Understanding and Enhancing Neural Network Verification Performance

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Dong Xu

May 2024

Acknowledgements

I want to express my sincere gratitude to my advisor, Matthew B. Dwyer. Thank you so much for your invaluable advice and direction over these years. You have encouraged me when I am down, called attention to me when I faltered, and always congrats me for my achievements. I will always remember that you wanted me to become an independent researcher who is opportunistic and vigilant. It would be impossible for me to achieve my academic goals without your assistance. Thank you sincerely, Matt.

I want to thank the most important person in my life, my dearest 王晨璐. We have gone such a long run being separated for a 12-hour time-zone difference for over 41 months. And now, you have become my wife. We will live together happily ever after.

I want to thank my parents, for your firm support of my studies abroad for so many years. I want to thank my grandmother, for thinking of me all the time. I want to thank my grandfather, although you are in another world now, I hope you are doing well. What's more, I want to thank my parents-in-law and grandparents-in-law, thank you for your trust and care, and I will take good care of her.

I want to thank all my colleagues in the Less Lab, Mitch, David, Carl, Meriel, Will, Fellipe, Trey, Sonyna, Rory, and Nusrat. Thank you all for your help and making my Ph.D. life more interesting.

I want to also thank my collaborators, David, Nusrat, and Sebastian Elbaum from UVa, Hai Doung and ThanVu Nguyen from GMU. Thank you so much for your inspiring research ideas that made all the publications possible. Lastly, I want to thank my committee members for spending their time reading and providing constructive feedback on this dissertation.

Abstract

With the rapid progression of machine learning, large-scale neural network models are being extensively implemented in safety-critical domains. Researchers have devised various techniques to evaluate the behaviors of these systems. One widely recognized approach involves using formal methods to validate or invalidate specifications that express desirable properties of system behaviors. Over the past six years, the neural network verification (NNV) community has developed more than 50 methods. Nevertheless, the community faces a challenge in generating benchmarks to effectively assess these approaches. Furthermore, the complexity of neural network models is growing at a much faster pace than the scalability of neural network verifiers. Therefore, there is a need for research on scaling NNV to apply them to real-world neural networks.

Understanding the performance characteristics of various NNV tools is crucial for their effective applications in practical scenarios and for the advancement of current methodologies. This dissertation focuses on two primary phases: understanding and enhancing the performance of NNV. In the initial phase, we introduce innovative automated approaches to systematically create diverse benchmarks for assessing existing verifiers, exploring their performance boundaries, and identifying bottlenecks. In the second phase, we effectively enhance the scalability of state-of-the-art neural network verifiers by addressing the recognized bottleneck known as "neuron stability." This is achieved by guiding the training process to produce neural networks with fewer unstable neurons; and by improving an existing verifier to stabilize neurons during the verification process, thereby significantly reducing the search space. The research conducted in this dissertation has led to the development of six open-source software artifacts for future research and development in the field.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

2 Background **8**

2.1 Neural Networks (NN) 8

2.2 Neuron Stability 9

2.3 Neural Network Verification (NNV) 10

2.4 Neural Network Verification Benchmarks 11

3 Related Work **13**

3.1 Software Verification Benchmarks 13

3.2 Neural Network Verification Benchmarks 15

3.3 Neural Network Verification Approaches 16

3.3.1 DPLL(T)-based NNV 17

3.4 Neural Network Stabilization Techniques 18

3.5 Neural Network Pruning Techniques 19

4 Unraveling Influential Factors in Neural Network Verification **20**

4.1 Identifying Factors 21

4.1.1 Potential Factors 21
4.1.2 Validating Factors 22
4.1.3 Factor Findings 25
4.2 Develop Balanced and Challenging Neural Network Verification Benchmarks 26
4.3 Conclusion 30

5 Systematic Generation of Diverse Benchmarks for Neural Network Verification 31
5.1 Approach 33
5.1.1 Factor Diverse Benchmarks 35
5.1.2 From Factor Covering Arrays to Verification Problems 37
5.1.3 Benchmark Generation 38
5.1.4 Implementation 40
5.1.5 The SwarmHost Verification Framework 41
5.1.6 Scaling Up Neural Networks with Enhanced R4V 42
5.2 Evaluation 42
5.2.1 Evaluation Setup 42
5.2.1.1 Selection of Seed Verification Problems 42
5.2.1.2 Selection of Verifiers 43
5.2.1.3 Selection of Factors and Levels 43
5.2.2 Selection of Metrics and Resources 44
5.2.3 Results 44
5.2.3.1 Comparing verifiers across a range of challenges 44
5.2.3.2 GDVB and benchmark requirements R1-R3 48
5.3 Conclusion 50

6 Adaptive Benchmark Generation for Neural Network Verification 51
6.1 Approach 52
6.1.1 Overview 53
6.1.2 The AdaGDVB Method 54

6.1.3 Examples 56
6.1.4 Implementation 58
6.2 Evaluation 59
6.2.1 Use Case One: Adaptive benchmarks 61
6.2.2 Use Case Two: Identifying Neural Network Verification Bottlenecks 62
6.3 Conclusion 63
7 Increasing Neuron Stability to Scale Neural Network Verification 64
7.1 Approach 66
7.1.1 Overview 66
7.1.2 Neuron Stability Estimation 70
7.1.3 Bias Shaping 71
7.1.4 Stable Pruning 73
7.1.5 Implementation 74
7.2 Evaluation 74
7.2.1 Study Design 75
7.2.2 RQ1: Stabilizing Neurons 77
7.2.3 RQ2: Enhancing Verification 79
7.2.4 Discussion 84
7.2.5 Threats to Validity 84
7.3 Conclusion 85
8 Harnessing Neuron Stability to Improve Verification 86
8.1 The VeriStable Approach 89
8.1.1 DPLL(T)-based Neural Network Verification 90
8.1.1.1 Boolean Representation 90
8.1.1.2 The DPLL search 91
8.1.1.3 Theory Solver 92
8.1.2 Improvements in VeriStable 92

8.1.2.1 Neuron Stability 92
8.1.2.2 Parallelism 94
8.1.3 Implementation 95
8.2 Experimental Design 96
8.2.1 Selection of NNV Benchmarks 96
8.2.2 Selection of Neural Network Verifiers Baselines 98
8.2.3 Experimental Setup 98
8.3 Results and Analysis 99
8.3.1 RQ1: Benefit of Stabilization 99
8.3.2 RQ2: Optimization Ablation Study 102
8.3.3 RQ3: Comparison with State-of-the-Art Neural Network Verifiers 103
8.4 Threats to Validity 105
8.5 Discussion 105
8.5.1 Property Specifications 106
8.5.2 Specification Format 106
8.6 Conclusion 107

9 Conclusion & Future Work **108**
9.1 Conclusion 108
9.2 Future Work 110
9.2.1 NNV Benchmark Generation 110
9.2.2 Training Verifiable Neural Networks 111
9.2.3 Enhancing the Scalability of the Neural Network Verifiers 111

A Neural Networks Artifacts and Verifiers **113**
A.1 Neural Network Datasets 113
A.2 Neural Network Architectures 114
A.3 Neural Network Verifiers 115

List of Figures

2.1 An FNN with ReLU. 11

3.1 NeuralSAT Architecture 18

4.1 A Set of Nine Factors That Affect Verifier Performance 23

4.2 Verification Problems Solved (Proved/Falsified) and Verification Time vs. Perturbation Radii of Three Neural Network Verifiers (**dashed lines**: problems proved; **dotted lines**: problems falsified; **solid lines**: solve time.) 28

5.1 SCR Score for Nine Verifiers on GDVB Benchmarks with $\text{MNIST}_{\text{ConvBig}}$ (left) and DAVE-2 (right) Seed Problems 46

5.2 Radar Plot with Maximum(solid) and Median(dotted) Values of the Two Artifacts . 47

5.3 Diversity Explored Across Factor Levels 49

6.1 Overview of AdaGDVB. 53

6.2 Conceptual Workflow of AdaGDVB (Factors: $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ with exponential growth; Dark grey: solvable problems, light grey: unsolvable problems; Circles: step 1, pentagons: step 2, triangles: step 3; Bold items in exploration: the two pivot points, P_u and P_o, at current iteration. Bold items in refinement: the performance boundary on the solvable side.) 57

6.3 AdaGDVB's Exploration Phase When Applied on α,β-CROWN Over the MNIST Network 58

6.4 AdaGDVB Generated Benchmarks (VPBs) for 4 Verifiers 60
6.5 VPBs of α,β-CROWN and NeuralSAT with 4× Granularity 61
6.6 Confirming the **neuron stability** Bottleneck Using an Instrumented Neurify Verifier 63

7.1 A Small Original Neural Network with Various Stability of Neurons 67
7.2 Applying the Bias Shaping Method on the Original Neural Network 67
7.3 Applying the Stable Pruning Method on the Original Neural Network 68
7.4 Stable Neurons(%) vs. Test Accuracy(%) per Model 78
7.5 Normalized Training Time 79
7.6 Solved Verification Problems vs. Test Accuracy(%) 81
7.7 Verification Time Speedup vs. Test Accuracy(%) 83

8.1 The Tree of Activation Patterns Computed by NeuralSAT (left) and VeriStable (right) at Corresponding Points during a Verification Run 88
8.2 Problems solved of NeuralSAT vs. VeriStable with various optimization settings 100
8.3 Stabilization Cost and Effectiveness during Verification 101

List of Tables

5.1 Mean & Variance of SCR and PAR-2 Scores Across Benchmarks (The darker and lighter gray boxes indicate the best and second best results) 45

7.1 Experimental Parameter Space 75

8.1 Benchmark Instances. U: `unsat`, S: `sat`, ?: `unknown`. 96

8.2 Problems Solved and Solving Time of NeuralSAT vs. VeriStable with Various Optimization Settings 100

8.3 A **Verifier**'s Rank (#) is Based on its VNN-COMP Score (**S**) on a Benchmark. (For each benchmark, the number of problems verified (**V**) and falsified (**F**) are shown) . 103

A.1 The Complete Set of Neural Network Architectures 114

A.2 The Complete Set of Neural Network Verifiers 115

List of Algorithms

1 The GDVB Algorithm 38

2 The AdaGDVB Algorithm 54

3 Training with Stabilizers 70

4 Bias Shaping 72

5 Stable Pruning 73

6 The VeriStable Algorithm. 90

7 The Stabilize Function 93

Chapter 1

Introduction

AI applications using machine learning algorithms are becoming well-known to the public because of their rapid development and deployment. They have been applied in various domains, ranging from self-driving vehicles, e.g., OpenPilot [102], online general language assistants, e.g., ChatGPT [98], recommendation systems [133], etc. While we are benefiting from the convenience of these novel applications, researchers continue to be concerned about their correctness when applied in safety-critical areas, such as airplane collision avoidance system [68], nuclear power plant control [125], healthcare applications [114], etc. To avoid tragedies happening in such areas, we must guarantee the safety of these AI applications.

One well-studied solution is to use formal methods to verify the neural networks. The process of verifying neural networks entails determining whether a given specification, known as the property, is satisfied by the network. An example of such a property is the local (per input) robustness property [152], which asserts that the network output is stable to small changes of a specific input:

$$\forall \rho \in [0, \epsilon] : |\mathcal{N}(x) - \mathcal{N}(x \pm \rho)| \leq \gamma,$$

where γ is a small value, ϵ represents the maximum *perturbation radius*, while x is a concrete input to the network also referred to as, the *center-point* of the local robustness property.

Recent years have witnessed significant development in neural network verification (NNV) to assure the correctness of machine learning applications. Over thirty NNV methods have been invented in the last six years. The variety of neural network verifiers covers a diverse range of algorithmic approaches including: reachability [134, 134, 124, 82, 53, 109, 110, 111, 112], optimization [87, 119, 14, 44, 131, 26, 101, 47, 89], and search [128, 127, 64, 130, 148, 5, 22, 72, 27]. Several falsification tools also have been developed to disprove the correctness of neural networks [105, 59, 35].

