1, \ldots, N\}$, the angle of the ith laser beam at time t where $\theta^{(t)}_1 = \theta^{(t)}_{\text{lidar}}$ and by $\theta^{(t)} = (\theta^{(t)}_1, \ldots, \theta^{(t)}_N)$ the vector of the angles of all the laser beams. While the heading angle of the LiDAR, $\theta^{(t)}_{\text{lidar}}$, changes as the robot pose changes over time, i.e., $\theta^{(t)}_{\text{lidar}} = f(x^{(t)})$ for some nonlinear function f, in this work we focus on the case when the heading angle of the LiDAR, $\theta^{(t)}_{\text{lidar}}$, is fixed over time and we will drop the superscript t from the notation. Such condition is satisfied in several real-world scenarios whenever the robot

is moving while maintaining a fixed pose (e.g. a quadrotor whose yaw angle is maintained constant).

For the ith laser beam, the observation signal $r_i(x^{(t)}) \in \mathbb{R}$ is the distance measured between the robot position $\zeta(x^{(t)})$ and the nearest obstacle in the θ_i direction, i.e.:

$$\begin{aligned} r_i(x^{(t)}) = \min_{\mathcal{O}_i \in \mathcal{O}} \min_{z \in \mathcal{O}_i} & \|z - \zeta(x^{(t)})\|_2 \\ \text{s.t.} \quad & \operatorname{atan2}\left(z - \zeta(x^{(t)})\right) = \theta_i. \end{aligned} \tag{5.3}$$

In this work, we restrict our attention to the case when the LiDAR scanner is ideal (with no noise) although the bounded noise case can be incorporated in the proposed framework. The final LiDAR image $d(x^{(t)}) \in \mathbb{R}^{2N}$ is generated by processing the observations $r(x^{(t)})$ as follows:

$$\begin{aligned} d_i(x^{(t)}) &= \left(r_i(x^{(t)}) \cos\theta_i,\ r_i(x^{(t)}) \sin\theta_i\right), \\ d(x^{(t)}) &= \left(d_1(x^{(t)}), \ldots d_N(x^{(t)})\right). \end{aligned} \tag{5.4}$$

As shown in (5.4), the LiDAR imaging function is non-linear which makes it hard to verify the safety specification. PeregriNN can verify this system only if the imaging constraints were linear. We proposed an algorithm that partitions the input space in a way, such that in each partition, the LiDAR imaging function is linear. We call this an imaging-adapted partitioning.

5.2.3 Imaging-Adapted Workspace Partitioning

We start by introducing the notation of the important geometric objects. We denote by $\textsc{Ray}(w, \theta)$ the ray originated from a point $w \in \mathcal{W}$ in the direction θ, i.e.:

$$\textsc{Ray}(w, \theta) = \{w' \in \mathcal{W} \mid \text{atan2}(w' - w) = \theta\}.$$

Similarly, we denote by $\textsc{Line}(w_1, w_2)$ the line segment between the points w_1 and w_2, i.e.:

$$\textsc{Line}(w_1, w_2) = \{w' \in \mathcal{W} \mid w' = \nu w_1 + (1 - \nu) w_2,\ 0 \leq \nu \leq 1\}.$$

For a convex polytope $P \subseteq \mathcal{W}$, we denote by $\textsc{Vert}(P)$, its set of vertices and by $\textsc{Edge}(P)$ its set of line segments representing the edges of the polytope.

