UNIVERSITY OF CALIFORNIA,
IRVINE

Formal Verification of Neural Networks: Algorithms and Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Haitham Khedr

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I extend my deepest gratitude to my academic advisor, Professor Yasser Shoukry, whose guidance, mentorship, and unwavering support have been invaluable throughout the course of my doctoral research. Your expertise, encouragement, and commitment to excellence have shaped not only this thesis but also my academic and professional growth.

I would like to express my gratitude to the National Science Foundation (NSF) for their financial support, which enabled me to conduct the experiments and research necessary for this thesis.

I am indebted to my colleagues and fellow researchers at Resilient Cyber-Physical Systems Lab, whose collaboration, discussions, and shared insights greatly enriched the quality of my work. I also want to thank all my co-authors and collaborators; Xiaowu Sun, James Ferlez, and Wael Fatnassi. It has been amazing working with all of you.

Heartfelt thanks to my friends and peers for their support, encouragement, and understanding during the challenging moments of this academic journey. Your friendship has been a source of strength and inspiration.

I want to acknowledge the support of my parents. Your love, understanding, and patience sustained me through the highs and lows of this doctoral pursuit.

In the journey of pursuing this PhD, I have been fortunate to have a companion and source of unwavering support—my beloved spouse. To you, I dedicate a profound acknowledgment, for your immeasurable contributions to my academic success and personal well-being. Your enduring patience, understanding, and encouragement have been the bedrock upon which I built this thesis. Through the long hours, the inevitable setbacks, and the moments of self-doubt, your belief in my abilities has fueled my perseverance.

Finally, I dedicate this thesis to my lovely newborn daughter and the countless individuals whose names may not be mentioned but who, in various ways, contributed to my academic and personal development. Your collective influence is woven into the fabric of this work.

Thank you all for being integral parts of my academic journey.

VITA

Haitham Khedr

EDUCATION

Doctor of Philosophy in Electrical Engineering and Computer Science **2023**
University of California, Irvine *Irvine, CA*

Bachelor of Science in Electrical and Computer Engineering **2016**
Ain Shams University *Cairo, Egypt*

PUBLICATIONS

- Xiaowu Sun, **Haitham Khedr**, and Yasser Shoukry, "Formal verification of neural network controlled autonomous systems," in Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, 2019, pp. 147–156
- **Haitham Khedr**, James Ferlez, and Yasser Shoukry, "PeregriNN: Penalized-Relaxation Greedy Neural Network Verifier," in International Conference on Computer Aided Verification, 2021, pp. 287–300
- James Ferlez, **Haitham Khedr**, and Yasser Shoukry, "FastBATLLNN: fast box analysis of two-level lattice neural networks," in 25th ACM International Conference on Hybrid Systems: Computation and Control, 2022, pp. 1–11
- **Haitham Khedr** and Yasser Shoukry, "CertiFair: A Framework for Certified Global Fairness of Neural Networks," in Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI-23)
- Wael Fatnassi*, **Haitham Khedr***, Valen Yamamoto, and Yasser Shoukry. "BERN-NN: Tight Bound Propagation For Neural Networks Using Bernstein Polynomial Interval Arithmetic." in Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control
- **Haitham Khedr** and Yasser Shoukry. "DeepBern-Nets: Taming the Complexity of Certifying Neural Networks using Bernstein Polynomial Activations and Precise Bound Propagation. (preprint)

ABSTRACT OF THE DISSERTATION

Formal Verification of Neural Networks: Algorithms and Applications

By

Haitham Khedr

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Irvine, 2023

Associate Professor Yasser Shoukry, Chair

Neural networks (NNs) have become the backbone of intelligent systems, with applications ranging from sensing modules to learning-based controllers. Their deployment in safety-critical domains such as healthcare and transportation underscores their significance. However, the fragility of NNs and their potential for dangerous mistakes, whether unintentional or adversarial, necessitates rigorous checks on their behavior. This thesis delves into the Formal Verification of NNs, a process that can provides formal guarantees on their behavior and ensures the reliability and robustness of these systems.

The cornerstone of this work is a novel algorithm, PeregriNN, that has advanced the Formal Verification of NNs. This algorithm has been used to verify various NN properties such as Adversarial Robustness, safety of autonomous systems, and in demonstrating the fairness of NNs by innovatively formulating the fairness property into a Formal Verification problem. Further, this thesis explores a new type of activation function that simplifies the formal verification process, albeit at the cost of increased training time. This exploration, coupled with a collaborative effort that led to the proposal of a polynomial approximation method for ReLU NNs to formally verify them, provides valuable insights into the balance between verification ease and computational efficiency. These methods offer promising approaches to the formal verification of neural networks, further enhancing the robustness and reliability of these systems. The thesis also extends the application of these theories and algorithms to the safety of autonomous systems with neural network controllers. This practical application underscores the real-world implications of formal verification in ensuring the safety and reliability of autonomous systems.

In summary, this thesis provides a comprehensive understanding of the formal verification of neural networks, underscoring the importance of algorithm development and its real-world applications. The findings from this study contribute significantly to this critical field of study, with implications for the fairness, safety, and robustness of NNs.

Chapter 1

Introduction

Neural networks (NNs) have revolutionized numerous fields, from healthcare to transportation, due to their ability to learn complex patterns and make accurate predictions. However, despite their widespread adoption and impressive capabilities, NNs are not without their challenges. This thesis delves into three critical issues associated with NNs: lack of robustness, fairness, and safety. Robustness in NNs refers to their ability to maintain performance when faced with adversarial inputs or slight modifications to the input data. Despite their sophisticated learning capabilities, NNs have shown to be surprisingly fragile. They can make dangerous mistakes when confronted with adversarial attacks or unexpected inputs, leading to significant concerns about their reliability. Fairness in NNs is another critical issue. As NNs are increasingly used in decision-making processes affecting individuals and communities, it is crucial that they do not perpetuate or amplify existing biases. However, ensuring fairness in NNs is a complex task due to the opaque nature of their decision-making process and the potential biases in the training data. Another crucial property of NN is safety which is particularly important when NNs are deployed in safety-critical domains. An unsafe NN can lead to catastrophic consequences, especially in applications like autonomous vehicles or healthcare systems. Ensuring the safety of NNs requires rigorous checks on their behavior and the ability to provide safety guarantees. This thesis focuses on the formal verification of neural networks as a means to address these challenges. Formal verification is a process that provides formal guarantees by mathematically proving the correctness of a system with respect to its specifications. In the context of neural networks, formal verification ensures that the network behaves as intended, even under unexpected inputs or adversarial attacks. To that end, this thesis proposes a novel solver, PeregriNN, for verifying feed-forward and convolutional NNs. PeregriNN is primarily designed to verify robustness properties of NNs. However, we have extended it to reason about more complex properties such as fairness by first formalizing the fairness property as a formal verification property and introducing the algorithmic changes needed to verify such a property. Additionally, this thesis introduces a novel NN activation function and studies its impact on the efficiency of NN verification, as well as its drawbacks during training. This exploration provides valuable insights into striking a balance between ease of verification and computational efficiency.

The thesis is organized as follows, Chapter 2, lays the groundwork for the thesis. It provides an overview of the preliminary topics and introduces the notation used throughout the thesis. This chapter serves as a foundation, ensuring that readers are equipped with the necessary background knowledge to understand the subsequent chapters. Chapter 3, delves into the heart of the thesis by introducing PeregriNN. This chapter discusses in detail the algorithmic enhancements that make PeregriNN a significant advancement in the formal verification of neural networks. The intricacies of the algorithm are explored, providing a deep understanding of its workings and its contributions to the field. Chapter 4, discusses overapproximation the output of a ReLU NN using Bernstein Polynomials. We show that using polynomials approximations yield tighter bounds that SOTA methods and can increase verification efficacy. Chapter 5, introduces DeepBern-Nets, a type of neural networks with a novel activation function that is more amenable to verification. This chapter discusses the benefits and challenges of this new activation function, providing a balanced view of its impact on the efficiency of neural network verification and its drawbacks during training. Chapters 6 and 7 consider applications of formal verification of NNs in different domains. Chapter 6 discusses the formalization of individual fairness as a formal verification problem. It then discusses how PeregriNN can be extended to verify fairness properties and proposes two innovative techniques for training neural networks that are individually fair. This chapter provides valuable insights into the intersection of fairness and formal verification of NNs. Chapter 7 considers another application where formal verification is crucial due to the safety-critical nature of autonomous systems. In this chapter, we discuss how to use formal verifiers to verify safety properties defined using Linear Temporal Logic (LTL) of a NN-controlled autonomous system.

Chapter 2

Background

In this chapter, we introduce all the notation and concepts needed to define our problem, as well as the formal definition 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

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

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 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

$$\Big\{ x \in \mathbb{R}^{k_0} \ \Big| \ x \in P_{y_0} \wedge \mathcal{NN}(x) \in P_{y_n} \wedge \Big(\bigwedge_{\ell=1}^{m} h_\ell(x, \mathcal{NN}(x)) \leq 0 \Big) \Big\} = \emptyset. \tag{2.2}$$

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 problems in where we are able to use this formulation to verify neural networks with different types of specifications.

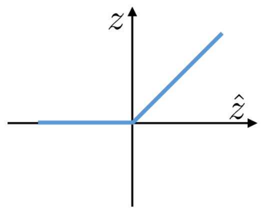
2.3 ReLU linear relaxation

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 [3] 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.

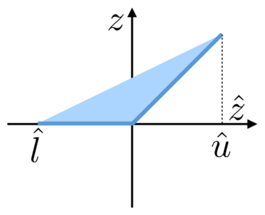
Formally, for a ReLU with input $\hat{z}$, the output set is given by $\{z \mid z = max(0, \hat{z})\}$. Given the lower and upper bounds $\hat{l}$ and $\hat{u}$ on the ReLU input, the relaxed convex set will be

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

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

2.4 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

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

$$\hat{u}_i = W_i^+ u_{i-1} + W_i^- l_{i-1} + b_i, \tag{2.4}$$

$$l_i = \max(0, \hat{l}_i), \tag{2.5}$$

$$u_i = \max(0, \hat{u}_i), \tag{2.6}$$

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 (2.3)-(2.6)

2.5 Symbolic interval analysis

To tighten the bounds computed by interval arithmetic, [4] 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.[4] 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.

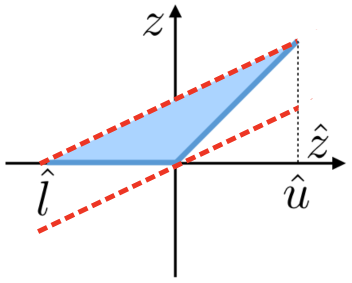


Figure 2.2: Symbolic interval analysis relaxations, the upper dotted line is the upper relaxation, and the lower line is the lower bound relaxation

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

$$eq^{ij}_{low}(x) = \frac{u^{ij}_{low}}{u^{ij}_{low} - l^{ij}_{low}} eq^{ij}_{low}(x),$$

$$eq^{ij}_{up}(x) = \frac{u^{ij}_{up}}{u^{ij}_{up} - l^{ij}_{up}} (eq^{ij}_{low}(x) - l^{ij}_{up}),$$

where u^{ij}_{low} and l^{ij}_{low} are the pre-activation concrete upper and lower bounds for the lower bound equation $\hat{eq}^{ij}_{low}$. Similarly, u^{ij}_{up} and l^{ij}_{up} 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:

$$u = \sum_{i|c_i>0} c_i x^u_i + \sum_{i|c_i\leq 0} c_i x^l_i \tag{2.7}$$

$$l = \sum_{i|c_i>0} c_i x^l_i + \sum_{i|c_i\leq 0} c_i x^u_i, \tag{2.8}$$

where x^u_i and x^l_i are the upper and lower bounds on the input to the neural network.

Chapter 3

PeregriNN: Neural Network Verifier

In this work, we introduce a new approach to formally verify the most commonly considered specifications for ReLU NNs – i.e. polytopic specifications on the input and output of the network. Like some other approaches, ours uses a relaxed convex program to mitigate the combinatorial complexity of the problem. However, unique in our approach is the way we use a convex solver not only as a linear feasibility checker, but also as a means of penalizing the amount of relaxation allowed in solutions. In particular, we encode each ReLU by means of the usual linear constraints, and combine this with a convex objective function that penalizes the discrepancy between the output of each neuron and its relaxation. This convex function is further structured to force the largest relaxations to appear closest to the input layer; this provides the further benefit that the most "problematic" neurons are conditioned as early as possible, when conditioning layer by layer. This paradigm can be leveraged to create a verification algorithm that is not only faster in general than competing approaches, but is also able to verify considerably more safety properties; we evaluated PEREGRiNN on a standard MNIST robustness verification suite to substantiate these claims.

3.1 Overview

We propose PeregriNN, an algorithm for efficiently and formally verifying the input/output behavior of ReLU NNs of the form 2.2. In this context, PeregriNN falls into the broad category of sound and complete *search and optimization* NN verifiers [5]. The *search* aspect of PeregriNN involves iterating over different combinations of neuron activation patterns to verify that each is compatible with the specified safety constraints (on the input and output of the network). Like other algorithms in this category, PeregriNN combines this search with *optimization* techniques to make inferences about the feasibility of full-network activation patterns on the basis of activation patterns of only a subset of neurons. The optimization in question reformulates the original NN feasibility problem into a relaxed convex feasibility problem to allow sound inferences: i.e. if the convex relaxation is infeasible, then the original NN problem may soundly be concluded to be infeasible. In this relaxed feasibility problem, the output of each individual neuron is assigned a relaxation variable that is decoupled from the actual output of that neuron. PeregriNN also uses a type of reachability analysis (symbolic interval analysis) both to enhance the optimization-based inference described above and as a source of additional sound inference itself. For this reason, PeregriNN's search procedure searches neurons in a layer-by-layer fashion, preferring to fix the phases of neurons closest to the input layer first.

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

Finally, we evaluated the performance of PeregriNN by using it to verify the adversarial robustness of networks trained on the MNIST [6] dataset. Our experiments show that PeregriNN is on average 1.27× faster than Neurify [4], 1.24× faster than Venus [7], 1.15× faster than nnenum [8], and 1.65× faster than Marabou [9]. It also proves 27 %, 19 %, 10 %, and 51 % more properties than the other solvers, respectively. PeregriNN's unique convex penalty augmentations are also considered in ablation experiments to validate their benefits.

Related work. Since PeregriNN is a sound and complete verification algorithm, we restrict our comparison to other sound and complete algorithms. NN verifiers can be grouped into roughly three categories: (i) SMT-based methods, which encode the problem into a Satisfiability Modulo Theory problem [3, 10, 9]; (ii) MILP-based solvers, which directly encode the verification problem as a Mixed Integer Linear Program [11, 12, 7, 13, 14, 15, 16, 17]; (iii) Reachability based methods, which perform layer-by-layer reachability analysis to compute the reachable set [8, 18, 19, 20, 21, 22, 23, 24]; and (iv) convex relaxations methods [25, 4, 26]. 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 [27]). These methods are most similar to PeregriNN: for example, [13, 28, 22] recursively refine the problem using input splitting, and [4] does so via neuron splitting. Other search and optimization methods include: Planet [3], which combines a relaxed convex optimization problem with a SAT solver to search over neurons' phases; and Marabou [9], which uses a

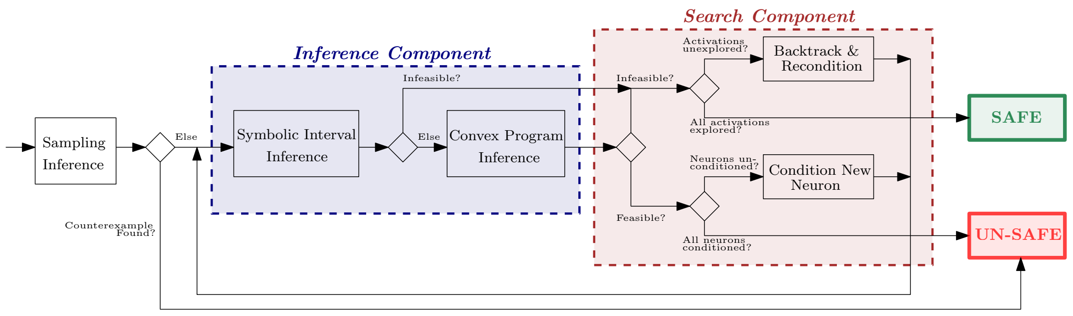


Figure 3.1: Block Diagram of the PeregriNN Algorithm

modified simplex algorithm.

3.2 Search and optimization

The general structure of PeregriNN is depicted in Fig. 3.1. Like other search and optimization based NN verifiers it has two main components: a *search component* and an *inference component*, and PeregriNN iterates back and forth between these these two components until termination. In particular, the search and inference components interact in the following way. The search component successively iterates over all possible on/off activations for each neuron; this is done by fixing these activations one neuron at a time, starting from the input layer and working towards the output layer. The process of fixing a neuron's activation is referred to as *conditioning its phase*: each neuron can be in either its active phase (operating linearly) or inactive phase (outputting zero). Thus, the search component provides the inference component a subset of neurons, each of which has been conditioned; the inference component then attempts to soundly reason about whether the remaining, unconditioned neurons can be operated in such a way as to violate the safety constraint. If the inference component soundly concludes safety for all possible activations of the remaining unconditioned neurons, then the search component backtracks, oppositely reconditioning one of the neurons that was already conditioned. Otherwise, if a sound safe conclusion is not made, then the search component uses information from the inference component to decide on a new neuron to condition, and the process repeats. The algorithm terminates if either a counterexample to safety is found, or else all possible neuron activations are considered without finding such a counterexample.

The convex program inference block is at the heart of the inference component and PeregriNN itself. In this block, PeregriNN, like other search and optimization solvers, uses a relaxed linear feasibility program where the output of each individual neuron is assigned a relaxation variable that is decoupled from the actual output of that neuron. Using the notation of section 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.

$$\begin{cases} y_i \geq 0,\ y_i \geq W_i y_{i-1} + b_i & \forall i = 1, \dots, n \\ y_0 \in P_{y_0},\ y_n \in P^{\mathsf{c}}_{y_n},\ \bigwedge_{\ell=1}^{m} h_\ell(y_0, y_n) \leq 0 \end{cases} \tag{3.1}$$

Importantly, if (3.1) is infeasible, then the original NN problem in (2.2) may be soundly concluded to be infeasible as well – and hence, safe. However, as described above, the primary function of the convex feasibility program is to use a set of conditioned neurons supplied by the search component in order to soundly reason about the remaining neurons. To do this, the conditioned neurons supplied by the search component are incorporated into the feasibility program (3.1) as *equality* constraints in the following way:

$$\text{Neuron } (y_i)_j \text{ ON:} \quad (y_i)_j = (W_i y_{i-1} + b_i)_j \wedge (y_i)_j \geq 0 \tag{3.2}$$

$$\text{Neuron } (y_i)_j \text{ OFF:} \quad (y_i)_j = 0 \wedge (W_i y_{i-1} + b_i)_j \leq 0. \tag{3.3}$$

Inferences created by the symbolic interval inference block using Symbolic Interval Analysis [22] are also incorporated using equality constraints like (3.2) and (3.3).

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

3.3 PeregriNN Enhancements

3.3.1 Sum-of-Slacks Penalty

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

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

3.3.2 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 [4].

3.3.3 Layer-wise-weighted Penalty

PeregriNN takes the "max-slack" neuron search priority one step further, though. Using techniques similar to those in [29], it is possible to show that there exists weights $q_1, \ldots, q_n$ such that solving (3.1) with the penalty

$$\min_{y_0,..,y_n} \sum_{i=0}^{n} \sum_{j=1}^{k_i} q_i s_{ij} \tag{3.4}$$

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 (3.4), 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).

As noted above, SMC [29] can be used to obtain layer-wise weights that guarantee concentration of slack in the earliest (shallowest) layer. However, these weights are often very large, since they depend on bounding the slack variables (most readily by over-approximation); the effect of this is possible computational instability in the convex program. Thus, as an *implementation* matter, we instead select these weights using a heuristic scheme characterized by two real-valued hyperparameters, λ_0 and γ. In particular, the weight of the i^{th} layer, q_i, is selected as $q_i = \lambda_0 \cdot \gamma^i$. In our experiments, we found the values $\lambda_0 = 10^{-7}$ and $\gamma = 10^3$ to effectively achieve the maximum slack concentration in the earliest layers.

3.3.4 Initial Counterexample Search by Sampling

Finally, PeregriNN extends a simple idea first introduced in [22] to rapidly identify counterexamples by means of sampling. The basic idea is to sample within a known region of the input to the NN (or the input to some deeper layer), and evaluate the NN (sub-NN) exactly on those samples in order to rapidly identify a counterexample; this approach help identify un-safe networks/properties early on. However, whereas [22] samples from within hyper-rectangle sets derived by symbolic interval analysis, PeregriNN uses the Volesti [30] Python library to uniformly sample points within the *polytopic* input constraint set, P_{y_0}, and thus applies to be more general input constraint sets in (2.2).

3.4 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 Section 2.2, 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 3.3. In particular, we compared the performance of a full implementation of PeregriNN – i.e. *exactly* as described in Section 3.3 – with implementations that are otherwise the same except for changing one and only one of the following: the penalty function used in the convex program inference block; the neuron prioritization used by the search component.

2. We compared PeregriNN against other state-of-the-art NN verifiers, both in terms of the time required to verify individual networks and properties and in terms of the number of properties proved with a common, fixed timeout.

Implementation. We implemented PeregriNN in Python, and used an off-the-shelf Gurobi 9.1 [31] convex optimizer for solving linear programs; the Volesti [30] Python interface was used to sample from the input polytope for the sampling inference block. For the other NN verifiers, we used publicly available implementations that were published by their creators (citations are included below). Each instance of of any verifier was run within its own single-core Virtual Box VM with 30 GB of memory; no more than 4 VMs were run concurrently on a host machine with 48 hyperthreaded cores and 256 GB of memory.

3.4.1 Adversarial Robustness Verification Task

Subsequent experiments used the testbench we describe in this section; it is largely identical to the PAT-FCN test in the VNN-COMP 2020 competition [32].

Table 3.1: 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>

Neural Networks.

We used three ReLU NNs to recognize digits using the standard MNIST training database; these NNs are exactly as in the PAT-FCN portion of [32]. The sizes of these fully-connected networks are described in Table 3.1. Each entry in the "Architecture" column of Table 3.1 is the number of number of neurons in a layer, from input layer on the left to output layer on the right.

Verification Properties.

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

Formally, let x' be a given image in category $t \in \{1, \dots, M\}$, 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 $m \in \{1, \dots, M\} \setminus \{t\}$ the set of inputs yielding classification of x' as m

$$\phi_m \triangleq \{x \mid x \in \mathbb{R}^{k_0},\ \|x - x'\|_\infty \leq \epsilon,\ z \in \mathbb{R}^{k_n},\ \max_{i=1,\dots,n} \mathcal{NN}(x)_i = \mathcal{NN}(x)_m\} \tag{3.5}$$

is empty. Note that each instance of (3.5) is compatible with the problem in (2.2).

Adversarial Robustness Verification Testbench

Our verification testbench was then constructed by selecting 50 test images from the MNIST test dataset; this set of test images includes the 25 used in the PAT-FCN portion of [32]. Each test instance was then a combination of one of those images, one of the networks from Table 3.1 and one the following two max-norm perturbations, $\epsilon = 0.02$ or $\epsilon = 0.05$; these perturbations are same ones used in PAT-FCN [32]. Thus, each verification test in our testbench can be identified by one of 300 tuples of the form: $(\mathit{net}, \mathit{image}, \mathit{perturb.}) \in \mathscr{TB} \triangleq \{\texttt{FC1}, \texttt{FC2}, \texttt{FC2}\} \times \{1, \ldots, 50\} \times \{0.02, 0.05\}$.

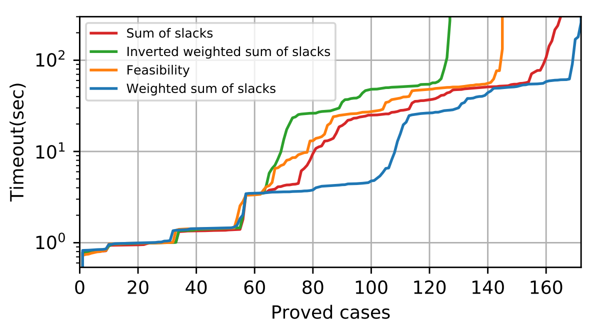
3.4.2 Ablation Experiments

In this series of experiments we evaluated the contribution that each of the primary PeregriNN enhancements made to its overall performance. This was done by comparing the full PeregriNN algorithm – as described in Section 3.3 – with altered versions that replace exactly one of those enhancements at a time.

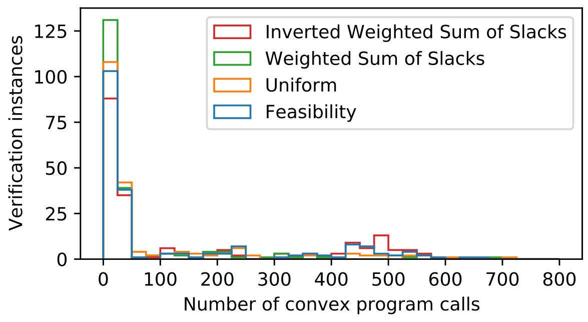
Note: removing core features of PeregriNN often resulted in much longer run times, so the experiments in this section use a testbench $\mathscr{TB}' \subset \mathscr{TB}$ that excludes all tests with one of the larger networks `FC2` or `FC3` *and* $\epsilon = 0.05$.

Penalty Function Ablation.

Our first ablation experiment evaluated the contribution of PeregriNN's unique penalty function features; see Section 3.3.1 and Section 3.3.3. In particular, we ran different variants



(a) Cactus plot; proved cases vs. timeout



(b) Histogram; number convex calls used

Figure 3.2: Performance of PeregriNN variants with different objective functions

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. [4] (i.e. simply using a constant penalty function of 1);
4. *"Inverted weighted sum of slacks"*: PeregriNN's own weighted sum of slacks penalty, except with the layer-wise weights applied in reverse order to force slack towards deeper layers rather than shallower ones (see also Section 3.3.3).

Fig. 3.2a shows a cactus plot of the number of proved cases vs. the timeout permitted to the algorithm: i.e. to prove at least a specified number of the test cases, each algorithm must have its timeout set at to the value of its curve in Fig. 3.2a. Fig. 3.2b 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. Fig. 3.2b plots the number of convex solver calls required for evenly spaced bins of convex solver calls.

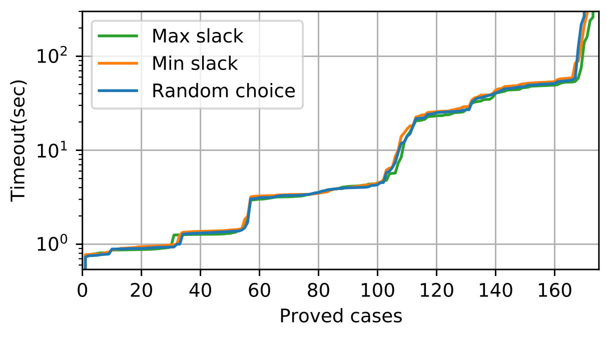
Conclusions: Fig. 3.2a demonstrates that PeregriNN's weighted sum of slacks has a clear benefit over both a uniformly weighted sum-of-slacks penalty and a plain feasibility convex program. For timeouts of longer than ≈ 1.2 seconds, PeregriNN overtakes the other two in terms of number of properties proved; even the uniform sum-of-slacks penalty considerably outperforms the feasibility convex program at similar timeouts. Note that *reversing* the layer-wise weights of PeregriNN's penalty function incurs a *performance hit*, especially for timeouts > 1.2 seconds. This suggests that driving slacks toward shallower layers, where the next neuron is conditioned, is the correct heuristic to apply. Fig. 3.2b also shows that going from feasibility to sum-of-slacks to weighted sum-of-slacks significantly reduces the number of test cases that require between 425 and 525 calls to the convex solver. This order of comparison shows a concomitant net influx of tests into the lowest bin of <25 convex calls; PeregriNN has the most test cases in this category, with ≈ 130 test cases proved in <25 convex solver calls.

Neuron Conditioning Priority Ablation.

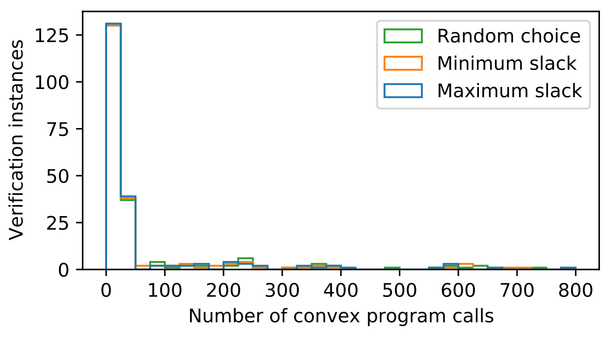
In the second ablation experiment, we evaluated the contribution of PeregriNN's maximum-slack neuron conditioning priority (see Section 3.3.2). To that end, we ran variants of PeregriNN with three different neuron conditioning priorities for the search component:

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

The performance of these algorithm variants is shown in Fig. 3.3a and Fig. 3.3b. As in the previous ablation experiment, Fig. 3.3a shows a cactus plot of the number of proved cases vs. the timeout, and Fig. 3.3b shows a histogram of the number of calls to the convex solver



(a) Cactus plot; proved cases vs. timeout



(b) Histogram; number convex calls used

Figure 3.3: Performance of PeregriNN variants with different conditioning priorities

required under each of the conditioning priorities.

Conclusions: Fig. 3.3a shows that PeregriNN's max-slack neuron priority allows it to prove slightly more properties than either a random neuron choice priority or the minimum-slack priority. The maximum slack priority also required the fewest total convex calls across all instances: it used 178 fewer than minimum slack and 686 fewer than a random choice. Thus, we conclude PeregriNN's max-slack heuristic slightly improves performance on this testbench.

3.4.3 Comparison with Other NN Verifiers

In this experiment, we evaluated PeregriNN with respect to a number of state-of-the-art NN verifiers on our adversarial robustness testbench, $\mathscr{TB}$. In particular, we ran the following tools on $\mathscr{TB}$: Venus [7]; Marabou [9]; Neurify [4]; and nnenum [8]. Venus was run with `st_ratio`$=0.4$, `depth_power`$=4$, `offline_deps` $=$ `True`, `online_deps` $=$ `True`, and `ideal_cuts` $=$ `True`; Marabou and Neurify were used with default parameters but `THREADS` $= 1$; and nnenum had `ADVERSARIAL_SEARCH` turned off. Each algorithm had its own one-core VM.

Fig. 3.4 contains a cactus plot showing the results for each of these algorithms, including PeregriNN. For a given number of test cases to be proved, Fig. 3.4 depicts the corresponding timeout required for each of the algorithm to prove that many cases. Of all the algorithms, PeregriNN was able to prove the most properties within the timeout limit of 600 seconds: PeregriNN was able to prove 190 properties; it was followed by nnenum, which proved 172; Venus, which proved 159; Neurify, which proved 149; and Marabou, which proved 125. Marabou consistently performed the worst, proving fewer cases than any other algorithm at every timeout. By contrast, Neurify was able to prove significantly more test cases than any other algorithm for extremely short timeouts, but it failed to prove more than 150 out of 300 test cases across the whole experiment. nnenum performed worse than Neurify on the way to proving 150 test cases, but it fared significantly better than either PeregriNN or Venus, which had more or less similar performance below this threshold. However, after ≈150 test cases, PeregriNN significantly outperformed all other algorithms: as the timeout was increased, PeregriNN proved additional properties at a rate significantly outpacing its closest competitor in this regime, nnenum. We further note that all algorithms proved a mixture of SAT and UNSAT properties.

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

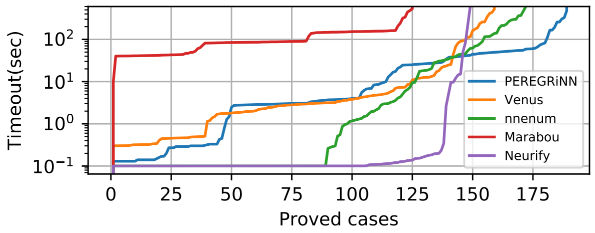


Figure 3.4: Cactus plot of various solvers on 300-case testbench, $\mathscr{TB}$

Chapter 4

BERN-NN: Tight Bound Propagation For Neural Networks Using Bernstein Polynomial Interval Arithmetic

In this paper, we present BERN-NN as an efficient tool to perform bound propagation of Neural Networks (NNs). Bound propagation is a critical step in wide range of NN model checkers and reachability analysis tools. Given a bounded input set, bound propagation algorithms aim to compute tight bounds on the output of the NN. So far, linear and convex optimizations have been used to perform bound propagation. Since neural networks are highly non-convex, state-of-the-art bound propagation techniques suffer from introducing large errors. To circumvent such drawback, BERN-NN approximates the bounds of each neuron using a class of polynomials called Bernstein polynomials. Bernstein polynomials enjoy several interesting properties that allow BERN-NN to obtain tighter bounds compared to those relying on linear and convex approximations. BERN-NN is efficiently parallelized on graphic processing units (GPUs). Extensive numerical results show that bounds obtained by BERN-NN are orders of magnitude tighter than those obtained by state-of-the-art verifiers such as linear programming and linear interval arithmetic. Moreoveer, BERN-NN is both faster and produces tighter outputs compared to convex programming approaches like alpha-CROWN.