In the software engineering field, the performance of programs is considered a non-functional requirement. Studies have shown that performance bugs require more effort and resources to detect and fix than functional bugs [66, 145, 96]. According to Jin et al., [66], "About two-thirds of performance bugs need inputs with special features to manifest." The current approaches detect performance bugs using rule-based checkers: metrics that record the execution of some particular functions of the target program. This process involves human knowledge to locate and instrument critical functions. Statistical methods combined with hypothesis-testing, search-based, and performance profiling techniques have proven effective in detecting bottlenecks in large software [103, 115, 90].

Given the multitude of approaches stemming from different algorithmic families, selecting a suitable verifier for a specific verification problem is crucial for their effective application in real-world scenarios. Understanding the performance characteristics of various verifiers across different use cases is essential. In the latest VNN-COMP, the performance of the participants varies up to more than 9×. For example, Leeson et al. developed a method to use a graph neural network to select the most appropriate software verification tool for a given instance and can improve the state-of-the-art by 12% [81].

The most comprehensive method to understand NNV algorithmic characteristics involves breaking down the data structures and algorithmic components within these algorithms, re-implementing all approaches within a common framework using standardized libraries, and subsequently evaluating them using consistent benchmarks. This approach is highly effective for comparing the performance of verification algorithms as it eliminates biases in implementations. However, it necessitates a deep understanding of software verification and significant time and resources to execute. Liu et al. conducted a study on a limited subset of basic verifiers [85], successfully identifying performance

disparities across various abstract domains. It is worth noting that keeping pace with the rapid advancements in the field makes such endeavors very costly.

Conversely, another prevalent method for comprehending performance characteristics among neural network verifiers involves conducting extensive empirical studies. This approach relies on large sets of benchmarks. The annual NNV competition, VNN-COMP [67, 7, 93, 23], assesses participants using more than a dozen neural network benchmarks. These benchmarks are updated annually based on contributions from competition participants, evolving over time through manual decisions.

Limitations of neural network verification benchmarks The limitation of the existing neural network verification benchmarks is the risk of potentially biased instances and the lack of diversity in problem difficulty. The majority of NNV benchmarks are created directly or indirectly by the developers of the verification tools. Typically, when a new verification method is introduced, the tool is tested on either the benchmarks developed by the tool's creators [68, 128, 45, 110, 112, 111, 109, 130], or on benchmarks from previously published studies [70, 92, 148, 28]. The VNN-COMP competitions have introduced a new set of benchmarks, which are sourced from verifier developers, including: ACAS-XU [68], FNNs from OVAL [28], CNNs from the OVAL benchmark [28], CNNs from ERAN verifiers [110, 109]. These NNV benchmarks primarily feature a limited number of network architectures, with most of the diversity stemming from variation in the center-point of the local robustness property. In general, the NNV community lacks ways to produce diverse and unbiased benchmarks.

Limitations of neural network verifiers The exponential complexity of the NNV problem has been demonstrated in previous studies [68]. However, the current neural network verifiers are not yet practical for proving properties of real-world neural networks [24]. Although the state-of-the-art verifiers can handle complex neural networks like the VGG architecture [108] from a decade ago, which consists of 138M parameters, they struggle to cope with the scale of modern large language model (LLM) neural networks. For instance, GPT models [25] contain a significantly larger number of parameters, ranging from 175B (v3) to 1.76T (v4). This growing disparity between the advancement of neural networks and verifiers underscores the urgent need to enhance the scalability of verifiers.

Such improvements are crucial to ensure the safe and reliable behavior of neural networks.

To enhance the performance of an existing neural network verifier, developers could take an ad-hoc manual approach by applying methods like those from the software engineering literature. Identifying performance bottlenecks in the NNV problem is akin to detecting performance bugs in general software but requires special treatment that we aim to support. For instance, the verifiers' performance relates to the interactions between network structure/behavior, the correctness property, and the algorithm. Understanding and predicting the interactions that lead to poor performance is nontrivial. This dissertation primarily concentrates on methodologies that aid in the performance analysis of neural network verifiers, including benchmark generation and automated empirical evaluations to push verifiers to their limits, as well as innovative strategies to boost the scalability of cutting-edge neural network verifiers.

Overview of approach Our strategy for improving the scalability of neural network verification performance involves two consecutive stages. In the first stage, we conduct thorough performance analyses to identify the bottlenecks in verifier performance by conducting a series of empirical studies. In the second stage, we develop innovative methods to address these bottlenecks to enhance the scalability of the neural network verifier. This includes creating neural networks that are easier to verify and directly addressing the performance bottlenecks during the verification process.

For starters, we initiated a research study aimed to discover potential factors that could influence NNV performance. The complexity of the verification problem depends upon several aspects of the neural network and the correctness property. Through our initial investigation, we successfully identified nine **factors** (**Def.** 4) that have a significant impact on the performance of the verifier in § 4.1. By manipulating the **levels** (**Def.** 5) of these factors, one can introduce variations in the difficulty of NNV problems. Through our innovative approaches, we have achieved systematic control over the levels of factors, thereby facilitating the generation of diverse benchmarks. These novel methods effectively eliminate any potential bias that may arise from the manual creation of verification instances, while also encompassing instances that exhibit variations in problem difficulty. Additionally, a new approach was created to regulate the complexity of the verification problems through the

manipulation of the **epsilon radius** factor. This has resulted in more equitable experimental design in our later research and the establishment of a new benchmark that is capable of showcasing greater variations in verifier performance compared to the VNN-COMP benchmarks. Consequently, resulting benchmarks facilitate more meaningful empirical studies, allowing for a comprehensive analysis of the verifiers' performance.

In the first stage, we distilled the knowledge from the software performance analysis to propose approaches with a set of automated frameworks, namely GDVB, AdaGDVB, and SwarmHost, to assist in reducing the workload of NNV developers when conducting performance analysis. The detailed workflow of performance analysis for neural network verifiers is described in § 6.2.1 and § 6.2.2. Initially, we leverage the automated generation of a space of benchmark instances using AdaGDVB that identifies the *verification performance boundary* (VPB, refer to **Def.** 7) of an NNV technique. The VPB comprises two sets of verification benchmark problems that exhibit minimal variation, where one set can be efficiently verified while the other cannot. Subsequently, we instrument the verifier to collect meta-information about the verification process. Following this, we conduct a more refined performance analysis by zooming in on the specific area of interest within the verifier's VPB, such as executing the instrumented verifier on a more refined subset of the VPB through the proposed frameworks like GDVB and SwarmHost. Finally, we confirm the performance bottlenecks by examining the relationships between the outcomes of the verifiers and the instrumented metrics.

In the second stage, we devise novel techniques to eliminate the identified performance limitations. By adhering to the aforementioned strategy, we effectively validated the presence of the **neuron stability** (refer to § 2.2) NNV performance bottleneck. On one hand, this led to the creation of two innovative methods for generating neural networks with more stable neurons by guiding the training process to produce fewer unstable neurons (refer to § 7). These stable training methods have the potential to significantly enhance the number of solved verification problems, up to five times, and speed up the verification process by a factor of 14, without compromising test accuracy and training time. On the other hand, this also served as inspiration to develop novel approaches aimed at directly reducing the number of unstable neurons in the verifiers (refer to § 8). The resulting new verifier exhibits a performance improvement of up to 12 times compared to the current state-of-the-art NNV

verifiers.

Contributions Our approach is beneficial not only to NNV researchers but also to NN developers. NNV researchers can leverage it to find bottlenecks, enhance verification algorithms, optimize implementations, etc. Neural network developers can use this approach to design neural networks that are easier to verify. The contributions of this dissertation include (1) the identification of 9 factors that influence NNV performance; (2) a method to control the difficulty of verification problems through epsilon radius search; (3) a systematic approach to generating diverse NNV benchmarks; (4) an automated approach to support identification of verification performance boundaries; (5) confirmation of the **neuron stability** performance bottleneck by applying the aforementioned approaches; (6) two novel methods to guide neural network training process to produce more stable neural networks to scale their verification; (7) an innovative technique to increase **neuron stability** during the verification process directly; and (8) implementations of five open-source tools: GDVB, AdaGDVB, SwarmHost, OCTOPUS and VeriStable.

The subsequent Chapters contain the following detailed contents. Chapter 2 (§ 2) provides an overview of all essential background information, definitions, and concepts necessary for the following chapters. Chapter 3 (§ 3) delves into work within the community related to this dissertation. In Chapter 4 (§ 4), we have identified 9 factors that impact verification performance and have devised a method to regulate the complexity of verification problems through epsilon radius search. Chapter 5 (§ 5) presents a new approach to generate diverse NNV benchmarks using the 9 influencing factors. Chapter 6 (§ 6) expands upon the foundation laid in Chapter 5 and introduces an automated approach to adaptively generate NNV benchmarks based on performance feedback from a specific verifier. The case study in this chapter has validated the effectiveness of the **neuron stability** apb of the Neurify verifier, indicating that increased **neuron stability** in the neural network makes solving a verification property more challenging. Chapter 7 (§ 7) capitalizes on the concept of **neuron stability** and introduces two innovative methods to decrease **neuron stability** in neural networks during training. Chapter 8 (§ 8) enhances an existing verifier, NeuralSAT, to produce VeriStable, by directly reducing **neuron stability** in the verification process to improve its scalability. Finally,

Chapter 9 (§ 9) concludes this dissertation and explores potential avenues for future research.

Chapter 2

Background

This chapter offers background knowledge of neural networks, neural network validation, and terminology used throughout the thesis.

2.1 Neural Networks (NN)

Neural networks are designed and trained to closely approximate target functions, $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. A network, $\mathcal{N} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, is comprised of L ordered hidden layers, $l_1, \ldots, l_L$, where $l_{in} = l_0$ is the input layer and $l_{out} = l_{L+1}$ is the output layer. Hidden layers are comprised of a set of *neurons* that accumulate a weighted sum of their inputs. Applying an *activation function*(σ) determines how to non-linearly scale that sum to compute the output from the layer. For an input $\boldsymbol{x}$, the composition of the hidden layers and the activation operates as,

$$\mathcal{N}(\boldsymbol{x}) = l_{out} \circ \sigma(l_L \circ \sigma(l_{L-1} \ldots \circ \sigma(l_1 \circ l_{in}(\boldsymbol{x})))).$$

There are numerous types of hidden layers, such as Fully connected (FC), Convolutional (Conv), Deconvolutional, Recurrent, Normalization, and Pooling layers, etc. Various activation functions are designed in the literature, e.g., Rectified Linear Units (ReLU), Sigmoid, Tanh, etc. We note

that ReLU networks are popular because they tend to be sparsely activated [56] and $\max(x, 0)$ is efficient to compute which leads to efficient training and inference. Moreover, they avoid the vanishing gradient problem [58] which speeds training convergence, especially in deep networks. Complex neural network architectures, formed by freely combining different types of layers and activation functions, are capable of solving intricate real-world problems. However, this flexibility also poses a challenge in terms of verifying their behaviors.

Given a neural network architecture, $\mathcal{N}(\cdot)$, the network is *trained* to define *weight* values, denoted θ, and *bias* values, denoted b, that are associated with each neuron's input. A trained network defines for input $\boldsymbol{x}$, the output $\mathcal{N}(\boldsymbol{x}; \theta, b)$; when it is clear from the context we drop θ, b and write $\mathcal{N}(\boldsymbol{x})$. Let $\hat{z}_{i,j}$ denote the value computed for the input of neuron j in hidden layer i prior to the application of the activation function – the pre-activation value – and $z_{i,j}$ the post-activation value. For a ReLU activation function, $z_{i,j} = \max(\hat{z}_{i,j}, 0)$. The input to layer i is computed as the weighted sum of the output of the prior layer, using the learned weights θ, and bias b. The semantics of $\mathcal{N}(\boldsymbol{x}; \theta, b)$ is given by the constraints as shown in Equation 2.1

$$\bigwedge_{i \in [1,L], j \in [1,M]} \left(\hat{z}_{i,j} = \sum_{k \in [1,M]} (\theta_{i,j,k} \cdot z_{i-1,j}) + b_{i,j} \wedge z_{i,j} = \max(\hat{z}_{i,j}, 0) \right) \tag{2.1}$$

with additional constraints relating the $z_{L,j}$ to the output layer, l_{out}, and $\boldsymbol{x} = z_{0,j}$.