Imaging-Adapted Partitions

The basic idea behind our algorithm is to partition the workspace into a set of polytopic sets (or regions) such that for each region $\mathcal{R}$ the LiDAR rays intersects with the same obstacle/workspace edge regardless of the robot positions $\zeta(x) \in \mathcal{R}$. To formally characterize this property, let $\mathcal{O}^\star = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \mathcal{O}_i$ be the set of all points in the workspace in which an obstacle or workspace boundary exists. Consider a workspace partition $\mathcal{R} \subseteq \mathcal{W}$ and a robot position $\zeta(x)$ that lies inside this partition, i.e., $\zeta(x) \in \mathcal{R}$. The intersection between the kth LiDAR laser beam $\textsc{Ray}(\zeta(x), \theta_k)$ and $\mathcal{O}^\star$ is a unique point characterized as:

$$z_{k,\zeta(x)} = \underset{z \in \mathcal{W}}{\text{argmin}} \, \|z - \zeta(x)\|_2 \quad \text{s.t.} \quad z \in \textsc{Ray}(\zeta(x), \theta_k) \cap \mathcal{O}^\star. \tag{5.5}$$

By sweeping $\zeta(x)$ across the whole region $\mathcal{R}$, we can characterize the set of all possible intersection points as:

$$\mathcal{L}_k(\mathcal{R}) = \bigcup_{\zeta(x)\in\mathcal{R}} z_{k,\zeta(x)}.$$

Using the set $\mathcal{L}_k(\mathcal{R})$ described above, we define the notion of imaging-adapted partitions as follows.

Definition 5.1 *A set $\mathcal{R} \subset \mathcal{W}$ is said to be an imaging-adapted partition if the following property holds:*

$$\mathcal{L}_k(\mathcal{R}) \text{ is a line segment} \quad \forall k \in \{1, \ldots, N\}.$$

Figure 5.2 shows concrete examples of imaging-adapted partitions. Imaging-adapted partitions enjoy the following property:

Lemma 5.2.1 *Consider an imaging-adapted partition $\mathcal{R}$ with corresponding sets $\mathcal{L}_1(\mathcal{R}), \ldots, \mathcal{L}_N(\mathcal{R})$. The LiDAR imaging function $d : \mathcal{R} \to \mathbb{R}^{2N}$ is an affine function of the form:*

$$d_k(\zeta(x)) = P_{k,\mathcal{R}}\zeta(x) + Q_{k,\mathcal{R}}, \qquad d = (d_1, \ldots, d_N) \tag{5.6}$$

for some constant matrices $P_{k,\mathcal{R}}$ and vectors $Q_{k,\mathcal{R}}$ that depend on the region $\mathcal{R}$ and the LiDAR angle θ_k.

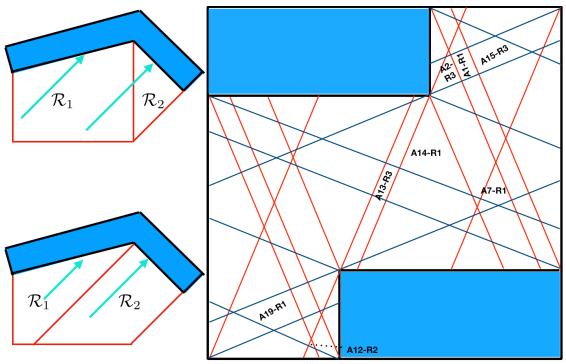


Figure 5.2: (left-up) A partitioning of the workspace that is *not* imaging-adapted. Within region $\mathcal{R}_1$, the LiDAR ray (cyan arrow) intersects with different obstacle edges depending on the robot position. (left-down) A partitioning of the workspace that is imaging-adapted. For both regions $\mathcal{R}_1$ and $\mathcal{R}_2$, the LiDAR ray (cyan arrow) intersects the same obstacle edge regardless of the robot position. (right) Imaging-adapted partitioning of the workspace.

5.2.4 Finite state abstraction

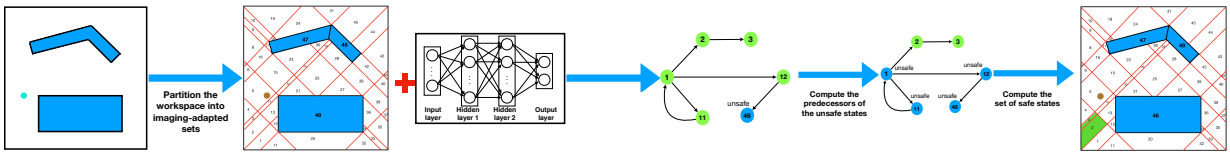


Figure 5.3: Pictorial representation of the proposed framework.