4.1 Introduction

Neural Networks (NNs) have become an increasingly central component of modern, safety-critical, cyber-physical systems like autonomous driving, autonomous decision-making in smart cities, and even autonomous landing in avionic applications. Thus, there is an increasing need to verify the safety and correctness [33, 34, 35] of NNs when they are used to control physical systems.

The problem of NN Verification has been well studied in literature [36]. Most NN verifiers rely mainly on either using linear relaxation and optimization [4, 25, 26, 37, 38, 39] to falsify a given property or prove its satisfaction, or reachability analysis to compute an over-approximation of the output set. The latter is specifically important for control applications where the property of interest is defined over a time horizon. Both techniques rely on overapproximation, hence, having tight output bounds is at the core of NN verification as it allows reasoning about NN properties in an efficient manner. For example, model checking the robustness of NNs against adversarial perturbations can be done by simply comparing the tight bounds of the outputs of the network. Moreover, networks used in control applications often involve multi-step reachability, and hence computing tight bounds is crucial to harness the accumulation of the error and hence be able to efficiently reason about the safety of the system.

Due to the non-convexity and non-linearity of NNs, the problem of finding the exact bounds of NN outputs is NP-hard[10]. Different tools have been proposed to find tight overapproximations of NN outputs. MILP-based methods [40, 16, 17, 12, 13, 15, 11, 14] encode the non-linear activations as linear and integer constraints. Reachability methods [23, 24, 19, 22, 21, 20, 18] use layer-by-layer reachability analysis (exact or overapproximation) of the network. Most of these methods either rely on convex *linear relaxation* of the non-linear activation functions to overapproximate the output of the NN, or try to find the exact bounds which are often intractable.

In this work, we explore using polynomials to approximate non-linear activations (e.g. ReLU). More specifically, we approximate non-linear activations using Bernstein polynomials which are constructed as a linear combination of the Bernstein basis polynomials [41]. The use of Bernstein polynomials is motivated by two reasons. First, based on the Stone-Weierstrass approximation theorem [42], Bernstein polynomials can uniformly approximate continuous activation functions. Second and most importantly, bounding a Bernstein polynomial is computationally cheap based on the interesting properties of Bernstein polynomials discussed in section 4.3. The goal of using higher-order polynomials versus linear relaxation is to get tight bounds on NNs which is crucial for verifying a large class of formal properties. This idea of using polynomials has inspired other researchers [43, 44, 1], however, the proposed tools suffer from scalability issues.

Our main contributions can be summarized as follows:

- We propose a tool that uses Bernstein polynomials to approximate ReLU activations and hence compute tighter NN bounds than state-of-the-art.
- The tool is designed with scalability in mind; hence, the entire operations can be accelerated using GPUs.
- We show that by using the proposed approximation, we are able to compute tighter output sets than alpha-Crown (winner of VNN22' competition[45] for Formal Verification of NNs) and other state-of-the-art bounding methods. For instance, BERN-NN approximations are twice reduced compared to alpha-Crown for actual NN's controllers. Moreover, Numerical results showed that Bern-NN can process neural networks with

more than 1000 neurons in less than 2 minutes

4.2 Problem Formulation

4.2.1 Notation:

General notation: We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ the vector of n real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n(\underline{d}, \overline{d}) = \big[\underline{d}_1, \overline{d}_1\big] \times \cdots \times \big[\underline{d}_n, \overline{d}_n\big] \subset \mathbb{R}^n$ the n-dimensional hyperrectangle where $\underline{d} = (\underline{d}_1, \cdots, \underline{d}_n)$ and $\overline{d} = \big(\overline{d}_1, \cdots, \overline{d}_n\big)$ are the lower and upper bounds of the hyperrectangle, respectively. We denote by x^T and A^T the transpose operation of the vector x and the matrix A. We denote by 0_n a vector that contains n zero values and by $0_{n\times m}$ the matrix of shape $n \times m$ that contains zeros. Finally, $A * B$ stands for the element-wise product between the multi-dimensional tensors A and B, and $A \otimes B$ stands for the Kronecker product between the matrices A and B.

Notation pertaining to multivariate polynomials: For a real-valued vector $x = \big(x_1, x_2, \cdots, x_n\big) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$, we denote by $x^K \in \mathbb{R}$ the scalar $x^K = x_1^{k_1} \times \ldots \times x_n^{k_n}$. Given two multi-indices $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this paper:

$$\begin{aligned}
&K + L = (k_1 + l_1, \cdots, k_n + l_n)\,,\\
&\binom{L}{K} = \binom{l_1}{k_1} \times \cdots \times \binom{l_n}{k_n},\\
&\sum_{K \le L} = \sum_{k_1 \le l_1} \cdots \sum_{k_n \le l_n}
\end{aligned}$$

Finally, a real-valued multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ is defined as:

$$\begin{aligned} p(x_1, \ldots, x_n) &= \sum_{k_1=0}^{l_1} \sum_{k_2=0}^{l_2} \cdots \sum_{k_n=0}^{l_n} a_{(k_1,\ldots,k_n)} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} \\ &= \sum_{K \leq L} a_K x^K, \end{aligned}$$

where $L = (l_1, l_2, \ldots, l_n)$ is the maximum degree of x_i for all $i = 1, \ldots, n$.

Notation pertaining to neural networks: In this paper, we consider H-layer, feed-forward, ReLU-based neural networks $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ defined as:

$$\begin{aligned} \mathcal{NN}(x) &= W^{(H)} z^{(H-1)} + b^{(H)} \\ z^{(H-1)} &= \sigma\left(W^{(H-1)} z^{(H-2)} + b^{(H-1)}\right) \\ &\vdots \\ z^{(1)} &= \sigma\left(W^{(1)} x + b^{(1)}\right) \end{aligned}$$

where σ is the ReLU activation function (i.e., $\sigma(z) = \max(0, z)$) that operates element-wise, $W^{(i)} \in \mathbb{R}^{h_i \times h_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{h_i}$ with $i \in \{1, \cdots, H\}$ are the weights and the biases of the network. For simplicity of notation, we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the pre-activation (input) and the post-activation (output) of the j-th neuron in the i-th layer.

4.2.2 Main Problem:

In this paper, we seek to find polynomials that upper and lower approximate the NN's outputs $\mathcal{NN}(x)$ whenever the NN's input x is confined within a pre-defined hypercube, i.e. $x \in I_n(\underline{d}, \overline{d})$.

Problem 1. *Given a neural network $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ and an input domain hypercube $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$. Find lower and upper approximate polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right)$,*

$\dots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$, such that:

$$\begin{aligned}
&\underline{p}_{\mathcal{NN},1}(x) \leq \mathcal{NN}_1(x) \leq \overline{p}_{\mathcal{NN},1}(x) \\
&\qquad\qquad\qquad \vdots \\
&\underline{p}_{\mathcal{NN},o}(x) \leq \mathcal{NN}_o(x) \leq \overline{p}_{\mathcal{NN},o}(x),
\end{aligned}$$

where with some abuse of notation, we use $\mathcal{NN}_i(x)$ to denote the ith output of the neural network $\mathcal{NN}$.

Note that the lower/upper bound polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right), \dots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$ depend on the input domain I_n. That is, for each value of I_n, we need to find different lower/upper bound polynomials. However, for the sake of simplicity of notation, we drop the dependency on I_n.

4.3 Tight bounds of ReLU Functions Using Bernstein Polynomials

To solve Problem 1, we rely on a class of polynomials called Bernstein polynomials which are defined as follows:

Definition 4.3.1. *(Bernstein Polynomials) Given a continuous function $g : \mathbb{R}^n \to \mathbb{R}$, an input domain (hypercube) $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$, and a multi-index $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, the polynomial:*

$$B_{g,L}(x) = \sum_{K \leq L} b^g_{K,L} Ber_{K,L}(x), \tag{4.1}$$

$$Ber_{K,L}(x) = \binom{L}{K} \frac{(x - \underline{d})^K \left(\overline{d} - x\right)^{L-K}}{\left(\overline{d} - \underline{d}\right)^L}, \tag{4.2}$$

$$b^g_{K,L} = g\bigg(\left(\overline{d}_1 - \underline{d}_1\right) \frac{k_1}{l_1} + \underline{d}_1, \cdots, \left(\overline{d}_n - \underline{d}_n\right) \frac{k_n}{l_n} + \underline{d}_n \bigg), \tag{4.3}$$

is called the Lth order Bernstein polynomial of g, where $Ber_{K,L}(x)$ and $b^g_{K,L}$ are called the Bernstein basis and Bernstein coefficients of g, respectively.

Bernstein polynomials are known to be capable of approximating any continuous function. That is, Bernstein approximation has an advantage compared to Taylor approximation because the latter relies on the function being differentiable. In this case, Taylor model can not approximate ReLU activation functions because they are not differentiable which makes Bernstein polynomials a good option to approximate ReLU functions. Bernstein polynomials have an interesting and useful property called *range enclosing property* which is defined as follows:

Definition 4.3.2. *(Range Enclosing Property [46]) Given a multi-dimensional polynomial $p(x)$ of order L that it defined over the region $I_n\left(\underline{d}, \overline{d}\right)$ with its Bernstein polynomial $B_{p,L} = \sum\limits_{K \leq L} b^p_{K,L}(x)\, Ber_{K,L}(x)$. The following holds for all $x \in I_n\left(\underline{d}, \overline{d}\right)$:*

$$\min_{K \leq L} b^p_{K,L} \;\leq\; p(x) \;\leq\; \max_{K \leq L} b^p_{K,L}. \tag{4.4}$$

The range enclosing property states that the minimum (maximum) over all the Bernstein coefficients is a lower (upper) bound for the polynomial p over the region $I_n\left(\underline{d}, \overline{d}\right)$. These bounds provided by the Bernstein coefficients are generally tighter than those given by interval arithmetic and many centered forms [47]. Note that the range enclosing property applies only when the Bernstein polynomial is used to approximate other polynomials p and other continuous functions g. Nevertheless, as we show in Section 4, these bounds will be helpful to provide tight bounds on the polynomials used to over/under approximate the individual neurons and hence obtain tight polynomial bounds on the NN's outputs.

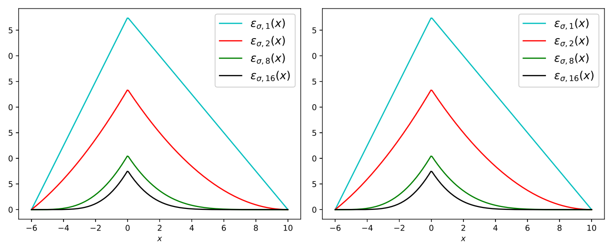


Figure 4.1: **(Top)** Bernstein polynomial approximations of ReLU activation for different approximation's order $L \in \{1, 2, 8, 16\}$, in the interval $I_1(-6, 10) = \big[-6, 10\big]$. **(Bottom)** Bernstein polynomial approximations of ReLU and their associated approximation errors for different approximation's order $L \in \{1, 2, 8, 16\}$ in the interval $I_1(-6, 10) = \big[-6, 10\big]$.

4.3.1 Over-Approximating ReLU functions using Bernstein Polynomials

We now study how to use Bernstein polynomials to over-approximate the ReLU function $\sigma : \mathbb{R} \to \mathbb{R}$ defined as $\sigma(x) = \max(0, x)$. While Bernstein polynomials can approximate any continuous function g, there is no guarantee that this Bernstein approximation is either over-approximation or under-approximation. The next result establishes an order between the ReLU function σ and its Bernstein approximation.

Proposition 4.3.3. *Given an interval* $I_1\left(\underline{d}, \overline{d}\right) = \left[\underline{d}, \overline{d}\right]$*, where* $0 \in \left[\underline{d}, \overline{d}\right]$ *and any approximation order* $L \geq 1$*. The following holds for all* $x \in I_1$*:*

$$\sigma(x) \leq B_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x).$$

Proof. This follows directly by substituting the function σ in the definition of Bernstein polynomials (4.1)-(4.3). □

In other words, Proposition 4.3.3 states that the Bernstein polynomial of σ is a guaranteed over-approximation of σ. This even holds *for any approximation order* L. Moreover, since the approximation error between a function g and its Bernstein approximation $B_{g,L}$ is known to decrease as L increases [48]. Then another consequence of Proposition 4.3.3 is that Bernstein polynomials produce a tighter over-approximation for ReLU functions as L increases.

Figure 4.1 emphasizes these conclusions pictorially where we show the Bernstein polynomials of σ with orders $L = 1, 2, 8, 16$. As shown in Figure 4.1 (Left), the Bernstein polynomials $B_{\sigma,L}(x)$ for $L =\in \{1, 2, 8, 16\}$ over-approximate the ReLU activation function over the entire input range. Furthermore, the over-approximation gets tighter to the actual ReLU by increasing the approximation order L. We note that using $L = 1$, the resulting Bernstein polynomial produces the well-studied linear convexification of the ReLU function which is used in state-of-the-art algorithms for bounding neural networks including Symbolic Interval Arithmetic (SIA) [4] and alpha-CROWN [49]. In other words, Bernstein polynomials can be seen as a generalization of these techniques.

4.3.2 Under-approximating ReLU functions using Bernstein polynomials

In addition to the over-approximation of the ReLU function σ, it is essential to establish a Bernstein under-approximation of σ which is captured by the following result.

Proposition 4.3.4. *Given an interval $I_1\left(\underline{d}, \overline{d}\right) = \left[\underline{d}, \overline{d}\right]$, where $0 \in \left[\underline{d}, \overline{d}\right]$, then the following holds for all $x \in I_1$:*

$$\underline{B}_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x) - \overline{B}_{\sigma,L}(0) \leq \sigma(x).$$

Proof. To prove the result, we define the approximation error $\epsilon_{\sigma,L}$ as:

$$\epsilon_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x) - \sigma(x).$$

We bound the maximum estimation error satisfies as follows:

$$\max_{x\in[\underline{d},\overline{d}]} \epsilon_{\sigma,L}(x) = \max_{x\in[\underline{d},\overline{d}]} \left(\overline{B}_{\sigma,L}(x) - \sigma(x)\right) \tag{4.5}$$

$$\overset{(a)}{=} \max_{x\in[\underline{d},0]} \overline{B}_{\sigma,L}(x) \tag{4.6}$$

$$\overset{(b)}{=} \overline{B}_{\sigma,L}(0) \tag{4.7}$$

where (a) follows from the fact that $\sigma(x) = 0$ for $x \in [\underline{d}, 0]$ and $\sigma(x) \geq 0$ for $x \in [0, \overline{d}]$ and hence the maximum of the equation is attained whenever $\sigma(x) = 0$. Equation (b) holds from the monotnicity of $\overline{B}_{\sigma,L}(x)$ when $x \in [\underline{d}, 0]$—the monotnicity follows directly from the definition of $\overline{B}_{\sigma,L}(x)$—and hence the maximum is attained when $x = 0$. It follows from the definition of $\epsilon_{\sigma,L}(x)$ that:

$$\sigma(x) = \overline{B}_{\sigma,L}(x) - \epsilon_{\sigma,L}(x) \geq \overline{B}_{\sigma,L}(x) - \max_{x\in[\underline{d},\overline{d}]} \epsilon_{\sigma,L}(x)$$
$$= \overline{B}_{\sigma,L}(x) - \overline{B}_{\sigma,L}(0) = \underline{B}_{\sigma,L}$$

which concludes the proof. □

Proposition 4.3.4 shows that the maximum error between the Bernstein over-approximation polynomial $\overline{B}_{\sigma,L}$ and the ReLU activation function σ is equal to the value of the Bernstein polynomial at 0, i.e., $\overline{B}_{\sigma,L}(0)$. This result has a direct consequence on the efficiency of our tool. It is enough to propagate over-approximation of the ReLU function and one can get an under-approximation directly by shifting the over-approximation polynomial.

Figure 4.1 (Right) emphasizes this fact pictorially. As it is shown in the figure, the maximum error $\epsilon_{\sigma,L}(x) = \overline{B}_{\sigma,L} - \sigma(x)$ is reached at $x = 0$ and is equal to $\overline{B}_{\sigma,L}\,(0)$.

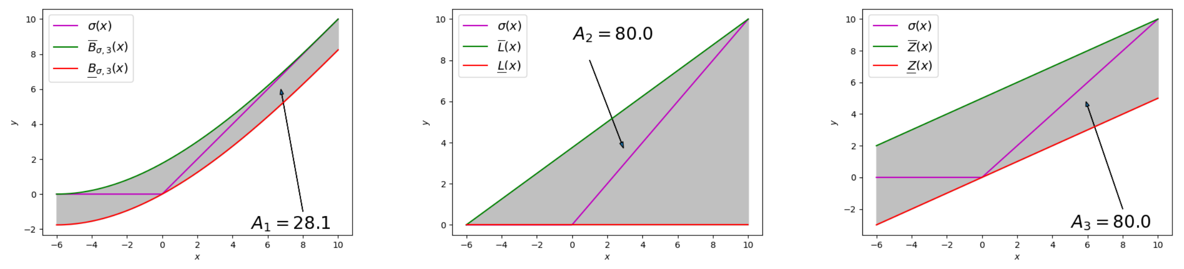


Figure 4.2: Illustrations of the over-approximation sets (shaded in gray) of the ReLU activation functions in the interval $[-6, 10]$ using different approaches: Bernstein approach (Left), triangulation approach (Center), and zonotope approach (Right). Green (Red)-colored curves represent the over-approximation (under-approximation) curves for every approach, respectively. A_i, $i \in \{1, 2, 3\}$, represents the over-approximation set's area for every approach.

Table 4.1: The area of the over-approximation set of the ReLU activation functions in the interval $[-6, 10]$ using different Bernstein approach for different approximation order L.

Approx. Method	Triangulation	Zonotope	Bernstein poly		
			$L = 2$	$L = 3$	$L = 8$
error	80.0	80.0	37.5	28.1	16.9

4.3.3 Comparing Bernstein Approximation Against Widely Used Approximations

The major advantage of using Bernstein polynomials is that they produce a tighter approximation for the response function of ReLU compared to the other state-of-the-art techniques. In particular, existing techniques focus on "convexifying" the response of the ReLU function through linear approximation/triangulation (Figure 4.2-middle) or zonotopes (Figure 4.2-right). Unlike these techniques, Bernstein polynomials lead to tighter non-convex approximations of the non-convex ReLU function. While it is direct to obtain a closed-form expression for the difference in the approximation error between Bernstein polynomials and triangulation/zonotope approximations, we, instead support our conclusions with the numerical example shown in Table 4.1 and highlighted in Figure 4.2. In this example, we compute the approximation error (highlighted in gray) which captures the quality of the over and under-approximations. As captured by this example, it is direct to see that Bernstein polynomials lead to tighter approximation. Moreover, such approximation gets tighter as the approximation order L increases.

4.4 Encoding Basic Bernstein Polynomial Operations Using Multi-Dimensional Tensors

While using Bernstein polynomials to approximate individual ReLU functions provides tighter bounds compared to other techniques, computing Bernstein polynomials via its definition in (4.1)-(4.3) is time-consuming. That is why state-of-the-art techniques have focused on linear (or convex) relaxations to obtain tractable computations. Nevertheless, in this section, we show that technological advances in Graphics Processing Units (GPUs) can be used to perform all the required operations to efficiently compute Bernstein polynomial approximations of individual neurons along with propagating these polynomials from one layer of the neural network to the next layer. Our main contribution of this section is to encode all necessary operations over Bernstein polynomials into additions and multiplication of multi-dimensional tensors that can be easily performed using GPUs.

4.4.1 Multi-dimensional tensor representation of Bernstein polynomials

We represent the Bernstein polynomial:

$$B_{g,L}\left(x\right) = \sum_{K \leq L} b^{g}_{K,L} Ber_{K,L}\left(x\right)$$

of function g and order L as a multi-dimensional tensor $\text{Ten}(B_{g,L})$ of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of $\text{Ten}(B_{g,L})$ is equal to the Bernstein coefficient $b^g_{K,L}$. The multi-dimensional tensor $\text{Ten}(B_{g,L})$ represent all the Bernstein coefficients $b^g_{K,L}$ of g, $\forall K \leq L$.

Example 1. *Consider the two-dimensional Bernstein polynomial:*

$$B_{g,L}(x_1, x_2) = \sum_{k_1=0}^{2} \sum_{k_2=0}^{3} b^g_{(k_1,k_2),L} Ber_{(k_1,k_2),L}(x_1, x_2)$$

with orders $L = (2, 3)$. Its two-dimensional tensor representation is written as follows:

$$Ten(B_{g,L}) = \begin{bmatrix} b^g_{(0,0),L} & b^g_{(0,1),L} & b^g_{(0,2),L} & b^g_{(0,3),L} \\ b^g_{(1,0),L} & b^g_{(1,1),L} & b^g_{(1,2),L} & b^g_{(1,3),L} \\ b^g_{(2,0),L} & b^g_{(2,1),L} & b^g_{(2,2),L} & b^g_{(2,3),L} \end{bmatrix}. \tag{4.8}$$

In a similar manner, we represent a multi-dimensional polynomial of order L written in the power series form $p(x) = \sum\limits_{K \leq L} a_K x^K$ as a multi-dimensional tensor $\text{Ten}(p)$ of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of $\text{Ten}(p)$ is equal to the coefficient a_K.

4.4.2 Multiplication of two multi-variate Bernstein polynomials

Multiplying two polynomials represented in the power series form on GPUs has been widely studied in the literature. Unlike power series, multiplying two Bernstein polynomials need extra handling [50]. In this subsection, we propose how to encode the multiplication of Bernstein polynomials using GPU implementations that were designed for power-series polynomials.

Given two multivariate polynomials written in a power series form, $p_1 = \sum\limits_{K \leq L_1} a^1_K x^K$ and $p_2 = \sum\limits_{K \leq L_2} a^2_K x^K$, and their tensor representation, $\text{Ten}(p_1)$ and $\text{Ten}(p_2)$, we use an efficient algorithm [51] that performs multivariate polynomial multiplications. We denote by

Prod (Ten (p_1) , Ten (p_2)) the tensor resulting from such multiplication, i.e.:

$$\text{Ten}\,(p_1 p_2) = \mathbf{Prod}\,(\text{Ten}\,(p_1)\,,\text{Ten}\,(p_2))\,.$$

Applying power-series-based algorithms to multiply two Bernstein polynomials produce incorrect results. Different algorithms were proposed for the case when the Bernstein polynomials are functions of one variable x_1 [52] and two variables x_1, x_2 [50]. Below, we generalize the procedure in [50] to account for Bernstein polynomials in n variables.

Proposition 4.4.1. *Given two multivariate Bernstein polynomials*

$$B_{g_1,L_1}(x) = \sum_{K \le L_1} b^{g_1}_{K,L_1} Ber_{K,L_1}(x)$$

and

$$B_{g_2,L_2}(x) = \sum_{K \le L_2} b^{g_2}_{K,L_2} Ber_{K,L}(x)$$

. The tensor representation of the Bernstein polynomial $B_{g_1,L_1}(x)B_{g_2,L_2}(x)$ can be computed as follows:

$$Ten\left(\tilde{B}_{g_1,L_1}\right) = Ten\,(B_{g_1,L_1}) * C_{L_1}, \tag{4.9}$$

$$Ten\left(\tilde{B}_{g_2,L_2}\right) = Ten\,(B_{g_2,L_2}) * C_{L_2}, \tag{4.10}$$

$$Ten\,(B_{g_1,L_1}B_{g_2,L_2}) = \frac{1}{C_{L_1+L_2}} * \boldsymbol{Prod}\left(Ten\left(\tilde{B}_{g_1,L_1}\right), Ten\left(\tilde{B}_{g_2,L_2}\right)\right). \tag{4.11}$$

where C_L is the multi-dimensional binomial tensor where its Kth component is equal to $\binom{L}{K}$, i.e, $(C_L)_K = \binom{L}{K}$. With some abuse of notation, we use $1/C_L$ to denote the multi-dimensional binomial tensor where its Kth component is equal to $\frac{1}{\binom{L}{K}}$.

The proof of Proposition 4.4.1 generalizes the argument in [50] to multi-dimensional inputs and is omitted for brevity. The Bernstein polynomials in (4.9) and (4.10) are called scaled

Bernstein polynomials [50] and enjoy the fact that their multiplication corresponds to the multiplication of power series polynomials. Hence we can use the power series **Prod** in (4.11) followed by the element-wise multiplication with the $\frac{1}{C_{L_1+L_2}}$ tensor to remove the effect of the scaling. Recall that we use $A * B$ to denote the element-wise multiplication between the tensors A and B, which can also be carried over using GPUs efficiently which renders all the steps in equations (4.9)-(4.11) to be efficiently implementable on GPUs. We refer to the equations (4.9)-(4.11) as $\textbf{Prod\_Bern}(B_{g_1,L_1}, B_{g_2,L_2})$.

Using **Prod_Bern**, one can compute the tensor corresponding to raising the function g to power i, where $i \in \mathbb{N}$ is an integer power, denoted by $\text{Ten}(B_{g^i,L})$ by applying the **Prod_Bern** procedure i times. We refer to this procedure as $\textbf{Pow\_Bern}(\text{Ten}(B_{g,L}), i)$.

4.4.3 Addition between two Bernstein polynomials

The authors in [52] studied how to add two Bernstein polynomials. However, their study is restricted to one-dimensional polynomials which are defined over the unity interval $I_1(x) = [0,1]$. We extend the argument to the general case with n inputs and any interval $I_n(\underline{d}, \overline{d})$ using the following result.

Proposition 4.4.2. *Given two Bernstein polynomials $B_{g_1,L_1}(x)$ and $B_{g_2,L_2}(x)$ with two different orders $L_1 = (l_1^1, \cdots, l_n^1)$ and $L_2 = (l_1^2, \cdots, l_n^2)$. Define $L_{sum} = \max(L_1, L_2)$, where the* max *operator is applied element-wise. The tensor representation of $B_{g_1+g_2,L_{sum}}$ can be computed as:*

$$L_{sum} = (\max(l_1^1, l_1^2), \ldots, \max(l_n^1, l_n^2)) \tag{4.12}$$

$$Ten\,(B_{g_1,L_{sum}}) = \textbf{\textit{Prod\_Bern}}\,(Ten\,(B_{g_1,L_1}), 1_{L_{sum}-L_1+1}) \tag{4.13}$$

$$Ten\,(B_{g_2,L_{sum}}) = \textbf{\textit{Prod\_Bern}}\,(Ten\,(B_{g_2,L_2}), 1_{L_{sum}-L_2+1}) \tag{4.14}$$

$$Ten\,(B_{g_1+g_2,L_{sum}}) = Ten\,(B_{g_1,L_{sum}}) + Ten\,(B_{g_2,L_{sum}}) \tag{4.15}$$

where 1_{L_e-L+1} is a multi-dimensional tensor of a shape $L_e - L + 1$ that contains just ones.

The proof of Proposition 4.4.2 generalizes the argument in [52] and is omitted for brevity. The operation in (4.13) and (4.14) is referred to as *degree elevation* in which we change the dimensions of the tensors ... Once both tensors are of the same dimension, we can add them element-wise. We denote by **Sum_Bern** the procedure defined by (4.12)-(4.15). Again, we note that all the operations in the **Sum_Bern** entail tensor element-wise multiplication and addition

4.5 BERN-NN algorithm

In this section, we provide the details of our tool, named BERN-NN. BERN-NN uses the tensor encoding discussed in Section 4 to propagate Bernstein polynomials that over- and under-approximate the different neurons in the network until over- and under-approximation polynomials for the final output of the network are computed.

4.5.1 Propagating bounds through single neuron

We first discuss how to propagate over- and under-approximations through neurons. Recall our notation that we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the input and output of the j-th neuron in the i-th layer. For ease of notation, we drop the i and j from the notation in this subsection.

Assume that we already computed the over- and under-approximations for the input of one of the hidden neurons, denoted by $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$, respectively. The objective is to compute the over- and under-approximations for the output of such a neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$, respectively. We proceed as follows.

Step 1: Compute input bounds for the neuron. Recall that the Bernstein coefficients depend on the input bounds of the function it aims to approximate. Since our aim is to approximate the scalar ReLU function of a neuron, we start by computing the bounds on the input to that neuron as follows:

$$lo = \min_{x \in I_n(\underline{d}, \overline{d})} \underline{B}_{\hat{z}, L_{\hat{z}}}(x), \qquad hi = \max_{x \in I_n(\underline{d}, \overline{d})} \overline{B}_{\hat{z}, L_{\hat{z}}}(x) \tag{4.16}$$

Thanks to the enclosure property (4.4), we can solve the optimization problems (4.16) by finding the minimum and the maximum coefficients of $\underline{B}_{\hat{z}, L_{\hat{z}}}$ and $\overline{B}_{\hat{z}, L_{\hat{z}}}$.

Step 2: Compute the polynomials $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$ that approximate the ReLU function. Given a user-defined approximation order L, the next step is to compute the Bernstein polynomials that over- and under-approximate the ReLU activation function σ denoted by $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$. These polynomials can be computed using the knowledge of lo and hi along with the definition of the Bernstein polynomial in (4.3). To facilitate the computations of the next step, we need to convert these polynomials into the corresponding power series form. This can be done by following the procedure in [53] to obtain:

$$p_{\overline{B}_{\sigma,L}}(x) = \sum_{K \le L} a_K^{\overline{B}_{\sigma,L}} x^K, \qquad p_{\underline{B}_{\sigma,L}}(x) = \sum_{K \le L} a_K^{\underline{B}_{\sigma,L}} x^K \tag{4.17}$$

Step 3: Propagate the bounds through the decomposition of polynomials. First, note that the following holds due to the monotonicity of the ReLU function σ and the fact that $z = \sigma(\hat{z})$:

$$\underline{B}_{\hat{z}, L_{\hat{z}}}(x) \le \hat{z}(x) \le \overline{B}_{\hat{z}, L_{\hat{z}}}(x) \Rightarrow \tag{4.18}$$

$$\underbrace{\sigma\left(\underline{B}_{\hat{z}, L_{\hat{z}}}(x)\right)}_{\underline{B}_{z, L_z}(x)} \le \underbrace{\sigma\Big(\hat{z}(x)\Big)}_{z(x)} \le \underbrace{\sigma\left(\overline{B}_{\hat{z}, L_{\hat{z}}}(x)\right)}_{\overline{B}_{z, L_z}(x)} \tag{4.19}$$

In other words, the post-bounds of the neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$ can be computed by composing the function σ with the under- and over-approximations of the neuron input $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$. Indeed such composition is hard to compute due to the nonlinearity in σ. Instead, we perform such composition with the over- and under-approximations of σ, $p_{\overline{B}_{\sigma,L}}$ and $p_{\underline{B}_{\sigma,L}}$, computed in Step 2, as:

$$\underline{B}_{z,L_z}(x) = \sum_{K \leq L} a_K^{\underline{B}_{\sigma,L}} \left(\underline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{4.20}$$

$$\overline{B}_{z,L_z}(x) = \sum_{K \leq L} a_K^{\overline{B}_{\sigma,L}} \left(\overline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{4.21}$$

Given the tensor representation $Ten(\underline{B}_{\hat{z},L_{\hat{z}}})$ and $Ten(\overline{B}_{\hat{z},L_{\hat{z}}})$, we can use the **Pow_Bern** and **Sum_Bern** procedures to perform the computations in (4.20) and (4.21) to calculate $Ten(\underline{B}_{z,L_z})$ and $Ten(\overline{B}_{z,L_z})$ with $L_z = L_{\hat{z}} * L$.