2.2 Neuron Stability

Propagating a single input $\boldsymbol{x}$ through network $\mathcal{N}(\boldsymbol{x})$ results in a pattern of ReLU activation in which each neuron is either *active*, $z_{i,j} = \max(\hat{z}_{i,j}, 0) = \hat{z}_{i,j}$, or *inactive*, $z_{i,j} = \max(\hat{z}_{i,j}, 0) = 0$. However, when a set of input values, e.g., a mini-batch, propagate through the network, this gives rise to pre-activation values for which a neuron is both active and inactive. When the set of pre-activation values spans 0 in this way, we say that neuron i, j is *unstable*. Otherwise, the neuron is *stable*, i.e., $max(\hat{z}_{i,j}) \leq 0 \vee min(\hat{z}_{i,j}) \geq 0$. The RS LOSS work [135] used interval propagation to estimate the stability of neurons. Note that neuron stability is only meaningful for piecewise linear activation

functions. When dealing with non-linear activation functions, one can employ abstraction techniques to transform them into piecewise linear abstractions [110].

Unstable neurons require verification approaches to reason about the disjunctions present in Equation 2.1. For each unstable neuron, there is a disjunction of the equation, $z_{i,j} = \hat{z}_{i,j} \vee 0$. In the worst case, if all neurons are unstable, then there are 2^{L*M} different ways of resolving the disjunctions. More generally, for a property, ϕ, only a subset of neurons will be unstable, $U_\phi \subseteq L \times M$, and, as we discuss in § 7 and § 8, controlling the size of this subset is a means of reducing the cost of NNV.

2.3 Neural Network Verification (NNV)

The neural network verification property ϕ composes a set of constraints over the inputs ($\phi_{\boldsymbol{x}}$) and another set of constraints associated with the output ($\phi_{\boldsymbol{y}}$). Verification of $\mathcal{N} \models \phi$ seeks to prove: $\forall \boldsymbol{x} \in \mathbb{R}^n : \phi_{\boldsymbol{x}}(\boldsymbol{x}) \Rightarrow \phi_{\boldsymbol{y}}(\mathcal{N}(\boldsymbol{x}))$. Generally, there are three categories of properties for the NNV problem: local robustness, safety, and consistency [1]. The most commonly studied property is the *local robustness property*. It defines a set of L^∞ norm perturbations over the input $\boldsymbol{x}$ of size ϵ to form a set of inputs x' that is similar to the original input, $|\boldsymbol{x} - \boldsymbol{x}'| \leq \epsilon$. This property states the network output remains the same for the classification problem as the original input, $\mathcal{N}(\boldsymbol{x}') = \mathcal{N}(\boldsymbol{x})$. For regression problems, the local robustness property allows a small amount of deviation(γ) to the network's output for given the input perturbations, $|\mathcal{N}(\boldsymbol{x}') - \mathcal{N}(\boldsymbol{x})| < \gamma$. Researchers have also developed other kinds of properties for classification problems. For instance, Toledo et al. introduced relational properties that establish relationships between classes, that is, if class a has the highest value for all inputs, then the output values for classes a and b are more similar to each other compared to the output values for classes a and c [120]. In addition, more complex properties like metamorphic specifications [30], which involve specific transformations in the input space leading to invariants in the network's output, have not been thoroughly investigated yet.

Recent work has demonstrated that a general class of specifications, where $\phi_{\boldsymbol{x}}$ and $\phi_{\boldsymbol{y}}$ are defined as half-space polytopes, can be reduced to local robustness specifications [105, 104]. This means that the essential complexity of NNV is present when verifying simpler local robustness specifications,

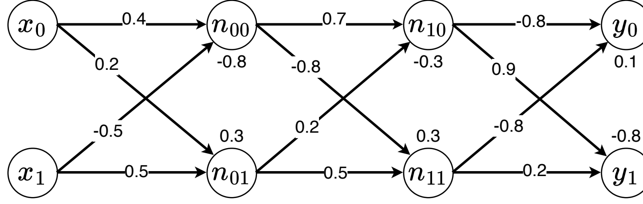


Fig. 2.1: An FNN with ReLU.

which state that $\forall \boldsymbol{x} \in c \pm \epsilon : \phi_y(\mathcal{N}(\boldsymbol{x}))$, for some constant input(center-point), c, and radius, ϵ, around it.

The inherent complexity of the NNV problem arises from the non-linear expressive power of activation functions in the neural networks – so it is generally unavoidable. Neural network verification seeks to prove or falsify whether a property specification is preserved by the neural network $\mathcal{N} \models \phi$. To prove a property, all inputs need to be considered while falsifying a property only needs one counter-example that violates the property constraints ϕ. A neural network verifier attempts to find a *counter-example* input to $\mathcal{N}$ that satisfies $\phi_{\boldsymbol{x}}$ but violates $\phi_{\boldsymbol{y}}$. If no such counter-example exists, ϕ is a valid property of $\mathcal{N}$. Otherwise, ϕ is not valid and the counter-example can be used to retrain or debug the neural network [64].

Example 2.3.1 Fig. 2.1 shows a simple ReLU neural network with two inputs $\{x_0, x_1\}$, four hidden neurons $\{n_{00}, n_{01}, n_{10}, n_{11}\}$, and two outputs $\{y_0, y_1\}$. The weights of a neuron are shown on its incoming edges, and the bias is shown above or below each neuron. The outputs of the hidden neurons are computed by the affine transformation and `ReLU`, e.g., $n_{00} = \texttt{ReLU}(0.4x_0 - 0.5x_1 - 0.8)$. The output neuron is computed with just the affine transformation, i.e., $y_0 = -0.8n_{10} - 0.8n_{11} + 0.1$. A valid property for this network is that the output is $y_0 > y_1$ for any inputs $x_0 \in [-2.0, 2.0], x_1 \in [-1.0, 1.0]$.

2.4 Neural Network Verification Benchmarks

Definition 1 *A neural network verification* ***problem*** *comprises a pair of a network and a property:* $\langle \mathcal{N}, \phi \rangle$.

The NNV problem evaluates whether the property is satisfied over the neural network. The outcome of a verification problem for a verifier indicates whether $\mathcal{N} \models \phi$ is *valid (unsatisfiable)*, *invalid (satisfiable)*, *unknown* – indicating that the problem cannot be determined to be either valid or invalid, or *out-of-resource (OOR, i.e., out of time or memory limit).*

Definition 2 *A neural network verification* ***benchmark*** *(B) contains a collection of neural network verification problems.*

To evaluate the performance of a neural network verifier, it is tested against a set of verification problems, a.k.a., a verification benchmark. As will be discussed in § 3, a good verification benchmark should contain problems that are diverse in structure and difficulty, represent verifier use cases; and evolve as verification technology advances. The same principles should also apply to NNV benchmarks.

Chapter 3

Related Work

This chapter examines the existing research in the field relating to neural network verification.

3.1 Software Verification Benchmarks

A key lesson learned by the community is that even though verification emphasizes the development of theoretical and algorithmic techniques, *advances in verification research often arise from understanding how different algorithmic and implementation approaches compare* – a process that requires empirical study. Empirical study in verification is common, but unlike many other fields of computer science, for decades it has organized *verification tool competitions* that serve as a regular and long-running form of community-driven empirical study. Researchers tracked the progress of SMT solvers over 6 years at these community-driven empirical studies and found that repeatedly "a certain solver presents a key idea that improves the performance in a particular division, and this idea is implemented by most solvers" in the following year [11]. Enabling the type of comparative studies that drive such advances requires *verification benchmarks* – a fact that the verification community has recognized for at least 25 years, e.g., [117, 75, 100, 16, 12].

Benchmarking in verification has evolved in response to the demands of empirical study within the field, e.g., [20, 63, 61, 15], to support two objectives: (A1) *assessment of the state-of-the-art*

and (A2) *comparison of alternative approaches*. In support of these, the verification community has favored benchmarks that: (R1) **are diverse in structure and difficulty**; (R2) **represent verifier use cases**; and (R3) **evolve as verification technology advances**.

The verification benchmarking and competition literature suggests that these requirements are widely accepted. For example, the TPTP benchmark's stated goals include R1 ("contains problems varying in difficulty"), R2 ("spans a diversity of subject matters"), and R3 ("is up-to-date", "provides a mechanism for adding new problems")[116]. Moreover, these requirements are promoted, either explicitly or implicitly, by many of the regularly held verification competitions. To meet R1 and R2 SAT competitions construct benchmarks that include problems from six different domains: software, hardware, AI, obstruction, combinatorial challenges, and theorem proving[63]. SAT competitions since 2017 have instituted a *bring your own benchmarks* policy that requires verifier developers to submit 20 new benchmarks with at least 10 that are "not too easy" or "too hard" – which helps to address R1 and R3. SMT competitions have used selection criteria that are biased towards these same requirements, e.g., "balancing the difficulty of benchmarks"[11].

The risk in letting technique developers choose their benchmark is selection bias – that the selected problems do not represent a broad or important population of problems. For example, if an SMT benchmark were selected based on the constraints generated by symbolic execution tools they would be structurally biased, consisting only of conjunctive formula. As another example, if an SAT benchmark were generated randomly, likely, a large portion of the benchmark would likely not represent realistic use cases.

Good benchmarks are expensive to develop, e.g., [21], but they are an invaluable resource for advancing a research community. When well-designed, they seek to balance requirements R1-R3 and to support a fair and accurate assessment of the state-of-the-art and comparison between alternative algorithmic and implementation approaches.

Verification competitions have undoubtedly been a positive force for developing high-quality verification benchmarks, but before their existence researchers were forced to develop their own "benchmarks" – a collection of verification problems on which they evaluate their techniques and perhaps others. This is the situation that the subfield of neural network verification finds itself in.

Verification Performance Metrics are standardized measurements, utilized in the verification community to evaluate the performance of verifiers against a verification benchmark. The community often discusses the importance of ranking tools fairly. Two commonly used and effective metrics for this purpose are solution-counting-ranking (SCR)[126] and penalized-average-ranking-2 (PAR-2)[65]. SCR is a simple metric that ranks verifiers based on the number of problems solved (both satisfiable and unsatisfiable), with tie-breaking determined by cumulative CPU time. On the other hand, PAR-2 is a more sophisticated metric that considers both the number of problems solved and the associated CPU time cost in its scoring. It equals the cumulative running times of the solved problems plus twice the time limit of the unsolved problems. In addition to these metrics, the performance of verifiers can also be assessed based solely on the number of problems solved and the processing time.

3.2 Neural Network Verification Benchmarks

NNV benchmarks are the primary means by which NNV technique performance is assessed. Broadly speaking, there are two types of benchmarks: first-party and third-party. First-party benchmarks are created by a neural network verifier developer when evaluating their tool, e.g., the well-known ACAS-Xu benchmark [68]. Such benchmarks are usually tested with a limited set of competitors beyond the developer. Third-party benchmarks are created by people other than neural network verifier developers to assess verifier performance, e.g., VNN-COMP [67, 7, 93, 23]. Such benchmarks tend to have a larger and more diverse set of problems.