Before we describe the proposed framework, we need to briefly recall the following definitions capturing the notion of a system and relations between different systems.

Definition 5.2 *An autonomous system $\mathcal{S}$ is a pair (X, δ) consisting of a set of states X and a set-valued map $\delta : X \to X$ representing the transition function. A system $\mathcal{S}$ is finite if X is finite. A system S is deterministic if δ is single-valued map and is non-deterministic if it is not deterministic.*

Definition 5.3 *Consider a deterministic system $\mathcal{S}_a = (X_a, \delta_a)$ and a non-deterministic system $\mathcal{S}_b = (X_b, \delta_b)$. A relation $Q \subseteq X_a \times X_b$ is a simulation relation from $\mathcal{S}_a$ to $\mathcal{S}_b$,*

and we write $\mathcal{S}_a \preccurlyeq_Q \mathcal{S}_b$*, if the following conditions are satisfied:*

1. *for every* $x_a \in X_a$ *there exists* $x_b \in X_b$ *with* $(x_a, x_b) \in Q$*,*
2. *for every* $(x_a, x_b) \in Q$ *we have that* $x'_a = \delta_a(x_a)$ *in* $\mathcal{S}_a$ *implies the existence of* $x'_b \in \delta_b(x_b)$ *in* $\mathcal{S}_b$ *satisfying* $(x'_a, x'_b) \in Q$*.*

Using the previous two definitions, we describe our approach as follows. As pictorially shown in Figure 5.3, given the autonomous robot system $\mathcal{S}_{\text{NN}} = (\mathcal{X}, \delta_{\text{NN}})$, where $\delta_{\text{NN}} : x \mapsto Ax + B\ \mathcal{NN}(d(x))$, our objective is to compute a finite state abstraction (possibly non-deterministic) $\mathcal{S}_{\mathcal{F}} = (\mathcal{F}, \delta_{\mathcal{F}})$ of $\mathcal{S}_{\text{NN}}$ such that there exists a simulation relation from $\mathcal{S}_{\text{NN}}$ to $\mathcal{S}_{\mathcal{F}}$, i.e., $\mathcal{S}_{\text{NN}} \preccurlyeq_Q \mathcal{S}_{\mathcal{F}}$. This finite state abstraction $\mathcal{S}_{\mathcal{F}}$ will be then used to check the safety specification.

The first step to compute the finite state abstraction is to compute an imaging-adapted partitioning $\mathcal{W}^\star$ of the workspace. Unfortunately, the number of partitions grows exponentially in the number of lasers N and the number of vertices of the polytopic obstacles. To harness this exponential growth, we compute an aggregate-partitioning $\mathcal{W}'$ using only a few laser angles (called primary lasers and denoted by θ_p). The resulting aggregate-partitioning $\mathcal{W}'$ would contain a smaller number of partitions such that each partition in $\mathcal{W}'$ represents multiple partitions in $\mathcal{W}^\star$.

After constructing the transition system $\mathcal{S}_{\text{NN}}$ using the imaging adapted partitioning. We use PeregriNN to verify the feasibility of transition between different pairs of states. Lastly, Using backward reachability analysis, we can compute the complete safe and unsafe sets.

5.2.5 Verification framework

Using the notion of imaging-adapted workspace partitioning, the imaging function inside each partition is now linear, and the transition between two workspace partitions can be easily verified using PeregriNN. Given the partitions $S_1, \ldots, S_w$, we can 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 [27], 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:

$$\left\{ x \middle| x \in \bigcup_{m=1}^{w} S_m,\ Ax + B\,\mathcal{NN}(Hx + d) \in \bigcup_{t=1}^{o} O_t \right\} = \emptyset. \tag{5.7}$$

Indeed, the system safety property (5.7) can be checked by solving $w \times o$ formulas of the form (2.8).