4.5.2 Propagating the bounds through one layer

Next, we discuss how to propagate the under- and over-approximation polynomials of the outputs of the $i-1$ layer denoted by $\underline{B}_{z_j^{(i-1)},\, L_z}, \overline{B}_{z_j^{(i-1)},\, L_z}, j \in \{1, \ldots, h_{i-1}\}$ to compute under- and over-approximation of the inputs of the neurons in the ith layer $\underline{B}_{\hat{z}_m^{(i)},\, L_{\hat{z}}}, \overline{B}_{z_m^{(i)},\, L_{\hat{z}}}, m \in \{1, \ldots, h_i\}$ of the neural network. Such bound propagation entails composing the under- and over-approximation polynomials $\underline{B}_{z_j^{(i-1)},\, L_z}, \overline{B}_{z_j^{(i-1)},\, L_z}$ with the weights of the ith layer of the neural network $W^{(i)}, b^{(i)}$. To that end, we define the set of positive and negative weights as:

$$W_+^{(i)} = \max\left(W^{(i)}, 0_{i\times(i-1)}\right) \qquad W_-^{(i)} = \min\left(W^{(i)}, 0_{i\times(i-1)}\right).$$

Similarly, for the outputs of the $i-1$ layer of the network, we define the vector of over-approximation polynomials and vector of the under-approximation polynomials as:

$$\overline{B}_{z^{(i-1)},\,L_z} = \left[\overline{B}_{z_1^{(i-1)},L_z} \ldots, \overline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$
$$\underline{B}_{z^{(i-1)},\,L_z} = \left[\underline{B}_{z_1^{(i-1)},L_z} \ldots, \underline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$

and for the inputs of the $i$the layer as:

$$\overline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\overline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \ldots, \overline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$
$$\underline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\underline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \ldots, \underline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$

Hence, the over- and under-approximations of the inputs of the ith layer can be efficiently computed as:

$$Ten\left(\overline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) * W_+^{(i)} + Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) * W_-^{(i)} + b^{(i)} \tag{4.22}$$
$$Ten\left(\underline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) * W_+^{(i)} + Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) * W_-^{(i)} + b^{(i)} \tag{4.23}$$

4.5.3 Mechanism of BERN-NN Polynomial Interval Arithmetic

We finally describe the proposed BERN-NN Polynomial Interval Arithmetic algorithm, depicted in Figure 4.3. For a neural network with n inputs $x_1, \ldots, x_n$, we initialize an over- and under-approximation Bernstein polynomials for each of the inputs, i.e.,:

$$\overline{B}_{z_i^{(0)},1} = \underline{B}_{z_i^{(0)},1} = \overline{B}_{z_i^{(0)},1} \qquad i \in \{1, \ldots, n\}.$$

Note that in the equation above, we used $z_i^{(0)}$ as a replacement of x_i to unify the notation with the remainder of the operations (see Figure 4.3). To compute the Bernstein polynomials

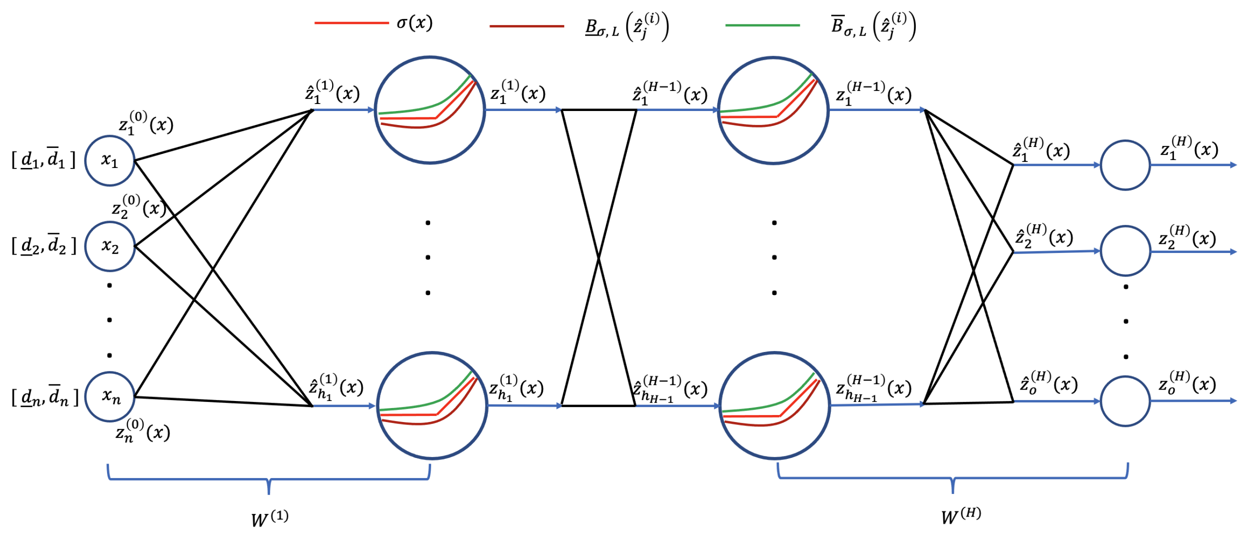


Figure 4.3: Mechanism of BERN-NN Polynomial Interval Arithmetic.

$\overline{B}_{z_i^{(0)},1}$ and $\underline{B}_{z_i^{(0)},1}$, we recall that the coefficients of such polynomials depend on the input domain. Hence, given a hypercube $I_n(\underline{d}, \overline{d})$ that bounds the input x of the neural network, we compute the tensor representation of these polynomials as:

$$Ten\left(\overline{B}_{z_1^{(0)},1}\right) = Ten\left(\underline{B}_{z_1^{(0)},1}\right) = \begin{bmatrix} \underline{d}_1 \\ \overline{d}_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{4.24}$$

$$Ten\left(\overline{B}_{z_2^{(0)},1}\right) = Ten\left(\underline{B}_{z_2^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \underline{d}_2 \\ \overline{d}_2 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{4.25}$$

$$\vdots \tag{4.26}$$

$$Ten\left(\overline{B}_{z_n^{(0)},1}\right) = Ten\left(\underline{B}_{z_n^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} \underline{d}_n \\ \overline{d}_n \end{bmatrix} \tag{4.27}$$

Next, we propagate these over- and under-approximation polynomials to the inputs of the first layer in the neural network using (4.22) and (4.23). Given a user-defined approximation order L, we propagate the polynomial approximations through the ReLU function using (4.20) and (4.21) for each of the neurons in layer 1. The produced over- and under-approximations of the outputs of all neurons are aggregated together in one tensor which is then propagated to the next layer. This process continues until we compute the over- and under-approximation polynomials of the outputs of the neural network, denoted by $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ for $j = 1, \ldots, o$. These polynomials are used as the solution of Problem 1.

It is important to note that the final Bernstein polynomials $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ have orders of L^{H-1} where L is the user-defined order of approximation of the ReLU function and H is the number of layers. This polynomial order increases exponentially with the number of hidden layers. Similarly, the shape of their multi-dimensional tensor representations is equal to $L^{H-1} + 1$ which increases exponentially with the number of hidden layers. To alleviate this problem, we introduce a parameter called *Lin*. Based on this parameter, we drop the orders of the post-bound over- and under-approximation polynomials to $[1, \cdots, 1]$. In other words, we linearize the approximation polynomials every *Lin* hidden layers. We use the algorithm in [54] to perform such linearization of the Bernstein polynomial. Luckily, this algorithm, like all the other operations in our BERN-NN involves tensor multiplications and additions and hence can be parallelized over GPUs efficiently.

Finally, note that one can always obtain absolute bounds on the inputs or outputs of any of the neurons (including the outputs of the neural network), thanks to the enclosure property of Bernstein polynomials (4.4). Such absolute bounds are useful for reachability analysis and model checkers.

4.5.4 GPU Implementation Details

To get the performance increase of GPUs without the complications of low-level languages, we implemented this tool in PyTorch. As mentioned above, we represent n-dimensional Bernstein polynomials as dense n-dimensional tensors. The tool becomes memory bound very quickly as the number of input nodes increases, making the number of dimensions in the tensors larger. In order to combat this, we use as many in-place operations as possible to avoid repeatedly allocating large chunks of memory during computation. Similarly, the multinomial coefficients used for degree elevation are used multiple times throughout the tool, and we cache each the first time they are generated to avoid spending time re-doing calculations and allocating additional memory.

We parallelized the tool on a node level: at each layer, the outputs of the last layer are passed to each node, which then can run independently of each other on separate GPUs. However, because the tensors become large very quickly, the gains in computation time only offset the overhead of copying tensors between GPUs when the neural network is particularly large. We collect and stack the outputs of all the nodes in one tensor and pass it to the next layer. When the polynomials are being composed with the ReLU approximation, each term is elevated to the highest degree expected of a composition between these two polynomials. This both ensures that the outputs of all the neurons can be stacked, as they are all the same shape and size, and also allows the multiplication of the stacked outputs of the last layer by the incoming weights to be a simple broadcasting multiplication, which is then easily parallelizable on a GPU.

We achieved additional performance gains by rewriting for-loops as element-wise tensor operations and by batching linear algebra operations like matrix multiplications and calculating the least-square solutions of matrices, both of which allow operations to be easily parallelized on GPUs and reduce the amount of time spent allocating many small patches of memory, instead doing a single large allocation.

4.6 Numerical Results

In this section, we perform a series of numerical experiments to evaluate the scalability and effectiveness of our tool. First, we conduct an ablation study to check the effect of varying different parameters (e.g., neural network width, neural network depth, ReLU approximation order) on the performance of our tool. We utilize two metrics:

- **Execution time**: which measures the time (in seconds) needed to compute the final Bernstein polynomials. Indeed, smaller values indicate better performance.
- **Relative volume of the output set**: this metric measures the "tightness" of the produced over- and under-approximation polynomials. Without loss of generality, we focus on neural networks with one output $z^{(H)}$ and we compute this metric as:

$$\text{Vol\_relative} = \frac{\text{Vol\_Output}}{\text{Vol\_Input}} \tag{4.28}$$

$$\text{Vol\_Input} = \prod_{i=1}^{n} \left(\overline{d}_i - \underline{d}_i \right) \tag{4.29}$$

$$\text{Vol\_Output} = \int \cdots \int_{I_n} \left(\overline{B}_{z^{(H)}}(x) - \underline{B}_{z^{(H)}}(x) \right) dx_1 \dots dx_n \tag{4.30}$$

  Indeed, smaller values of this metric indicate tighter approximations of the output set.

After the ablation study, we compare our tool with a set of state-of-the-art bound computation tools—including the winner of the last 2022 Verification of Neural Network (VNN) competition [45]—to study the relative performance.

Setup: We implemented our tool in Python3.9 using PyTorch for all tensor arithmetic. We run all our experiments using a single GeForce RTX 2080 Ti GPU and two 24-core Intel(R) Xeon(R). We like to note that the throughput of the tool can be increased by utilizing multiple GPU to process different neurons in parallel in a batch-processing fashion. However, in this section, we focus on using only one GPU and we leave the generalization of our algorithm to utilize multiple GPUs for future work.

4.6.1 Ablation study

The effect of varying the ReLU's order of approximation:

We study the effect of varying the ReLU's order of approximation L for a fixed NN architecture on the execution time and the output's relative volume space of our tool. In Figure 4.4, we report the statistical results for 50 random networks of a fixed architecture. Figure 4.4 (top) shows that increasing the approximation order increases the execution time. On the other hand, Figure 4.4 (bottom) shows that the relative volume of the output set significantly decreases with increasing the order of approximation. The results of both figures highlight the trade-off between the tightness of the output bounds and the execution time as a function of the ReLU approximation order L.

The effect of varying the input's dimension:

We study the effect of varying the input's dimension n, for a fixed NN architecture on the execution time of our tool. Figure 4.5 shows that the execution time for computing the output set grows linearly for smaller values of n but seems to grow more rapidly after $n = 7$. This suggests that the proposed tool can be used efficiently for many control applications.

The effect of increasing the number of neurons per layer:

We study the effect of varying the number of neurons per layer N_e, for a fixed NN architecture $[3, N_e, N_e, 1]$ on the execution time of our tool. Figure 4.6 summarizes the execution times

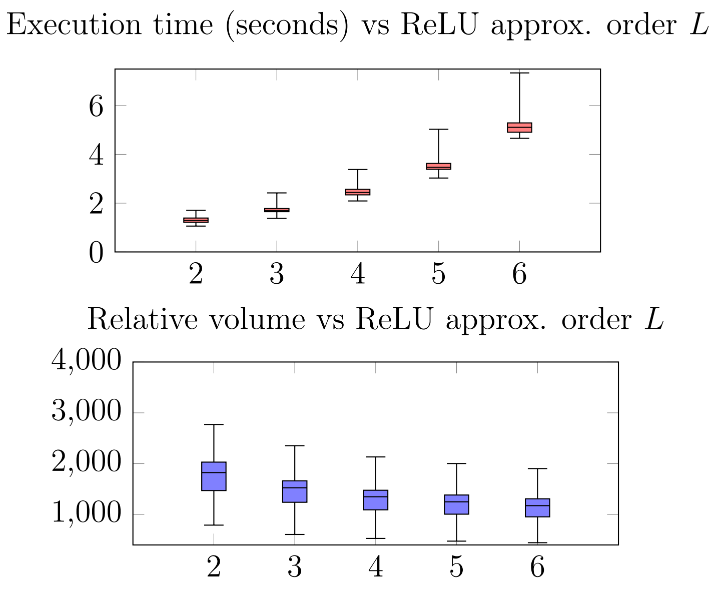


Figure 4.4: Effect of varying the ReLU's order of approximation L for a NN architecture $[2, 20, 20, 1]$ on the execution time of our tool (top) and the relative volume of the output set (bottom). We set $n = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

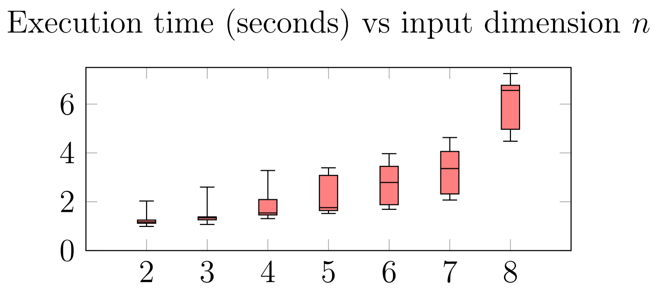


Figure 4.5: Effect of varying the input's dimension n for a NN architecture $[n, 20, 20, 1]$ on the execution time our tool. We set $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

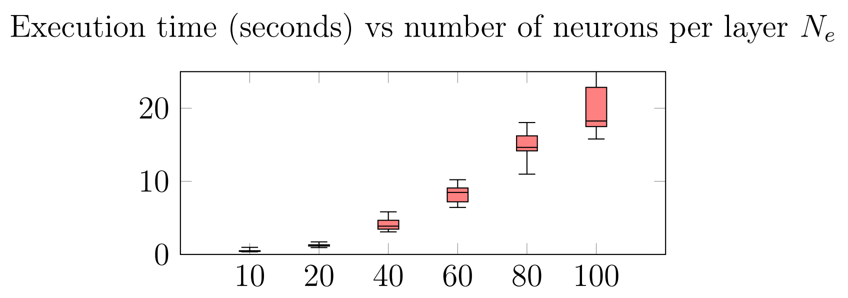


Figure 4.6: Effect of varying the number of neurons per layer N_e for a NN architecture $[2, N_e, N_e, 1]$ on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

with a varying number of neurons per layer. The results show that increasing the number of neurons per layer highly affects the execution time. This is due to the expensive arithmetic and memory operations for large tensors that represent the Bernstein polynomials. Nevertheless, this increase in execution time can be harnessed by using multiple GPUs to compute bounds for different nodes in parallel along with using the same GPU to process multiple nodes simultaneously.

The effect of increasing the number of hidden layers:

We study the effect of varying the number of hidden layers n_h, with 20 neurons in every hidden layer, on the execution time of our tool. Unlike the effect of increasing the number of neurons per layer, the results in Figure 4.7 show that the execution time almost grows linearly with the number of hidden layers.

Scalability analysis of Bern-NN:

We finally try to study the execution time of Bern-NN for relatively large neural networks. In this study, we add extra layers with 100 neurons each and report the execution time in Figure 4.8 for random neural networks. As shown in the figure, Bern-NN can process neural

Execution time (seconds) vs number of layers n_h

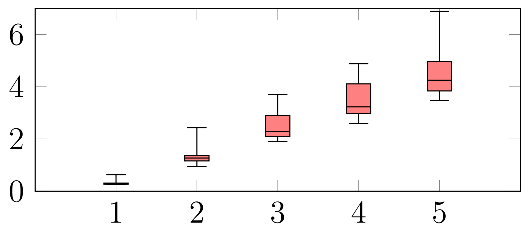


Figure 4.7: Effect of varying the number of hidden layers n_h, for a NN architecture $[2, 20, .., 20, 1]$ with 20 neurons in every hidden layer on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

Execution time (seconds) vs total number of neurons

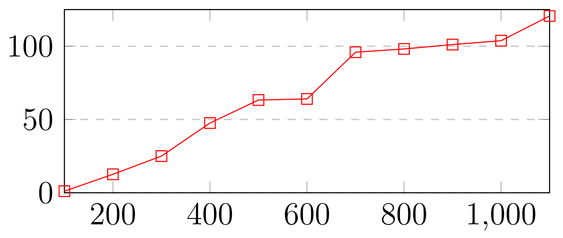


Figure 4.8: Scalability of the Bern-NN tool as a function of increasing the total number of neurons.

networks with more than 1000 neurons in less than 2 minutes.

4.6.2 Comparison against other tools

In this subsection, we compare the performance of our tool in terms of execution time and the output set's relative volume compared to bound propagation tools such as Symbolic Interval Analysis (SIA)[4], alpha-CROWN [49], and reachability analysis tool such as POLAR [1]. We note that alpha-CROWN [49] was the winner of the 2022 VNN competition and we compare Bern-NN against the bound propagation algorithm used within alpha-CROWN as a representative tool for all the bound propagation techniques. Moreover, alpha-CROWN is also designed to harness the computational powers of GPUs. We compare Bern-NN against

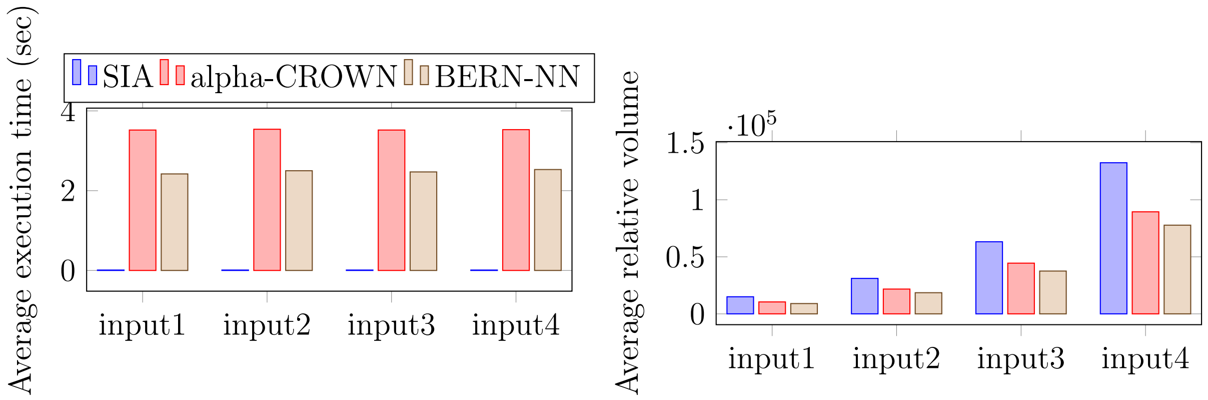


Figure 4.9: Performance results in terms of average execution times (top) and relative volume (bottom) for BERN-NN, SIA, and alpha-CROWN for different input spaces. The NN's architecture is $[2, 20, 20, 1]$. The ReLU's order of approximation is $L = 4$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. Input1 $= I_n = [-5, 5]^2$, Input2 $= I_n = [-10, 10]^2$, Input3 $= I_n = [-20, 20]^2$, Input4 $= I_n = [-40, 40]^2$.

POLAR since it also uses polynomials (Taylor Model with a Bernstein error correction) to compute bounds on the output of neural networks. POLAR [1] outperforms other reachability-based tools and hence is a representative tool for such techniques.

Comparison against SIA and alpha-CROWN for random NN

We compare the performance of our tool to SIA and alpha-CROWN for random neural networks with $[2, 20, 20, 1]$ architecture for different hyperrectangle input spaces (Figure 4.9). We also compare the performance as the input dimension of the network increases (Figure 4.10). The results show that SIA is the fastest in terms of execution time for all different input hyperrectangles due to the simplicity of its computations. However, its relative volume is the highest. On the other hand, Bern-NN's relative volume is the smallest for all different input spaces thanks to its tight higher-order ReLU approximations. Compared to alpha-CROWN (which also runs on GPUs), Bern-NN is both faster and produces tighter bounds leading to an average of 25% reduction in execution time with an average of 10% reduction in the relative volume metric. This shows the practicality of Bern-NN for control applications.

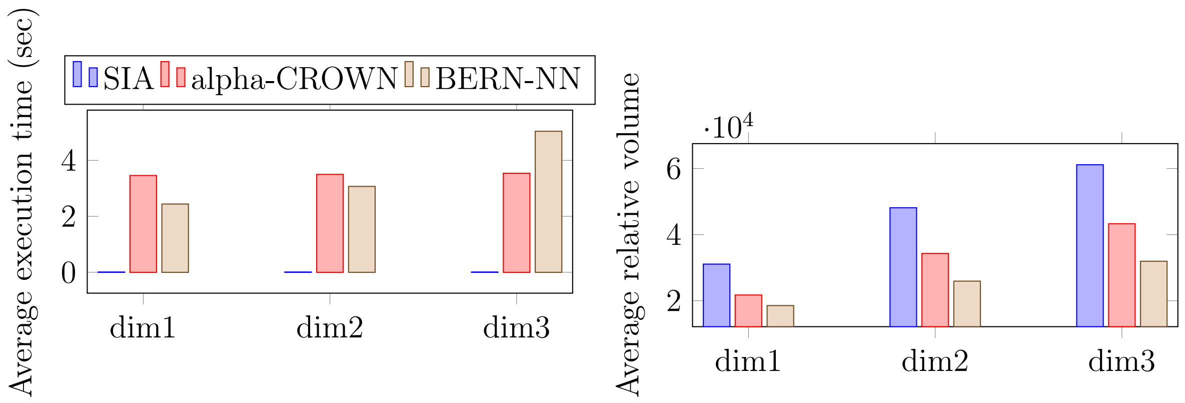


Figure 4.10: Performance results in terms of average execution times (top) and relative volume (bottom) for BERN-NN, SIA, and alpha-CROWN for input's dimensions n. The NN's architecture is $[n, 20, 20, 1]$. the input's space is $[-10, 10]^n$. The ReLU's order of approximation is $L = 4$, $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. dim1 $= n = 2$, dim2 $= n = 3$, dim3 $= n = 4$.

Case Study for Control Benchmarks

In this experiment, we test different tools on benchmarks of NN controllers (used by POLAR) to evaluate the tightness of their estimated bounds. Table 4.3 shows the architecture of the networks used in each benchmark. Table 4.2 summarizes the performance of the tools with respect to the average execution time and average relative volume for six control benchmarks. The results show that Bern-NN provides the tightest estimate for the output set for all benchmarks except Benchmark 3. We would like to highlight that the tight approximation provided by Bern-NN is important for control applications because the specification of interest is usually defined over a time horizon and require multi-step reachability, hence, tighter bounds at each step are crucial. Lastly, Bern-NN is faster than alpha-CROWN over all benchmarks except Benchmark 5. However, SIA and POLAR are faster than Bern-NN but provide looser bound estimates. Each benchmark is run with five different hyperrectangles that are all centered around zero and have a radius $r \in \{1, 1.5, 2, 2.5, 3\}$.

Table 4.2: Performance results in terms of average execution times and volume for BERN-NN, SIA, alpha-CROWN, and POLAR, for 5 different input's spaces $I_n\left(\underline{d}, \overline{d}\right)$ for 6 benchmarks [1]. The ReLU's order of approximation is $L = 2$, $Lin = 0$.

Tool	Benchmark 1		Benchmark 2		Benchmark 3		Benchmark 4		Benchmark 5		Benchmark 6	
	time	volume	time	volume	time	volume	time	volume	time	volume	time	volume
SIA	**0.01**	2.544	**0.02**	6.05	**0.01**	1.02	**0.01**	9.41	**0.02**	53.38	**0.02**	2.03
$CROWN$	2.9	3.1	3.49	5.50	3.54	**0.73**	3.13	17.04	3.80	77.72	4.10	2.4
$Bern-NN$	0.84	**1.62**	1.30	**5.4**	1.09	0.81	1.15	**6.21**	41.7	**35.85**	3.25	**1.38**
$POLAR$	0.21	25.43	0.284	51.80	0.29	18.81	0.42	33.32	5.52	432.75	0.81	7.00

Table 4.3: Architectures of POLAR Benchmarks

	Architecture
Benchmark 1	[2,20,20,1]
Benchmark 2	[2,20,20,1]
Benchmark 3	[2,20,20,1]
Benchmark 4	[3,20,20,1]
Benchmark 5	[3,100,100,1]
Benchmark 6	[4,20,20,20,1]

4.7 Conclusion

In conclusion, we presented Bern-NN, a tool for computing higher-order tight bounds for NNs by approximating non-linear ReLU activations using Bernstein polynomials. We provided GPU-based computational machinery to handle tensor arithmetic for manipulating polynomials as well as bounding them using the properties of Bernstein polynomials. We conducted extensive experiments to evaluate the scalability of our tool as well as compare its estimated bounds with state-of-the-art methods. The results showed that our tool can process neural networks with thousands of neurons in a few minutes. These results also show that our tool outperforms state-of-the-art tools in terms of computing tighter bounds while reducing the execution time compared to other tools.

Chapter 5

DeepBern-Nets

As discussed in the previous chapters, formal certification of Neural Networks (NNs) is crucial for ensuring their safety, fairness, and robustness. Unfortunately, on the one hand, sound and complete certification algorithms of ReLU-based NNs do not scale to large-scale NNs. On the other hand, incomplete certification algorithms—based on propagating input domain bounds to bound the outputs of the NN—are easier to compute, but they result in loose bounds that deteriorate with the depth of NN, which diminishes their effectiveness. In this paper, we ask the following question; can we replace the ReLU activation function with one that opens the door to incomplete certification algorithms that are easy to compute but can produce tight bounds on the NN's outputs? We introduce DeepBern-Nets, a class of NNs with activation functions based on Bernstein polynomials instead of the commonly used ReLU activation. Bernstein polynomials are smooth and differentiable functions with desirable properties such as the so-called range enclosure and subdivision properties. We design a novel Interval Bound Propagation (IBP) algorithm, called Bern-IBP, to efficiently compute tight bounds on DeepBern-Nets outputs. Our approach leverages the properties of Bernstein polynomials to improve the tractability of neural network certification tasks while maintaining the accuracy of the trained networks. We conduct comprehensive experiments in adversarial robustness and reachability analysis settings to assess the effectiveness of the proposed Bernstein polynomial activation in enhancing the certification process. Our proposed framework achieves high certified accuracy for adversarially-trained NNs, which is often a challenging task for certifiers of ReLU-based NNs. Moreover, using Bern-IBP bounds for certified training results in NNs with state-of-the-art certified accuracy compared to ReLU networks. This work establishes Bernstein polynomial activation as a promising alternative for improving neural network certification tasks across various NNs applications. The code for DeepBern-Nets is publicly available[1].

5.1 Introduction

Deep neural networks (NNs) have revolutionized numerous fields with their remarkable performance on various tasks, ranging from computer vision and natural language processing to healthcare and robotics. As these networks become integral components of critical systems, ensuring their safety, security, fairness, and robustness is essential. It is unsurprising, then, the growing interest in the field of certified machine learning, which resulted in NNs with enhanced levels of robustness to adversarial inputs [55, 56, 57, 58], fairness [59, 60, 61, 62], and correctness [63].

While certifying the robustness, fairness, and correctness of NNs with respect to formal properties is shown to be NP-hard [10], state-of-the-art certifiers rely on computing upper/lower bounds on the output of the NN and its intermediate layers [39, 38, 64, 65, 37]. Accurate bounds can significantly reduce the complexity and computational effort required during the certification process, facilitating more efficient and dependable evaluations of the network's behavior in diverse and challenging scenarios. Moreover, computing such bounds has opened the door for a new set of "certified training" algorithms [66, 67, 68] where these bounds are

[1] https://github.com/rcpsl/DeepBern-Nets

used as a regularizer that penalizes the worst-case violation of robustness or fairness, which leads to training NNs with favorable properties. While computing such lower/upper bounds is crucial, current techniques in computing lower/upper bounds on the NN outputs are either computationally efficient but result in loose lower/upper bounds or compute tight bounds but are computationally expensive. In this paper, we are interested in algorithms that can be both computationally efficient and lead to tight bounds.

This work follows a Design-for-Certifiability approach where we ask the question; can we replace the ReLU activation function with one that allows us to compute tight upper/lower bounds efficiently? Introducing such novel activation functions designed with certifiability in mind makes it possible to create NNs that are easier to analyze and certify during their training. Our contributions in this paper can be summarized as follows:

1. We introduce DeepBern-Nets, a NN architecture with a new activation function based on Bernstein polynomials. Our primary motivation is to shift some of the computational efforts from the certification phase to the training phase. By employing this approach, we can train NNs with known output (and intermediate) bounds for a predetermined input domain which can accelerate the certification process.

2. We present Bern-IBP, an Interval Bound Propagation (IBP) algorithm that computes tight bounds of DeepBern-Nets leading to an efficient certifier.

3. We show that Bern-IBP can certify the adversarial robustness of adversarially-trained DeepBern-Nets on MNIST and CIFAR-10 datasets even with large architectures with millions of parameters. This is unlike state-of-the-art certifiers for ReLU networks, which often fail to certify robustness for adversarially-trained ReLU NNs.

4. We show that employing Bern-IBP during the training of DeepBern-Nets yields high certified robustness on the MNIST and CIFAR-10 datasets with robustness levels that

are comparable—or in many cases surpassing—the performance of the most robust ReLU-based NNs reported in the SOK benchmark.

We believe that our framework, DeepBern-Nets and Bern-IBP, enables more reliable guarantees on NN behavior and contributes to the ongoing efforts to create safer and more secure NN-based systems, which is crucial for the broader deployment of deep learning in real-world applications.

5.2 DeepBern-Nets: Deep Bernstein Polynomial Networks

5.2.1 Bernstein polynomials preliminaries

Bernstein polynomials form a basis for the space of polynomials on a closed interval [69]. These polynomials have been widely used in various fields, such as computer-aided geometric design [69], approximation theory [70], and numerical analysis [71], due to their unique properties and intuitive representation of functions. A general polynomial of degree n in Bernstein form on the interval $[l, u]$ can be represented as:

$$P_n^{[l,u]}(x) = \sum_{k=0}^{n} c_k b_{n,k}^{[l,u]}(x), \qquad x \in [l, u] \tag{5.1}$$

where $c_k \in \mathbb{R}$ are the coefficients associated with the Bernstein basis $b_{n,k}^{[l,u]}(x)$, defined as:

$$b_{n,k}^{[l,u]}(x) = \frac{\binom{n}{k}}{(u-l)^n}(x-l)^k(u-x)^{n-k}, \tag{5.2}$$

with $\binom{n}{k}$ denoting the binomial coefficient. The Bernstein coefficients c_k determine the shape and properties of the polynomial $P_n^{[l,u]}(x)$ on the interval $[l, u]$. It is important to note that

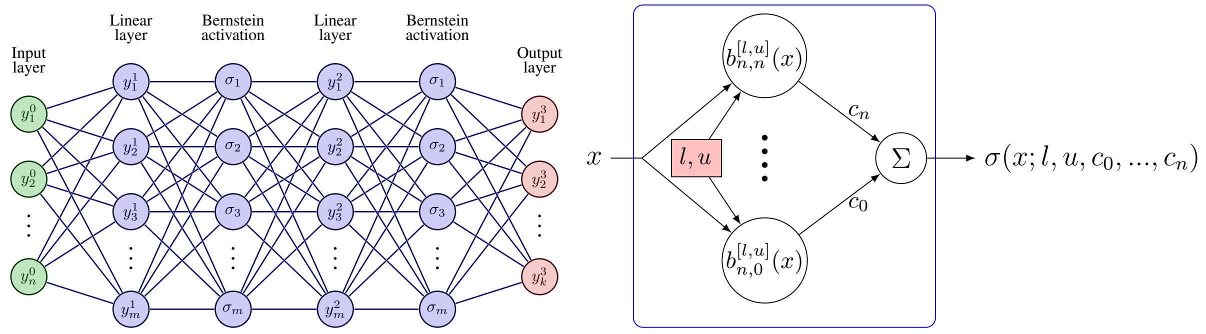


Figure 5.1: (Left) shows the structure of a DeepBern-Nets with two hidden layers. DeepBern-Nets are similar to Feed Forward NNs except that the activation function is a Bernstein polynomial. (Right) shows a simplified computational graph of a degree n Bernstein activation. The Bernstein basis is evaluated at the input x using l and u computed during training, and the output is then computed as a linear combination of the basis functions weighted by the learnable Bernstein coefficients c_k.

unlike polynomials represented in power basis form, the representation of a polynomial in Bernstein form depends on the domain of interest $[l, u]$ as shown in equation 5.1.

5.2.2 Neural Networks with Bernstein activation functions

We propose using Bernstein polynomials as non-linear activation functions σ in feed-forward NNs. We call such NNs as DeepBern-Nets. Like feed-forward NNs, DeepBern-Nets consist of multiple layers, each consisting of linear weights followed by non-linear activation functions. Unlike conventional activation functions (e.g., ReLU, sigmoid, tanh, ..), Bernstein-based activation functions are parametrized with learnable Bernstein coefficients $\boldsymbol{c} = c_0, \ldots, c_n$, i.e.,

$$\sigma(x; l, u, \boldsymbol{c}) = \sum_{k=0}^{n} c_k b_{n,k}^{[l,u]}(x), \qquad x \in [l, u], \tag{5.3}$$

where x is the input to the neuron activation, and the polynomial degree n is an additional hyper-parameter of the Bernstein activation and can be chosen differently for each neuron. Figure 5.1 shows a simplified computational graph of the Bernstein activation and how it is used to replace conventional activation functions.

Training of DeepBern-Nets. Since Bernstein polynomials are defined on a specific domain (equation 5.2), we need to determine the lower and upper bounds ($\boldsymbol{l}^{(k)}$ and $\boldsymbol{u}^{(k)}$) of the inputs to the Bernstein activation neurons in layer k, during the training of the network. To that end, we assume that the input domain $\mathcal{D}$ is bounded with the lower and upper bounds (denoted as $\boldsymbol{l}^{(0)}$ and $\boldsymbol{u}^{(0)}$, respectively) known during training. We emphasize that our assumption that $\mathcal{D}$ is bounded and known is not conservative, as the input to the NN can always be normalized to $[0, 1]$, for example.