The annual NNV competition VNN-COMP [67, 7, 93, 23] provides an overview of the NNV field every year. While its main task is to rank the performance of the verifiers, the baseline is that it has to provide a fair set of benchmarks to examine the participants. Over the last four years, it has collected over 20 benchmarks from various domains, e.g., image recognition, image generation, image prediction, aircraft control, power system, vision, etc. The benchmarks vary in, datasets, number of input dimensions, number of neurons, number of parameters, types of layers, and activation functions. The strategy it used is similar to SMT/SAT competitions, where VNN-COMP asks participants to also contribute in creating benchmarks for attending. Though it removes the difficulty of creating

fair benchmarks for the competition, the potential risk is that a verifier is biased towards its own submitted benchmarks. The VNN-COMP rule also allows verifiers to customize their tool for each benchmark, thereby introducing an additional concern regarding the verifiers' ability to perform consistently in the competition compared to real-world scenarios, e.g., α,β-CROWN has over 100 parameters that can be tuned.

Hence, the VNN-COMP benchmarks encompass a variety of initial benchmarks, and their inherent biases tend to persist over time, making them susceptible to being easily solved by state-of-the-art NNV techniques. Nevertheless, the rapid advancement of contemporary neural network verifiers renders many of the benchmarks in VNN-COMP quickly outdated. Our research, as outlined in § 5, reveals that a modern verifier can successfully tackle 89% of the problems in under 30 seconds. Consequently, there is an urgent requirement for comprehensive NNV benchmarks.

3.3 Neural Network Verification Approaches

Research on NNV is extensive and continuously expanding. This section provides an overview of established techniques and their accompanying tool implementations.

The NNV algorithm survey[85] is notable for analyzing algorithmic characteristics and comprehending the performance of a small set of neural network verifiers. This study summarizes the algorithms of various NNV techniques, rewrites the algorithm in the same programming language, and assesses the verifier's performance. Due to the extensive effort put into the reimplementation process, it can extract additional intermediate data and analyze the efficiency of each abstract domain. Nonetheless, the rewrite of verification methods is very costly, and its study primarily concentrates only on early verifiers and relatively minor verification tasks.

Several approaches have been introduced to verify a neural network behavior in recent years [85]. One class of verifiers, including α,β-CROWN [129], Nnenum [8], ERAN [110], and MN-BaB [51] overapproximate ReLU behavior which allows them to calculate an overapproximation of the entire neural network efficiently. When the verification property is satisfiable, some incomplete techniques, like ERAN, simply return *unknown*, but others, like Nnenum, α,β-CROWN or MN-BaB, perform

a case split on unstable neurons to refine the over-approximation. Another class of verifiers, including Marabou [70] and Planet [45], explore the space of case-splits to formulate separate constraint queries that constitute verification conditions. Here again, the number of possible case splits leads to exponential complexity.

Constraint-based approaches, e.g., ReLuplex [68], and its successor Marabou [70, 69], `DLV` [64], `Planet` [45], and `MIPVerify` [119] encode the problem as a constraint-solving task. These techniques transform NNV into a constraint problem, solvable using tools like SMT solvers (Planet, DLV) or SAT-based approach with custom simplex and MILP solvers (Reluplex, Marabou).

Abstraction-based approaches, e.g., `AI`^{2} [53], ERAN [92, 111, 110] (`DeepZ`, `RefineZono`, `DeepPoly`, `K-ReLU`), MN-BaB [51], ReluVal [128], Neurify [127], verinet [62], NNV [123], Nnenum [6, 8], `CROWN` [148], and CROWN [129], leverage abstract domains to tackle scalability. These techniques employ various abstract domains, such as in NNV include intervals [128], zonotopes [110], polytopes [111, 143], and starsets/imagestars [8, 123], to improve scalability. To address spurious counterexamples due to overapproximations, these methods often iterate to check counterexamples and refine abstractions.

3.3.1 DPLL(T)-based NNV

NNV is NP-Complete [68] and thus can be formulated as an SAT or SMT checking problem. Direct application of SMT solvers does not scale to the large and complex formulae encoding real-world, complex neural networks. While custom solvers, like Planet and ReLuplex, retain the soundness, completeness, and termination of SMT and improve on the performance of a direct SMT encoding, they do not scale to handle realistic neural networks [7].

While abstraction is crucial to the performance of NNV techniques, recent work on NeuralSAT [41] shows that combining it with the DPLL(T) approach of modern SMT solvers [74, 91, 10] can further improve the scalability of NNV. Fig. 3.1 gives an overview of NeuralSAT, which consists of a theory solver (Deduce) and standard DPLL components (everything else).

NEURALSAT constructs a propositional formula representing neuron activation status (Boolean Abstraction) and searches for satisfying truth assignments while employing a neural network-specific theory solver to check feasibility concerning neural network constraints and properties. The process integrates standard DPLL components, which include deciding variable assignments, and performing Boolean constraint propagation (BCP), with neural network-specific theory solving (Deduce), which uses LP solving and the polytope abstraction to check the satisfiability of assignments with the property of interest. If satisfiability is confirmed, it continues with new assignments; otherwise, it analyzes and learns conflict clauses (Analyze Conflict) to backtrack. NEURALSAT continues its search process until it either proves the property (`unsat`) or finds a total assignment (`sat`). In § 8, we describe how these DPLL components are adapted and incorporated into our new NNV approach.

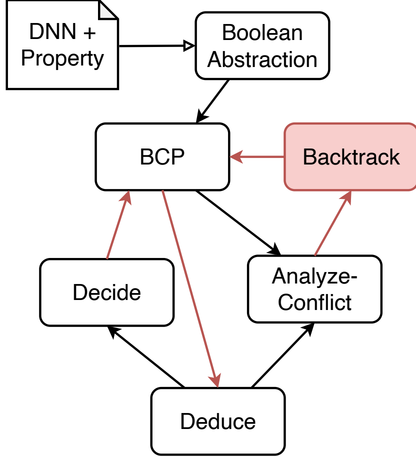


Fig. 3.1: NEURALSAT Architecture

3.4 Neural Network Stabilization Techniques

A ReLU neuron is *stable* relative to a given specification when it is in either its active or inactive phase for all inputs satisfying the specification's precondition. Researchers have observed that stable neurons have the potential to improve verifier performance, since they tend to linearize the otherwise highly non-linear computation encoded in the neural network. These methods typically require modifying the network specification. In practice, researchers employ heuristics to apply neuron stability to ReLU during the training of neural networks. For example, the RS Loss approach [135, 139] incorporates regularization techniques to train more stable weights. The linearity grafting technique [29, 146] directly replaces ReLU activation functions with strictly linear activation functions to achieve stability. Both unstructured and structured neural network pruning [151] can also help network stabilization.

RS Loss[135] is a regularization technique that induces neuron stability in the training process.

The RS LOSS, L_R is blended with the regular training loss L_T to yield a weighted sum as the optimization target, $L = L_T + w_R \times L_R$, where w_R is the hyperparameter to control the degree of stabilization. The RS LOSS term L_R is formulated as $L_R = \sum_{i=1}^{n} -\text{TANH}(1 + \underline{\hat{z}_i} \times \bar{\hat{z}}_i)$ where $\underline{\hat{z}}$ and $\bar{\hat{z}}$ are the lower and upper bounds of the pre-activation values. NRS Loss [151] is a variant of RS LOSS that regularizes the pre-batch normalization (BN) bounds instead of pre-activation bounds.

3.5 Neural Network Pruning Techniques

Neural network pruning involves optimizing a pre-trained neural network by decreasing the number of parameters or computational resources it requires, all while maintaining its predictive accuracy. *DropNet*[118] is a structured model compression method to generate sparse and reduced neural networks based on the *lottery ticket hypothesis* [52]. According to the hypothesis, a dense network contains a subnetwork that can match the test accuracy of the base network if trained in isolation. DropNet iteratively prunes a predefined percentage of less important neurons by setting their weights to zero. Although the iteration process is resource expensive, the flatness of the error landscape at the end of training limits the fraction of weights that can be pruned, hence sharp pruning at once reduces the network accuracy[99].

While the initial purpose of pruning was preserving network accuracy only, recent studies have revealed that pruning can significantly increase a network's robustness and scale robustness verification [151]. The removal of non-linearity from the insignificant neurons by converting them to linear functions has been proposed in literature [29]. However, the existence of linear activation functions in a network can sometimes result in unnecessary computational costs, as the networks are supposed to work on complex data and linear functions are incapable of handling the complexity. Also, special treatments are required to handle these non-standard architectures in network inference and verification.

Chapter 4

Unraveling Influential Factors in Neural Network Verification

The process of improving the verification of neural networks commences with comprehending the performance of the verifier. The initial stage of the verifier analysis involves understanding the attributes of the verification problem itself. This chapter delves into the identification of nine separate **factors** that impact the performance of neural network verification. Furthermore, it provides a comprehensive discussion on how manipulating the parameter epsilon radius has the potential to modify the complexity of the verification problems and to balance the quantity of unsatisfiable and satisfiable instances.

The content presented in this chapter establishes groundwork that paves the way for further exploration and improvement of verifier performance. To illustrate, (1) a variety of influencing factors are employed to create diverse benchmarks in § 5; (2) an adaptive search algorithm is formulated in § 6 to determine the verification performance boundaries of different verification approaches based on these influencing factors[1]; and (3) the identification of epsilon radius controlling, which governs the adherence to ground truth in verification problem instances, is utilized to regulate the level of

[1]The factor study introduced in this chapter is part of the published in the CAV2020 conference [140].

difficulty in § 7.

4.1 Identifying Factors

As mentioned in § 1, the verification community has taken steps to establish policies that encourage the use of diverse benchmarks. The inclusion of diversity in a benchmark is important for two reasons: (a) it showcases the range of applicability of a verification technology, and (b) it reveals performance variations both within and across different verification technologies. For instance, the selection process for the SMT competition benchmarks aims to incorporate an equal number of satisfiable and unsatisfiable benchmarks at various levels of difficulty, as stated in [11]. This is because the SMT community recognizes that the satisfiability or unsatisfiability of a benchmark problem can significantly impact the performance of verifiers. Determining unsatisfiability typically involves considering all possible variable assignments, which is generally more computationally expensive than finding a single satisfiable assignment.

To determine an initial set of factors for neural network verifiers, we began with an analysis of the literature, which identified several candidate factors, and then conducted a targeted and exploratory **factor study** to identify whether *manipulating a factor could influence some performance measure of some neural network verifier*. This study only aims to identify such factors and does not seek to characterize the complex relationship between factors and neural network verifier performance; for example, we do not aim to capture a comprehensive set of factors, assess the independence of or relations between factors, or rank factors in terms of their degree of influence.

4.1.1 Potential Factors

Throughout the literature review process, it became apparent that there is a scarcity of published papers that explicitly delve into the factors affecting performance in the realm of NNV. However, it is worth noting that almost all of these papers do provide metrics about the verification issues they have successfully addressed.