Chapter 6

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 chapter 2, and it is described subsequently in detail. In this context, we evaluated PeregriNN with two objectives described as follows.

1. We conducted ablation experiments for all of PeregriNN's novel features as described in section 4.1. In particular, we compared the performance of a full implementation of PeregriNN – i.e. *exactly* as described in section 4.1 – 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.

Table 6.1: 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>

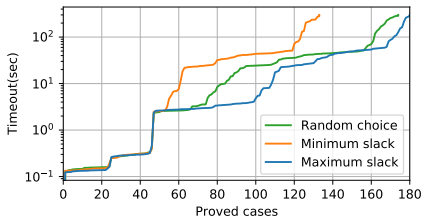
Implementation. We implemented PeregriNN in Python, and used an off-the-shelf Gurobi 9.1 [24] convex optimizer for solving linear programs; the Volesti [13] Python interface was used to sample from the input polytope for the sampling inference block. For the other NN verifiers, we used publicly available 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. The VMs were run no more than 4 at a time on a host machine with 48 hyperthreaded cores and 256 GB of memory; this ensured that each VM had adequate access to hardware resources.

6.1 Adversarial Robustness Verification Task

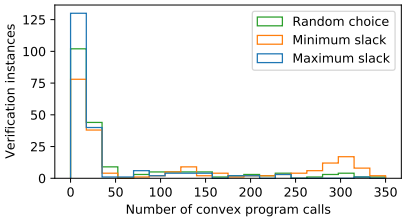
Subsequent experiments used the testbench we describe in this chapter.

Neural Networks. We used three ReLU NNs to recognize digits using the standard MNIST training database. The size and architectures of these networks are described in Table 6.1. Each entry in the "Architecture" column of Table 6.1 describes the architecture of the corresponding network in terms of number of neurons per 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 in-



(a) Cactus plot; proved cases vs. timeout



(b) Histogram; number convex calls used

Figure 6.1: Performance of PeregriNN variants with different conditioning priorities

put 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.

Adversarial Robustness Verifier Testbench Our verification testbench was then constructed by selecting 50 test images from the MNIST test dataset. Each test instance was then a combination of one of those images, one of the networks from Table 6.1 and one the following two max-norm perturbations, $\epsilon = 0.02$ or $\epsilon = 0.05$. Thus, each verification test in our testbench can be identified by one of 300 tuples of the form: $(net, image, perturb.) \in \mathscr{TB} \triangleq \{\texttt{FC1}, \texttt{FC2}, \texttt{FC2}\} \times \{1, \ldots, 50\} \times \{0.02, 0.05\}$.

6.1.1 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 section 4.1 – 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* $\epsilon = 0.05$.

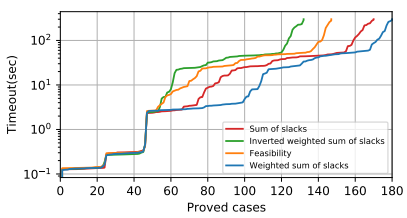
Penalty Function Ablation.

Our first ablation experiment evaluated the contribution of PeregriNN's unique penalty function features; see subsection 4.1.1 and subsection 4.1.3. In particular, we ran different variants of PeregriNN with the following penalty functions used inside the convex program inference block:

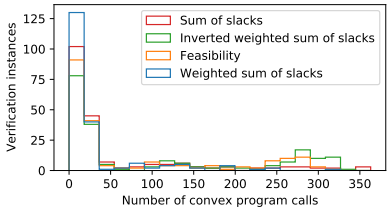
1. *"Weighted sum of slacks"*: PeregriNN's own weighted sum of slacks penalty;
2. *"Sum of slacks"*: A sum-of-slacks penalty with equal weighting on all layers;
3. *"Feasibility"*: A feasibility-only convex program such as the one used in other tools, e.g. [31] (i.e. simply using a constant penalty function 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 force slack towards deeper layers rather than shallower ones (see also subsection 4.1.3).