Using the bounds on the input domain $\boldsymbol{l}^{(0)}$ and $\boldsymbol{u}^{(0)}$ and the learnable parameters of the NNs (i.e., weights of the linear layers and the Bernstein coefficients $\boldsymbol{c}$ for each neuron), we

Algorithm 1 Training step of an L-layer DeepBern-Net $\mathcal{NN}$

```
1: Given: Training Batch $(\mathcal{X}, \boldsymbol{t})$ and input bounds $[\boldsymbol{l}^{(0)}, \boldsymbol{u}^{(0)}]$
2: Initialize all parameters
3: Set the learning rate $\alpha$
4: ▷ Forward propagation
5: Set $\boldsymbol{y}^{(0)} = \mathcal{X}$
6: Set $\mathcal{B}^{(0)} = [\boldsymbol{l}^{(0)}, \boldsymbol{u}^{(0)}]$
7: for $i = 1....L$ do
8:     if layer $i$ is Bernstein activation then
9:         $\boldsymbol{l}^{(i)}, \boldsymbol{u}^{(i)} \leftarrow \mathcal{B}^{(i-1)}$          ▷ Store Input bounds of the Bernstein layer
10:        for each neuron $z$ in layer $i$ do
11:            Let $\boldsymbol{c}_{\boldsymbol{z}}^{(\boldsymbol{i})}$ be the Bernstein coefficients for neuron $z$ of the $i$-th layer
12:            $\mathcal{B}_z^{(i)} \leftarrow [\min_j c_{zj}^{(i)}, \max_j c_{zj}^{(i)}]$
13:        $\mathcal{B}^{(i)} \leftarrow [\mathcal{B}_0^{(i)}, \mathcal{B}_1^{(i)}, ..., \mathcal{B}_m^{(i)}]$          ▷ $m$ denotes the number of neurons in layer $i$
14:    else
15:        $\mathcal{B}^{(i)} \leftarrow$IBP$(\mathcal{B}^{(i-1)})$
16:    $\boldsymbol{y}^{(i)} \leftarrow$ forward$(\boldsymbol{y}^{(i-1)})$          ▷ Regular forward step
17: ▷ Backpropagation
18: Compute the loss function: $\mathcal{L}(\boldsymbol{y}^{(L)}, \boldsymbol{t})$
19: Compute the gradients with respect to all model parameters (including Bernstein coefficients)
20: for each Parameter $\theta$ do          ▷ Weights, biases, and Bernstein coefficients $c_k$
21:     $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$
```

will update the bounds $\boldsymbol{l}^{(k)}$ and $\boldsymbol{u}^{(k)}$ with each step of training by propagating $\boldsymbol{l}^{(0)}$ and $\boldsymbol{u}^{(0)}$ through all the layers in the network. Unlike conventional non-linear activation functions where symbolic bound propagation relies on linear relaxation techniques [4, 22], the Bernstein polynomial enclosure property allows us to bound the output of an n-th order Bernstein activation in $\mathcal{O}(n)$ operations (Algorithm 1-line 12). We start by reviewing the enclosure property of Bernstein polynomials as follows.

Property 1 (Enclosure of Range [72])**.** *The enclosure property of Bernstein polynomials states that for a given polynomial $P_n^{[l,u]}(x)$ of degree n in Bernstein form on an interval $[l,u]$, the polynomial lies within the convex hull of its Bernstein coefficients. In other words, the Bernstein polynomial is bounded by the minimum and maximum values of its coefficients c_k regardless of the input x.*

$$\min_{0\leq k\leq n} c_k \leq P_n^{[l,u]}(x) \leq \max_{0\leq k\leq n} c_k, \qquad \forall x \in [l,u]. \tag{5.4}$$

Algorithm 1 outlines how to use the enclosure property to propagate the bounds from one layer to another for a single training step in an L-layer DeepBern-Net. In contrast to normal training, we calculate the worst-case bounds for the inputs to all Bernstein layers by propagating the bounds from the previous layers. Such bound propagation can be done for linear layers using interval arithmetic [36]—referred to in Algorithm 1-line 15 as Interval Bound Propagation (IBP)—or using Property 1 for Bernstein layers (Algorithm 1-Line 12). We store the resulting bounds for each Bernstein activation function. Then, we perform the regular forward step. The parameters are then updated using vanilla backpropagation, just like conventional NNs. During inference, we directly use the stored layer-wise bounds $\boldsymbol{l}^{(k)}$ and $\boldsymbol{u}^{(k)}$ (computed during training) to propagate any input through the network. In Appendix B.3.3, we show that the overhead of computing the bounds $\boldsymbol{l}^{(k)}$ and $\boldsymbol{u}^{(k)}$ during training adds between $0.2\times$ to $5\times$ overhead for the training, depending on the order n of the Bernstein activation function and the size of the network.

Stable training of DeepBern-Nets. Using polynomials as activation functions in deep NNs has attracted several researchers' attention in recent years [73, 74]. A major drawback of using polynomials of arbitrary order is their unstable behavior during training due to exploding gradients–which is prominent with the increase in order [74]. In particular, for a general nth order polynomial in power series $f_n(x) = w_0 + w_1 x + \ldots + w_n x^n$, its derivative is $df_n(x)/dx = w_1 + \ldots + n w_n x^{n-1}$. Hence training a deep NN with multiple polynomial activation functions suffers from exploding gradients as the gradient scales exponentially with the increase in the order n for $x > 1$.

Luckily, and thanks to the unique properties of Bernstein polynomials, DeepBern-Net does not suffer from such a limitation as captured in the next result, whose proof is given in Appendix B.1.1.

Proposition 5.2.1. *Consider the Bernstein activation function $\sigma(x; l, u, \boldsymbol{c})$ of arbitrary order n. The following holds:*

1. $\left|\frac{d}{dx}\sigma(x; l, u, \boldsymbol{c})\right| \leq 2n \max_{k \in \{0,\ldots,n\}} |c_k|$,
2. $\left|\frac{d}{dc_i}\sigma(x; l, u, \boldsymbol{c})\right| \leq 1$ *for all* $i \in \{0, \ldots, n\}$.

Proposition 5.2.1 ensures that the gradients of the proposed Bernstein-based activation function depend only on the value of the learnable parameters $\boldsymbol{c} = (c_0, \ldots, c_n)$. Hence, the gradients do not explode for $x > 1$. This feature is not enjoyed by the polynomial activation functions in [74] and leads to better stable training properties when the Bernstein polynomials are used as activation functions. Moreover, one can control these gradients by adding a regularizer–to the objective function–that penalizes high values of c_k, which is common for other learnable parameters, i.e., weights of the linear layer. Proof of Proposition 5.2.1 is in Appendix B.1.1

5.3 Bern-IBP: Certification using Bernstein Interval Bound Propagation

5.3.1 Certification of global properties using Bern-IBP

We consider the certification of global properties of NNs. Global properties need to be held true for the entire input domain $\mathcal{D}$ of the network. For simplicity of presentation, we will assume that the global property we want to prove takes the following form:

$$\forall \boldsymbol{y}^{\mathbf{(0)}} \in \mathcal{D} \Longrightarrow y^{(L)} = \mathcal{NN}(\boldsymbol{y}^{(0)}) > 0 \tag{5.5}$$

where $y^{(L)}$ is a scalar output and $\mathcal{NN}$ is the NN of interest. Examples of such global properties include the stability of NN-controlled systems [75] as well as global individual fairness [62].

In this paper, we focus on the incomplete certification of such properties. In particular, we certify properties of the form (5.5) by checking the lower/upper bounds of the NN. To that end, we define the lower $\mathcal{L}$ and upper $\mathcal{U}$ bounds of the NN within the domain $\mathcal{D}$ as any real numbers that satisfy:

$$\mathcal{L}\left(\mathcal{NN}(\boldsymbol{y}^{(0)}), \mathcal{D}\right) \leq \min_{\boldsymbol{y}^{(0)} \in \mathcal{D}} \mathcal{NN}(\boldsymbol{y}^{(0)}), \qquad \mathcal{U}\left(\mathcal{NN}(\boldsymbol{y}^{(0)}), \mathcal{D}\right) \geq \max_{\boldsymbol{y}^{(0)} \in \mathcal{D}} \mathcal{NN}(\boldsymbol{y}^{(0)}) \tag{5.6}$$

Incomplete certification of (5.5) is equivalent to checking if $\mathcal{L}\left(\mathcal{NN}(\boldsymbol{y}^{(0)}), \mathcal{D}\right) > 0$. Thanks to the Enclosure of Range (Property 1) of DeepBern-Nets, one can check the condition $\mathcal{L}\left(\mathcal{NN}(\boldsymbol{y}^{(0)}), \mathcal{D}\right) > 0$ in constant time, i.e., $\mathcal{O}(1)$, by simply checking the minimum Bernstein coefficients of the output layer.

5.3.2 Certification of local properties using Bern-IBP

Local properties of NNs are the ones that need to be held for subsets S of the input domain $\mathcal{D}$, i.e.,

$$\forall \boldsymbol{y}^{(\mathbf{0})} \in S \subset \mathcal{D} \Longrightarrow y^{(L)} = \mathcal{NN}(\boldsymbol{y}^{(0)}) > 0 \tag{5.7}$$

Examples of local properties include adversarial robustness and the safety of NN-controlled vehicles [33, 76, 77]. Similar to global properties, we are interested in incomplete certification by checking whether $\mathcal{L}\left(\mathcal{NN}(\boldsymbol{y}^{(0)}), S\right) > 0$.

The output bounds stored in the Bernstein activation functions are the worst-case bounds for the entire input domain $\mathcal{D}$. However, for certifying local properties over $S \subset \mathcal{D}$, we need to refine these output bounds on the given sub-region S. To that end, for a Bernstein activation layer k with input bounds $[\boldsymbol{l}^{(k)}, \boldsymbol{u}^{(k)}]$ (computed and stored during training), we can obtain tighter output bounds thanks to the following subdivision property of Bernstein polynomials.

Property 2 (Subdivision [72])**.** *Given a Bernstein polynomial $P_n^{[l,u]}(x)$ of degree n on the interval $[l, u]$, the coefficients of the same polynomial on subintervals $[l, \alpha]$ and $[\alpha, u]$ with $\alpha \in [l, u]$ can be computed as follows. First, compute the intermediate coefficients c_j^k for $k = 0, ..., n$ and $j = k, ..., n$*

$$c_j^k = \begin{cases} c_j & \textit{if } k = 0 \\ (1-\tau)c_{j-1}^{k-1} + \tau c_j^{k-1} & \textit{if } k > 0 \end{cases}, \quad c_i' = c_i^i \quad c_i'' = c_n^{n-i} \quad i = 0 \dots n,$$

where $\tau = \frac{\alpha - l}{u - l}$. *Next, the polynomials defined on each of the subintervals* $[l, \alpha]$ *and* $[\alpha, u]$ *are:*

$$P_n^{[l,\alpha]}(x) = \sum_{k=0}^{n} c'_k b_{n,k}^{[l,\alpha]}(x), \qquad P_n^{[\alpha,u]}(x) = \sum_{k=0}^{n} c''_k b_{n,k}^{[\alpha,u]}(x).$$

Indeed, we can apply the Subdivision property twice to compute the coefficients of the polynomial $P_n^{[\alpha,\beta]}$. Computing the coefficients on the subintervals allows us to tightly bound the polynomial using property 1. Therefore, given a DeepBern-Net trained on $\mathcal{D} = [\boldsymbol{l}^{(0)}, \boldsymbol{u}^{(0)}]$, we can compute tighter bounds on the subregion $S = [\hat{\boldsymbol{l}}^{(0)}, \hat{\boldsymbol{u}}^{(0)}]$ by applying the subdivision property (Property 2) to compute the Bernstein coefficients on the sub-region S, and then use the enclosure property (Property 1) to compute tight bounds on the output of the activation equivalent to the minimum and maximum of the computed Bernstein coefficients. We do this on a layer-by-layer basis until we reach the output of the NN. Implementation details of this approach is given in Appendix B.2.

5.4 Experiments

Implementation: Our framework has been developed in Python, and is designed to facilitate the training of DeepBern-Nets and certify local properties such as Adversarial Robustness and certified training. We use PyTorch [78] for all neural network training tasks. To conduct our experiments, we utilized a single GeForce RTX 2080 Ti GPU in conjunction with a 24-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz. Only 8 cores were utilized for our experiments.

5.4.1 Experiment 1: Certification of Adversarial Robustness

The first experiment assesses the ability to compute tight bounds on the NN output and its implications for certifying NN properties. To that end, we use the application of adversarial robustness, where we aim to certify that a NN model is not susceptible to adversarial examples within a defined perturbation set. The results in [2, 79] show that state-of-the-art IBP algorithms fail to certify the robustness of NNs trained with Projected Gradient Descent (PGD), albeit being robust, due to the excessive errors in the computed bounds, which forces designers to use computationally expensive sound and complete algorithms. Thanks to the properties of DeepBern-Nets, the bounds computed by Bern-IBP are tight enough to certify the robustness of NNs without using computationally expensive sound and complete tools. To that end, we trained several NNs using the MNIST [6] and CIFAR-10 [80] datasets using PGD. We trained both Fully Connected Neural Networks (FCNN) and Convolutional Neural Networks (CNNs) on these datasets with Bernstein polynomials of orders $2, 3, 4, 5,$ and 6. For detailed information regarding the model architectures, please refer to Appendix B.3.2. Further information about the training procedure can be found in Appendix B.3.1.

Formalizing adversarial robustness as a local property

Given a NN model $\mathcal{NN} : [0,1]^d \to \mathbb{R}^o$, a concrete input $\boldsymbol{x_n}$, a target class t, and a perturbation parameter ϵ, the adversarial robustness problem asks that the NN output be the target class t for all the inputs in the set $\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x_n}\|_\infty \leq \epsilon\}$. In other words, a NN is robust whenever:

$$\forall \boldsymbol{x} \in S(\boldsymbol{x_n}, \epsilon) = \{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x_n}\|_\infty \leq \epsilon\} \quad \implies \quad \mathcal{NN}(\boldsymbol{x})_t > \mathcal{NN}(\boldsymbol{x})_i, \ i \neq t$$

where $\mathcal{NN}(\boldsymbol{x})_t$ is the NN output for the target class and $\mathcal{NN}(\boldsymbol{x})_i$ is the NN output for any class i other that t. To certify the robustness of a NN, one can compute a lower bound on the adversarial robustness $\mathbb{L}_{\text{robust}}$ for all classes $i \neq t$ as:

$$\mathbb{L}_{\text{robust}}(\boldsymbol{x_n}, \epsilon) = \min_{i \neq t} \left(\mathcal{L}\big(\mathcal{NN}(\boldsymbol{x})_t, S(\boldsymbol{x_n}, \epsilon)\big) - \mathcal{U}\big(\mathcal{NN}(\boldsymbol{x})_i, S(\boldsymbol{x_n}, \epsilon)\big) \right) \tag{5.8}$$

$$\leq \min_{i \neq t} \left(\min_{\boldsymbol{x} \in S(\boldsymbol{x_n}, \epsilon)} \mathcal{NN}(\boldsymbol{x})_t - \mathcal{NN}(\boldsymbol{x})_i \right) \tag{5.9}$$

Indeed, the NN is robust whenever $\mathbb{L}_{\text{robust}} > 0$. Nevertheless, the tightness of the bounds $\mathcal{L}(\mathcal{NN}(\boldsymbol{x})_t, S(\boldsymbol{x_n}, \epsilon))$ and $\mathcal{U}(\mathcal{NN}(\boldsymbol{x})_i, S(\boldsymbol{x_n}, \epsilon))$ plays a significant role in the ability to certify the NN robustness. The tighter these bounds, the higher the ability to certify the NN robustness.

Experiment 1.1: Tightness of output bounds - Bern-IBP vs IBP

For each trained neural network, we compute the lower bound on robustness $\mathbb{L}_{\text{robust}}(\boldsymbol{x_n}, \epsilon)$ using Bern-IBP and using state-of-the-art Interval Bound Propagation (IBP) that does not take into account the properties of DeepBern-Nets. In particular, for this experiment, we used auto_LiRPA[39], a tool that is part of $\alpha\beta$-CROWN[39]—the winner of the 2022 Verification of Neural Network (VNN) competition [79]. Figure 5.2 shows the difference between the bound $\mathbb{L}_{\text{robust}}(\boldsymbol{x_n}, \epsilon)$ computed by Bern-IBP and the one computed by IBP using a semi-log scale. The raw data for the adversarial robustness bound $\mathbb{L}_{\text{robust}}(\boldsymbol{x_n}, \epsilon)$ for both Bern-IBP and IBP is given in Appendix B.3.4.

The results presented in Figure 5.2 clearly demonstrate that Bern-IBP yields significantly tighter bounds in comparison to IBP. Figure 5.2 also shows that for all values of ϵ, the bounds computed using IBP become exponentially looser as the order of the Bernstein activations increase, unlike the bounds computed with Bern-IBP, which remain precise even for higher-order Bernstein activations or larger values of ϵ. The raw data in Appendix B.3.4 provide a clearer view on the superiority of computing $\mathbb{L}_{\text{robust}}(\boldsymbol{x_n}, \epsilon)$ using Bern-IBP compared to IBP.

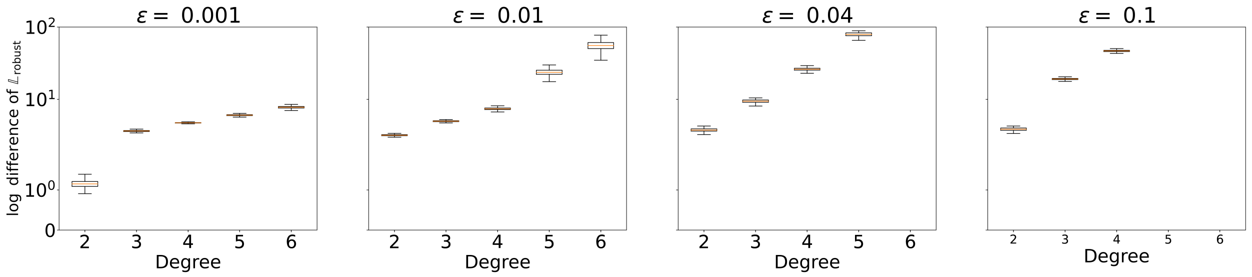


Figure 5.2: A visual representation of the tightness of bounds computed using Bern-IBP compared to IBP. The figure shows the log difference between $\mathbb{L}_{\text{robust}}$ computed using Bern-IBP and IBP for NNs with varying orders of and different values of ϵ. The figure demonstrates the enhanced precision and scalability of the Bern-IBP method in computing tighter bounds, even for higher-order Bernstein activations and larger values of ϵ, as compared to the naive IBP method.

Experiment 1.2: Certification of Adversarial Robustness using Bern-IBP

Next, we show that the superior precision of bounds calculated using Bern-IBP can lead to efficient certification of adversarial robustness. Here, we define the certified accuracy of the NN as the percentage of the data points (in the test dataset) for which an adversarial input can not change the class (the output of the NN). Table 5.1 contrasts the certified accuracy for the adversarially-trained (using 100-step PGD) DeepBern-Nets of orders 2, 4, and 6, using

Table 5.1: A comparison of certified accuracy and verification time for neural networks with Bernstein polynomial activations using both IBP and Bern-IBP methods and varying values of ϵ. The table also presents the upper bound on certified accuracy calculated using a 100-step PGD attack. The results highlight the superior performance of Bern-IBP in certifying robustness properties compared to IBP.

Dataset	Model (# of params)	Test acc. (%)	ϵ	IBP		Bern-IBP		U.B (PGD)
				Time (s)	Certified acc. (%)	Time (s)	Certified acc. (%)	Certified acc. (%)
MNIST	CNNa_4 (190,426)	97.229	0.01	3.45	0	1.43	88.69	95.97
			0.03	3.41	0	1.42	72.12	92.53
			0.1	3.26	0	1.39	65.22	75.27
	CNNb_2 (905,882)	97.14	0.01	4.38	0	2.07	80.21	95.42
			0.03	4.58	0	2.11	56.49	90.57
			0.1	4.61	0	1.97	72.35	78.6
CIFAR-10	CNNa_6 (258,626)	46.77	1/255	3.29	0	1.82	27.74	33.53
			2/255	3.25	0	1.83	33.49	35.81
	CNNb_4 (1,235,994)	54.66	1/255	5.17	0	4.45	28.55	42.86
			2/255	5.14	0	4.33	14.7	36.73

both IBP and Bern-IBP methods and varying values of ϵ. As observed by the table, IBP fails to certify the robustness of all the NNs. On the other hand, Bern-IBP achieved high certified accuracy for all the NNs with varying values of ϵ. Finally, we use the methodology reported in [39] to upper bound the certified accuracy using 100-step PGD attack.

It is essential to mention that IBP's inability to certify the robustness of NNs is not unique to DeepBern-Nets. In particular, as shown in [2, 79], most certifiers struggle to certify the robustness of ReLU NNs when trained with PGD. This suggests the power of DeepBern-Nets, which can be efficiently certified—in a few seconds even for NNs with millions of parameters, as shown in Table 5.1—using incomplete certifiers thanks to the ability of Bern-IBP to compute tight bounds.

5.4.2 Experiment 2: Certified training using Bern-IBP

In this experiment, we demonstrate that the tight bounds calculated by Bern-IBP can be utilized for certified training, achieving state-of-the-art results. Although a direct comparison with methods from certified training literature is not feasible due to the use of Bernstein polynomial activations instead of ReLU activations, we provide a comparison with state-of-the-art certified accuracy results from the SOK benchmark [2] to study how effectively can Bern-IBP be utilized for certified training. We trained neural networks with the same architectures as those in the benchmark to maintain a similar number of parameters, with the polynomial order serving as an additional hyperparameter. The training objective adheres to the certified training literature [81], incorporating the bound on the robustness loss in the objective as follows:

$$\min_{\theta} \underset{(\boldsymbol{x},y)\in(X,Y)}{\mathbb{E}} \left[(1-\lambda)\mathcal{L}_{\text{CE}}(\mathcal{NN}_{\theta}(\boldsymbol{x}), y; \theta) + \lambda \mathcal{L}_{\text{RCE}}(S(\boldsymbol{x}, \epsilon), y; \theta))\right], \tag{5.10}$$

where $\boldsymbol{x}$ is a data point, y is the ground truth label, $\lambda \in [0,1]$ is a weight to control the certified training regularization, $\mathcal{L}_{\text{CE}}$ is the cross-entropy loss, θ is the NN parameters, and $\mathcal{L}_{\text{RCE}}$ is computed by evaluating $\mathcal{L}_{\text{CE}}$ on the upper bound of the logit differences computed[81] using a bounding method.

For DeepBern-Nets, $\mathcal{L}_{\text{RCE}}$ is computed using Bern-IBP during training, while the networks in the SOK benchmark are trained using CROWN-IBP [81]. Table 5.2 illustrates that employing Bern-IBP bounds for certified training yields state-of-the-art certified accuracy (certified with Bern-IBP) on these datasets, comparable to—or in many cases surpassing—the performance of ReLU networks. The primary advantage of using Bern-IBP lies in its ability to compute highly precise bounds using a computationally cheap method, unlike the more sophisticated bounding methods for ReLU networks, such as α-Crown. For more details about the exact architecture of the NNs, please refer to Appendix B.3.2

Table 5.2: A comparison of certified accuracy for NNs with Bernstein polynomial activations versus ReLU NNs as in the SOK benchmark [2]. The certified accuracy is computed using Bern-IBP for NNs with polynomial activations, and the method yielding highest certified accuracy as reported in SOK for ReLU NNs. The table highlights the effectiveness of Bern-IBP in achieving competitive certification while utilizing a very computationally cheap method for tight bound computation.

Model	MNIST Certified acc. (%)				CIFAR-10 Certified acc. (%)			
	$\epsilon = 0.1$		$\epsilon = 0.3$		$\epsilon = 2/255$		$\epsilon = 8/255$	
	DeepBern-Net (%)	SOK (%)	DeepBern-Net (%)	SOK (%)	DeepBern-Net (%)	SOK (%)	DeepBern-Net (%)	SOK (%)
FCNNa	**72**	68	**31**	25	**38**	33	**28**	27
FCNNb	**86**	85	**57**	54	**39**	37	**26**	25
FCNNc	**80**	80	**51**	22	**36**	32	**31**	30
CNNa	**95**	95	82	**88**	45	**46**	31	**34**
CNNb	**95**	94	77	**85**	**49**	49	**37**	35
CNNc	87	**89**	72	**87**	38	**51**	32	**38**

5.4.3 Experiment 3: Tight reachability analysis of NN-controlled Quadrotor using Bern-IBP

In this experiment, we study the application-level impact of using Bernstein polynomial activations in comparison to ReLU activations with respect to the tightness of reachable sets in the context of safety-critical applications. Specifically, we consider a 6D linear dynamics system $\dot{x} = Ax + Bu$ representing a Quadrotor (used in [82, 83, 84]), controlled by a nonlinear NN controller where $u = \mathcal{NN}(x)$. To ensure a fair comparison, both sets of networks are trained on the same datasets, using the same architectures and training procedures. The only difference between the two sets of networks is the activation function used (ReLU vs. Bernstein polynomial).

After training, we perform reachability analysis with horizon $T = 6$ on each network using the respective bounding methods: Crown and α-Crown for ReLU networks and the proposed Bern-IBP for Bernstein polynomial networks. We compute the volume of the reachable sets after each step for each network. The results are visualized in Figure 5.3, comparing the error in the volume of the reachable sets for both ReLU and Bernstein polynomial networks. The error is computed with respect to the true volume of the reachable set for each network, which is computed by heavy sampling. As shown in Figure 5.3, using Bern-IBP on the NN with Bernstein polynomial can lead to much tighter reachable sets compared to SOTA bounding methods for ReLU networks. This experiment provides insights into the potential benefits of using Bernstein polynomial activations for improving the tightness of reachability bounds, which can have significant implications for neural network certification for safety-critical systems.

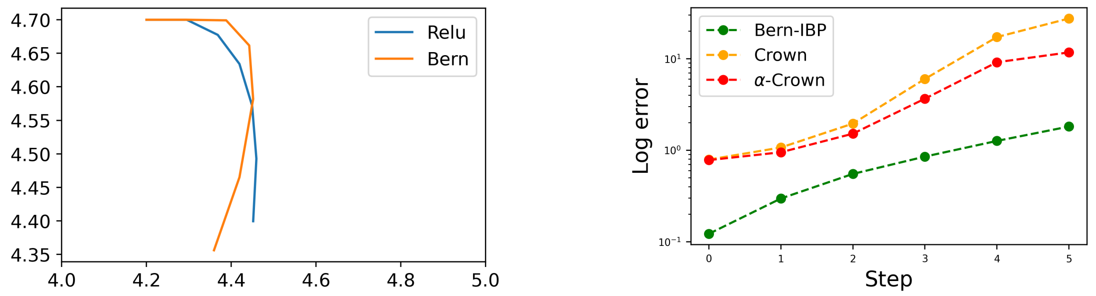


Figure 5.3: (Left) The trajectory of the Quadrotor for the ReLU and Bernstein polynomial networks. (Right) the error in the reachable set volume $e = (\hat{V} - V)/V$ for each of the networks after each step. $\hat{V}$ is the estimated volume using the respective bounding method and V is the true volume of the reachable set using heavy sampling

5.5 Discussion and limitations

Societal impact. The societal impact of utilizing Bernstein polynomial activations in neural networks lies in their potential to enhance the reliability and interpretability of AI systems, enabling improved safety, fairness, and transparency in various real-world applications.

Limitations. While Bernstein polynomials offer advantages in the context of certification, they also pose some limitations. One limitation is the increased computational complexity during training compared to ReLU networks.

Chapter 6

CertiFair: A Framework for Certified Global Fairness of Neural Networks

We consider the problem of whether a Neural Network (NN) model satisfies global individual fairness. Individual Fairness (defined in [85]) suggests that *similar* individuals with respect to a certain task are to be treated *similarly* by the decision model. In this work, we have two main objectives. The first is to construct a verifier which checks whether the fairness property holds for a given NN in a classification task or provide a counterexample if it is violated, i.e., the model is fair if all similar individuals are classified the same, and unfair if a pair of similar individuals are classified differently. To that end, We construct a sound and complete verifier that verifies global individual fairness properties of ReLU NN classifiers using distance-based similarity metrics. The second objective of this paper is to provide a method for training provably fair NN classifiers from unfair (biased) data. We propose a fairness loss that can be used during training to enforce fair outcomes for similar individuals. We then provide provable bounds on the fairness of the resulting NN. We run experiments on commonly used fairness datasets that are publicly available and we show that global individual fairness can be improved by 96 % without significant drop in test accuracy.

6.1 Introduction

Neural Networks (NNs) have become an increasingly central component of modern decision-making systems, including those that are used in sensitive/legal domains such as crime prediction [86], credit assessment [87], income prediction [87], and hiring decisions. However, studies have shown that these systems are prone to biases [61] that deem their usage *unfair* to unprivileged users based on their age, race, or gender. The bias is usually either inherent in the training data or introduced during the training process. Mitigating algorithmic bias has been studied in the literature [59, 60, 88] in the context of group and individual fairness. However, the fairness of the NN is considered only *empirically* on the test data with the hope that it represents the underlying data distribution.

Unlike the *empirical* techniques for fairness, we are interested to provide *provable certificates* regarding the fairness of a NN classifier. In particular, we focus on the "global individual fairness" property which states that a NN classification model is globally individually fair if *all similar* pairs of inputs x, x' are assigned the same class. We use a feature-wise closeness metric instead of an ℓ_p norm to evaluate similarity between individuals, i.e, a pair x, x' is similar if for all features i, $|x_i - x'_i| \leq \delta_i$. Given this fairness notion, the objective of this paper is twofold. First, it aims to provide a sound and complete formal verification framework that can automatically certify whether a NN satisfy the fairness property or produce a concrete counterexample showing two inputs that are not treated fairly by the NN. Second, this paper provides a training procedure for certified fair training of NNs even when the training data is biased.

Challenge: Several existing techniques focus on generalizing ideas from *adversarial robustness* to reason about NN fairness [89, 90]. By viewing unfairness as an adversarial noise that can flip the output of a classifier, these techniques can certify the fairness of a NN *locally*, i.e., in the neighborhood of a given individual input. In contrast, this paper focuses on *global* fairness properties where the goal is to ensure that the NN is fair with respect to *all* the similar inputs in its domain. Such a fundamental difference precludes the use of existing techniques from the literature on adversarial robustness and calls for novel techniques that can provide provable fairness guarantees.

This work: We introduce CertiFair, a framework for certified global fairness of NNs. This framework consists of two components. First, a verifier that can prove whether the NN satisfies the fairness property or produce a concrete counterexample that violates the fairness property. This verifier is motivated by the recent results in the "relational verification" problem [91] where the goal is to verify *hyperproperties* that are defined over pairs of program traces. Our approach is based on the observation that the global individual fairness property (6.1) can be seen as a *hyperproperty* and hence we can generalize the concept of *product programs* to *product NNs* that accepts a pair of inputs (x, x'), instead of a single input x, and generates two independent outputs for each input. A global fairness property for this product NN can then be verified using existing NN verifiers. Moreover, and inspired by methods in certified robustness, we also propose a training procedure for *certified fairness* of NNs. Thanks again to the product NN, mentioned above, one can establish upper bounds on fairness and use it as a regularizer during the training of NNs. Such a regularizer will promote the fairness of the resulting model, even if the data used for training is biased and can lead to an unfair classifier. While such *fairness regularizer* will enhance the fairness of the model, one needs to check if the fairness property holds globally using the sound and complete verifier mentioned above.

Contributions: Our main contributions are:

- To the best of our knowledge, we present the first sound and complete NN verifier for global individual fairness properties.

- A method for training NN classifiers with a modified loss that enforces fair outcomes for similar individuals. We provide bounds on the loss in fairness which constructs a certificate on the fairness of the trained NN.
- We applied our framework to common fairness datasets and we show that global fairness can be achieved with a minimal loss in performance.

6.2 Preliminaries

Our framework supports regression and multi-class classification models, however for simplicity, we only present our framework for binary classification models $h : \mathbb{R}^n \rightarrow \{0, 1\}$ of the form $h(x) = t(f_\theta(x))$ where t is a threshold function with threshold equals to 0.5. Moreover, we assume f_θ is an L-layer NN with ReLU hidden activations and parameters $\theta = ((W_1, b_1), \dots, (W_L, b_L))$ where (W_i, b_i) denotes the weights and bias of the ith layer. We also assume the activation function of the last layer of f_θ is a sigmoid function. The NN accepts an input vector x where the components $x_i \in \mathbb{R}$ (the set of real numbers) or $x_i \in \mathbb{Z}$ (the set of integer numbers). This is suitable for most of the datasets where some features of the input are numerical while others are categorical. In this paper, we are interested in the following fairness property:

Definition 6.2.1 (Global Individual Fairness [85, 92]). *A model $f_\theta(x)$ is said to satisfy the global individual fairness property ϕ if the following holds:*

$$\forall x, x' \in D_\phi \quad s.t. \quad d(x, x') = 1 \Longrightarrow h(f_\theta(x)) = h(f_\theta(x')), \tag{6.1}$$

where $d : \mathcal{R}^n \times \mathcal{R}^n \rightarrow \{1, 0\}$ is a similarity metric that evaluates to 1 when x and x' are similar and D_ϕ is the input domain of x for property ϕ defined as $D_\phi := D_\phi^0 \times ... \times D_\phi^{n-1}$ with

$D_\phi^i := \{x_i \mid l_i \leq x_i \leq u_i\}$ for some bounds l_i and u_i.