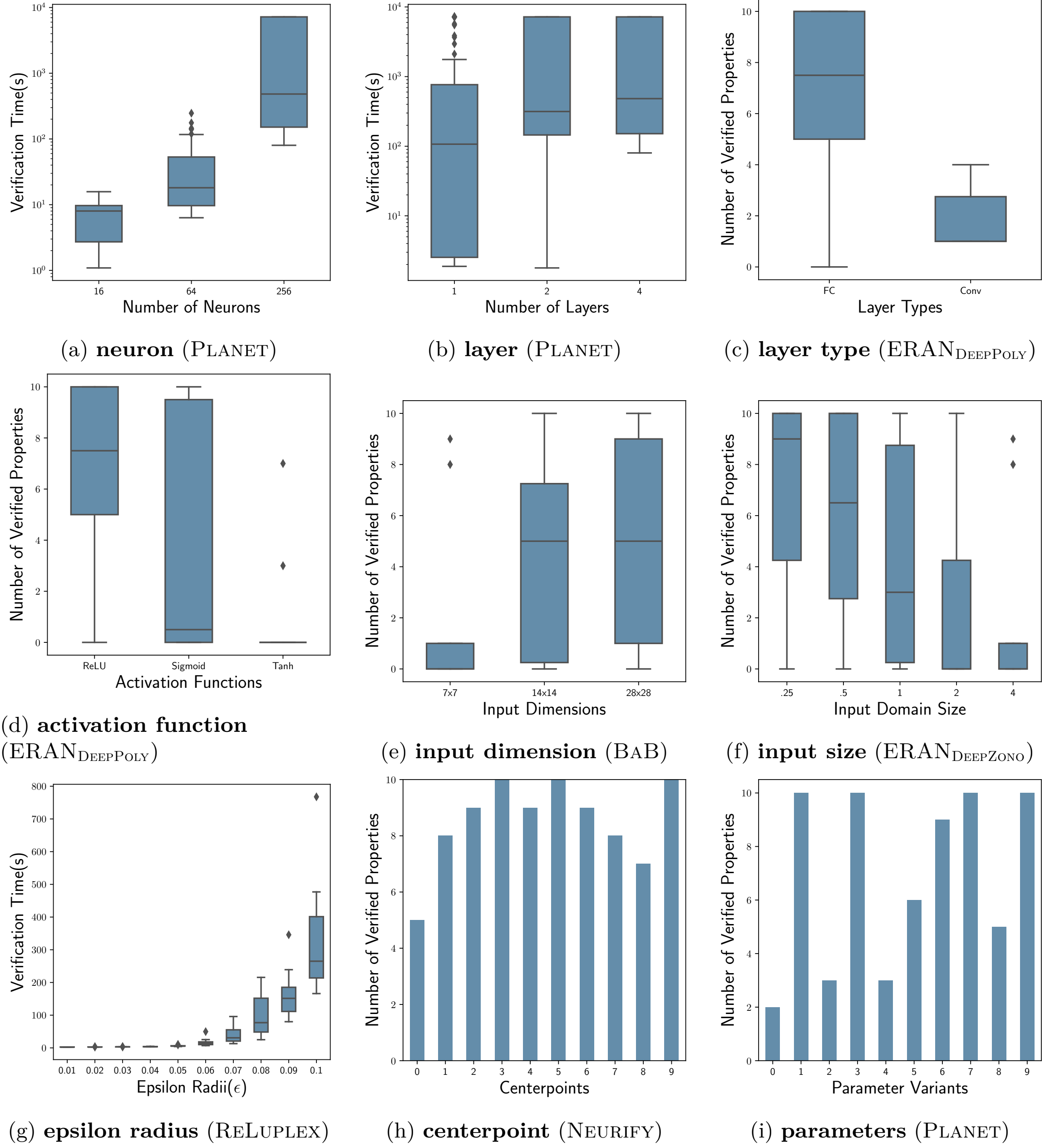
Evaluation results for ReLuplex present data on verifier outcome and solve time for local

robustness properties that vary in the input center point and radius [68]; most subsequent papers report similar property variation. Evaluation results for RobustVerifier present a study of varying the number of layers in neural networks and its impact on verifier performance[84]. Evaluation results for ERAN and its successor, MN-BaB, present performance variations across a range of networks varying in the number of layers, layer types, and neurons[53, 111, 110, 112, 51]. Bunel et al. [28] were the first that we are aware of to explicitly vary factors of NNV problems. They found that the performance varied with input dimension, number of neurons per layer, and number of layers across a set of 6 different verifiers. All the other papers published on NNV in recent years have used verification problems that varied, in an ad-hoc fashion, over a subset of the above factors [51, 94, 143, 50].

We study factors associated with both the neural network and the properties. Based on the literature analysis, we identified 4 factors related to the neural network: *number of neurons*(**neuron**), *number of layers*(**layer**), the *type of layers*(**layer type**), the *input dimension*(**input dimension**). We conjectured that an additional 3 factors might impact verifier performance: *the type of activation function*(**activation function**), the *input domain size*(**input size**), and the *learned parameters*(**parameters**). In addition, we identified 2 factors related to the property: perturbation size(**epsilon radius**) and input data center point(**centerpoint**). For robustness properties, scaling perturbation epsilon radius involves increasing the size of the input domain which will involve *more DNN behaviors* in verification. Switching a center point involves moving it to a different location in the input domain which will involve *different DNN behaviors* in verification. As a result, we propose a total number of 9 factors that can potentially influencing verification performance to be studied next step.

4.1.2 Validating Factors

As in other verification domains, neural network verifier performance is multi-faceted. In this factor study, we consider both **number of problems solved** and **solve time**. We say that the result of a verification problem is *solved* if a verifier determines conclusively that the property is *UNSAT* or *SAT*, result as opposed to *unknown*, *error*, and *out of resources* (time and memory).



(a) **neuron** (PLANET) (b) **layer** (PLANET) (c) **layer type** ($\text{ERAN}_{\text{DeepPoly}}$)

(d) **activation function** ($\text{ERAN}_{\text{DeepPoly}}$) (e) **input dimension** (BaB) (f) **input size** ($\text{ERAN}_{\text{DeepZono}}$)

(g) **epsilon radius** (ReLuplex) (h) **centerpoint** (Neurify) (i) **parameters** (PLANET)

Fig. 4.1: A Set of Nine Factors That Affect Verifier Performance

Our exploratory factor study is opportunistic in that we seek to find a verification problem for which manipulation of a selected factor exhibits performance variation. Towards this end, we

conducted a series of trials where we varied a factor hypothesized to influence verification performance while holding all other factors constant, and reported the results in Fig. 4.1.

We studied variations of networks for the MNIST (§ A.1) dataset and considered local robustness properties. We used different verifiers across the study: RELUPLEX, PLANET, NEURIFY, BAB, ERAN with the DeepPoly (DP) and DeepZono (DZ) abstract domains, as a subset of the full verifier list depicted in § A.3. We briefly describe the trials and then summarize the outcome as follows:

1. **neuron**: The architecture of the neural network was fixed, with 4 fully connected layers using ReLU activation functions, and the total number of neurons was varied (16, 64, 256) – they were spread evenly across layers. Each network is trained 10 times and verified on 100 local robustness properties. Fig. 4.1a plots the number of neurons versus verification time for PLANET.

   ***Finding**: Verification time can increase with the number of neurons.*

2. **layer**: We use the same context as for the neuron factor study, except that we fixed the number of neurons at 256 and varied the number of layers (1,2,4). Fig. 4.1b plots the number of layers versus verification time for PLANET.

   ***Finding**: Verification time can increase with the number of layers.*

3. **layer type**: We use a pair of two-layer neural networks, with the same number of neurons, where one has a fully-connected layer and the other a convolutional layer. Each network is trained 10 times and verified on 10 local robustness properties. Fig. 4.1c plots layer type versus the number of solved properties using ERAN_{DP}. ***Finding**: Verification problems solved can vary with layer type.*

4. **activation function**: We use the fully-connected network from the layer types study, we generated three networks by altering the activation function from ReLU to use Sigmoid and Tanh. The training setup and properties remain the same as in the previous trial. Fig. 4.1d plots the activation function versus the number of properties solved using ERAN_{DP}.

   ***Finding**: Verification problems solved can vary with the activation function.*

5. **input dimension**: We use 3 architectures that differ only in their input dimension which is scaled ($\frac{1}{16}, \frac{1}{4}, 1$) relative to the original problem. The training setup and properties are from the layer-type study. Fig. 4.1e plots the input dimension versus the number of properties solved using BaB.

   ***Finding**: Verification problems solved can increase with increasing input dimension.*

6. **input size**: We use 5 architectures that differ only in the range of values of their inputs which are scaled ($\frac{1}{4}, \frac{1}{2}, 1, 2, 4$) based on the original problem. The training setup and properties are from the layer-type study. Fig. 4.1f plots the input size versus the number of solved properties using ERAN_{DZ}.

   ***Finding**: Verification problems solved can decrease with increasing input domain size.*

7. **epsilon radius**: We use a single-layer network and reuse the training setup and properties from the layer type study. We scale the properties ($0.01 - 0.1$) to generate verification problems. Fig. 4.1g plots property scaling versus the verification time using ReLuplex.

   ***Finding**: Verification time can increase with increasing property scale.*

8. **centerpoint**: We replicated the property scale study, but held the scale fixed and translated the center point of the local robustness property to 10 other locations. Fig. 4.1h plots the number of DNNs for each of the 10 translated properties solved using Neurify. ***Finding**: Verification problems solved can vary with property translation.*

9. **parameters**: Building of the property studies, we explore the verification of 10 scaled property variants across the same network trained 10 times with different initial weights. Fig. 4.1i plots the number of solved properties using Planet. ***Finding**: Verification problems solved can vary with the learned weights of the network.*

4.1.3 Factor Findings

As a result, the existence of all 9 **factor** hypotheses are empirically validated based on the above factor study, such that at least one verifier is sensitive to changes in the **level** of each **factor**. Various

factors influence the performance of different NNV tools differently – in terms of time or problems solved. For example, we found that: varying input dimension impacts BaB's accuracy, but not ReLuplex's; varying input domain size impacts ERAN_{DZ}'s accuracy, but not Neurify's; and varying property scale impacts ReLuplex's verification time, but not Neurify's. Note that the purpose of this study is to confirm the existence of a set of **factor**s that influence the neural network verification performance.

This exploratory study provides a starting set of viable factors that impact verification performance. These factors can be used to parameterize the GDVB approach to produce verification problem benchmarks in which those factors are systematically varied to parameterize the benchmark(§ 5). Furthermore, systematically parameterizing and searching the factor space allows one to identify the verification performance boundary of the given verifier(§ 6).

4.2 Develop Balanced and Challenging Neural Network Verification Benchmarks

As mentioned in § 2, the NNV problem consists of a network and a property pair denoted as $\langle \mathcal{N}, \phi \rangle$. The complexity of the NNV problem is determined by the intricacy of the neural network $\mathcal{N}$ and the expressiveness of the property ϕ. The factor study(refer to § 4.1) identifies 9 **factors** that can impact the performance of the verifier. Among these factors, 7 are associated with the neural network, while 2 are related to the property specification. This section specifically focuses on a particular factor: **epsilon radius**. This factor defines the permissible range of perturbation in the local robustness property, which in turn affects the determination of the verification problem as either UNSAT or SAT.

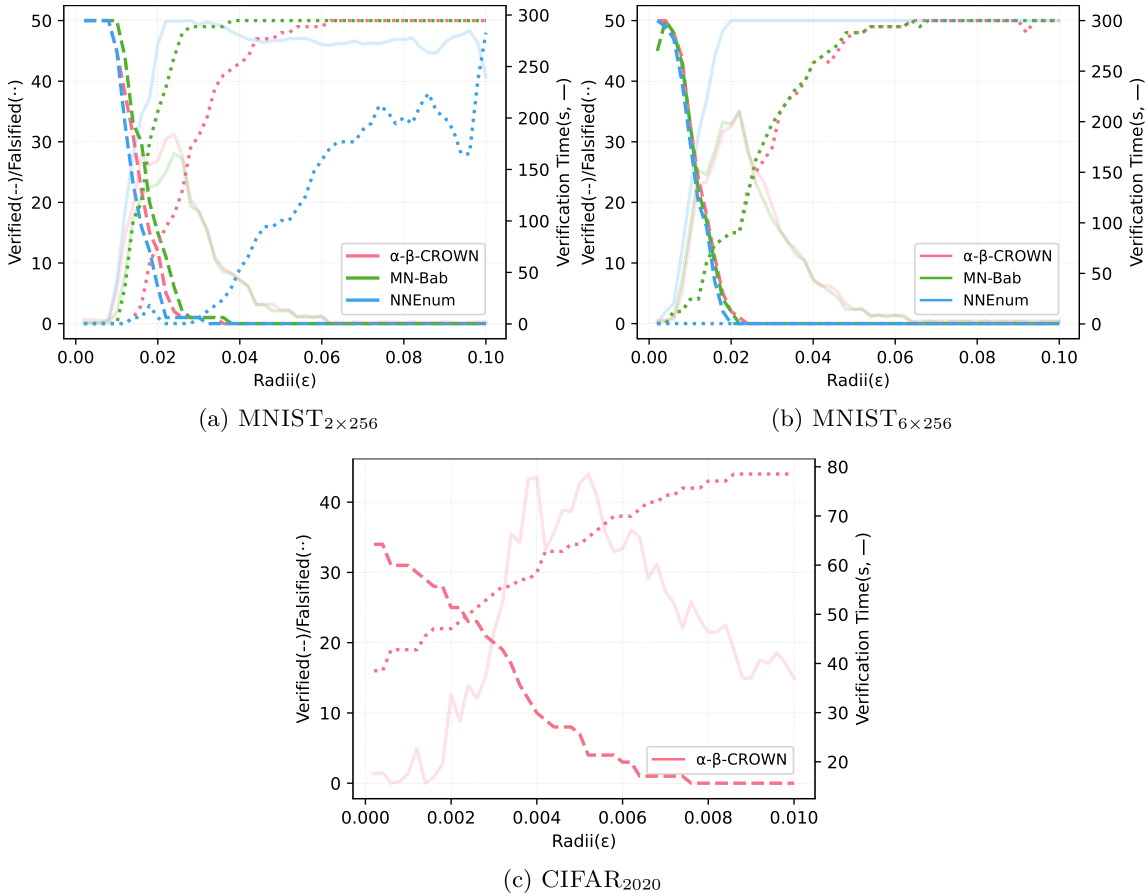
Controlling the ground truth of the verification problems is crucial to create balanced verification benchmarks. The SMT solver competition emphasizes the importance of including an equal number of satisfiable and unsatisfiable benchmarks when designing these benchmarks (Barrett et al., 2013). As mentioned in § 2.3, the effort and complexity involved in verifying a property as either UNSAT or SAT can vary significantly. In general, falsifying a property by finding a single counter-example is

much easier when compared to proving a property, ensuring a post-condition of a program for all possible input sets.