Figure 6.2a 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 Figure 6.2a. Figure 6.2b 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 6.2b plots the number of convex solver calls required for evenly spaced bins of convex solver calls.

<u>*Conclusions:*</u> Figure 6.2a 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



(a) Cactus plot; proved cases vs. timeout



(b) Histogram; number convex calls used

Figure 6.2: Performance of PeregriNN variants with different objective functions

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 of $\approx$1.2 seconds. This suggests that driving slacks toward shallower layers, where the next neuron is conditioned, is the correct heuristic to apply. Figure 6.2b 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 225 and 325 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 nearly 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 subsection 4.1.2). 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.

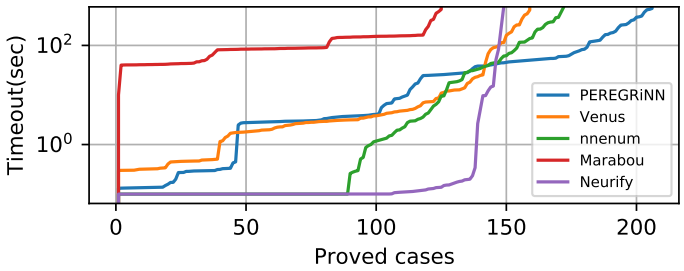


Figure 6.3: Cactus plot of various solvers on 300-case testbench, $\mathscr{TB}$

The performance of these algorithm variants is shown in Figure 6.1a and Figure 6.1b. As in the previous ablation experiment, Figure 6.1a shows a cactus plot of the number of proved cases vs. the timeout, and Figure 6.1b shows a histogram of the number of calls to the convex solver required under each of the conditioning priorities.

Conclusions: Figure 6.1a shows that PeregriNN's maximum-slack neuron priority allows it to prove more properties for a given timeout than either a random neuron choice priority or the minimum-slack choice priority. Figure 6.1b also shows that the benefit of this neuron priority is reflected in a dramatic reduction in the number of test cases where 250-350 convex solver calls are required; PeregriNN's maximum-slack priority has the most test cases requiring <25 calls.

6.1.2 Comparison with Other NN Verifiers

In this experiment, we evaluated the performance of 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 all 300 test cases in $\mathscr{TB}$: Venus [7]; Marabou [19]; Neurify [31]; and nnenum [5]. Venus was used 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 with *THREADS* = 1; and nnenum was used with *ADVERSARIAL_SEARCH* turned off. As noted, each algorithm ran inside its own single-core VM.

Figure 6.3 contains a cactus plot showing the results for each of these algorithms, including PeregriNN. For a given number of test cases to be proved, Figure 6.3 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 300 seconds: PeregriNN was able to prove 206 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 $\approx$150 test cases, PeregriNN significantly outperformed all other algorithms: as the timeout was increased, PeregriNN continued prove additional properties at a rate significantly outpacing its closest competitor in this regime, nnenum.

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.

6.2 Safety of a neural network controlled quadrotor

We use PeregriNN to verify (5.7) 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 6.2 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 (5.7) for NNs with different architectures and recording the verification time. Table 6.3 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 [27], which uses SMC solvers [25], are capable of handling at most networks with 1000 ReLUs. Comparing PeregeriNN to SMC solver in [27], we conclude that PeregeriNN can verify networks that are 2 orders of magnitude larger than SMC with 1900 times less execution time.

Table 6.2: 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 6.3: **(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

Chapter 7

Conclusion

This thesis introduce a new solver, PeregriNN, 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.

PeregriNN is implemented in Python and it uses Gurobi off the shelf convex optimizer. It extends symbolic analysis methods with new contributions. We used our tool to verify NNs trained for different tasks like adversarial robustness of networks on the MNIST dataset and the safety of a NN controlled quadrotor. 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.