In this paper, we utilize the feature-wise similarity metric d defined as:

$$d(x, x') = \begin{cases} 1 & \text{if } |x_i - x'_i| \leq \delta_i \qquad \forall i \in \{1, \dots n\} \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

This feature-wise similarity metric allows the fairness property ϕ in (6.1) to capture several other fairness properties as special cases as follows:

Definition 6.2.2 (Individual discrimination [93])**.** *A model $f_\theta(x)$ is said to be nondiscriminatory between individuals if the following holds:*

$$\forall x = (x_s, x_{ns}), x' = (x'_s, x'_{ns}) \in D_\phi \quad s.t. \quad x_{ns} = x'_{ns} \text{ and } x_s \neq x'_s \implies h(f_\theta(x)) = h(f_\theta(x')),$$

where x_s and x_{ns} denotes the sensitive attributes and non-sensitive attributes of x, respectively.

Indeed, the individual discrimination corresponds to a global individual fairness property by setting $\delta_i = 0$ in (6.2) for the non-sensitive attributes. Another definition of fairness[90] states that two individuals are similar if their numerical features differ by no more than α. Again, this can be represented by the closeness metric simply by setting $\delta_i = 0$ for categorical attributes and $\delta_i = \alpha$ for numerical attributes.

Based on Definition 1 above, we can formally verify whether the fairness property ϕ holds by checking if the set of counterexamples (or violations) $\mathcal{C}$ is empty, where $\mathcal{C}$ is defined as:

$$\mathcal{C} = \left\{ (x, x') \middle| x, x' \in D_\phi, \bigwedge_{i=0}^{n-1} |x_i - x'_i| < \delta_i, h(x) \neq h(x') \right\} = \emptyset. \tag{6.3}$$

6.3 Global Individual Fairness as a hyperproperty

In this section, we draw the connection between the verification of global individual fairness properties (6.1) and hyperproperties in the context of program verification. On the one hand, several local properties of NNs (e.g., adversarial robustness) are considered trace properties, i.e., properties defined on the input-output behavior of the NN. In this case, one can search the input space of the NN to find a *single* input (or counterexample) that leads to an output that violates the property. In the domain of adversarial robustness, a counterexample corresponds to a disturbance to an input that can change the classification output of a NN. On the other hand, other properties, like the global fairness properties, can not be modeled as trace properties. This stems from the fact that one can not judge the correctness of the NN by considering individual inputs to the NN. Instead, finding a counterexample to the fairness property will entail searching over *pairs* of inputs and comparing the NN outputs of these inputs. Properties that require examining pairs or sets of traces (input-outputs of a program) are defined as *hyperproperties* [91].

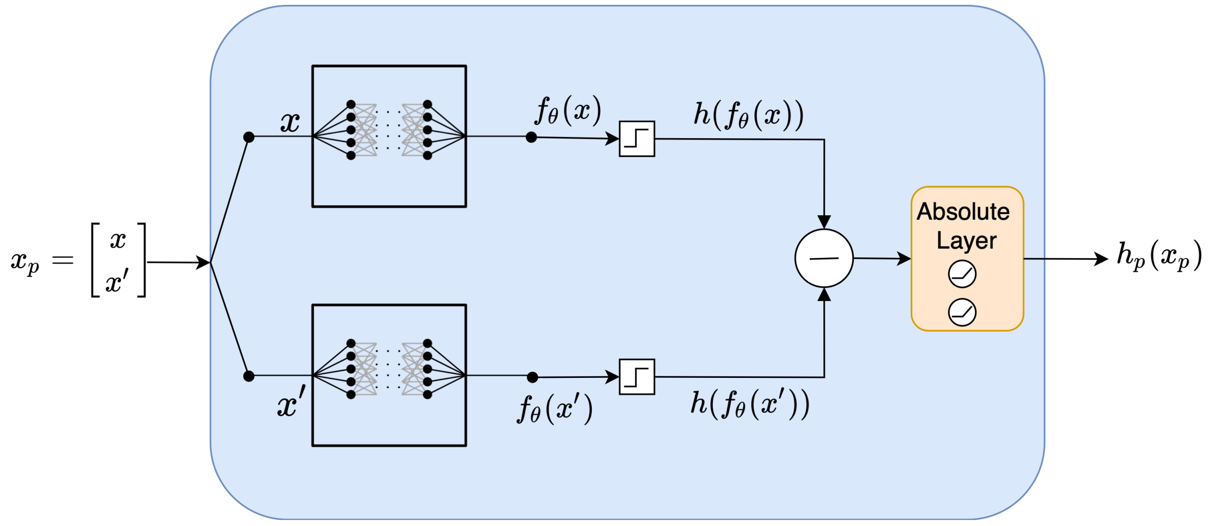


Figure 6.1: Construction of the Product NN.

Modeling global individual fairness as a hyperproperty leads to a direct certification framework. In particular, a key idea in the hyperproperty verification literature is the notion of a product program that allows the reduction of the hyperproperty verification problem to a standard verification problem [91]. A product program is constructed by composing two copies of the original program together. The main benefit is that the hyperproperties of the original program become trace properties of the product program that can be verified using standard techniques. Motivated by this observation, our framework CertiFair generalizes the concept of *product programs* into *product NNs* (described in detail in Section 6.4.2 and shown in Figure 6.1) that accepts a pair of inputs and generates a pair of two independent outputs. We then use the product network to verify fairness (hyper)properties using standard techniques.

6.4 CertiFair: A Framework for Certified Fairness of Neural Networks

As mentioned earlier in section 6.3, the fairness property can be viewed as a hyperproperty of the NN. We propose the use of a product NN that can reduce the verification of such hyperproperty into standard trace (input/output) property. In this section, we first explain how to construct the product NN followed by how to use it to encode the fairness verification problem into ones that are accepted by off-the-shelf NN verifiers. Next, we discuss how to use this product NN to derive a fairness regualrizer that can be used during training to obtain a certified fair NN.

6.4.1 Product Neural Network

Given a neural network f_θ, the product network f_{θ_p} is basically a side-to-side composition of f_θ with itself. More formally, the parameters vector θ_p of the product NN is defined as:

$$\theta_p = \left(\left(\begin{bmatrix} W_1 & \mathbf{0} \\ \mathbf{0} & W_1 \end{bmatrix}, \begin{bmatrix} b_1 \\ b_1 \end{bmatrix} \right), \ldots, \left(\begin{bmatrix} W_L & \mathbf{0} \\ \mathbf{0} & W_L \end{bmatrix}, \begin{bmatrix} b_L \\ b_L \end{bmatrix} \right) \right) \tag{6.4}$$

where (W_i, b_i) are the weights and biases of the ith layer of f_θ. The input to the product network f_{θ_p} is a pair of concatenated inputs $x_p = (x, x')$. Finally, we add an output layer that results in an output $h_p \in \{0, 1\}$ defined as: $h_p(f_{\theta_p}(x_p)) = |h(f_\theta(x)) - h(f_\theta(x'))|$ where the absolute value operator $|.|$ can be implemented using ReLU nodes by noticing that $|a| = max(a, 0) + max(-a, 0)$. Figure 6.1 summarizes this construction.

6.4.2 Fairness Verification

Using the product network defined above, we can rewrite the set of counterexamples $\mathcal{C}$ in (6.3) as:

$$\mathcal{C}_p = \left\{ x_p \middle| x_p \in D_\phi \times D_\phi, \ \bigwedge_{i=0}^{n-1} |x_i - x'_i| < \delta_i, \ h_p(x_p) > 0 \right\} \tag{6.5}$$

which corresponds to the standard verification of NN input-output properties in equation (2.2), albeit being defined over the product network inputs and outputs..

To check the emptiness of the set $\mathcal{C}_p$ in (6.5) (and hence certify the global individual fairness property), we need to search the space $D_\phi \times D_\phi$ to find at least one counterexample that violates the fairness property, i.e., a pair $x_p = (x, x')$ that represent similar individuals who are classified differently by the NN. Finding such a counterexample is, in general, NP-hard due to the non-convexity of the ReLU NN f_{θ_p}. To that end, we use PeregriNN [38] as our

NN verifier. Briefly, PeregriNN overapproximates the highly non-convex NN with a linear convex relaxation for each ReLU activation. This is done by introducing two optimization variables for each ReLU, a pre-activation variable $\hat{y}$ and a post-activation variable y. The non-convex ReLU function can then be overapproximated by a triangular region of three constraints; $y \geq 0$, $y \geq \hat{y}$, and $y \leq \frac{u}{u-l}(\hat{y} - l)$, where l,u are the lower and upper bounds of $\hat{y}$ respectively. The solver tries to check whether the approximate problem has no solution or iteratively refine the NN approximation until a counterexample that violates the fairness constraint is found. PeregriNN employs other optimizations in the objective function to guide the refinement of the NN approximation but the details of these methods are beyond the scope of this paper. We refer the reader to chapter 3 for more details on the internals of the solver.

Proposition 6.4.1. *Consider a NN model f_θ and a fairness property ϕ—either representing a Global Individual Fairness property (Definition 6.2.1) or an Individual Discrimination property (Definition 6.2.2). Consider a set of counterexamples $\mathcal{C}_p$ computed using a NN verifier applied to the product network f_{θ_p}. The NN satisfies the property ϕ whenever the set $\mathcal{C}_p$ is empty.*

Proof. This result follows directly from the equivalence between the sets $\mathcal{C}$ in (6.3) and $\mathcal{C}_p$ in (6.5) along with the NN verifiers (like PeregriNN) being sound and complete and hence capable of finding any counterexample, if one exists. □

6.4.3 Certified Fair Training

In this section, we formalize a *fairness regularizer* that can be used to train certified fair models. In particular, we propose two fairness regularizers that correspond to *local* and *global* individual fairness. We provide the formal definitions of both these regularizers below and their characteristics.

Local Fairness Regularizer $\mathcal{L}_f^l$ using Robustness around Training Data: Our first proposed regularizer is motivated by the *robustness* regularizers used in the literature of certified robustness [26, 81]. The regualrizer, denoted by $\mathcal{L}_f^l$, aims to minimize the average loss in fairness across all training data. More formally, given a training point (x, y) and NN parameters θ, let $\mathcal{L}(f_\theta(x), y; \theta) = -[y \log(f_\theta(x)) + (1 - y) \log(1 - f_\theta(x))]$ be the standard binary cross-entropy loss. The fairness regularizer $\mathcal{L}_f^l$ can then be defined as:

$$\begin{aligned}\mathcal{L}_f^l(\theta) &= \mathbb{E}_{(x,y)\in(X,Y)}\left[\max_{\substack{x'\in D_\phi\\ d(x,x')=1}} \mathcal{L}(f_\theta(x'), y; \theta)\right] \\ &= \mathbb{E}_{(x,y)\in(X,Y)}\left[\max_{x'\in S_\phi(x)} -y\log(f_\theta(x')) - (1-y)\log(1-f_\theta(x'))\right]\end{aligned} \tag{6.6}$$

In other words, the regularizer above aims to measure the expected value (across the training data) for the worst-case loss of fairness due to points x' that are assigned to different classes. Indeed, the regualrizer (6.6) is not differentiable (with respect to the weights θ) due to the existence of the max operator. Nevertheless, one can compute an upper bound of (6.6) and aims to minimize this upper bound instead. Such upper bound can be derived as follows:

$$\max_{\substack{x'\in D_\phi\\ d(x,x')=1}} \mathcal{L}(f_\theta(x'), y; \theta) = \max_{\substack{x'\in D_\phi\\ d(x,x')=1}} \begin{cases} -\log(1 - f_\theta(x')) & \text{if } y=0 \\ -\log(f_\theta(x')) & \text{if } y=1 \end{cases} \tag{6.7}$$

$$\leq \begin{cases} -\log(1 - \theta^T \overline{w}_{S_\phi(x)}) & \text{if } y=0 \\ -\log(\theta^T \underline{w}_{S_\phi(x)}) & \text{if } y=1 \end{cases} \tag{6.8}$$

where $\theta^T \overline{w}_{S_\phi(x)}$ and $\theta^T \underline{w}_{S_\phi}$ are the linear upper/lower bound of $f_\theta(x')$ inside the set $S_\phi(x) = \{x' \in D_\phi | d(x, x') = 1\}$. Such linear upper/lower bound of $f_\theta(x')$ can be computed using off-the-shelf bounding techniques like Symbolic Interval Analysis [4] and α-Crown [49]. We denote by $\overline{\mathcal{L}}(y; \theta)$ the right hand side of the inequality in (6.8) which depends only on the label y and the NN parameters θ. Now the fairness property can be incorporated in training by optimizing the following problem over θ (the NN parameters):

$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\in(X,y)} \left[(1-\lambda_f)\underbrace{\mathcal{L}(f_\theta(x), y; \theta)}_{\text{natural loss}} + \lambda_f \underbrace{\overline{\mathcal{L}}(y;\theta)}_{\text{local fairness loss}} \right], \tag{6.9}$$

where λ_f is a regularization parameter to control the trade-off between fairness and accuracy.

Although easy to compute and incorporate in training, the regularizer $\mathcal{L}^l_f(\theta)$ (and its upper bound) defined above suffers from a significant drawback. It focuses on the fairness *around* the samples presented in the training data. In other words, although the aim was to promote *global* fairness, this regularizer is effectively penalizing the training only in the *local* neighborhood of the training data. Therefore, its effectiveness depends greatly on the quality of the training data and its distribution. Poor data distribution may lead to the poor effect of this regularizer. Next, we introduce another regularizer that avoids such problems.

Global Fairness Regularizer $\mathcal{L}^g_f$ using Product Network: To avoid the dependency on data, we introduce a novel fairness regularizer capable of capturing global fairness during the training. Such a regularizer is made possible thanks to the product NN defined above. In particular, the global fairness regularizer $\mathcal{L}^g_f(\theta)$ is defined as:

$$\mathcal{L}^g_f(\theta) = \max_{\substack{(x,x')\in D_\phi \times D_\phi \\ d(x,x')=1}} |f_\theta(x) - f_\theta(x')| \tag{6.10}$$

In other words, the regualizer $\mathcal{L}^g_f(\theta)$ in (6.10) aims to penalize the worst case loss in global fairness. Similar to (6.6), the $\mathcal{L}^g_f(\theta)$ is also non-differentiable with respect to θ. Nevertheless, and thanks to the product NN, we can upper bound $\mathcal{L}^g_f(\theta)$ as:

$$\mathcal{L}^g_f(\theta) \le \max_{(x,x')\in D_\phi \times D_\phi} |f_\theta(x) - f_\theta(x')| = \max_{x_p \in D_\phi \times D_\phi} f_p(x_p) \le \theta^T \overline{w}_{D_\phi} \tag{6.11}$$

where $\theta^T \overline{w}_{D_\phi}$ is the linear upper bound of the product network among the domain $D_\phi \times D_\phi$. Again, such bound can be computed using Symbolic Interval Analysis and α-Crown on the product network after replacing the output h_p with $f_p = |f_\theta(x) - f_\theta(x')|$. It is crucial to note that the upper bound in (6.11) depends only on the domain D_ϕ. Hence, the fairness property can now be incorporated into training by minimizing this upper bound as:

$$\min_\theta \underset{(x,y)\in(X,y)}{\mathbb{E}} \left[(1-\lambda_f) \underbrace{\mathcal{L}(f_\theta(x), y; \theta)}_{\text{natural loss}} \right] + \lambda_f \underbrace{\theta^T \overline{w}_{D_\phi}}_{\text{global fairness loss}}, \tag{6.12}$$

where the fairness loss is now outside the $\mathbb{E}[\ .\ ]$ operator.

In the next section, we show that the global fairness regularizer $\mathcal{L}_f^g(\theta)$ empirically outperforms the local fairness regularizer $\mathcal{L}_f^l(\theta)$. We end up our discussion in this section with the following result:

Proposition 6.4.2. *Consider a NN model f_θ and a fairness property ϕ—either representing a Global Individual Fairness property (Definition 6.2.1) or an Individual Discrimination property (Definition 6.2.2). Consider a NN model f_θ trained using the objective function in* (6.12). *If $\theta^T \overline{w}_{D_\phi} = 0$ by the end of the training, then the resulting f_θ is guaranteed to satisfy ϕ.*

Proof. The result follows directly from equation (6.11). □

Indeed, the result above is just a sufficient condition. In other words, the NN may still satisfy the fairness property ϕ even if $\theta^T \overline{w}_{D_\phi} > 0$. Such cases can be handled by applying the verification procedure in Section 6.4.2 after training the NN.

6.5 Experimental evaluation

We present an experimental evaluation to study the effect of our proposed fairness regularizers and hyperparameters on the global fairness. We evaluated CertiFair on four widely investigated fairness datasets (Adult [87], German [87], Compas [94], and Law School [95]). All datasets were pre-processed such that any missing rows or columns were dropped, features were scaled so that they're between $[0, 1]$, and categorical features were one-hot encoded.

Implementation: We implemented our framework in a Python tool called CertiFair that can be used for training and verification of NNs against an individual fairness property. CertiFair uses Pytorch for all NN training and a publicly available implementation of PeregriNN [38] as a NN verifier. We run all our experiments using a single GeForce RTX 2080 Ti GPU and two 24-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz (only 8 cores were used for these experiments).

Table 6.1: Comparison between verifying local and global fairness properties on the Adult dataset for different similarity constraint (distance δ_i).

δ_i	Certified local fairness (%)	Certified global fairness (%)
0.02	89.25	6.56
0.03	100.00	65.98
0.05	81.42	6.40
0.07	100.00	57.39
0.1	99.95	66.35

Measuring global fairness using verification: While the verifier (described in Section 6.4.2) is capable of finding concrete counterexamples that violate the fairness, it is also important to *quantify* a bound on the fairness. In these experiments, the certified global fairness is quantified as the percentage of partitions of the input space with zero counterexamples. In particular, the input space is partitioned using the categorical features, i.e., the number of partitions is equal to the number of different categorical assignments and each partition corresponds to one categorical assignment. Note that the numerical features don't have to be fixed inside each partition (property dependant). To verify the property globally, we run the verifier on each partition of the input space and verify the fairness property. Finally, we count the number of verified fair partitions (normalized by the total number of partitions).

Fairness properties: In the experimental evaluation, we consider two classes of fairness properties. The first class $\mathcal{P}_1$ is the one in definition 6.2.2 where two individuals are similar if they only differ in their sensitive attribute. The second class of properties $\mathcal{P}_2$ relaxes the first by also allowing numerical attributes to be close (not identical), this is allowed under definition 6.2.1 of global individual fairness by setting $\delta_i > 0$ for numerical attributes.

6.5.1 Experiment 1: Global Individual Fairness vs. Local Individual Fairness

In this experiment, we empirically show that NNs with high *local* individual fairness does not necessarily result into NNs with *global* individual fairness. In particular, we train a NN on the Adult dataset and considered multiple fairness properties (all from class $\mathcal{P}_2$ defined above) by varying δ_i. Note that δ_i is equal for all features i within the same property, but is different from one property to another. Next, we use PeregeriNN verifier to find counterexamples for both the *local* fairness (by applying the verifier to the trained NN) and the *global* fairness (by applying the verifier to the product NN). We measure the fairness of the NN for both cases and report the results in Table 6.1. The results indicate that verifying *local* fairness may result in incorrect conclusions about the fairness of the model. In particular, rows1-4 in the table shows that counterexamples were not found in the neighborhood of the training data (reflected by the 100% certified local fairness), yet verifying the product NN was capable of finding counterexamples that are far from the training data leading to accurate conclusions

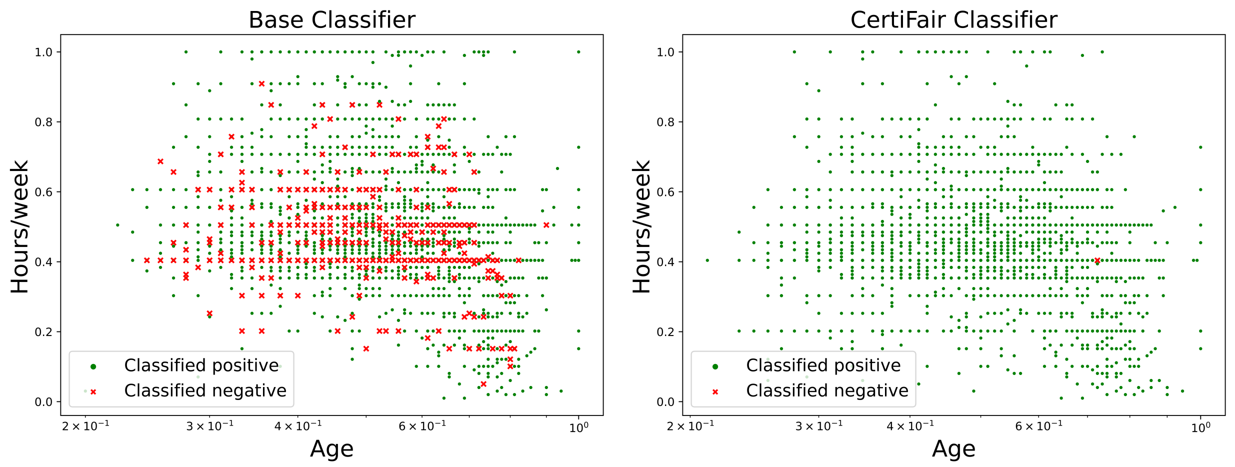


Figure 6.2: Comparison between the base and CertiFair classifiers in terms of fairness as defined in 6.2.2. We show the classifications for the minority group of the adult dataset projected on two features; Age, and hours worked per week. The figure shows that the base classifier suffers from biases against identical individuals who are of different race (red markers). CertiFair is able to drastically improve the individual fairness on this dataset with only 2% reduction in accuracy.

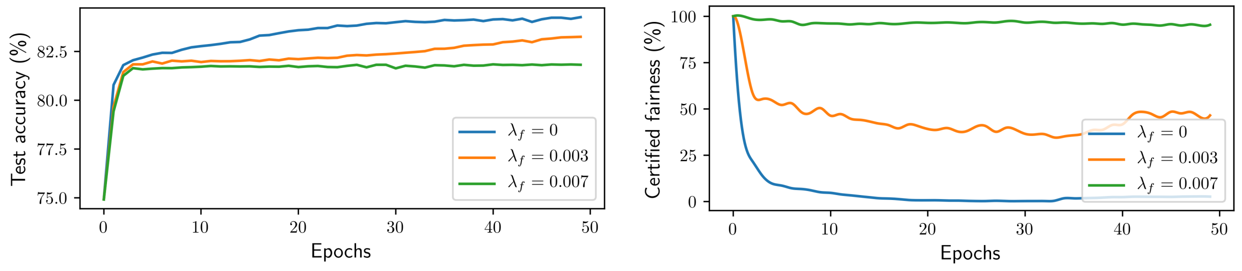


Figure 6.3: Test accuracy (left) and certified fairness (right) across training epochs when training a NN with the fairness constraint ($\lambda_f = 0.007$) and without it ($\lambda_f = 0$) on the Adult dataset.

about the NN fairness.

6.5.2 Experiment 2: Effects of Incorporating the Fairness Regularizer

We investigate the effect of using the global fairness regularizer (defined in (6.11)) on the decisions of the NN classifier when trained on the Adult dataset. The fairness property for this experiment is of class $\mathcal{P}_1$. To investigate the predictor's bias, we first project the data points on two numerical features (age and hours/week). Our objective is to check whether the points that are classified positively for the privileged group are also classified positively for the non privileged group. Figure 6.2 (left) shows the predictions for the unprivileged group when using the base classifier ($\lambda_f = 0$). Green markers indicate points for which individuals from both privileged and non privileged groups are treated equally while red markers show individuals from the non privileged group that did not receive the same NN output compared to the corresponding identical ones in the privileged group. Figure 6.2 (right) shows the same predictions but using the fair classifier ($\lambda_f = 0.03$), the predictions from this classifier drastically decreased the discrimination between the two groups while only decreasing the accuracy by 2%. These results suggest that we can indeed regularize the training to improve the satisfaction of the fairness constraint without a drastic change in performance.

We also investigate how the certified fairness changes across epochs of training. To that end, we train a NN for the German dataset and evaluate the test accuracy as well as the certified global fairness after each epoch of training for two different values of λ_f. Figure 6.3 shows the underlying trade-off between achieving fairness versus maximizing accuracy. As expected, lower values of λ_f results in lower loss in accuracy (compared to the base case with $\lambda_f = 0$) while having lower effect on the fairness. The results also show that a small sacrifice of the accuracy can lead to significant enhancement of the fairness as shown for the $\lambda_f = 0.007$ case.

6.5.3 Experiment 3: Certified Fair Training

Experiment setup: The objective of this experiment is to compare the two regularizers, the local fairness regularizer in (6.8) and the global fairness regularizer in (6.11). To that end, we performed a grid search over the learning rate α, the fairness regularization parameter λ_f, and the NN architecture to get the best test accuracy across all datasets. Best performance was obtained with a NN that consists of two hidden layers of 20 neurons (except for the German dataset, where we use 30 neurons per layer), learning rate $\alpha = 0.001$, global fairness

Table 6.2: Comparison between a base classifier ($\lambda_f = 0$) and CertiFair classifier with different fairness regularizers

Constraint	Dataset	Test Accuracy (%)			Positivity Rate(%)			Certified Global Fairness (%)		
		Base	$\mathcal{L}_f^g$	$\mathcal{L}_f^l$	Base	$\mathcal{L}_f^g$	$\mathcal{L}_f^l$	Base	$\mathcal{L}_f^g$	$\mathcal{L}_f^l$
$\mathcal{P}_2$	Adult	84.55	82.34	83.81	20.8	17.4	21.79	6.40	100.00	61.92
	German	75.30	73.00	70.00	79.00	72.00	83.00	8.64	95.06	86.41
	Compas	68.30	65.08	63.19	61.55	66.00	69.40	47.22	100.00	100.00
	Law	87.60	79.92	78.69	21.51	9.10	25.03	6.87	51.87	78.54
$\mathcal{P}_1$	Adult	84.55	82.33	83.25	20.80	18.13	20.15	1.77	97.86	95.31
	German	75.30	72.66	69.66	79.00	83.14	81.71	14.81	92.59	82.71
	Compas	68.30	65.08	63.82	61.55	66.00	71.60	47.22	100.00	100.00
	Law	87.60	84.90	86.69	21.42	17.50	21.63	34.16	70.10	86.45

regularization parameter λ_f equal to 0.01 for Adult and Law School, 0.005 for German, and 0.1 for Compas dataset, and local fairness regularization parameter λ_f equal to 0.95 for Adult, 0.9 for Compas, 0.2 for German, and 0.5 for Law School. We trained the models for 50 epochs with a batch size of 256 (except for Law School, where the batch size is set to 1024) and used Adam Optimizer for learning the NN parameters θ. All the datasets were split into 70% and 30% for training and testing, respectively.

Effect of the choice of the fairness regularizer: We investigate the certified global fairness for the two regularizers introduced in Section 6.4.3. Table 6.2 summarizes the results for $\mathcal{P}_1$ and $\mathcal{P}_2$ fairness properties across different datasets. For each property and dataset, we compare the test accuracy, positivity rate (percentage of points classified as 1), and the certified global fairness of the base classifier (trained with $\lambda_f = 0$) and the CertiFair classifier trained twice with two different fairness regularizers $\mathcal{L}_f^l$ and $\mathcal{L}_f^g$. Compared to the base classifier, training the NN with the global fairness regularizer $\mathcal{L}_f^g$ significantly increases the certified global fairness with a small drop in the accuracy in most of the cases except for the Law School dataset, where the test accuracy dropped by 7 % on $\mathcal{P}_2$ but the global fairness increased by 55 %. Compared to the local regularizer $\mathcal{L}_f^l$, the global regularizer achieves higher global fairness and comparable (if not better) test accuracy on all datasets except Law

Table 6.3: Effect of fairness regularization parameter λ_f on the test accuracy and certified fairness.

λ_f	1×10^{-4}	5×10^{-4}	7×10^{-4}	5×10^{-3}	7×10^{-3}	1×10^{-2}	2×10^{-2}	5×10^{-2}
Test accuracy (%)	74.33	74.33	75.00	72.33	72.66	73.00	72.6	66.33
Certified global fairness (%)	8.64	14.81	13.58	66.66	82.71	95.06	98.76	100

School. We think that this might be due to the network's limited capacity to optimize both objectives. We also report the positivity rate (number of data points classified positively) for the classifiers. This metric is important because most of these datasets are unbalanced, and hence the classifiers can trivially skew all the classifications to a single label and achieve high fairness percentage. Thus it is desired that the positivity rate of the CertiFair classifier to be close to the one of the base classifier to ensure that it is not trivial. Lastly, we conclude that even though the local regularizer improves the global fairness, the global regularizer can achieve higher degrees of certified global fairness without a significant decrease in test accuracy, and of course, it avoids the drawbacks of the local regularizer discussed in Section 6.4.3.

Effect of the fairness regularization parameter: In this experiment, we investigate the effect of the fairness regularization parameter λ_f on the classifier's accuracy and fairness. The parameter λ_f controls the trade-off between the accuracy of the classifier and its fairness, and tuning this parameter is usually dependent on the network/dataset. To that end, we trained a two-layer NN with 30 neurons per layer for the German dataset using 8 different values for λ_f and summarized the results in Table 6.3. The fairness property verified is of class $\mathcal{P}_2$. The results show that the global fairness satisfaction can increase without a significant drop in accuracy up to a certain point, after which the fairness loss is dominant and results in a significant decrease in the classifier's accuracy.

6.6 Related work

Group fairness: Group fairness considers notions like demographic parity [96], equality of odds, and equality of opportunity [97]. Tools that verify notions of group fairness assume knowledge of a probabilistic model of the population. FairSquare [98] relies on numerical integration to formally verify notions of group fairness; however, it does not scale well for NNs. VeriFair [99] considers probabilistic verification using sampling and provides soundness guarantees using concentration inequalities. This approach is scalable to big networks, but it does not provide worst-case proof.

Individual fairness: More related to our work is the verification of individual fairness properties. LCIFR [90] proposes a technique to learn fair representations that are provably certified for a given individual. An encoder is *empirically* trained to map similar individuals to be within the neighborhood of the given individual and then apply NN verification techniques to this neighborhood to certify fairness. The property verified is a *local* property with respect to the given individual. On the contrary, our work focuses on the global fairness properties of a NN. It also avoids the empirical training of similarity maps to avoid affecting the soundness and completeness of the proposed framework. In the context of individual global fairness, a recent work [92] proposed a sound but incomplete verifier for linear and kernelized polynomial/radial basis function classifiers. It also proposed a meta-algorithm for the global individual fairness verification problem; however, it is not clear how it can be used to design sound and complete NN verifiers for the fairness properties. Another line of work [100] focuses on proving *dependency* fairness properties which is a more restrictive definition of fairness since it requires the NN outputs to avoid any dependence on the sensitive attributes. The method employs forward and backward static analysis and input space partitioning to verify the fairness property. As mentioned, this definition of fairness is different from the individual fairness we are considering in this work and is more restrictive.

NN verification: This work is algorithmically related to NN verification in the context of adversarial robustness. However, adversarial robustness is a local property of the network given a nominal input, and a norm bounded perturbation. Moreover, the robustness property does not consider the notion of sensitive attributes. The NN verification literature is extensive, and the approaches can be grouped into three main groups: (i) SMT-based methods, which encode the problem into a Satisfiability Modulo Theory problem [9, 10, 3]; (ii) MILP-based solvers, which solves the verification problem exactly by encoding it as a Mixed Integer Linear Program [16, 17, 12, 13, 15, 11, 14]; (iii) Reachability based methods [23, 24, 19, 22, 21, 20, 18], which perform layer-by-layer reachability analysis to compute a reachable set that can be verified against the property; and (iv) convex relaxations methods [4, 25, 26, 37, 38, 39]. Generally, (i), (ii), and (iii) do not scale well to large networks. On the other hand, convex relaxation methods use a branch and bound approach to refine the abstraction.

6.7 Discussion

On the contention between Group and Individual fairness: Group fairness is the requirement that different groups should be treated similarly regardless of individual merits. It is often thought of as a contradictory requirement to individual fairness. However, this has been an issue of debate [101]. Thus, it's not clear how our framework for training might affect the group fairness requirement and is left for further investigation.

Can fairness be achieved by dropping sensitive attributes from data? Fairness through unawareness is the process of learning a predictor that does not explicitly use sensitive attributes in the prediction process. However, dropping the sensitive attributes is not sufficient to remove discrimination as it can be highly predictable implicitly from other features. It has been shown [102, 103, 104, 105] that discrimination highly occurs in different systems such as housing, criminal justice, and education that do not explicitly depend on sensitive attributes in their predictions.

Chapter 7

Formal Verification of Neural Network Controlled Autonomous Systems

In this paper, 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 utilize a Satisfiability Modulo Convex (SMC) encoding to enumerate all the possible assignments of different ReLUs. To accelerate this process, we develop a pre-processing algorithm that could rapidly prune the space of feasible ReLU assignments. Finally, we demonstrate the efficiency of the proposed algorithms using numerical simulations with the increasing complexity of the neural network controller.

7.1 Introduction

From simple logical constructs to complex deep neural network models, Artificial Intelligence (AI)-agents are increasingly controlling physical/mechanical systems. Self-driving cars, drones, and smart cities are just examples of such systems to name a few. However, regardless of the explosion in the use of AI within a multitude of cyber-physical systems (CPS) domains, the safety and reliability of these AI-enabled CPS is still an under-studied problem. It is then unsurprising that the failure of these AI-controlled CPS in several, safety-critical, situations leads to human fatalities [106].