We notice that the **epsilon radius** factor is a unique parameter in the verification problem. Modifying any factors associated with the neural network requires retraining, which is a costly process. Altering the **centerpoint** factor involves selecting a different test input center point, and this factor is considered "unordered" because the distance from the center point inputs to the neural network's decision boundary is unknown.

However, the **epsilon radius** factor, which represents the size of the epsilon radius, is an "ordered" factor. When the **epsilon radius** value is small, the property's ground truth tends to be UNSAT, i.e., in extreme cases, the trivial property where **epsilon radius** $= 0$ is always unsatisfiable. The trivial property defines a single input point x, and the output of the neural network is always a point $\mathcal{N}(x)$. Therefore, it can never be on more than one side of the decision boundary, making the trivial property always unsatisfiable. On the other hand, when the **epsilon radius** value is larger, the property's ground truth tends to be SAT. In the extreme case, the global property where **epsilon radius** $= 1$ is always satisfiable. The global property encompasses the entire possible input set for the neural network. For any non-trivial neural network, it is always possible to find a point in the input set, known as a counter-example, that lies on the other side of the decision boundary compared to the center point.

Based on the aforementioned conjecture, we design a research study aimed at comprehending the impact of **epsilon radius** on the ground truths of the verification problems. The study is conducted on the $\text{MNIST}_{2\times256}$, $\text{MNIST}_{6\times256}$, and CIFAR_{2020} (Tab. A.1) architectures. The $\text{MNIST}_{2\times256}$ and $\text{MNIST}_{6\times256}$ networks are fully connected MNIST ReLU networks comprising two or six hidden layers, respectively, where each layer consists of 256 neurons. CIFAR_{2020} is a convolutional network containing two convolutional layers and one fully connected layer. To assess the benchmark, we introduce variations in the **epsilon radius** factor, incrementing it by $2e-4$ from 0.01 to 0.1 for the MNIST networks, and incrementing it by $2e-5$ from 0.001 to 0.01 for the CIFAR_{2020} network. For every epsilon radius, we evaluate 50 center points, resulting in a total of 7.5K verification problems. The experiment is constrained by a time limit of 300 seconds and a memory limit of 8GB. To evaluate



(a) MNIST$_{2\times256}$

(b) MNIST$_{6\times256}$

(c) CIFAR$_{2020}$

Fig. 4.2: Verification Problems Solved (Proved/Falsified) and Verification Time vs. Perturbation Radii of Three Neural Network Verifiers (**dashed lines**: problems proved; **dotted lines**: problems falsified; **solid lines**: solve time.)

the benchmark, we employ three cutting-edge verifiers: α,β-CROWN, MN-BaB, and Nnenum chosen from Tab. A.2.

The graphs in Fig. 4.2 illustrate the verification problems that were either proven or falsified, as well as the corresponding verification time for each of the three verifiers over the three network architectures. The X-axis stands for the incremental **epsilon radius** selections. The solid lines are the average verification times of 50 problems. The dashed lines are the number of problems proved to be UNSAT, and the dotted lines are the number of problems falsified to be SAT. It is

evident from 4.2a and 4.2b that for the MNIST architectures, the MN-BaB verifier outperformed the other two verifiers. MN-BaB was able to prove and falsify a greater number of properties compared to the other two verifiers, regardless of the epsilon radii, while maintaining a lower overall time cost. Although all three verifiers demonstrated similar capabilities in proving properties, their ability to falsify properties differed significantly. MN-BaB is the best verifier at falsification and α,β-CROWN is the second. Nnenum being the worst of the three in falsification, and even fails to falsify $\text{MNIST}_{6\times256}$ and CIFAR_{2020} networks due to memory limitation. 4.2c only depicts the results of α,β-CROWN. This is because Nnenum failed to falsify properties due to the memory limitation, and MN-BaB's implementation simply doesn't support the CIFAR_{2020} convolutional architecture.

Interestingly, the graphs in Fig. 4.2 reveal that the **epsilon radius** factor reaches cross-points at 0.016 for the MNIST networks and 0.0022 for the CIFAR_{2020} network, i.e., the number of unsatisfiable (UNSAT) and satisfiable (SAT) problems becomes equal. Additionally, the verification times for all three verifiers peak slightly after this cross-point. This suggests that the verification problems in the vicinity of the cross-point pose a significant challenge. These specific robustness problems lie on the decision boundary, thus demanding verification methods to spend additional effort to evaluate their validity, i.e., spend extra SMT calls or more iterations in abstract refinements.

It is worth noting that verification problems of the MNIST networks with an **epsilon radius** value less than 0.008 or greater than 0.04 can be easily proven to be UNSAT or falsified to be SAT. We believe these problems are not valuable to be considered when designing verification benchmarks for empirical studies. However, it is important to emphasize that this evaluation of the epsilon radii region is specific to this particular MNIST neural network architecture. The cross-points and the challenging region may vary for different networks.

The **epsilon radius** study suggests that manipulating the epsilon radius proves to be a successful method for managing both the ground truth and the complexity of the verification problems. Problems with epsilon values that are relatively smaller or larger are deemed too simple to prove or falsify, and therefore, it is advisable to refrain from incorporating such problems when designing an NNV benchmark. This benchmark design philosophy significantly impacts the experimental design

discussed in § 7. We also envision more future research alone this line of work in § 8.

4.3 Conclusion

This chapter presents an exploratory study that effectively identified nine factors that impact verification performance. Additionally, it delves into the examination of the **epsilon radius**, demonstrating the potential of manipulating NNV benchmark difficulty by controlling the **epsilon radius** values. These fundamental findings serve as a foundation for our subsequent research endeavors. For example, in § 5, a novel approach is proposed to generate diverse benchmarks by controlling the combination of factors. § 6 focuses on exploring the factor space to determine verification performance boundaries for a specific verifier. In § 7, the experimental design utilizes the **epsilon radius** factor to regulate the difficulty of the benchmarks. Furthermore, § 8 evaluates a more challenging benchmark using the proposed method from § 5, surpassing the state-of-the-art benchmarks in VNN-COMP.

Chapter 5

Systematic Generation of Diverse Benchmarks for Neural Network Verification

Over the last six years, there has been a significant emergence of over 50 neural network verification approaches (refer to § 3.3). These verification methods exhibit variations in terms of algorithmic families, performance capabilities, supported architectures, properties, and network formats, among other factors. Consequently, comprehending the disparities in their performance has become an imperative and formidable undertaking to effectively utilize them in practical applications.

The NNV community has developed an extensive collection of benchmarks (refer to § 3.2) that exhibit variations in architectures, datasets, software domains, and more. Many of the NNV benchmarks are created by the verifier developers [45, 128, 110]. However, the risk in letting technique developers choose their own benchmark is selection bias – that the selected problems do not represent a broad or important population of problems. Moreover, the complexity and diversity of these benchmarks have rapidly become outdated. Notably, approximately 89% of the problems in VNN-COMP are deemed excessively simple, as they can be efficiently solved within 30 seconds by

state-of-the-art verifiers on contemporary hardware [42].

The verification community is committed to finding benchmarks that fulfill specific criteria (see § 3.1). The three primary requirements are,

- diversity in difficulty (**R1**) [116],
- reflecting real-world applications (**R2**) [24],
- having a reasonable level of difficulty (**R3**) [11]

This chapter reports a novel approach, GDVB [1], the first *framework for systematic **G**eneration of **D**iverse neural network **V**erification **B**enchmarks*, that meets the de-facto requirements for verification benchmarks, **R1-R3**, for the rapidly evolving field of NNV. GDVB takes a **generative** approach to benchmark development – an approach that has risen in popularity in recent years [78, 2, 144]. Unlike, other generative benchmark approaches GDVB seeks to systematically cover variations in verification problems that are known to influence verifier performance.

Towards that end, GDVB is parameterized by: (1) a set of **factor**s known to influence the performance of neural network verifiers; (2) a **coverage** goal that determines the combination of factors that should be reflected in the benchmark; and (3) a **seed** verification problem from which a set of variant problems are generated. From these parameters, it computes a constrained mixed-level covering array[32] defining a set of factor-value tuples. Each tuple defines how the seed verification problem can be transformed to give rise to a verification problem capable of exposing performance variation in a neural network verifier.

As a benchmark generator GDVB naturally meets requirement **R3**. By starting from a seed network representing an NNV use case, GDVB is guaranteed to meet **R2**. As we discuss in §5.1, the use of factors allows GDVB to produce systematically diverse verification problems both in terms of structure and difficulty to meet requirement **R1**.

Moreover, GDVB offers the potential to reduce selection bias in performing evaluations of NNV approaches since it assures coverage of a space of performance-related factors. Finally, GDVB is designed to support the rapidly evolving field of NNV by allowing the generation of benchmarks,

[1]The GDVB approach introduced in this chapter is published in the CAV2020 conference [140].

e.g., from new seeds as verifiers improve, as new performance factors are identified, and to target challenge problems in different neural network domains, e.g., regression models for autonomous UAV navigation [88, 113].

The benchmarks produced by GDVB exhibit a significantly greater range of diversity compared to the benchmarks utilized in published papers of Neurify and ERAN. The extensive analysis illustrates that the utilization of GDVB benchmarks effectively distinguishes 9 cutting-edge verifiers in 2020. Our subsequent research conducted four years after the publication has confirmed that the benchmarks generated by GDVB continue to possess a much higher level of diversity in terms of network complexity and difficulty when compared to the most recent VNN-COMP benchmarks as discussed in § 3.2.

The contributions of this chapter are:

1. identification of the need for unbiased and diverse benchmarks for NNV
2. the specification of a verification benchmark as the solution to a constrained mixed-level covering array problem
3. the GDVB algorithm for computing a benchmark from a verification problem by transforming the neural network and property specification
4. the evaluation of GDVB on multiple state-of-the-art neural network verifiers in 2020 using different seed verification problems that demonstrate how GDVB results can support the evaluation of verifiers;
5. a verification execution and analysis framework named SwarmHost
6. and the open-source framework of GDVB.

5.1 Approach

The goal of GDVB is to meet requirements **R1-R3** by producing a *factor diverse* benchmark that (a) reflects aspects of the complexity encoded in a real verification problem that acts as a seed for

generation $\langle \mathcal{N}_s, \phi_s \rangle$, (b) varies aspects of the problem that are related to verifier performance, (c) accounts for interactions among those factors, and (d) is only comprised of well-defined verification problems.