7.1 Summary of contributions

PeregriNN extends previous work by introducing novel contributions summarized as:

- *Sum-of-Slacks Penalty.* We introduced an convex optimization objective function that corresponds to the sum of slack variables in each layer. This incetivizes the convex solver to choose an actual inputoutput response of the network.
- *Max-Slack Conditioning Priority.* PeregriNN uses a max-slack heuristic to choose which neuron to condition. This neuron slack variable can be thought of as a measure of how problematic each neuron is.
- *Layer-wise-Weighted Penalty.* In addition to using a sum-of-slacks penalty, PeregriNN also uses a weighted penalty of those slack variables. These weights encourage the solver to push problematic neurons towards the start of the network. This leads to conditioning on earlier neurons and hence tightening the bounds of the outputs.
- *Initial Counterexample Search by Sampling.* PeregriNN also incorporates a sampling step early on to try to prove unsafe properties quickly.

7.2 Future work

PeregriNN performs better than SOTA methods on the MNIST benchmark. However, there are still room for improvements. This is a brief list of improvements and research directions that we think are promising:

- Many of the computationally expensive steps in PeregriNN, can be done in parallel using a multi-core system. For example, conditioning on different neurons (exploring different paths of the search tree) in parallel. In some problems, there is a need to solve multiple independant verification problems (e.g. Adversarial robustness). Each of those problems can be run in parallel.

- PeregriNN currently performs well on problem with large input space. However, for problems with small input space, other methods like input splitting would be faster. An interesting topic for future research is to build a unified framework for splitting the input space and the neurons, together with finding the suitable heuristics for splitting.

- This work focuses only on verifying neural networks to check if it satisfies a property or not. An interesting research direction is to figure out how the network can be changed in order to satisfy the specification, or even better, how can the training process be altered to generate a network that satisfies a specification.

- Studying the safety of a neural network controlled system as a binary property is restrictive in my opinion. Designing an autonomous system that is much safer -but not necessarily 100% safe- than a benchmark system, can by itself be a huge success. In the context of autonomous driving for example, designing autonomous cars that are statistically much safer that human drivers is a satisfying goal. This motivates for studying the safety of such systems in a probabilistic fashion without treating the safety property as a restrictive binary metric.

Bibliography

[1] Tesla's autopilot was involved in another deadly car crash. `https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/`.

[2] Uber's self-driving car saw the woman it killed, report says. `https://www.wired.com/story/uber-self-driving-crash-arizona-ntsb-report/`.

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

[4] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson. Improved geometric path enumeration for verifying relu neural networks. In *International Conference on Computer Aided Verification*, pages 66–96. Springer, 2020.

[5] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson. Improved Geometric Path Enumeration for Verifying ReLU Neural Networks. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, volume 12224 of *Lecture Notes in Computer Science*, pages 66–96. Springer International Publishing, 2020.

[6] 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*, pages 2613–2621, 2016.

[7] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient verification of relu-based neural networks via dependency analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3291–3299, 2020.

[8] 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*, 21(2020), 2020.

[9] C.-H. Cheng, G. Nührenberg, and H. Ruess. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017.

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

[11] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In *UAI*, volume 1, page 2, 2018.

[12] R. Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

[13] I. Z. Emiris and V. Fisikopoulos. Practical Polytope Volume Approximation. *ACM Transactions on Mathematical Software*, 44(4):38:1–38:21, 2018.

[14] 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*, pages 11423–11434, 2019.

[15] M. Fischetti and J. Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

[16] 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)*, pages 3–18. IEEE, 2018.

[17] 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*, pages 169–178, 2019.

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

[19] 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*, pages 443–452. Springer, 2019.

[20] H. Khedr, J. Ferlez, and Y. Shoukry. Effective formal verification of neural networks using the geometry of linear regions, 2020.

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

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

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

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

[25] 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*, 106(9):1655–1679, 2018.

[26] 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.

[27] 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*, pages 147–156, 2019.

[28] 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.

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

[30] 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.

[31] 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*, pages 6367–6377, 2018.

[32] 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)*, pages 1599–1614, 2018.

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

[34] 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.

[35] 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*, 29(11):5777–5783, 2018.