Motivated by the urgency to study the safety, reliability, and potential problems that can rise and impact the society by the deployment of AI-enabled systems in the real world, several works in the literature focused on the problem of designing deep neural networks that are robust to the so-called adversarial examples [107, 108, 109, 110, 111, 112, 113]. Unfortunately, these techniques focus mainly on the robustness of the learning algorithm with respect to data outliers without providing guarantees in terms of safety and reliability of the decisions made by these neural networks. To circumvent this drawback, recent works focused on three main techniques namely (i) testing of neural networks, (ii) falsification (semi-formal verification) of neural networks, and (iii) formal verification of neural networks.

Representatives of the first class, namely testing of neural networks, are the works reported in [114, 115, 116, 117, 118, 119, 120, 121, 122, 123] in which the neural network is treated as a white box, and test cases are generated to maximize different coverage criteria. Such coverage criteria include neuron coverage, condition/decision coverage, and multi-granularity testing criteria. On the one hand, maximizing test coverage gives system designers confidence that the networks are reasonably free from defect. On the other hand, testing does not formally guarantee that a neural network satisfies a formal specification.

To take into consideration the effect of the neural network decisions on the entire system behavior, several researchers focused on the falsification (or semi-formal verification) of autonomous systems that include machine learning components [124, 125, 126]. In such falsification frameworks, the objective is to generate corner test cases that forces a violation of system-level specifications. To that end, advanced 3D models and image environments are used to bridge the gap between the virtual world and the real world. By parametrizing the input to these 3D models (e.g., position of objects, position of light sources, intensity of light sources) and sampling the parameter space in a fashion that maximizes the falsification of the safety property, falsification frameworks can simulate several test cases until a counterexample is found [124, 125, 126].

While testing and falsification frameworks are powerful tools to find corner cases in which the neural network or the neural network enabled system may fail, they lack the rigor promised by formal verification methods. Therefore, several researchers pointed to the urgent need of using formal methods to verify the behavior of neural networks and neural network enabled systems [127, 128, 129, 130, 131, 132]. As a result, recent works in the literature attempted the problem of applying formal verification techniques to neural network models.

Applying formal verification to neural network models comes with its unique challenges. First and foremost is the lack of widely-accepted, precise, mathematical specifications capturing the correct behavior of a neural network. Therefore, recent works focused entirely on verifying neural networks against simple input-output specifications [133, 3, 134, 135, 136, 137]. Such input-output techniques compute a guaranteed range for the output of a deep neural network given a set of inputs represented as a convex polyhedron. To that end, several algorithms that exploit the piecewise linear nature of the Rectified Linear Unit (ReLU) activation functions (one of the most famous nonlinear activation functions in deep neural networks) have been proposed. For example, by using binary variables to encode piecewise linear functions, the constraints of ReLU functions are encoded as a Mixed-Integer Linear Programming (MILP). Combining output specifications that are expressed in terms of Linear Programming (LP), the verification problem eventually turns to a MILP feasibility problem [136, 138].

Using off-the-shelf MILP solvers does not lead to scalable approaches to handle neural networks with hundreds and thousands of neurons [3]. To circumvent this problem, several MILP-like solvers targeted toward the neural network verification problem are proposed. For example, the work reported in [133] proposed a modified Simplex algorithm (originally used to solve linear programs) to take into account ReLU nonlinearities as well. Similarly, the work reported in [3] combines a Boolean satisfiability solving along with a linear over-approximation of piecewise linear functions to verify ReLU neural networks against convex specifications. Other techniques that exploit specific geometric structures of the specifications are also proposed [139, 23]. A thorough survey on different algorithms for verification of neural networks against input-output range specifications can be found in [140] and the references within.

Unfortunately, the input-output range properties are simplistic and fail to capture the safety and reliability of cyber-physical systems when controlled by a neural network. Recent works showed how to perform reachability-based verification of closed-loop systems in the presence of learning components [141, 20, 142]. Reachability analysis is performed by either separately estimating the output set of the neural network and the reachable set of continuous dynamics [141], or by translating the neural network controlled system into a hybrid system [20]. Once the neural network controlled system is translated into a hybrid system, off-the-shelf existing verification tools of hybrid systems, such as SpaceEx [143] for piecewise affine dynamics and Flow* [144] for nonlinear dynamics, can be used to verify safety properties of the system. Another related technique is the safety verification using barrier certificates [145]. In such approach, a barrier function is searched using several simulation traces to provide a certificate that unsafe states are not reachable from a given set of initial states.

Differently from the previous work—in the literature of formal verification of neural network controlled system—we consider, in this paper, the case in which the robotic system is equipped with a LiDAR scanner that is used to sense the environment. The LiDAR image is then processed by a neural network controller to compute the control inputs. Arguably, the ability of neural networks to process high-bandwidth sensory signals (e.g., cameras and LiDARs) is one of the main motivations behind the current explosion in the use of machine learning in robotics and CPS. Towards this goal, we develop a framework that can reason about the safety of the system while taking into account the robot continuous dynamics, the workspace configuration, the LiDAR imaging, and the neural network.

In particular, the contributions of this paper can be summarized as follows:
1- A framework for formally proving safety properties of autonomous robots equipped with LiDAR scanners and controlled by neural network controllers.
2- A notion of imaging-adapted partitions along with a polynomial-time algorithm for processing the workspace into such partitions. This notion of imaging-adapted partitions plays a significant role in capturing the LiDAR imaging process.
3- A Satisfiability Modulo Convex (SMC)-based algorithm combined with an SMC-based pre-processing for computing finite abstractions of neural network controlled autonomous systems.

7.2 Problem Formulation

7.2.1 Notation

The symbols $\mathbb{N}$, $\mathbb{R}, \mathbb{R}^+$ and $\mathbb{B}$ denote the set of natural, real, positive real, and Boolean numbers, respectively. The symbols $\wedge, \neg$ and $\rightarrow$ denote the logical AND, logical NOT, and logical IMPLIES operators, respectively. Given two real-valued vectors $x_1 \in \mathbb{R}^{n_1}$ and $x_2 \in \mathbb{R}^{n_2}$, we denote by $(x_1, x_2) \in \mathbb{R}^{n_1+n_2}$ the column vector $[x_1^T, x_2^T]^T$. Similarly, for a vector $x \in \mathbb{R}^n$, we denote by $x_i \in \mathbb{R}$ the ith element of x. For two vectors $x_1, x_2 \in \mathbb{R}^n$, we denote by $\max(x_1, x_2)$ the element-wise maximum. For a set $S \subset \mathbb{R}^n$, we denote the boundary and the interior of this set by ∂S and $\text{int}(S)$, respectively. Given two sets S_1 and S_2, $f : S_1 \rightrightarrows S_2$ and $f : S_1 \rightarrow S_2$ denote a set-valued and ordinary map, respectively. Finally, given a vector $z = (x, y) \in \mathbb{R}^2$, we denote by $\text{atan2}(z) = \text{atan2}(y, x)$.

7.2.2 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:

$$x^{(t+1)} = Ax^{(t)} + Bu^{(t)}, \tag{7.1}$$

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. The matrices A and B represent the robot dynamics and have appropriate dimensions. For

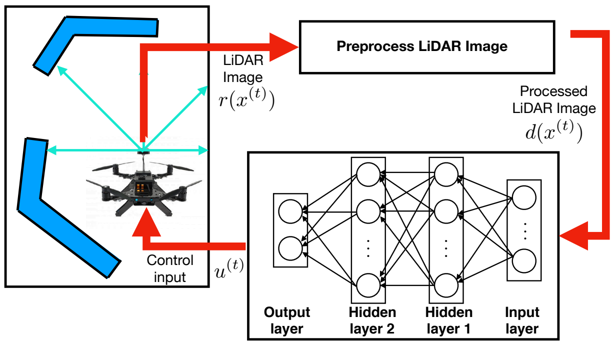


Figure 7.1: Pictorial representation of the problem setup under consideration.

a robot with nonlinear dynamics that is either differentially flat or feedback linearizable, the state space model (7.1) 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.

7.2.3 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 $i \in \{1, \dots, N\}$, the angle of the ith laser beam at time t where $\theta^{(t)}_1 = \theta^{(t)}_{\text{lidar}}$ and by $\theta^{(t)} = (\theta^{(t)}_1, \dots, \theta^{(t)}_N)$ 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 paper 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.:

$$\begin{aligned} r_i(x^{(t)}) = \min_{\mathcal{O}_i \in \mathcal{O}} \min_{z \in \mathcal{O}_i} & \|z - \zeta(x^{(t)})\|_2 \\ \text{s.t.} \quad & \operatorname{atan2}\left(z - \zeta(x^{(t)})\right) = \theta_i. \end{aligned} \tag{7.2}$$

In this paper, we will 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:

$$\begin{aligned} d_i(x^{(t)}) &= \left(r_i(x^{(t)}) \cos\theta_i,\ r_i(x^{(t)}) \sin\theta_i\right), \\ d(x^{(t)}) &= \left(d_1(x^{(t)}), \ldots d_N(x^{(t)})\right). \end{aligned} \tag{7.3}$$

7.2.4 Neural Network Controller

We consider a pre-trained neural network controller $f_{\text{NN}} : \mathbb{R}^{2N} \to \mathbb{R}^m$ that processes the LiDAR images to produce control actions with L internal and fully connected layers in addition to one output layer. Each layer contains a set of M_l neurons (where $l \in \{1, \ldots, L\}$) with Rectified Linear Unit (ReLU) activation functions. ReLU activation functions play an important role in the current advances in deep neural networks [146]. For such neural network architecture, the neural network controller $u^{(t)} = f_{\text{NN}}(d(x^{(t)}))$ can be written as:

$$\begin{aligned} h^{1(t)} &= \max\left(0,\ W^0 d(x^{(t)}) + w^0\right), \\ h^{2(t)} &= \max\left(0,\ W^1 h^{1(t)} + w^1\right), \\ &\vdots \\ h^{L(t)} &= max\left(0,\ W^{L-1} h^{L-1(t)} + w^{L-1}\right), \\ u^{(t)} &= W^L h^{L(t)} + w^L, \end{aligned} \tag{7.4}$$

where $W^l \in \mathbb{R}^{M_l \times M_{l-1}}$ and $w^l \in \mathbb{R}^{M_l}$ are the pre-trained weights and bias vectors of the neural network which are determined during the training phase.

7.2.5 Robot Trajectories and Safety Specifications

The trajectories of the robot whose dynamics are described by (7.1) when controlled by the neural network controller (7.2)-(7.4) starting from the initial condition $x_0 = x^{(0)}$ is denoted by $\eta_{x_0} : \mathbb{N} \to \mathbb{R}^n$ such that $\eta_{x_0}(0) = x_0$. A trajectory η_{x_0} is said to be safe whenever the robot position does not collide with any of the obstacles at all times.

Definition 7.2.1 (Safe Trajectory). *A robot trajectory η_{x_0} is called safe if $\zeta(\eta_{x_0}(t)) \in \mathcal{W},\ \zeta(\eta_{x_0}(t)) \notin \mathcal{O}_i,\ \forall \mathcal{O}_i \in \mathcal{O},\ \forall t \in \mathbb{N}$.*

Using the previous definition, we now define the problem of verifying the system-level safety of the neural network controlled system as follows:

Problem 2. *Consider the autonomous robot whose dynamics are governed by* (7.1) *which is controlled by the neural network controller described by* (7.4) *which processes LiDAR images described by* (7.2)-(7.3)*. Compute the set of safe initial conditions $\mathcal{X}_{safe} \subseteq \mathcal{X}$ such that any trajectory η_{x_0} starting from $x_0 \in \mathcal{X}_{safe}$ is safe.*

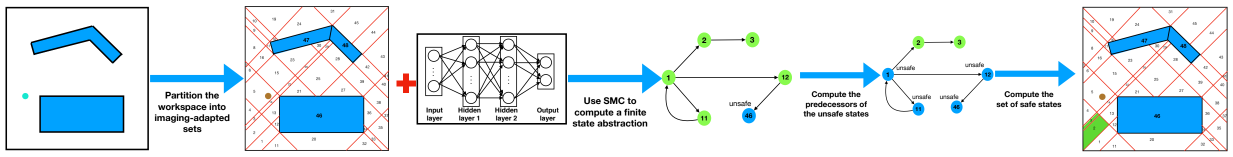

Partition the workspace into imaging-adapted sets
Input layer
Hidden layer 1
Hidden layer 2
Output layer
Use SMC to compute a finite state abstraction
unsafe
Compute the predecessors of the unsafe states
unsafe
Compute the set of safe states

Figure 7.2: Pictorial representation of the proposed framework.

7.3 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 7.3.1. *An autonomous system $\mathcal{S}$ is a pair (X, δ) consisting of a set of states X and a set-valued map $\delta : X \rightrightarrows X$ 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 not deterministic.*

Definition 7.3.2. *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 7.2, given the autonomous robot system $\mathcal{S}_{\text{NN}} = (\mathcal{X}, \delta_{\text{NN}})$, where $\delta_{\text{NN}} : x \mapsto Ax + Bf_{\text{NN}}(d(x))$, 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 difficulty in computing the finite state abstraction $\mathcal{S}_{\mathcal{F}}$ is the nonlinearity in the relation between the robot position $\zeta(x)$ and the LiDAR observations as captured by equation (7.2). However, we notice that we can partition the workspace based on the laser angles $\theta_1, \ldots, \theta_N$ along with the vertices of the polytopic obstacles such that the map d (defined in equation (7.3) which maps the robot position to the processed observations) is an affine map as shown in Section 7.4. Therefore, as summarized in Algorithm 2, the first step is to compute such partitioning $\mathcal{W}^\star$ of the workspace (WKSP-PARTITION, line 2 in Algorithm 2). While WKSP-PARTITION focuses on partitioning the workspace $\mathcal{W}$, one needs to partition the remainder of the state space $\mathcal{X}$ (STATE-SPACE-PARTITION, line 7 in Algorithm 2) to compute the finite set of abstract states $\mathcal{F}$ along with the simulation relation Q that maps between states in $\mathcal{X}$ and the corresponding abstract states in $\mathcal{F}$, and vice versa.

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 $\mathcal{W}^\star$. Similarly, we can compute a corresponding aggregate set of states $\mathcal{F}'$ as:

$$s' = \{s \in \mathcal{F} \mid \exists x \in w', w' \in \mathcal{W}', (x, s) \in Q\}$$

where each aggregate state s' is a set representing multiple states in $\mathcal{F}$. Whenever possible, we will carry out our analysis using the aggregated-partitioning $\mathcal{W}'$ (and $\mathcal{F}'$) and use the fine-partitioning $\mathcal{W}^\star$ only if deemed necessary. Details of the workspace partitioning and computing the corresponding affine maps representing the LiDAR imaging function are given in Section 7.4.

The state transition map $\delta_{\mathcal{F}}$ is computed as follows. First, we assume a transition exists between any two states s and s' in $\mathcal{F}$ (line 8- 9 in Algorithm 2). Next, we start eliminating unnecessary transitions. We observe that regions in the workspace that are adjacent or within some vicinity are more likely to force the need of transitions between their corresponding abstract states. Similarly, regions in the workspace that are far from each other are more likely to prohibit transitions between their corresponding abstract states. Therefore, in an attempt to reduce the number of computational steps in our algorithm, we check the transition feasibility between a state $s \in \mathcal{F}$ and an aggregate state $s' \in \mathcal{F}'$. If our algorithm (`CHECK-FEASIBILITY`) asserted that the neural network δ_{NN} prohibits the robot from transitioning between the regions corresponding to s and s' (denoted by $\mathcal{X}_s$ and $\mathcal{X}_{s'}$, respectively), then we conclude that no transition in $\delta_{\mathcal{F}}$ is feasible between the abstract state s and all the abstract states $s^\star$ in s' (lines 15-21 in Algorithm 2). This leads to a reduction in the number of state pairs that need to be checked for transition feasibility. Conversely, if our algorithm (`CHECK-FEASIBILITY`) asserted that the neural network δ_{NN} allows for a transition between the regions corresponding to s and s', then we proceed by checking the transition feasibility between the state s and all the states $s^\star$ contained in the aggregate state $s^\star$ (lines 23-27 in Algorithm 2).

Checking the transition feasibility (`CHECK-FEASIBILITY`) between two abstract states entails reasoning about the robot dynamics, the neural network, along with the affine map representing the LiDAR imaging computed from the previous workspace partitioning. While the robot dynamics is assumed linear, the imaging function is affine, the technical difficulty lies in reasoning about the behavior of the neural network controller. Thanks to the ReLU activation functions in the neural network, we can encode the problem of checking the transition feasibility between two regions as formula φ, called monotone Satisfiability Modulo Convex (SMC) formula [147, 148], over Boolean and convex constraints representing, respectively, the ReLU phases and the dynamics, the neural network weights, and the imaging constraints. In addition to using the SMC solver to check the transition feasibility (`CHECK-FEASIBILITY`) between abstract states, it will be used also to perform some pre-processing of the neural network function δ_{NN} (lines 11-13 in Algorithm 2) which is going to speed up the process of checking the the transition feasibility. Details of the SMC encoding and the strategy to check transition feasibility (`CHECK-FEASIBILITY`) are given in Section 7.5.

Once the finite state abstraction $\mathcal{S}_{\mathcal{F}}$ and the simulation relation Q is computed, the next step is to partition the finite states $\mathcal{F}$ into a set of unsafe states $\mathcal{F}_{\text{unsafe}}$ and a set of safe states $\mathcal{F}_{\text{safe}}$ using the following fixed-point computation:

$$\mathcal{F}^k_{\text{unsafe}} = \begin{cases} \{s \in \mathcal{F} \mid \exists x \in \mathcal{X} : (x,s) \in Q, \\ \qquad\qquad \zeta(x) \in \mathcal{O}_i, \mathcal{O}_i \in \mathcal{O}\} & k = 0 \\ \mathcal{F}^{k-1}_{\text{unsafe}} \bigcup_{s \in \mathcal{F}^{k-1}_{\text{unsafe}}} \textsc{Pre}(s) & k > 0 \end{cases}$$

$$\mathcal{F}_{\text{unsafe}} = \lim_{k \to \infty} \mathcal{F}^k_{\text{unsafe}}, \qquad \mathcal{F}_{\text{safe}} = \mathcal{F} \setminus \mathcal{F}_{\text{unsafe}}.$$

where the $\mathcal{F}^0_{\text{unsafe}}$ represents the abstract states corresponding to the obstacles and workspace boundaries, $\mathcal{F}^k_{\text{unsafe}}$ with $k > 0$ represents all the states that can reach $\mathcal{F}^0_{\text{unsafe}}$ in k-steps, and $\textsc{Pre}(s)$ is defined as:

$$\textsc{Pre}(s) = \{s' \in \mathcal{F} \mid s \in \delta_{\mathcal{F}}(s')\}.$$

The remaining abstract states are then marked as the set of safe states $\mathcal{F}_{\text{safe}}$. Finally, we can compute the set of safe states $\mathcal{X}_{\text{safe}}$ as:

$$\mathcal{X}_{\text{safe}} = \{x \in \mathcal{X} \mid \exists s \in \mathcal{F}_{\text{safe}} : (x,s) \in Q\}.$$

These computations are summarized in lines 31-40 in Algorithm 2.

7.4 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.

7.4.1 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:

$$z_{k,\zeta(x)} = \underset{z \in \mathcal{W}}{\operatorname{argmin}} \|z - \zeta(x)\|_2 \quad \text{s.t.} \quad z \in \textsc{Ray}(\zeta(x), \theta_k) \cap \mathcal{O}^\star. \tag{7.5}$$

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

$$\mathcal{L}_k(\mathcal{R}) = \bigcup_{\zeta(x) \in \mathcal{R}} z_{k,\zeta(x)}. \tag{7.6}$$

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

Definition 7.4.1. *A set $\mathcal{R} \subset \mathcal{W}$ is said to be an imaging-adapted partition if the following property holds:*

$$\mathcal{L}_k(\mathcal{R}) \textit{ is a line segment} \quad \forall k \in \{1, \ldots, N\}. \tag{7.7}$$

Figure 7.3 shows concrete examples of imaging-adapted partitions. Imaging-adapted partitions enjoy the following property:

Lemma 7.4.2. *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:*

$$d_k(\zeta(x)) = P_{k,\mathcal{R}}\zeta(x) + Q_{k,\mathcal{R}}, \qquad d = (d_1, \ldots, d_N) \tag{7.8}$$

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.

7.4.2 Partitioning the Workspace

Motivated by Lemma 7.4.2, our objective is to design an algorithm that can partition the workspace $\mathcal{W}$ into a set of imaging-adapted partitions. As summarized in Algorithm 3, our algorithm starts by computing a set of line segments $\mathcal{G}$ that will be used to partition the workspace (lines 1-5 in Algorithm 3). This set of line segments $\mathcal{G}$ are computed as follows. First, we define the set $\mathcal{V}$ as the one that contains all the vertices of the workspace and the obstacles, i.e., $\mathcal{V} = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \textsc{Vert}(\mathcal{O}_i)$. Next, we consider rays originating from all the vertices in $\mathcal{V}$ and pointing in the opposite directions of the angles $\theta_1, \ldots, \theta_N$. By intersecting these rays with the obstacles and picking the closest intersection points, we acquire the line segments $\mathcal{G}$ that will be used to partition the workspace. In other words, $\mathcal{G}$ is computed as:

$$\mathcal{G}_k = \{\textsc{Line}(v, z) \mid v \in \mathcal{V},\ z = \underset{z \in \textsc{Ray}(v, \theta_k + \pi) \cap \mathcal{O}^\star}{\operatorname{argmin}} \|z - v\|_2\}$$

$$\mathcal{G} = \bigcup_{k=1}^{N} \mathcal{G}_k \tag{7.9}$$

Thanks to the fact that the vertices v are fixed, finding the intersection between $\textsc{Ray}(v, \theta_k + \pi)$ and $\mathcal{O}^\star$ is a standard ray-polytope intersection problem which can be solved efficiently [149].

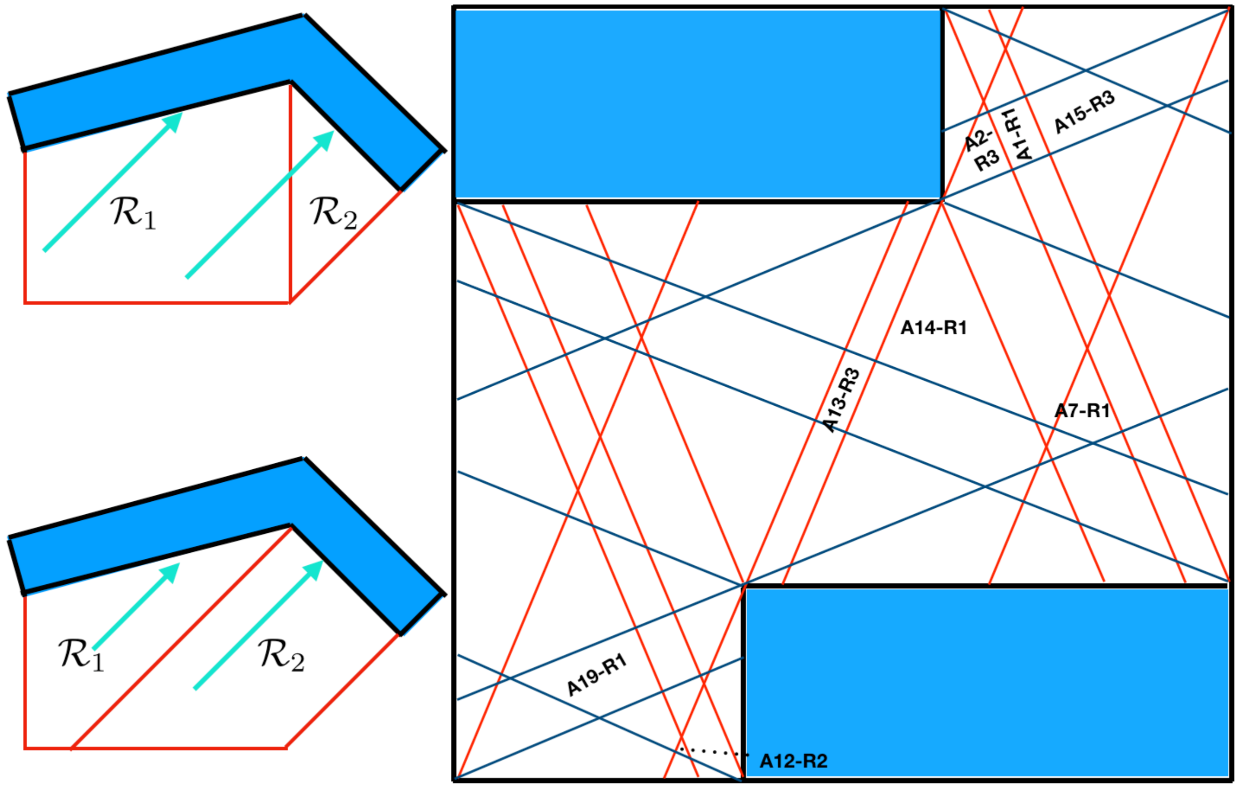


Figure 7.3: (left-up) A partitioning of the workspace that is *not* imaging-adapted. Within region $\mathcal{R}_1$, the LiDAR ray (cyan arrow) intersects with different obstacle edges depending on the robot position. (left-down) A partitioning of the workspace that is imaging-adapted. For both regions $\mathcal{R}_1$ 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 used in Section 7.6.

The next step is to compute the intersection points $\mathcal{P}$ between the line segments $\mathcal{G}$ and the edges of the obstacles $\mathcal{E} = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \text{Edge}(\mathcal{O}_i)$. A naive approach will be to consider all combinations of line segments in $\mathcal{G} \cup \mathcal{E}$ and test them for intersection. Such approach is combinatorial and would lead to an execution time that is exponential in the number of laser angles and vertices of obstacles. Thanks to the advances in the literature of computational geometry, such intersection points can be computed efficiently using the plane-sweep algorithm [149]. The plane-sweep algorithm simulates the process of sweeping a line downwards over the plane. The order of the line segments $\mathcal{G} \cup \mathcal{E}$ from left to right as they intersect the sweep line is stored in a data structure called the sweep-line status. Only segments that are adjacent in the horizontal ordering need to be tested for intersection. Though the sweeping process can be visualized as continuous, the plane-sweep algorithm sweeps only the endpoints of segments in $\mathcal{G} \cup \mathcal{E}$, which are given beforehand, and the intersection points, which are computed on the fly. To keep track of the endpoints of segments in $\mathcal{G} \cup \mathcal{E}$ and the intersection points, we use a balanced binary search tree as data structure to support insertion, deletion, and searching in $O(log\ n)$ time, where n is number of elements in the data structure.

The final step is to use the line segments $\mathcal{G} \cup \mathcal{E}$ and their intersection points $\mathcal{P}$, discovered by the plane-sweep algorithm, to compute the workspace partitions. To that end, consider the undirected planar graph whose vertices are the intersection points $\mathcal{P}$ and whose edges are $\mathcal{G} \cup \mathcal{E}$, denoted by $\textsc{Graph}(\mathcal{P}, \mathcal{G} \cup \mathcal{E})$. The workspace partitions are equivalent to finding subgraphs of $\textsc{Graph}(\mathcal{P}, \mathcal{G} \cup \mathcal{E})$ such that each subgraph contains only one simple cycle [1]. To find these simple cycles, we use a modified Depth-First-Search algorithm in which it starts from a vertex in the planar graph and then traverses the graph by considering the rightmost turns along the vertices of the graph. Finally, the workspace partitions are computed as the convex hulls of all the vertices in the computed simple cycles. It follows directly from the fact that each region is constructed from the vertices of a simple cycle that there exists no line segment in $\mathcal{G} \cup \mathcal{E}$ that intersects with the interior of any region, i.e., for any workspace partition $\mathcal{R}$, the following holds:

$$\text{int}(\mathcal{R}) \cap e = \emptyset \qquad \forall e \in \mathcal{G} \cup \mathcal{E} \tag{7.10}$$

This process is summarized in lines 12-20 in Algorithm 3. An important property of the regions determined by Algorithm 3 is stated by the following proposition.

[1] A cycle in an undirected graph is called simple when no repetitions of vertices and edges, other than the starting and ending vertex.

Proposition 7.4.3. *Consider a workspace partition $\mathcal{R}$ that is computed by Algorithm 3 and satisfies* (7.10)*. The following property holds for any LiDAR ray with angle θ_k:*

$$\exists e \in \mathcal{E} \qquad \textit{such that} \qquad \mathcal{L}_k(\mathcal{R}) \subseteq e,$$

where $\mathcal{L}_k(\mathcal{R})$ is defined in (7.6)*.*

We conclude this section by stating our first main result, quantifying the correctness and complexity of Algorithm 3.

Theorem 7.4.4. *Given a workspace with polytopic obstacles and a set of laser angles $\theta_1, \ldots, \theta_N$, then Algorithm 3 computes the partitioning $\mathcal{R}_1, \ldots, \mathcal{R}_r$ such that:*

1. $\mathcal{W} = \bigcup_{i=1}^{r} \mathcal{R}_i$,
2. *$\mathcal{R}_i$ is an imaging-adapted partition* $\quad \forall i = 1, \ldots, r$,
3. *$d : \mathcal{R}_i \to \mathbb{R}^{2N}$ is affine* $\quad \forall i = 1, \ldots, r$.

Moreover, the time complexity of Algorithm 3 is $O(M \log\, M + I \log\, M)$, where $M = |\mathcal{G} \cup \mathcal{E}|$ is cardinality of $\mathcal{G} \cup \mathcal{E}$, and I is number of intersection points between segments in $\mathcal{G} \cup \mathcal{E}$.

7.5 Computing the Finite State Abstraction

Once the workspace is partitioned into imaging-adapted partitions $\mathcal{W}^\star = \{\mathcal{R}_1, \ldots, \mathcal{R}_r\}$ and the corresponding imaging function is identified, the next step is to compute the finite state transition abstraction $\mathcal{S}_\mathcal{F} = (\mathcal{F}, \delta_\mathcal{F})$ of the closed loop system along with the simulation relation Q. The first step is to define the state space $\mathcal{F}$ and its relation to $\mathcal{X}$. To that end, we start by computing a partitioning of the state space $\mathcal{X}$ that respects $\mathcal{W}^\star$. For the sake of simplicity, we consider $\mathcal{X} \subset \mathbb{R}^n$ that is n-orthotope, i.e., there exists constants $\underline{x}_i, \overline{x}_i \in \mathbb{R}, i = 1, \ldots, n$ such that:

$$\mathcal{X} = \{x \in \mathbb{R}^n \mid \underline{x}_i \leq x_i < \overline{x}_i, \quad i = 1, \ldots, n\}$$

Now, given a discretization parameter $\epsilon \in \mathbb{R}^+$, we define the state space $\mathcal{F}$ as:

$$\begin{aligned} \mathcal{F} = \{(k_1, k_3, \ldots, k_n) \in \mathbb{N}^{n-1} \mid 1 \leq k_1 \leq r, \\ 1 \leq k_i \leq \frac{\overline{x}_i - \underline{x}_i}{\epsilon}, i = 3, \ldots, n\} \end{aligned} \tag{7.11}$$

where r is the number of regions in the partitioning $\mathcal{W}^\star$. In other words, the parameter ϵ is used to partition the state space into hyper-cubes of size ϵ in each dimension $i = 3, \ldots, n$. A state $s \in \mathcal{F}$ represents the index of a region in $\mathcal{W}^\star$ followed by the indices identifying a hypercube in the remaining $n - 2$ dimensions. Note that for the simplicity of notation, we assume that $\overline{x}_i - \underline{x}_i$ is divisible by ϵ for all $i = 1, \ldots, n$. We now define the relation $Q \subseteq \mathcal{X} \times \mathcal{F}$ as:

$$\begin{aligned} Q = \{(x, s) \in \mathcal{X} \times \mathcal{F} \mid s = (k_1, k_3, \ldots, k_n), x = (\zeta(x), x_3, \ldots, x_n), \\ \zeta(x) \in \mathcal{R}_{k_1}, \underline{x}_i + \epsilon(k_i - 1) \leq x_i < \underline{x}_i + \epsilon k_i, \\ i = 3, \ldots, n\}. \end{aligned} \tag{7.12}$$

Finally, we define the state transition function $\delta_{\mathcal{F}}$ of $\mathcal{S}_{\mathcal{F}}$ as follows:

$$\begin{aligned} & (k'_1, k'_3, \ldots k'_n) \in \delta_{\mathcal{F}}((k_1, k_3, \ldots k_n)) \text{ if} \\ & \quad \exists x = (\zeta(x), x_3, \ldots, x_n) \in \mathcal{R}_{k_1}, \underline{x}_i + \epsilon(k_i - 1) \leq x_i < \underline{x}_i + \epsilon k_i, \\ & \quad x' = (\zeta(x'), x'_3, \ldots, x'_n) \in \mathcal{R}_{k'_1}, \underline{x}_i + \epsilon(k'_i - 1) \leq x'_i < \underline{x}_i + \epsilon k'_i, \\ & \quad \text{s.t.} \quad x' = Ax + Bf_{NN}(d(x)). \end{aligned} \tag{7.13}$$

It follows from the definition of $\delta_{\mathcal{F}}$ in (7.13) that checking the transition feasibility between two states s and s' is equivalent to searching for a robot initial and goal states along with a LiDAR image that will force the neural network controller to generate an input that moves the robot between the two states while respecting the robots dynamics. In the reminder of this section, we focus on solving this feasibility problem.