Definition 3 *Constrained Mixed-level Covering Array (Def. 2.9 from [32])*
$CMCA(N; t, k, (|v_1|, |v_2|, ..., |v_k|), C)$ is an $N \times k$ array on $|v|$ symbols, where $|v| = \sum_{i=0}^{k} |v_i|$, with the following properties: 1) Each column $i(1 \leq i \leq k)$ contains only elements from a set S_i of size $|v_i|$, 2) the rows of each $N \times t$ subarray cover all t-tuples of values from the t columns at least one time, and 3) all rows are models of C.

A covering array defines a systematic method for testing how combinations of parameter values influence system performance [33]. A covering array is an $N \times k$ array. The k columns represent *factors* that may influence performance and cells can take on v *levels* – defining settings for factors. The N rows of the array define combinations of factor-levels. Arrays are defined to achieve a *strength* of the coverage, t. $t = 2$ defines pairwise strength, which means that all pairs of levels for all factors are present in some row of the covering array.

Definition 4 *A neural network verification performance **factor**, f, is a directly modifiable parameter of the verification problem, that contributes to the verification problem's difficulty, such that at least one verifier is sensitive to.*

Definition 5 *For a given seed verification problem, the **level**, l_f, of a factor, f, is a scaling ratio of the desired new verification problem when compared to the seed verification problem.*

We require a richer form of covering array that permits the number of levels to vary with different factors, i.e., a mixed-level covering array (MCA), and that can constrain specified factor-level combinations, e.g., by forbidding their inclusion in the MCA. By modeling each factor as a variable and its levels as the domain of the variable, one can express constraints as propositional logic formulae over equality terms; if the levels are ordered then richer underlying theories can be applied. A constrained-MCA defines an MCA that is consistent with a given constraint, C.

Rather than synthesize random verification problems, we seed the generation process to generate a benchmark that reflects the complexity of the seed problem. This permits benchmarks to be generated to reflect the challenges present in different neural network problem subdomains. Factors, like those described in § 4.1, may interact; changes to one factor may mask or amplify neural network verifier performance changes arising from another. Exploring all combinations of factors is expensive, but by using covering arrays we can systematically explore interactions among factors. Accounting for such interactions helps to produce a benchmark that is *less biased* than one that only covers individual factor variations. Not all combinations of factors are possible. For example, if one reduces the number of layers in a network to 0, then it is not possible to preserve the number of neurons in the original network. Thus, benchmark generation must take into account constraints among factors to ensure that only well-defined problems are included in a benchmark.

5.1.1 Factor Diverse Benchmarks

Consider a set of **factor**s (**Def.** 4), F, with a set of **level**s (**Def.** 5), L_f, for each factor, $f \in F$; we refer to L_f as the *level set* of f. For a verification problem, p, let $l(p)$ be the set of factor levels corresponding to the problem. A benchmark, B, is a set of verification problems, and we can denote the factor levels for the benchmark as $l(B) = \{l(p) \mid p \in B\}$.

The simplest form of diversity for a benchmark is requiring that all individual factor levels be present in at least one verification problem, $\forall f \in F : \forall l \in L_f : \exists p \in l(B) : l \in p$. However, this diversity fails to account for interactions among factors. The simplest form of interaction-sensitive diversity considers pairs of factors, but as we discuss below our approach generalizes to any arity of factor-level coverage.

For a pair of factors, $f, f' \in F$, the Cartesian product of their level sets defines the set of all pairwise combinations of their levels. Across all factors the set of such pairs is

$$\textsc{pairs}(F) = \{(l, l') \mid f, f' \in F \wedge f \neq f' \wedge l \in L_f \wedge l' \in L_{f'}\}$$

A *pairwise diverse benchmark* is one in which

$$\forall (x,y) \in \textsc{pairs}(F) : \exists p \in l(B) : (x,y) \in \{(x',y') \mid x' \in p \wedge y' \in p\}$$

Constraints on allowable combinations of factors serve to restrict a benchmark. A pairwise exclusion constraint, $\gamma(F) \subseteq \textsc{pairs}(F)$, requires that

$$\forall (x,y) \in \gamma(F) : \forall p \in l(B) : \neg(x \in p \wedge y \in p)$$

We write γ when F is understood from the context.

The arity of factor-level coverage and exclusion constraints can vary independently. It is common for factor-level coverage to be uniform and to generalize it to t-way coverage, i.e., to require coverage of the elements of the Cartesian product of the level sets of t factors. On the other hand, as observed in prior work [32], constraints generally involve a mix of arity. To denote this generality we define $\Gamma \subseteq \bigcup_i \gamma_i$ where γ_i defines the set of possible i-way exclusion constraints.

Example 5.1.1 *Factor Diversity*

Consider the DAVE-2 network which accepts 100 by 100 color images and infers an output indicating the steering angle [97]. DAVE-2 consists of 5 convolutional layers with 55296, 17424, 3888, 3136, and 1600 neurons, respectively, followed by 4 fully connected layers with 1164, 100, 50, and 10 neurons, respectively. All 82668 neurons use ReLU activations. One can define a local robustness property for DAVE-2 as

$$\phi = \forall \boldsymbol{x} \in i \pm 0.02 : \|\text{DAVE-2}(\boldsymbol{x}) - \text{DAVE-2}(i)\| \leq 5$$

which states that for a given input image, i, all inputs within a distance of 0.02 will result in an inferred steering angle within 5 degrees of the angle for i. These yield the verification problem $\langle \text{DAVE-2}, \phi \rangle$.

Consider factors for the number of neurons, number of convolutional layers, and number of fully connected layers; a tuple $(\#neuron, \#conv, \#fc)$ represents levels for these factors. For each factor consider two percentage levels: 100% and 50%. A neuron factor level of 50% indicates that

a version of DAVE-2 with 41334 neurons is required. In the absence of constraints, an example pairwise factor diverse benchmark for $\langle \text{DAVE-2}, \phi \rangle$ consists of the following four verification problems: $(100\%, 100\%, 100\%)$, $(100\%, 50\%, 50\%)$, $(50\%, 100\%, 50\%)$, and $(50\%, 50\%, 100\%)$. The property ϕ is constant across the benchmark.

5.1.2 From Factor Covering Arrays to Verification Problems

Given a set of factors, $F = \{f_1, f_2, \ldots, f_{|F|}\}$, and levels, L_{f_i}, a t-way factor diverse benchmark of k verification problems is specified by

$$CMCA(|F|; t, k, (|L_{f_1}|, |L_{f_2}|, \ldots, |L_{f_{|F|}}|), \Gamma)$$

Each element in this mixed-level covering array specifies how to construct a verification problem in the benchmark from the seed problem.

Levels are operationalized as transformations on verification problems. We assume a sufficient set of transformations, Δ, such that a verification problem can be transformed into a form that achieves any level of any factor

$$\forall f \in F : \forall l_f \in L_f : \exists \delta \in \Delta : l_f \in l(\delta(\langle \mathcal{N}_s, \phi_s \rangle))$$

The definition of Δ and L_i must be coordinated to achieve this property.

A per-factor transformation $\delta \in \Delta$ may impact a single component of a verification problem, e.g., reducing the number of neurons in a neural network does not impact the property, or both components, e.g., the input dimension impacts the neural network and the property by transforming the input data domain. The set of all transformations Δ defines the set of verification problems that can be produced by the application of a set of per-factor transformations to the seed problem,

$$\Delta(\langle \mathcal{N}_s, \phi_s \rangle) = \{\langle \mathcal{N}, \phi \rangle \mid \langle \mathcal{N}, \phi \rangle = \delta_{f_1} \circ \delta_{f_2} \ldots \circ \delta_{f_{|F|}}(\langle \mathcal{N}_s, \phi_s \rangle) \wedge \delta_i \in \Delta\}$$

The set of all possible factor level combinations is $\Pi_{f \in F} L_f$, i.e., the product of all of the per-factor

Alg. 1: The GDVB Algorithm

Data: A seed verification problem $\langle \mathcal{N}_s, \phi_s \rangle$, a set of factors F and constraints Γ, and a coverage goal t.

Result: A benchmark of NNV problems B

```
1 C ← genCMCA(F, Γ, t)
2 B ← ∅
3 for c ∈ C do
4 |  B ← B ∪ transform(⟨N_s, φ_s⟩, c)
5 end
```

levels. The set of t-way factor level combinations is

$$c_t = \{c | a \in \Pi_{f \in F} L_f \wedge c \subseteq a \wedge |c| = t\}$$

allowing for the interpretation of $|F|$-tuples as sets.

Definition 6 *Given a set of factors F, with associated factor levels L_f, a t-way factor diverse benchmark, B, for a seed problem $\langle \mathcal{N}_s, \phi_s \rangle$ with exclusion constraints Γ is defined by the following:*

- $B \subseteq \Delta(\langle \mathcal{N}_s, \phi_s \rangle)$;
- $\forall \langle \mathcal{N}, \phi \rangle \in B : \forall \gamma \in \Gamma : \gamma \not\subseteq l(\langle \mathcal{N}, \phi \rangle)$; *and*
- $\forall c \in c_t - \Gamma : \exists \langle \mathcal{N}, \phi \rangle \in B : c \subseteq l(\langle \mathcal{N}, \phi \rangle)$

5.1.3 Benchmark Generation

GDVB is defined in Alg. 1. We use existing techniques, e.g., Automated Combinatorial Testing for Software(ACTS) [76], for generating a CMCA for constraints specified as logical formulae where factors are variables and levels are values for those variables. A CMCA is a set of k-tuples. Each such tuple defines the target level for each factor for a problem in the generated benchmark. Those levels are used to transform the given seed verification problem and the resultant problem is accumulated in the benchmark.

TRANSFORM uses different approaches to transform the seed neural network and the property. Neural network transformation builds on an approach called R4V that automates architectural

transformations by scaling (1) the number of neurons in a fully connected layer, (2) the number of kernels in a convolutional layer, (3) the input dimension, or (4) the range of values within an input dimension [106]. The first 3 of these require changes to the structure of the neural networks and the last two require changes to the training data, e.g., reshaping, renormalizing. R4V ensures that the network is well-defined after transformation. TRANSFORM maps factor-levels to per-layer scale parameters for R4V.

R4V permits the training of a network using network distillation which we find advantageous for GDVB because: it accelerates the training process, and it drives training to match the accuracy of the problem neural network to that of $\mathcal{N}_s$, which reduces variation in accuracy across B. We adopt R4V so that after each training epoch, the learned parameters and the validation accuracy are recorded. When training finishes, we select the models associated with the highest validation accuracy. Training is performed using the training data and hyperparameters for $\mathcal{N}_s$.

Whereas R4V can be used to directly manipulate neural network architecture-related factors, it can only indirectly affect the learned weights. To address this, we adopt the approach taken throughout the machine learning literature – train a network on multiple initial seeds and report performance across seeds. Thus, each neural network in B is trained multiple times, thereby producing a benchmark comprised of $s * |B|$ verification problems, where is the desired number of seeds.

Example 5.1.2 *Neural Network Transformation*

Consider this element of the CMCA described above: $\langle(50\%, 100\%, 50\%), \phi\rangle$, applied to DAVE-2. TRANSFORM would compute that 50% of the fully connected layers should be present in the resultant neural network and randomly select 2 of the 4 layers to scale by 0. The fully-connected layers are chosen at random since the layer count factor does not consider layer ordering. If we consider the case where the layers with 100 and 50 neurons are dropped, this will eliminate 150 neurons. The other transformation required is to reduce the number of neurons by half. To do that all remaining layers will be scaled by $\frac{82668*0.5-150}{82688} = 0.498$.