7.5.1 SMC Encoding of NN

We translate the problem of checking the transition feasibility in $\delta_{\mathcal{F}}$ into a feasibility problem over a monotone SMC formula [147, 148] as follows. We introduce the Boolean indicator variables b_j^l with $l = 1, \dots, L$ and $j = 1, \dots, M_l$ (recall that L represents the number of layers in the neural network, while M_l represents the number of neurons in the lth layer). These Boolean variables represent the phase of each ReLU, i.e., an asserted b_j^l indicates that the output of the jth ReLU in the lth layer is $h_j^l = (W^{l-1}h^{l-1} + w^{l-1})_j$ while a negated b_j^l indicates that $h_j^l = 0$. Using these Boolean indicator variables, we encode the problem of checking the transition feasibility between two states $s = (k_1, k_3, \dots, k_n)$ and $s' = (k'_1, k'_3, \dots, k'_n)$ as:

$$\exists\, x, x' \in \mathbb{R}^n, u \in \mathbb{R}^m, d \in \mathbb{R}^{2N}, \tag{7.14}$$

$$(b^l, h^l, t^l) \in \mathbb{B}^{M_l} \times \mathbb{R}^{M_l} \times \mathbb{R}^{M_l}, \quad l \in \{1, \dots, L\}$$

subject to:

$$\zeta(x) \in \mathcal{R}_{k_1} \;\wedge\; \underline{x}_i + \epsilon(k_i - 1) \le x_i < \underline{x}_i + \epsilon k_i,\ i = 3, \dots, n \tag{7.15}$$

$$\wedge \zeta(x') \in \mathcal{R}_{k'_1} \wedge \underline{x}_i + \epsilon(k'_i - 1) \le x'_i < \underline{x}_i + \epsilon k'_i,\ i = 3, \dots, n \tag{7.16}$$

$$\wedge\, x' = Ax + Bu \tag{7.17}$$

$$\wedge\, d_k = P_{k,\mathcal{R}_{k_1}} \zeta(x) + Q_{k,\mathcal{R}_{k_1}}, \qquad k = 1, \dots, N \tag{7.18}$$

$$\wedge \left(t^1 = W^0 d + w^0\right) \;\wedge\; \left(\bigwedge_{l=2}^{L} t^l = W^{l-1}h^{l-1} + w^l\right) \tag{7.19}$$

$$\wedge \left(u = W^L h^L + w^L\right) \tag{7.20}$$

$$\wedge \bigwedge_{l=1}^{L} \bigwedge_{j=1}^{M_j} b_j^l \rightarrow \left[\left(h_j^l = t_j^l\right) \wedge \left(t_j^l \geq 0\right) \right] \tag{7.21}$$

$$\wedge \bigwedge_{l=1}^{L} \bigwedge_{j=1}^{M_j} \neg b_j^l \rightarrow \left[\left(h_j^l = 0\right) \wedge \left(t_j^l < 0\right) \right] \tag{7.22}$$

where (7.15)-(7.16) encode the state space partitions corresponding to the states s and s'; (7.17) encodes the dynamics of the robot; (7.18) encodes the imaging function that maps the robot position into LiDAR image; (7.19)-(7.22) encodes the neural network controller that maps the LiDAR image into a control input.

Compared to Mixed-Integer Linear Programs (MILP), monotone SMC formulas avoid using encoding heuristics like big-M encoding which leads to numerical instabilities. The SMC decision procedures follow an iterative approach combining efficient Boolean Satisfiability (SAT) solving with numerical convex programming. When applied to the encoding above, at each iteration the SAT solver generates a candidate assignment for the ReLU indicator variables b_j^l. The correctness of these assignments are then checked by solving the corresponding set of convex constraints. If the convex program turned to be infeasible, indicating a wrong choice of the ReLU indicator variables, the SMC solver will identify the set of "Irreducible Infeasible Set" (IIS) in the convex program to provide the most succinct explanation of the conflict. This IIS will be then fed back to the SAT solver to prune its search space and provide the next assignment for the ReLU indicator variables. SMC solvers were shown to better handle problems (compared with MILP solvers) for problems with relatively large number of Boolean variables [148].

7.5.2 Pruning Search Space By Pre-processing

While a neural network with M ReLUs would give rise to 2^M combinations of possible assignments to the corresponding Boolean indicator variables, we observe that only several of those combinations are feasible for each workspace region. In other words, the LiDAR imaging function along with the workspace region enforces some constraints on the inputs to the neural network which in turn enforces constraints on the subsequent layers. By performing pre-processing on each of the workspace regions, we can discover those constraints and augment it to the SMC encoding (7.15)-(7.22) to prune combinations of assignments of the ReLU indicator variables.

To find such constraints, we consider an SMC problem with the fewer constraints (7.15), (7.18)-(7.22). By iteratively solving the reduced SMC problem and recording all the IIS conflicts produced by the SMC solver, we can compute a set of counter-examples that are unique for each region. By iteratively invoking the SMC solver while adding previous counter-examples as constraints until the problem is no longer satisfiable, we compute the set $\mathcal{R}$-CONFLICTS which represents all the counter-examples for region $\mathcal{R}$. Finally, we add the following constraint:

$$\bigvee_{c \in \mathcal{R}\text{-}\textsc{Conflicts}} \neg c \tag{7.23}$$

to the original SMC encoding (7.15)-(7.22) to prune the set of possible assignments to the ReLU indicator variables. In Section 7.6, we show that pre-processing would result in several orders of magnitude reduction in the execution time.

7.5.3 Correctness of Algorithm 2

We end our discussion with the following results which assert the correctness of the whole framework described in this paper. We first start by establishing the correctness of computing the finite state abstraction $\mathcal{S}_\mathcal{F}$ along with the simulation relation Q as follows:

Proposition 7.5.1. *Consider the finite state abstraction $\mathcal{S}_\mathcal{F} = (\mathcal{F}, \delta_\mathcal{F})$ where $\mathcal{F}$ is defined by* (7.11) *and $\delta_\mathcal{F}$ is defined by* (7.13) *and computed by means of solving the SMC formu-*

las (7.15)-(7.23)*. Consider also the system* $\mathcal{S}_{NN} = (\mathcal{X}, \delta_{NN})$ *where* $\delta_{NN} : x \mapsto Ax + Bf_{NN}(d(x))$*. For the relation* Q *defined in* (7.12)*, the following holds:* $\quad \mathcal{S}_{NN} \preccurlyeq_Q \mathcal{S}_{\mathcal{F}}$*.*

Recall that Algorithm 2 applies standard reachability analysis on $\mathcal{S}_{\mathcal{F}}$ to compute the set of unsafe states. It follows directly from the correctness of the simulation relation Q established above that our algorithm computes an over-approximation of the set of unsafe states, and accordingly an under-approximation of the set of safe states. This fact is captured by the following result that summarizes the correctness of the proposed framework:

Theorem 7.5.2. *Consider the safe set* $\mathcal{X}_{safe}$ *computed by Algorithm 2. Then any trajectory* η_x *with* $\eta_x(0) \in \mathcal{X}_{safe}$ *is a safe trajectory.*

While Theorem 7.5.2 establishes the correctness of the proposed framework in Algorithm 2, two points needs to be investigated namely (i) complexity of Algorithm 2 and (ii) maximality of the set $\mathcal{X}_{\text{safe}}$. Although Algorithm 3 computes the imaging-adapted partitions efficiently (as shown in Theorem 7.4.4), analyzing a neural network with ReLU activation functions is shown to be NP-hard. Exacerbating the problem, Algorithm 2 entails analyzing the neural network a number of times that is exponential in the number of partition regions. In addition, floating point arithmetic used by the SMC solver may introduce errors that are not analyzed in this paper. In Section 7.6, we evaluate the efficiency of using the SMC decision procedures to harness this computational complexity. As for the maximality of the computed $\mathcal{X}_{\text{safe}}$ set, we note that Algorithm 2 is not guaranteed to search for the maximal $\mathcal{X}_{\text{safe}}$.

7.6 Results

We implemented the proposed verification framework as described by Algorithm 2 on top of the SMC solver named *SATEX* [150]. All experiments were executed on an Intel Core i7 2.5-GHz processor with 16 GB of memory.

Table 7.1: Scalability results for the WKSP-PARTITION Algorithm

Number of Vertices	**Number of Lasers**	**Number of Regions**	**Time [s]**
8	8	111	0.0152
	38	1851	0.3479
	118	17237	5.5300
10	8	136	0.0245
	38	2254	0.4710
	118	20343	6.9380
12	8	137	0.0275
	38	2418	0.5362
	120	23347	8.0836
	218	76337	37.0572
	298	142487	86.6341

7.6.1 Scalability of the Workspace Partitioning Algorithm

As the first step of our verification framework, imaging-adapted workspace partitioning is tested for numerical stability with increasing number of laser angles and obstacles. Table 7.1 summarizes the scalability results in terms of the number of computed regions and the execution time grows as the number of LiDAR lasers and obstacle vertices increase. Thanks to adopting well-studied computational geometry algorithms, our partitioning process takes less than 1.5 minutes for the scenario where a LiDAR scanner is equipped with 298 lasers (real-world LiDAR scanners are capable of providing readings from 270 laser angles).

7.6.2 Computational Reduction Due to Pre-processing

The second step is to pre-process the neural network. In particular, we would like to answer the following question: given a partitioned workspace, how many ReLU assignments are feasible in each region, and if any, what is the execution time to find them out. Recall that a ReLU assignment is feasible if there exist a robot position and the corresponding LiDAR image that will lead to that particular ReLU assignment.

Thanks to the IIS counterexample strategy, we can find all feasible ReLU assignments in pre-processing. Our first observation is that the number of feasible assignments is indeed much smaller compared to the set of all possible assignments. As shown in Table 7.2, for a neural network with a total of 32 neurons, only 11 ReLU assignments are feasible (within the region under consideration). Comparing this number to $2^{32} = 4.3E9$ possibilities of ReLU assignments, we conclude that pre-processing is very effective in reducing the search space by several orders of magnitude.

Furthermore, we conducted an experiment to study the scalability of the proposed pre-processing for an increasing number of ReLUs. To that end, we fixed one choice of workspace

Table 7.2: Execution time of the SMC-based pre-processing as a function of the neural network architecture.

Number of Hidden Layers	**Total Number of Neurons**	**Number of Feasible ReLU Assignments**	**Number of Counter-examples**	**Time [s]**
1	32	11	60	2.7819
	72	31	183	11.4227
	92	58	265	18.4807
	102	68	364	43.2459
	152	101	540	78.3015
	172	146	778	104.4720
	202	191	897	227.2357
	302	383	1761	656.3668
	402	730	2614	1276.4405
	452	816	4325	1856.0418
	502	1013	3766	2052.0574
	552	1165	4273	4567.1767
	602	1273	5742	6314.4890
	652	1402	5707	7166.3059
	702	1722	6521	8813.1829
2	22	3	94	1.3180
	42	19	481	10.9823
	62	35	1692	53.2246
	82	33	2685	108.2584
	102	58	5629	292.7412
	122	71	9995	739.4883
	142	72	18209	2098.0220
	162	98	34431	6622.1830
	182	152	44773	12532.8552
3	32	5	319	5.7227
	47	7	5506	148.8727
	62	45	72051	12619.5353
4	22	9	205	10.4667
	42	5	1328	90.1148

regions while changing the neural network architecture. The execution time, the number of generated counterexamples, along with the number of feasible ReLU assignments are given in

Table 7.2. For the case of neural networks with one hidden layer, our implementation of the counterexample strategy is able to find feasible ReLU assignments for a couple of hundreds of neurons in less than 4 minutes. In general, the number of counterexamples, and hence feasible ReLU assignments, and execution time grows with the number of neurons. However, the number of neurons is not the only deciding factor. Our experiments show that the depth of the network plays a significant role in affecting the scalability of the proposed algorithms. For example, comparing the neural network with one hidden layer and a hundred neurons per layer versus the network with two layers and fifty neurons per layer we notice that both networks share the same number of neurons. Nevertheless, the deeper network resulted in one order of magnitude increase regarding the number of generated counterexamples and one order of magnitude increase in the corresponding execution time. Interestingly, both of the architectures share a similar number of feasible ReLU assignments. In other words, similar features of the neural network can be captured by fewer counterexamples whenever the neural network has fewer layers. This observation can be accounted for the fact that counterexamples that correspond to ReLUs in early layers are more powerful than those involves ReLUs in the later layers of the network.

In the second part of this experiment, we study the dependence of the number of feasible ReLU assignments on the choice of the workspace region. To that end, we fix the architecture of the neural network to one with 2 hidden layers and 40 neurons per layer. Table 7.3 reports the execution time, the number of counterexamples, and the number of feasible ReLU assignments across different regions of the workspace. In general, we observe that the number of feasible ReLU assignments increases with the size of the region.

Table 7.3: Execution time of the SMC-based pre-processing as a function of the workspace region. Region indices are shown in Figure 7.3.

Region Index	**Number of Feasible ReLU Assignments**	**Number of Counter-examples**	**Time [s]**
A2-R3	33	2685	108.2584
A14-R1	55	4925	215.8251
A13-R3	7	1686	69.4158
A1-R1	25	2355	99.2122
A7-R1	26	3495	139.3486
A12-R2	3	1348	54.4548
A15-R3	25	3095	121.7869
A19-R1	38	4340	186.6428

7.6.3 Transition Feasibility

Following our verification streamline, the next step is to compute the transition function of the finite state abstraction $\delta_{\mathcal{F}}$, i.e., check transition feasibility between regions. Table 7.4 shows performance comparison between our proposed strategy that uses counterexamples obtained from pre-processing and the SMC encoding without preprocessing. We observe that the SMC encoding empowered by counterexamples, generated through the pre-processing phase, scales more favorably compared to the ones that do not take counterexamples into account leading to 2-3 orders of magnitude reduction in the execution time. Moreover, and thanks to the pre-processing counter-examples, we observe that checking transition feasibility becomes less sensitive to changes in the neural network architecture as shown in Table 7.4.

Table 7.4: Performance of the SMC-based encoding for computing $\delta_{\mathcal{F}}$ as a function of the neural network (timeout = 1 hour).

Number of Hidden Layers	**Total Number of Neurons**	**Time [s] (Exploit Counter-examples)**	**Time [s] (Without Counter-examples)**
1	82	0.5056	50.1263
	102	7.1525	timeout
	112	12.524	timeout
	122	18.0689	timeout
	132	20.4095	timeout
2	22	0.1056	15.8841
	42	4.8518	timeout
	62	3.1510	timeout
	82	2.6112	timeout
	102	11.0984	timeout
	122	3.8860	timeout
	142	0.7608	timeout
	162	2.7917	timeout
	182	193.6693	timeout
3	32	0.3884	388.549
	47	0.9034	timeout
	62	59.393	timeout

Algorithm 2 VERIFY-NN($\mathcal{X}, \delta_{\text{NN}}$)

```
1:  Step 1: Partition the workspace
2:  $(\mathcal{W}^\star, \mathcal{W}') = \text{WKSP-PARTITION}(\mathcal{W}, \mathcal{O}, \theta_p, \theta_p)$
3:
4:
5:  Step 2: Compute the finite state abstraction $\mathcal{S}_\mathcal{F}$
6:  Step 2.1: Compute the states of $\mathcal{S}_\mathcal{F}$
7:  $(\mathcal{F}, \mathcal{F}', Q) = \text{STATE-SPACE-PARTITON}(\mathcal{W}^\star, \mathcal{W}')$
8:  for each $s$ and $s'$ in $\mathcal{F}$ do
9:      $\delta_\mathcal{F}.\text{ADD-TRANSITION}(s, s')$
10: Step 2.2: Pre-process the neural network
11: for each $s$ in $\mathcal{F}$ do
12:     $\mathcal{X}_s = \{x \in \mathcal{X} \mid (x, s) \in Q\}$
13:     $CE_s = \text{PRE-PROCESS}(\mathcal{X}_s, \delta_{\text{NN}})$
14: Step 2.3: Compute the transition map $\delta_\mathcal{F}$
15: for each $s$ in $\mathcal{F}$ and $s'$ in $\mathcal{F}'$ where $s \notin s'$ do
16:     $\mathcal{X}_s = \{x \in \mathcal{X} \mid (x, s) \in Q\}$
17:     $\mathcal{X}_{s'} = \{x \in \mathcal{X} \mid (x, s^\star) \in Q,\ \forall s^\star \in s'\}$
18:     STATUS = CHECK-FEASIBILITY($\mathcal{X}_s, \mathcal{X}_{s'}, \delta_{\text{NN}}, CE_s$)
19:     if STATUS == INFEASIBLE then
20:         for each $s^\star$ in $s'$ do
21:             $\delta_\mathcal{F}.\text{REMOVE-TRANSITION}(s, s^\star)$
22:     else
23:         for each $s^\star$ in $s'$ do
24:             $\mathcal{X}_{s^\star} = \{x \in \mathcal{X} \mid (x, s^\star) \in Q\}$
25:             STATUS = CHECK-FEASIBILITY($\mathcal{X}_s, \mathcal{X}_{s^\star}, \delta_{\text{NN}}, CE_s$)
26:             if STATUS == INFEASIBLE then
27:                 $\delta_\mathcal{F}.\text{REMOVE-TRANSITION}(s, s^\star)$
28:
29:
30: Step 3: Compute the safe set
31: Step 3.1: Mark the abstract states corresponding to obstacles and workspace
    boundary as unsafe

        $\mathcal{F}^0_{\text{unsafe}} = \{s \in \mathcal{F} \mid \exists x \in \mathcal{X} : (x, s) \in Q,\ \zeta(x) \in \mathcal{O}_i, \mathcal{O}_i \in \mathcal{O}\}$

32: Step 3.2: Iteratively compute the predecessors of the abstract unsafe states
33: STATUS = FIXED-POINT-NOT-REACHED
34: while STATUS == FIXED-POINT-NOT-REACHED do
35:     $\mathcal{F}^k_{\text{unsafe}} = \mathcal{F}^{k-1}_{\text{unsafe}} \cup \text{PRE}(\mathcal{F}^{k-1}_{\text{unsafe}})$
36:     if $\mathcal{F}^k_{\text{unsafe}} == \mathcal{F}^{k-1}_{\text{unsafe}}$ then
37:         STATUS = FIXED-POINT-REACHED
38: $\mathcal{F}_{\text{safe}} = \mathcal{F} \setminus \mathcal{F}_{\text{unsafe}}$
39: Step 3.3: Compute the set of safe states
40: $\mathcal{X}_{\text{safe}} = \{x \in \mathcal{X} \mid \exists s \in \mathcal{F}_{\text{safe}} : (x, s) \in Q\}$
41: Return $\mathcal{X}_{\text{safe}}$
```

Algorithm 3 WKSP-PARTITION $(\mathcal{W}, \mathcal{O}, \theta, \theta_p)$

```
 1: Step 1: Generate partition segments
 2: $\mathcal{O}^\star = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \mathcal{O}_i, \quad \mathcal{V} = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \textsc{Vert}(\mathcal{O}_i), \quad \mathcal{E} = \bigcup_{\mathcal{O}_i \in \mathcal{O}} \textsc{Edge}(\mathcal{O}_i)$
 3: for $k \in \{1, \ldots, N\}$ do
 4:     Use a ray-polygon intersection algorithm to compute:
        $$\mathcal{G}_k = \{\textsc{Line}(v, z) \mid v \in \mathcal{V},\ z = \underset{z \in \textsc{Ray}(v, \theta_k + \pi) \cap \mathcal{O}^\star}{\operatorname{argmin}} \|z - v\|_2\}$$
 5: $\mathcal{G} = \bigcup_{k \in \theta} \mathcal{G}_k, \qquad \mathcal{G}' = \bigcup_{k \in \theta_p} \mathcal{G}_k$
 6:
 7:
 8: Step 2: Compute intersection points
 9: $\mathcal{P} = \text{PLANE-SWEEP}(\mathcal{G} \cup \mathcal{E}), \qquad \mathcal{P}' = \text{PLANE-SWEEP}(\mathcal{G}' \cup \mathcal{E})$
10:
11:
12: Step 3: Construct the partitions
13: $\textsc{Cycles} = \textsc{Find-Vertices-Of-Simple-Cycle}(\textsc{Graph}(\mathcal{P}, \mathcal{G} \cup \mathcal{E}))$
14: $\textsc{Cycles}' = \textsc{Find-Vertices-Of-Simple-Cycle}\ (\textsc{Graph}(\mathcal{P}', \mathcal{G}' \cup \mathcal{E}))$.
15: for $c \in \textsc{Cycles}$ do
16:     $\mathcal{R} = \textsc{Convex-Hull}(c)$
17:     $\mathcal{W}^\star$.ADD($\mathcal{R}$)
18: for $c \in \textsc{Cycles}'$ do
19:     $\mathcal{R}' = \textsc{Convex-Hull}(c)$
20:     $\mathcal{W}'$.ADD($\mathcal{R}'$)
21: Return $\mathcal{W}^\star, \mathcal{W}'$
```

Chapter 8

Conclusion

This thesis has delved into the critical domain of formal verification, presenting a comprehensive exploration of algorithms and applications that contribute to the robustness and reliability of NN systems. Throughout this research endeavor, we introduced three novel algorithms, each addressing distinct challenges in formal verification, and examined their practical applications through the lens of two specific use cases.

The first algorithm, PeregriNN, laid the foundation for verifiying formal properties of NNs by introducing key enhancements to existing verifiers. Its implementation and evaluation demonstrated promising results compared to SOTA methods.

The second algorithm, Bern-NN, addressed one key challenge in NN verification, which is tight overapproximation of ReLU NNs. To that end, we introduced an overapproximation algorithm using Bernstein Polynomials and showed that it yields bounds that are tighter than SOTA methods.

The third algorithm, Bern-IBP, is for tightly bounding the output of DeepBern-Nets. DeepBern-Nets are NNs with Bernstein Polynomial activation functions, they are shown to be easier to verify using the novel Bern-IBP algorithm. The whole idea of DeepBern-Nets capitalizes on the desireable properties of Bernstein Polynomials.

Building upon these algorithmic contributions, we then transitioned to the practical realm by presenting two applications. We consider the formal verification of indidual fairness properties and the safety of a NN-controlled autonomous system.

In conclusion, this thesis does not only contribute novel algorithms to the field of formal verification for NNs but also demonstrates their practical significance through real-world applications. The intersection of theoretical advancements and tangible implementations underscores the potential of formal verification in shaping the future of safe and trustworthy artificial intelligence systems. As we move forward, the insights gained from this research pave the way for further exploration, refinement, and widespread adoption of formal verification techniques in the ever-evolving landscape of NN applications.

Bibliography

[1] C. Huang, J. Fan, X. Chen, W. Li, and Q. Zhu, "Polar: A polynomial arithmetic framework for verifying neural-network controlled systems," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 414–430, Springer, 2022.

[2] L. Li, T. Xie, and B. Li, "Sok: Certified robustness for deep neural networks," *arXiv preprint arXiv:2009.04131*, 2020.

[3] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis* (D. D'Souza and K. N. Kumar, eds.), pp. 269–286, Springer, 2017.

[4] 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* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, pp. 6367–6377, 2018.

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

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

[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*, vol. 34, pp. 3291–3299, 2020.

[8] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson, "Improved Geometric Path Enumeration for Verifying ReLU Neural Networks," in *Computer Aided Verification* (S. K. Lahiri and C. Wang, eds.), vol. 12224 of *Lecture Notes in Computer Science*, pp. 66–96, Springer International Publishing, 2020.

[9] 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 *Computer Aided Verification* (I. Dillig and S. Tasiran, eds.), pp. 443–452, Springer International Publishing, 2019.

[10] 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.

[11] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma, “Strong mixed-integer programming formulations for trained neural networks,” *Mathematical Programming*, vol. 183, no. 1, pp. 3–39, 2020.

[12] 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*, vol. 29, pp. 2613–2621, 2016.

[13] 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. 42, pp. 1–39, 2020.

[14] C.-H. Cheng, G. Nührenberg, and H. Ruess, “Maximum resilience of artificial neural networks,” in *Automated Technology for Verification and Analysis* (D. D’Souza and K. Narayan Kumar, eds.), pp. 251–268, Springer, 2017.

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

[16] A. Lomuscio and L. Maganti, “An approach to reachability analysis for feed-forward relu neural networks,” 2017.

[17] V. Tjeng, K. Xiao, and R. Tedrake, “Evaluating robustness of neural networks with mixed integer programming,” 2017.

[18] 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* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, pp. 11423–11434, Curran Associates, Inc., 2019.

[19] 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.

[20] 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*, HSCC ’19, (New York, NY, USA), pp. 169–178, Association for Computing Machinery, 2019.

[21] H.-D. Tran, X. Yang, D. Manzanas 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," in *Computer Aided Verification* (S. K. Lahiri and C. Wang, eds.), pp. 3–17, Springer International Publishing, 2020.

[22] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pp. 1599–1614, USENIX Association, 2018.

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

[24] 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.

[25] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli, "A dual approach to scalable verification of deep networks.," in *Uncertainty in Artificial Intelligence* (A. Globerson and R. Silva, eds.), vol. 1, pp. 550–559, 2018.

[26] E. Wong and J. Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," 2017.

[27] S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari, "Output range analysis for deep neural networks," 2017.

[28] V. R. Royo, R. Calandra, D. M. Stipanovic, and C. Tomlin, "Fast neural network verification via shadow prices," 2019.

[29] 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.

[30] I. Z. Emiris and V. Fisikopoulos, "Practical Polytope Volume Approximation," *ACM Transactions on Mathematical Software*, vol. 44, no. 4, pp. 38:1–38:21, 2018.

[31] "Gurobi optimizer 9.1."

[32] "International Verification of Neural Networks Competition 2020 (VNN-COMP'20)." https://sites.google.com/view/vnn20.

[33] 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.

[34] X. Sun and Y. Shoukry, "Provably correct training of neural network controllers using reachability analysis," *arXiv preprint arXiv:2102.10806*, 2021.

[35] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal analysis and redesign of a neural network-based aircraft taxiing system with verifai," in *International Conference on Computer Aided Verification*, pp. 122–134, Springer, 2020.

[36] C. Liu, T. Arnon, C. Lazarus, C. Strong, C. Barrett, M. J. Kochenderfer, *et al.*, "Algorithms for verifying deep neural networks," *Foundations and Trends® in Optimization*, vol. 4, no. 3-4, pp. 244–404, 2021.

[37] P. Henriksen and A. Lomuscio, "Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis,"

[38] H. Khedr, J. Ferlez, and Y. Shoukry, "Peregrinn: Penalized-relaxation greedy neural network verifier," in *Computer Aided Verification* (A. Silva and K. R. M. Leino, eds.), (Cham), pp. 287–300, Springer International Publishing, 2021.

[39] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter, "Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification," *arXiv preprint arXiv:2103.06624*, 2021.

[40] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Sherlock-a tool for verification of neural network feedback systems: demo abstract," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pp. 262–263, 2019.

[41] R. T. Farouki, "The bernstein polynomial basis: A centennial retrospective," *Computer Aided Geometric Design*, vol. 29, no. 6, pp. 379–419, 2012.

[42] L. De Branges, "The stone-weierstrass theorem," *Proceedings of the American Mathematical Society*, vol. 10, no. 5, pp. 822–824, 1959.

[43] S. Dutta, X. Chen, and S. Sankaranarayanan, "Reachability analysis for neural feedback systems using regressive polynomial rule inference," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pp. 157–168, 2019.

[44] J. Fan, C. Huang, X. Chen, W. Li, and Q. Zhu, "Reachnn*: A tool for reachability analysis of neural-network controlled systems," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 537–542, Springer, 2020.

[45] S. Bak, C. Liu, and T. Johnson, "The second international verification of neural networks competition (vnn-comp 2021): Summary and results," *arXiv preprint arXiv:2109.00498*, 2021.

[46] A. P. Smith, "Fast construction of constant bound functions for sparse polynomials," *Journal of Global Optimization*, vol. 43, no. 2, pp. 445–458, 2009.

[47] V. Stahl, *Interval methods for bounding the range of polynomials and solving systems of nonlinear equations.* na, 1995.

[48] J. Garloff, "Convergent bounds for the range of multivariate polynomials," in *International Symposium on Interval Mathematics*, pp. 37–56, Springer, 1985.

[49] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh, "Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers," *arXiv preprint arXiv:2011.13824*, 2020.

[50] J. Sánchez-Reyes, "Algebraic manipulation in the Bernstein form made simple via convolutions," *Computer-Aided Design*, vol. 35, no. 10, pp. 959–967, 2003.

[51] D. A. Popescu and R. T. Garcia, "Multivariate polynomial multiplication on GPU," *Procedia Computer Science*, vol. 80, pp. 154–165, 2016.

[52] R. T. Farouki and V. Rajan, "Algorithms for polynomials in Bernstein form," *Computer Aided Geometric Design*, vol. 5, no. 1, pp. 1–26, 1988.

[53] S. Ray and P. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients.," *Reliab. Comput.*, vol. 17, no. 1, pp. 40–71, 2012.

[54] J. Garloff and A. P. Smith, "Guaranteed affine lower bound functions for multivariate polynomials," in *PAMM: Proceedings in Applied Mathematics and Mechanics*, vol. 7, pp. 1022905–1022906, Wiley Online Library, 2007.

[55] I. J. Goodfellow, J. Shlens, and C. S. Szegedy, "Explaining and harnessing adversarial examples," 2014.

[56] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," 2016.

[57] 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 *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, USENIX Association, 2018.

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

[59] B. H. Zhang, B. Lemoine, and M. Mitchell, "Mitigating unwanted biases with adversarial learning," in *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 335–340, 2018.

[60] D. Xu, S. Yuan, L. Zhang, and X. Wu, "Fairgan: Fairness-aware generative adversarial networks," in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 570–575, IEEE, 2018.

[61] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, "A survey on bias and fairness in machine learning," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–35, 2021.

[62] H. Khedr and Y. Shoukry, "Certifair: A framework for certified global fairness of neural networks," *arXiv preprint arXiv:2205.09927*, 2022.

[63] Y. Yang and M. Rinard, "Correctness verification of neural networks," *arXiv preprint arXiv:1906.01030*, 2019.

[64] C. Ferrari, M. N. Mueller, N. Jovanović, and M. Vechev, "Complete verification via multi-neuron relaxation guided branch-and-bound," in *International Conference on Learning Representations.*

[65] S. Bak, "Nnenum: Verification of relu neural networks with optimized abstraction refinement," in *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, (Berlin, Heidelberg), p. 19–36, Springer-Verlag, 2021.

[66] B. Zhang, D. Jiang, D. He, and L. Wang, "Rethinking lipschitz neural networks for certified l-infinity robustness," *arXiv preprint arXiv:2210.01787*, 2022.

[67] Z. Lyu, M. Guo, T. Wu, G. Xu, K. Zhang, and D. Lin, "Towards evaluating and training verifiably robust neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4308–4317, 2021.

[68] M. N. Müller, F. Eckert, M. Fischer, and M. Vechev, "Certified training: Small boxes are all you need," *arXiv preprint arXiv:2210.04871*, 2022.

[69] R. T. Farouki, "The bernstein polynomial basis: A centennial retrospective," *Comput. Aided Geom. Des.*, vol. 29, p. 379–419, aug 2012.

[70] W. Qian, M. D. Riedel, and I. Rosenberg, "Uniform approximation and bernstein polynomials with coefficients in the unit interval," *European Journal of Combinatorics*, vol. 32, no. 3, pp. 448–463, 2011.

[71] R. Farouki and V. Rajan, "On the numerical condition of polynomials in bernstein form," *Computer Aided Geometric Design*, vol. 4, no. 3, pp. 191–216, 1987.

[72] J. Titi, *Matrix Methods for the Tensorial and Simplicial Bernstein Forms with Application to Global Optimization.* PhD thesis, 01 2019.

[73] J. Wang, L. Chen, and C. W. W. Ng, "A new class of polynomial activation functions of deep learning for precipitation forecasting," in *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, WSDM '22, (New York, NY, USA), p. 1025–1035, Association for Computing Machinery, 2022.

[74] V. Gottemukkula, "Polynomial activation functions," 2020.

[75] W. Wu, J. Chen, and J. Chen, "Stability analysis of systems with recurrent neural network controllers," *IFAC-PapersOnLine*, vol. 55, no. 12, pp. 170–175, 2022. 14th IFAC Workshop on Adaptive and Learning Control Systems ALCOS 2022.

[76] N. Kochdumper, H. Krasowski, X. Wang, S. Bak, and M. Althoff, "Provably safe reinforcement learning via action projection using reachability analysis and polynomial zonotopes," *IEEE Open Journal of Control Systems*, vol. 2, pp. 79–92, 2023.

[77] U. Santa Cruz and Y. Shoukry, "Nnlander-verif: A neural network formal verification framework for vision-based autonomous aircraft landing," in *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pp. 213–230, Springer, 2022.

[78] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[79] M. N. Müller, C. Brix, S. Bak, C. Liu, and T. T. Johnson, "The third international verification of neural networks competition (vnn-comp 2022): Summary and results," *arXiv preprint arXiv:2212.10376*, 2022.

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

[81] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh, "Towards stable and efficient training of verifiably robust neural networks," *arXiv preprint arXiv:1906.06316*, 2019.

[82] M. Everett, G. Habibi, C. Sun, and J. P. How, "Reachability analysis of neural feedback loops," *IEEE Access*, vol. 9, pp. 163938–163953, 2021.

[83] H. Hu, M. Fazlyab, M. Morari, and G. J. Pappas, "Reach-sdp: Reachability analysis of closed-loop systems with neural network controllers via semidefinite programming," in *2020 59th IEEE conference on decision and control (CDC)*, pp. 5929–5934, IEEE, 2020.

[84] D. M. Lopez, P. Musau, H.-D. Tran, and T. T. Johnson, "Verification of closed-loop systems with neural network controllers," in *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems* (G. Frehse and M. Althoff, eds.), vol. 61 of *EPiC Series in Computing*, pp. 201–210, EasyChair, 2019.

[85] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel, "Fairness through awareness," in *Proceedings of the 3rd innovations in theoretical computer science conference*, pp. 214–226, 2012.

[86] T. Brennan, W. Dieterich, and B. Ehret, "Evaluating the predictive validity of the compas risk and needs assessment system," *Criminal Justice and behavior*, vol. 36, no. 1, pp. 21–40, 2009.

[87] D. Dua and C. Graff, "UCI machine learning repository," 2017.

[88] N. Mehrabi, U. Gupta, F. Morstatter, G. V. Steeg, and A. Galstyan, "Attributing fair decisions with attention interventions," *arXiv preprint arXiv:2109.03952*, 2021.

[89] M. Yurochkin, A. Bower, and Y. Sun, "Training individually fair ml models with sensitive subspace robustness," *arXiv preprint arXiv:1907.00020*, 2019.

[90] A. Ruoss, M. Balunovic, M. Fischer, and M. Vechev, "Learning certified individually fair representations," *Advances in Neural Information Processing Systems*, vol. 33, pp. 7584–7596, 2020.

[91] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *International Symposium on Formal Methods*, pp. 200–214, Springer, 2011.

[92] P. G. John, D. Vijaykeerthy, and D. Saha, "Verifying individual fairness in machine learning models," in *Conference on Uncertainty in Artificial Intelligence*, pp. 749–758, PMLR, 2020.

[93] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 625–635, Association for Computing Machinery, 2019.

[94] J. Angwin, J. Larson, S. Mattu, and L. Kirchner, "How we analyzed the compas recidivism algorithm," 2016.

[95] L. F. Wightman, *LSAC national longitudinal bar passage study.* Law School Admission Council, 1998.

[96] M. Feldman, S. A. Friedler, J. Moeller, C. Scheidegger, and S. Venkatasubramanian, "Certifying and removing disparate impact," in *proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 259–268, 2015.

[97] M. Hardt, E. Price, and N. Srebro, "Equality of opportunity in supervised learning," *Advances in neural information processing systems*, vol. 29, 2016.

[98] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: Probabilistic verification of program fairness," *Proc. ACM Program. Lang.*, vol. 1, oct 2017.

[99] O. Bastani, X. Zhang, and A. Solar-Lezama, "Probabilistic verification of fairness properties via concentration," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[100] C. Urban, M. Christakis, V. Wüstholz, and F. Zhang, "Perfectly parallel fairness certification of neural networks," *Proc. ACM Program. Lang.*, vol. 4, nov 2020.

[101] R. Binns, "On the apparent conflict between individual and group fairness," in *Proceedings of the 2020 conference on fairness, accountability, and transparency*, pp. 514–524, 2020.

[102] E. Bonilla-Silva, *Racism without racists: Color-blind racism and the persistence of racial inequality in the United States.* Rowman & Littlefield Publishers, 2006.

[103] A. E. Taslitz, "Racial blindsight: The absurdity of color-blind criminal justice," *Ohio St. J. Crim. L.*, vol. 5, p. 1, 2007.

[104] D. Pedreshi, S. Ruggieri, and F. Turini, "Discrimination-aware data mining," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, (New York, NY, USA), p. 560–568, Association for Computing Machinery, 2008.

[105] Amazon, "Amazon doesn't consider the race of its customers. should it?."

[106] Wikipedia, "List of autonomous car fatalities." https://en.wikipedia.org/wiki/List_of_autonomous_car_fatalities.

[107] A. Ferdowsi, U. Challita, W. Saad, and N. B. Mandayam, "Robust deep reinforcement learning for security and safety in autonomous vehicle systems," *arXiv preprint*, 2018.

[108] T. Everitt, G. Lea, and M. Hutter, "AGI safety literature review," *arXiv preprint*, 2018.

[109] M. Charikar, J. Steinhardt, and G. Valiant, "Learning from untrusted data," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 47–60, ACM, 2017.

[110] J. Steinhardt, P. W. W. Koh, and P. S. Liang, "Certified defenses for data poisoning attacks," in *Advances in Neural Information Processing Systems*, pp. 3520–3532, 2017.

[111] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli, "Towards poisoning of deep learning algorithms with back-gradient optimization," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp. 27–38, ACM, 2017.

[112] A. Paudice, L. Muñoz-González, and E. C. Lupu, "Label sanitization against label flipping poisoning attacks," *arXiv preprint*, 2018.

[113] W. Ruan, M. Wu, Y. Sun, X. Huang, D. Kroening, and M. Kwiatkowska, "Global robustness evaluation of deep neural networks with provable guarantees for l0 norm," *arXiv preprint*, 2018.

[114] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18, ACM, 2017.

[115] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," *arXiv preprint arXiv:1708.08559*, 2017.

[116] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 408–426, Springer, 2018.

[117] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," *arXiv preprint*, 2018.

[118] L. Ma, F. Juefei-Xu, J. Sun, C. Chen, T. Su, F. Zhang, M. Xue, B. Li, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems," *arXiv preprint*, 2018.

[119] J. Wang, J. Sun, P. Zhang, and X. Wang, "Detecting adversarial samples for deep neural networks through mutation testing," *arXiv preprint*, 2018.

[120] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," *arXiv preprint*, 2018.

[121] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Proceedings of the AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.

[122] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic autonomous driving system testing," *arXiv preprint*, 2018.

[123] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," *arXiv preprint*, 2018.

[124] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," in *NASA Formal Methods Symposium*, pp. 357–372, Springer, 2017.

[125] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," *arXiv preprint arXiv:1804.06760*, 2018.

[126] Z. Zhang, G. Ernst, S. Sedwards, P. Arcaini, and I. Hasuo, "Two-layered falsification of hybrid systems guided by monte carlo tree search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[127] Z. Kurd and T. Kelly, "Establishing safety criteria for artificial neural networks," in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pp. 163–169, Springer, 2003.

[128] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Towards verified artificial intelligence," *arXiv preprint*, 2016.

[129] S. A. Seshia, A. Desai, T. Dreossi, D. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue, "Formal specification for deep neural networks," *arXiv preprint*, 2018.

[130] J. Leike, M. Martic, V. Krakovna, P. A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, and S. Legg, "AI safety gridworlds," *arXiv preprint*, 2017.

[131] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," *arXiv preprint*, 2018.

[132] K. Scheibler, L. Winterer, R. Wimmer, and B. Becker, "Towards verification of artificial neural networks," in *Workshop on Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pp. 30–40, 2015.

[133] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*, pp. 97–117, Springer, 2017.

[134] R. Bunel, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar, "A unified view of piecewise linear neural network verification," *arXiv preprint*, 2018.

[135] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," *arXiv preprint*, 2018.

[136] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Output range analysis for deep feedforward neural networks," in *NASA Formal Methods Symposium*, pp. 121–138, Springer, 2018.

[137] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *International Conference on Computer Aided Verification*, pp. 243–257, Springer, 2010.

[138] V. Tjeng and R. Tedrake, "Verifying neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[139] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai 2: Safety and robustness certification of neural networks with abstract interpretation," in *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[140] W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson, "Verification for machine learning, autonomy, and neural networks survey," *arXiv preprint arXiv:1810.01989*, 2018.

[141] W. Xiang and T. T. Johnson, "Reachability analysis and safety verification for neural network control systems," *arXiv preprint arXiv:1805.09944*, 2018.

[142] M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano, "Reachability analysis for neural agent-environment systems," in *Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2018)*, AAAI, 2018.

[143] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *International Conference on Computer Aided Verification (CAV)*, pp. 379–395, Springer, 2011.

[144] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *International Conference on Computer Aided Verification (CAV)*, pp. 258–263, Springer, 2013.

[145] C. E. Tuncali, J. Kapinski, H. Ito, and J. V. Deshmukh, "Reasoning about safety of learning-enabled components in autonomous cyber-physical systems," in *Proceedings of the 55th Annual Design Automation Conference (DAC)*, ACM, 2018.

[146] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[147] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "SMC: Satisfiability Modulo Convex optimization," in *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pp. 19–28, ACM, 2017.

[148] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Smc: Satisfiability modulo convex programming [40pt]," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1655–1679, 2018.

[149] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational geometry: algorithms and applications.* Springer-Verlag TELOS, 2008.

[150] "SatEX Solver," June 2018.

[151] E. Doha, A. Bhrawy, and M. Saker, "On the derivatives of bernstein polynomials: an application for the solution of high even-order differential equations," *Boundary Value Problems*, vol. 2011, pp. 1–16, 2011.

[152] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

Appendix A

CertiFair

A.1 Datasets

In this section, we provide a detailed description of the fairness datasets used in Section 6.5. We preprocess all the datasets such that all numerical features are scaled to $[0, 1]$ and the categorical features are one-hot encoded. The datasets are split into train and test sets that amount for 70% and 30% of the actual data, respectively.

Adult: The adult dataset is considered one of the most commonly used datasets for fairness-aware classification studies. The task is to predict whether the annual income of an individual exceeds \$50000 US dollars based on demographic characteristics. We consider the sensitive attribute for this dataset to be gender, with the privileged group being males.

German: The German credit dataset is used for credit assessment, i.e. decide if granting a credit to an individual is risky or not. It contains 1000 instances with no missing values.

Compas: The Compas dataset is used to predict whether a criminal will be re-offending within two years. It contains 5278 preprocessed instances. We consider the sensitive attribute for this dataset to be race, with the privileged group being Caucasian.

Law School: The dataset contains records for law school admission of different universities in the United States. The goal is to predict whether a law student will pass the bar exam. The dataset contains 112630 preprocessed records. We consider the sensitive attribute for this dataset to be race, with the privileged group being White.

Table A.1 summarizes the number of positive labels in each of the datasets. It is important to compare the percentage of satisfaction of a fairness property with this value, because a naive classifier can achieve 100 % fairness by classifying all points positevly.

A.2 Fairness constraints

In this section we provide a detailed explanation of the $\mathcal{P}_2$ fairness properties used to generate Table 6.2

Adult: The numerical domain is defined as $D_\phi = [0.4, 1] \times [0.5, 0.7] \times [0, 0.3], \times [0, 0.2] \times [0.5, 0.75]$. The similarity parameter $\delta_i = 0.05 \quad \forall i \in \{1, \dots n\}$

German: The numerical domain is defined as $D_\phi = [0, 1]^n$. The similarity parameter $\delta_i = 0.05 \quad \forall i \in \{1, \dots n\}$.

Table A.1: Number of data points classified positively. This metric is important when investigating fairness of classifiers. A naive classifier that classifies all points positively is 100 % fair.

Dataset	Positivity rate (%)
Adult	24.17
German	66.33
Compas	52.95
Law School	26.34

Compas: The numerical domain is defined as $D_\phi = [0,1]^n$. The similarity parameter $\delta_i = 0.02 \quad \forall i \in \{1, \dots n\}$.

Law School: The numerical domain is defined as $D_\phi = [0,1]^n$. The similarity parameter $\delta_i = 0.03 \quad \forall i \in \{1, \dots n\}$.

A.3 Additional experiments

We provide additional experiments similar to the ones in Table 6.2 for different values of the regularization parameter λ_f. Specifically, we consider properties of class $\mathcal{P}_2$ and compare between the impact of the global regularizer $\mathcal{L}_F^g$ and $\mathcal{L}_F^l$ for five randomly picked values of $\lambda_f \in \{0.01, 0.03, 0.07, 0.1, 0.5\}$. We observe that the global regularizer is able to enforce the fairness property with small values of λ_f, but after a certain threshold, the fairness loss dominates the loss and the accuracy starts to decrease. The local fairness regularizer seems to have very small effect for small values of λ_f but starts to improve fairness for larger values starting from $\lambda_f = 0.1$ for this set of properties and datasets.

The data in Table A.2 enforces our conclusions in Section 5.3 that the global fairness regularizer outperforms the local fairness regularizer in terms of providing better balance of fairness and accuracy. For example, in the German dataset, the local fairness regularizer was able to achieve 100% fairness for $\lambda_f = 0.5$ but with a drop of accuracy from 75.30% to 68.3%. On the other hand, the global fairness was able to achieve the same 100% fairness with a much smaller $\lambda_f = 0.03$ and a reduction in the accuracy from 75.30% to 73%. The same conclusion can be drawn among the Adult and Compas datasets.

Table A.2: Comparison between global and local fairness regularizers for varying values of λ_f

λ_f	Dataset	Test Accuracy (%)			Certified Fairness(%)		
		Base	$\mathcal{L}_f^g$	$\mathcal{L}_f^l$	Base	$\mathcal{L}_f^g$	$\mathcal{L}_f^l$
0.001	Adult	84.55	84.33	85.24	6.40	84.11	29.21
	German	75.30	73.00	74.33	8.64	95.06	17.28
	Compas	68.30	67.86	68.87	47.22	44.44	16.66
	Law	87.60	86.39	87.39	6.87	21.45	6.04
0.003	Adult	84.55	84.05	84.76	6.40	95.83	33.22
	German	75.30	73.00	72.00	8.64	100.00	13.58
	Compas	68.30	67.42	68.18	47.22	97.22	19.44
	Law	87.60	76.86	86.64	6.87	76.87	0.83
0.007	Adult	84.55	83.75	84.88	6.40	100.00	41.40
	German	75.30	72.66	71.33	8.64	100.00	22.22
	Compas	68.30	64.89	68.62	47.22	100.00	33.33
	Law	87.60	74.21	85.89	6.87	100.00	1.66
0.1	Adult	84.55	83.43	84.88	6.40	100.00	50.78
	German	75.30	72.66	70.33	8.64	100.00	33.33
	Compas	68.30	65.21	67.42	47.22	100.00	36.11
	Law	87.60	73.92	85.44	6.87	99.79	1.45
0.5	Adult	84.55	82.56	84.82	6.40	100.00	57.65
	German	75.30	69.30	68.30	8.64	100.00	100.00
	Compas	68.30	64.90	65.40	47.22	100.00	66.66
	Law	87.60	73.71	81.01	6.87	100.00	76.04

Appendix B

DeepBern-Nets

B.1 Bernstein Polynomials

B.1.1 Proof of Proposition 5.2.1

Proof. Before we prove our result, we review the following properties of Bernstein polynomials.

Property 3 (Positivity [72]). *Bernstein basis polynomials are non-negative on the interval $[l, u]$, i.e., $b_{n,k}^{[l,u]}(x) \geq 0$ for all $x \in [l, u]$.*

Property 4 (Partition of Unity [72]). *The sum of Bernstein basis polynomials of the same degree is equal to 1 on the interval $[l, u]$, i.e., $\sum_{k=0}^{n} b_{n,k}^{[l,u]}(x) = 1$, $\forall x \in [l, u]$.*

Property 5 (Closed under differentiation [151]). *The derivative of an n-degree Bernstein polynomial is n multiplied by the difference of two $(n-1)$-degree Bernstein polynomials. Concretely,*

$$\frac{d}{dx} b_{n,k}^{[l,u]}(x) = n \left(b_{n-1,k-1}^{[l,u]}(x) - b_{n-1,k}^{[l,u]}(x) \right)$$

Now, it follows from Property 5 that:

$$\begin{aligned}
\left|\frac{d}{dx}\sigma(x;l,u,\boldsymbol{c})\right| &= \left|\sum_{k=0}^{n} c_k n \left(b^{[l,u]}_{n-1,k-1}(x) - b^{[l,u]}_{n-1,k}(x)\right)\right| \\
&\leq \sum_{k=0}^{n} \left|c_k n b^{[l,u]}_{n-1,k-1}(x)\right| + \sum_{k=0}^{n} \left|c_k n b^{[l,u]}_{n-1,k}(x)\right| \\
&\leq n \max_k |c_k| \sum_{k=0}^{n} \left|b^{[l,u]}_{n-1,k-1}(x)\right| + n \max_k |c_k| \sum_{k=0}^{n} \left|b^{[l,u]}_{n-1,k}(x)\right| \\
&\overset{(a)}{=} n \max_k |c_k| \sum_{k=0}^{n} b^{[l,u]}_{n-1,k-1}(x) + n \max_k |c_k| \sum_{k=0}^{n} b^{[l,u]}_{n-1,k}(x) \\
&\overset{(b)}{=} n \max_k |c_k| + n \max_k |c_k| (1 + b^{[l,u]}_{n-1,n}(x)) \overset{(c)}{=} 2n \max_k |c_k|
\end{aligned}$$

where (a) follows from Property 3; (b) follows from Property 4, and (c) follows from the definition of Bernstein basis and the fact that the binomial coefficient $\binom{n-1}{n} = 0$.

Similarly,

$$\left|\frac{d}{dc_i}\sigma(x;l,u,\boldsymbol{c})\right| = \left|b^{[l,u]}_{n,i}(x)\right| \overset{(d)}{=} b^{[l,u]}_{n,i}(x) \overset{(e)}{\leq} 1.$$

where (d) follows from Property 3 and (e) follows from both Properties 3 and 4 which implies that Bernstein basis satisfy $0 \leq b^{[l,u]}_{n,i}(x) \leq 1$.

□

B.1.2 Example to demonstrate properties of Bernstein polynomials

To demonstrate the properties of Bernstein polynomials, we present a simple example to represent the polynomial $f(x) = x^3 + x^2 - x + 1$ for all $x \in [0,1]$ using the Bernstein form. Any polynomial expressed in power series form can be converted to Bernstein form by employing a closed-form expression [72] to calculate the Bernstein coefficients. For instance,

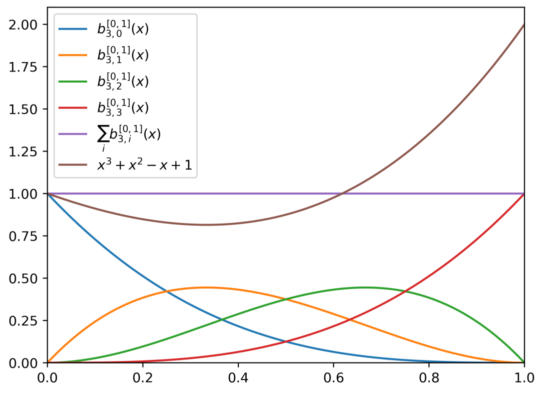


Figure B.1: A visual representation of the polynomial $f(x) = x^3 + x^2 - x + 1$ along with the Bernstein basis polynomials of degree three $b_{3,i}^{[0,1]}$ for $x \in [0, 1]$. The basis polynomials exhibit positivity and unity partition properties, while the range of its Bernstein coefficients bounds the range of the polynomial.

$f(x) = x^3 + x^2 - x + 1 = \sum_{i=0}^{3} c_i b_{3,i}^{[0,1]}$ for $x \in [0, 1]$, with $c_0 = 1$, $c_1 = c_2 = \frac{2}{3}$, and $c_3 = 2$. Figure B.1 illustrates a plot of the polynomial $f(x)$ and the Bernstein basis polynomials $b_{3,i}^{[0,1]}$. As depicted in the figure, the basis polynomials are positive (Property 3) and sum to 1 (Property 4). The range of the polynomial is constrained by the Bernstein coefficients' range, which is $[\frac{2}{3}, 2]$ (Property 1). Lastly, applying the subdivision property to compute the coefficients of the Bernstein polynomial on $[0.6, 0.8]$ results in $c_0 = 1.352$, $c_1 = 1.184$, $c_2 = 1.0613$, and $c_3 = 0.976$. With the new coefficients, we can use the range enclosure property to infer that the polynomial's range on $[0.6, 0.8]$ is $[0.976, 1.352]$.

B.2 Implementation of Bern-IBP

In this section, we discuss the implementation details of Bern-IBP and how it can be applied to certify global and local properties.

Following the discussion in section 5.3.1, we can check if a global property holds by examining the output bounds of the $\mathcal{NN}$. Algorithm 4 provides a procedure for incomplete certification, which relies on Property 1 to efficiently compute bounds on the output and check if the property holds. The output bounds are simply the minimum and maximum of the Bernstein coefficients of the last layer. The bounds computed using the Bernstein coefficients are not only much tighter than IBP bounds (as demonstrated in experiment 5.4.1), but they are also more computationally efficient, as they do not require any matrix-vector operations

Algorithm 4 Incomplete Certification of a **global** output property $y^{(L)} = \mathcal{NN}(\boldsymbol{y}^{(0)}) > 0$

```
1: Given: Neural Network $\mathcal{NN}$ with $L$ layers, and input bounds $\boldsymbol{l}^{(0)}, \boldsymbol{u}^{(0)}$
2: $l^{(L)} = \min_i c_i^{(L)}$
3: if $l^{(L)} > 0$ then
4:     return SAT
5: else
6:     return UNKNOWN
```

Certification of local properties defined on a subset of the input domain $\mathcal{D}$ can benefit from computing tighter bounds on the outputs of the NN using Bern-IBP. Algorithm 5 propagates the input bounds on a layer-by-layer basis, for linear and convolutional layers, we propagate the bounds using IBP. For Bernstein layers, we first apply the subdivision property to compute a new set of Bernstein coefficients to represent the polynomial on a subregion of $[\boldsymbol{l}^{(k)}, \boldsymbol{u}^{(k)}]$, then, using the new coefficients, we apply the enclosure property to bound the output of the Bernstein activation. This procedure result in much tighter bounds compared to IBP (as shown in Experiment 5.4.1) and Appendix B.3.4

B.3 Additional information on numerical experiments

B.3.1 Experimental Setup

Datasets. In our MNIST and CIFAR-10 experiments, we employ `torchvision.datasets` to load the datasets, maintaining the original data splits. While we normalize the input images for CIFAR-10, we do not apply any data augmentation techniques. To evaluate the

Algorithm 5 Incomplete Certification of a **local** output property $y^{(L)} = \mathcal{NN}(\boldsymbol{y}^{(0)}) > 0$

```
1: Given: Neural Network $\mathcal{NN}$ with $L$ layers, and input bounds $\boldsymbol{l}^{(0)}, \boldsymbol{u}^{(0)}$
2: for $i = 1....L$ do
3:     if $type(layer\ i)$ is Linear or Conv then
4:         $\hat{\boldsymbol{l}}^{(i)}, \hat{\boldsymbol{u}}^{(i)} \leftarrow \text{IBP}(layer\ i, [\boldsymbol{l}^{(i-1)}, \boldsymbol{u}^{(i-1)}])$
5:     else
6:         for each neuron $z$ in $layer\ i$ do        ▷ Actual implementation is vectorized
7:             $\tau \leftarrow \frac{\hat{l}_z^{(i)} - l_z^{(i)}}{u_z^{(i)} - l_z^{(i)}}$ ▷ $l_z^{(i)}$ and $u_z^{(i)}$ denote lower and upper bounds for neuron $z$ for the entire input domain $\mathcal{D}$
8:             for $k = 0....n$ do
9:                 for $j = k....n$ do
10:                    $c_j^k \leftarrow \begin{cases} c_j & \text{if } k = 0 \\ (1-\tau)c_{j-1}^{k-1} + \tau c_j^{k-1} & \text{if } k > 0 \end{cases}$
11:                    $\boldsymbol{c}' \leftarrow c_i^i$
12:                    $\boldsymbol{c}'' \leftarrow c_n^{n-i}$        ▷ $\boldsymbol{c}''$ are the coefficients of the polynomial on $[\hat{l}_z, ub_z]$
13:                    ▷ # Lines 10 to 16 need to be executed twice to compute the coefficients of the polynomial on $[\hat{l}_z, ub_z]$. Omitted for simplicity
14:            $\hat{l}_z^{(i)} = \min_i \boldsymbol{c}'', \quad \hat{u}_z^{(i)} = \max_i \boldsymbol{c}''$
15: if $\hat{l}^{(L)} > 0$ then
16:     return SAT
17: else
18:     return UNKNOWN
```

certified accuracy of our models, we utilize the test set during the certification process.

Certified training. During certified training, our models are trained using the Adam optimizer [152] for 100 epochs (unless otherwise specified) with an initial learning rate of 5e−3. We incorporate an exponential learning rate decay of 0.999 that begins after 50 epochs. For the MNIST dataset, we employ a batch size of 512, while for CIFAR-10, we use a batch size of 256, except for larger models where a batch size of 128 is utilized. Prior to incorporating the robust loss into the objective, we perform 10 warmup epochs for MNIST and 20 for CIFAR-10. The total loss comprises a weighted combination of the natural cross-entropy loss and the robust loss. The weight follows a linear schedule after the warmup phase, gradually increasing to optimize more for the robust loss towards the end of training. In terms of evaluation, the primary metric is certified accuracy, which represents the percentage of test examples for which the model can confidently make correct predictions within the given l_∞ perturbation radius.

Bernstein activations. We use the same value of the hyperparameter n for all neurons in the network. For a Bersntein activation layer with m neurons, we initialize the Bernstein coefficients from a normal distribution $c_k \sim \mathcal{N}(\mathbf{0}, \sigma^2)$, where $\sigma^2 = \frac{1}{m}$.

B.3.2 Models Architecture

Table B.1 lists the architecture, polynomial order and number of parameters for the Neural networks used to compare the certified robustness with ReLU networks from SOK[2] benchmark.

Table B.1: Neural Network Models

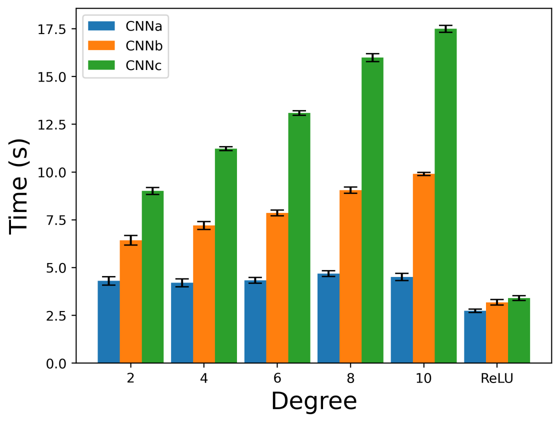
Model	Structure	Degree		# of Parameters	
		MNIST	CIFAR-10	MNIST	CIFAR-10
FCNNa	[20,20,10]	4	3	16,530	62,250
FCNNb	[100,100,100,10]	8	3	102,410	329,710
FCNNc	[100,100,100,100,100,100,100,10]	10	10	147,810	376,610
CNNa	[CONV16,CONV16,100,10]	10	12	219,250	296,090
CNNb	[CONV16,CONV16,CONV32,CONV32,512,10]	4	8	953,946	1,360,922
CNNc	[CONV32,CONV32,CONV64,CONV64,512,512,10]	2	7	2,118,954	2,966,570

B.3.3 Training time

In this section, we study the computational complexity of training DeepBern-Nets.

Figure B.2 (left) shows the average epoch time and the standard deviation for training DeepBern-Nets. We trained NNs with three different architectures and with increasing

Bernstein activation order on the MNIST dataset. The figure shows that for each architecture, the training time seems to grow linearly with the polynomial order (used in the activation functions), except for the small architecture (CNNa). This is due to the fact that higher-order polynomials introduce more parameters into the network and the fact that the cost of computing the Bernstein bounds during training also scales with the order of the polynomial. We also report the training time of a ReLU network with the same architecture to contrast an important underlying trade-off; Bernstein activations are trained with certifiability in mind, which comes with the extra computational cost during training.



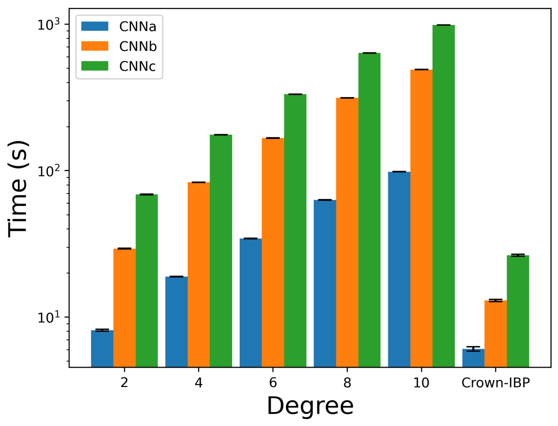


Figure B.2: (Left) Training time (per epoch) for three different model architectures and increasing order of Bernstein polynomials. (Right) training time (per epoch) for training networks with certified training objective functions (i.e., Bern-IBP must be used with every epoch to compute the loss in the certified training loss) and increasing Bernstein order on the MNIST dataset. Crown-IBP execution times are reported for ReLU networks with the same architecture.

Figure B.2 (right) shows the average epoch time ad the standard deviation for certified training of DeepBern-Nets using Bern-IBP. We also report the certified training epoch time for ReLU networks of the same architecture using Crown-IBP. We observe a similar trend of linear increase in training time with increasing the order of Bernstein activation.

B.3.4 Tightness of output bounds - Bern-IBP vs IBP

In this section, we complement Experiment 5.4.1 by reporting the raw data for computing the lower bound on the robustness margin as defined in 5.8 computed using Bern-IBP and IBP on the MNIST dataset. Tables B.2, B.3, B.4, and B.5 present the mean, median, minimum, and maximum values for the lower bounds using both methods on NNs of increasing order and different values of ϵ, respectively. The model architecture is CNNb as described in B.3.2. The tables clearly demonstrate that Bern-IBP achieves significantly higher precision than IBP in bounding DeepBern-Nets. This improvement is observed consistently across all DeepBern-Nets orders and various epsilon values. Bern-IBP outperforms IBP by orders of magnitude, highlighting its effectiveness in providing tighter bounds.

Table B.2: Raw values of the average of $\mathbb{L}_{\text{robust}}$ in Experiment 1.1. The results show that Bern-IBP results in orders of magnitude tighter values for $\mathbb{L}_{\text{robust}}$ compared with IBP.

Order	$\epsilon = 0.001$		$\epsilon = 0.01$		$\epsilon = 0.04$		$\epsilon = 0.1$	
	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP
2	3.25	6.44	-23.75	0.33	-47.85	-4.39	-48.10	0.02
3	-31.35	6.91	-145.52	-0.30	-13175.76	-8.39	-2.16e+8	-104.32
4	-109.46	6.75	-1779.87	-0.33	-8.4e+11	-0.38	-2.53e+21	-8.36
5	-410.41	6.94	-2.65e+31	2.67	-inf	-0.40	-inf	-7.69
6	-2429.93	7.05	-inf	1.13	-inf	-11.63	-inf	-42.75

Table B.3: Raw values of the median of $\mathbb{L}_{\text{robust}}$ in Experiment 1.1. The results show that Bern-IBP results in orders of magnitude tighter values for $\mathbb{L}_{\text{robust}}$ compared with IBP.

Order	$\epsilon = 0.001$		$\epsilon = 0.01$		$\epsilon = 0.04$		$\epsilon = 0.1$	
	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP
2	3.38	6.54	-23.6	0.43	-46.71	-4.14	-47.25	0.19
3	-31.01	7.07	-145.36	-0.03	-12108.94	-8.14	-1.99e+8	-104.78
4	-105.52	6.9	-1584.41	-0.1	-2.12e+11	-0.13	-1.91e+20	-8.36
5	-404.44	7.17	-1.88e+10	2.96	-2.34e+34	-0.17	-inf	-7.93
6	-2334.94	7.22	-1.32e+24	1.56	-inf	-11.25	-inf	-42.37

Table B.4: Raw values of the minimum of $\mathbb{L}_{\text{robust}}$ in Experiment 1.1. The results show that Bern-IBP results in orders of magnitude tighter values for $\mathbb{L}_{\text{robust}}$ compared with IBP.

Order	$\epsilon = 0.001$		$\epsilon = 0.01$		$\epsilon = 0.04$		$\epsilon = 0.1$	
	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP
2	-20.16	-16.63	-42.72	-16.56	-83.7	-22.22	-71.33	-8.25
3	-96.55	-12.16	-205.09	-14.02	-34962.84	-22.91	-2302369792	-137.07
4	-3550.07	-10.15	-56758.56	-13.72	-1.09065E+15	-9.23	-8.24695E+24	-23.03
5	-1345.89	-11.78	-2.2861E+35	-12.93	-inf	-8.68	-inf	-18.11
6	-109130.05	-12.24	-inf	-17.03	-inf	-30.47	-inf	-72.53

Table B.5: Raw values of the maximum of $\mathbb{L}_{\text{robust}}$ in Experiment 1.1. The results show that Bern-IBP results in orders of magnitude tighter values for $\mathbb{L}_{\text{robust}}$ compared with IBP.

Order	$\epsilon = 0.001$		$\epsilon = 0.01$		$\epsilon = 0.04$		$\epsilon = 0.1$	
	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP	IBP	Bern-IBP
2	14.83	18.01	-10.33	11.24	-23.36	5.61	-28	5.42
3	-12.53	20.98	-96.18	7.18	-2952.79	1.21	-13615902	-76.27
4	-71.81	18.97	-767.27	7.59	-307653536	4.35	-1.80601E+17	1.4
5	-249.03	16.76	-8781314	12.62	-7.72777E+25	4.26	-inf	2.62
6	-1055.1	19.99	-1.13219E+15	10.94	-inf	0.18	-inf	-19.47