Property transformation builds on a domain-specific language (DSL) for specifying NNV properties defined by the *deep neural network verification framework* (DNNV) [104]. Specifications in this Python-based DSL are parametric and TRANSFORM maps factor-levels to those parameters. For

example, Listing 5.1 defines the parametric local robustness property ϕ that is centered at the image stored at "path/to/image", has radius 0.02, and can be translated and scaled through parameters `t` and `s`, respectively. Restricting factors to levels that are supported by TRANSFORM and using CMCA algorithms that meet § 2 **Def.** 3 ensures that GDVB produces a solution that meets § 2 **Def.** 6.

```
1  import numpy as np
2  N = Network("N")
3  s = Parameter("s", float, default=1.0)
4  e=0.02 * s
5  x = Image("path/to/image")
6  t = np.load(Parameter("t", str, "path/to/zeros.npy"))
7  x = x + t
8  Forall(x_,
9    Implies(
10   (x - e) < x_ < (x + e),
11   abs(N(x_) - N(x)) <= 5
12 )
```

Listing 5.1: Parametric Property ϕ

5.1.4 Implementation

An instance of GDVB has been developed to accommodate a range of factors that have been determined based on the findings of the study in § 4.1. These factors are characterized by *percentage-based levels*, and are accompanied by a set of constraints that ensure benchmark problems are both non-trivial and capable of efficient training.

The instantiation of GDVB includes support for the following factors: the total number of neurons in the neural network (**neuron**), the number of Fully Connected layers (**fully connected layer**), the number of Convolutional layers (**convolutional connected layer**), the dimension of the input (**input dimension**), the size of each input dimension (**input size**), the perturbation size of the property (**epsilon radius**), and the center points of the property (**centerpoint**). However, it is

important to note that the **activation function** factor is currently not supported due to limitations within the underlying R4V library. We anticipate addressing this limitation in future iterations of GDVB.

We note that the implementation of GDVB is flexible in that it permits the customization of levels, as we demonstrate in the next section, to generate a benchmark that focuses on variation in a subset of factors. More generally, GDVB can easily be extended to support additional factors and levels for which an instance of TRANSFORM can be defined. We expect that GDVB will evolve in this way as studies of neural network verifiers are performed. The implementation of GDVB is open-source and can be accessed at the following URL: `https://github.com/edwardxu0/GDVB`.

5.1.5 The SwarmHost Verification Framework

While the main purpose of GDVB is to generate diverse NNV benchmarks. It also contains a pipeline that supports verifier executions and result analysis for easier analysis of empirical studies. It currently supports three verification frameworks: DNNV, DNNF, and the novel SwarmHost framework.

Due to the fast development of the NNV field, most of the supported verifiers in DNNV are now outdated. The GDVB coalition is extended with a new verification framework called SwarmHost. This new framework supports more modern verifiers like α,β-CROWN [129], MN-BaB [51], Nnenum [8], verinet [62], and NeuralSAT [41]. Additionally, it can generate local robustness properties in the VNNLIB format [38] supported by these tools. Moreover, it provides APIs that enable uniform execution and result parsing to aid our later research, such as AdaGDVB and OCTOPUS. One notable feature of SwarmHost is its modular design, which eases the integration of new verifiers. To add a new verifier to the pipeline, users only need to implement a Python interface for verifier execution and result parsing. The implementation of SwarmHost is open-source and can be accessed at the following URL: `https://github.com/edwardxu0/SwarmHost`.

5.1.6 Scaling Up Neural Networks with Enhanced R4V

Note that when the **level** of a **factor** is greater than 1, this means that the resulting verification problem is "larger" than the original seed problem. GDVB supports scaling up of the **epsilon radius**, **input dimension**, **input size**, and **neuron** factors natively. However, when dealing with the number of **fully connected layer** and the number of **convolutional connected layer** factors, additional work is required with the underlying R4V library. We have enhanced the R4V library used by GDVB to support the addition of both fully connected and convolutional layers to an existing network. Users can also specify the location, type, size, and activation functions of the added layer. With this extension, GDVB is not limited by starting with a large and complex neural network. Instead, users can easily begin with a small network and still generate a challenging benchmark by scaling up the verification problems. The enhanced implementation of R4V is open-source and can be accessed at the following URL: `https://github.com/edwardxu0/R4V`.

5.2 Evaluation

In this section, we showcase the potential uses of GDVB across a series of artifacts and verifiers, while highlighting the challenges it helps to systematically address.

5.2.1 Evaluation Setup

5.2.1.1 Selection of Seed Verification Problems

The evaluation section examines two GDVB benchmarks through the utilization of two seed verification problems: $\text{MNIST}_{\text{ConvBig}}$ and DAVE-2 (Tab. A.1). The $\text{MNIST}_{\text{ConvBig}}$ network comprises 4 Convolutional and 3 Fully Connected layers. It stands as the most extensive MNIST network within the ERAN (§ 3.2) benchmarks, encompassing 48,074 neurons and 1,974,762 parameters. On the other hand, the DAVE-2 network is composed of 5 Convolutional and 5 Fully Connected layers, housing 82,669 neurons and 2,116,983 parameters. Both seed verification problems employ a 0.02 epsilon radius.

5.2.1.2 Selection of Verifiers

A set of state-of-the-art verifiers in 2020 are selected from Tab. A.2 to be assessed with the 2 GDVB benchmarks, including: ReLuplex, Planet, BaB, Neurify, and ERAN. We use Branch-and-Bound (BaB), as well as a variation of Branch-and-Bound with Smart-Branching (BaBSB). Additionally, we evaluate the ERAN verifier with 4 available abstract domains: $\text{ERAN}_{\text{DeepZono}}$, $\text{ERAN}_{\text{DeepPoly}}$, $\text{ERAN}_{\text{RefineZono}}$, and $\text{ERAN}_{\text{RefinePoly}}$. The 9 representative verifiers spam across multiple algorithmic families, including: search, optimization, and reachability (refer to § 3.3.)

5.2.1.3 Selection of Factors and Levels

To ensure consistency, all applicable influencing **factor**s discovered in § 4 are parameterized into quintile settings, i.e., 5 evenly distributed **level**s per **factor**. The **activation function** factor is left out to be the same ReLU activation function as the seed problems. The **layer** and **layer type** are combined to form the two new factors: the number of Fully Connected layers and the number of Convolutional layer factors. The levels of these two factors use exactly the number of the actual layers, such that the number of layers is always an integer. All the other **factor**s have quintle **level**s, i.e., $(\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1)$ relative to the seed verification problem. For **centerpoint**, we select a set of five center points that shift on a different instance of the test data; unlike the above levels, this level is unordered.

As for the constraints, Γ, is constructed based on whether certain combinations of the factor levels will permit valid neural network architectures. These constraints were developed iteratively based on feedback from the R4V tool, which reports when TRANSFORM has specified an invalid neural network, and when training failed to closely approximate the accuracy of the seed network.

Example 5.2.1 Our instantiation of GDVB exclusion constraints for DAVE-2 are as follows: (1) $fc = 0 \land conv = 0$, (2) $conv = 0 \land neu \geq 20$, (3) $conv = 0 \land idm \geq 80$, and (4) $conv = 100 \land idm = 20$. The first of these requires that some layer be present. The second and third are related to the blowup in the size of fully-connected layers that results from dropping all convolutional layers, which makes training difficult; limiting the total number of neurons and the reduction of input dimension mitigates

this. The fourth constraint ensures that the input dimension reduction results in a meaningful network; without it, the dimensionality reduction achieved by sequences of convolutional layers yields an invalid network, i.e., the input to some layer is smaller than the kernel size.

5.2.2 Selection of Metrics and Resources

To evaluate verifier performance, we use the SCR and the PAR-2 (refer to § 3.1) metrics. The SCR counts solely and number of solved problems and PAR-2 also aggregates to solve time. All training and verification took place under CentOS Linux 7. R4V transformation and distillation jobs ran on NVIDIA 1080Ti GPUs. Verification jobs were limited to 4 hours and ran on 2.3GHz and 2.2GHz Xeon processors with 64GB of memory, for DAVE-2 and $\text{MNIST}_{\text{ConvBig}}$, respectively.

5.2.3 Results

5.2.3.1 Comparing verifiers across a range of challenges

Consider the use case where a researcher is attempting to compare a new verifier (e.g., a new algorithm, a revised implementation, or an extension to an existing approach) against existing verifiers. As shown earlier, for such a comparison to be meaningful, many factors must be considered and properly explored. Given a seed network, a property, a set of factors, and a coverage goal, GDVB can generate a benchmark that helps to reduce bias in conducting such an evaluation.

For this use case, we consider seed networks and local robustness properties similar to those from the $\text{ERAN}_{\text{DeepZono}}$ study [110] for the $\text{MNIST}_{\text{ConvBig}}$ verification problem and local robustness properties based on those from the Neurify study [127] for the DAVE-2 verification problem. We run an instance of GDVB using the factors and levels described in Sect. 5.2.1.3, a coverage strength of 2, and train 5 versions of each network to account for stochastic parameter variation. The total time to generate and train GDVB ($\text{MNIST}_{\text{ConvBig}}, \ldots$) was 24.3 hours and the resulting 30 verification problems took 401.8 hours to run across all 9 verifiers. For GDVB (DAVE-2, . . .) 44 verification problems were generated with training and verification times of 158.2 hours and 772.4 hours, respectively. CMCA generation took less than a minute for both problems. Each problem in

	MNIST$_{\text{ConvBig}}$		DAVE-2	
Verifier	SCR	PAR-2	SCR	PAR-2
ERAN$_{\text{DeepZono}}$	11.40±0.49	18,126.80±488.27	7.20±1.94	24,496.20±1,176.59
ERAN$_{\text{DeepPoly}}$	21.00±0.89	9,206.00±806.70	18.40±2.15	17,443.00±1,344.65
ERAN$_{\text{RefineZono}}$	10.20±0.40	19,252.60±343.66	5.80±2.14	25,236.60±1,253.90
ERAN$_{\text{RefinePoly}}$	12.60±1.02	16,981.40±930.71	10.20±1.83	22,250.60±1,186.44
Neurify	22.00±1.10	8,636.20±1,008.63	19.20±2.56	17,247.80±1,397.05
Planet	7.00±0.63	23,145.60±468.18	3.40±1.62	27,268.60±775.56
BaB	0.20±0.40	28,689.80±220.40	0.00±0.00	28,800.00±0.00
BaBSB	0.00±0.00	28,800.00±0.00	0.00±0.00	28,800.00±0.00
ReLuplex	3.20±0.40	25,757.80±381.40	4.40±1.02	26,023.60±635.90

Tab. 5.1: Mean & Variance of SCR and PAR-2 Scores Across Benchmarks (The darker and lighter gray boxes indicate the best and second best results)

the benchmark must be trained and verified in sequence, but across problems, they can be parallelized. We exploited this to reduce the cost of running the benchmarks to 4.9 hours for MNIST$_{\text{ConvBig}}$ and 7.9 hours for DAVE-2. We measured the SCR and PAR-2 scores for the nine verifiers across the benchmarks.

The results are shown in Table 5.1. Since the SCR and PAR-2 score trends are the same we depict just SCR in Fig. 5.1. Boxplots show the SCR scores for a verifier across all the generated problems; variation in plots arises from the 5 trained versions of the networks for each problem. For each box, the middle line represents the median, the box-bounds are the first and third quartiles, and the whiskers represent minimal and maximal values.

The plot for MNIST$_{\text{ConvBig}}$ on the left of Fig. 5.1 shows that **the GDVB benchmark with the MNIST$_{\text{ConvBig}}$ seed can identify considerable performance variation across verifiers**, with ERAN$_{\text{DeepPoly}}$ and Neurify accurately verifying a median of over 20 properties, the rest of the ERAN-variants verifying between 10 and 13 properties, and the remaining tools verifying between 0 and 8 properties. The results are consistent when we employ DAVE-2 as the seed network, with **marked differences among groups of verifiers**. However, the generated problems turned out to be more challenging across all verifiers. ERAN$_{\text{DeepPoly}}$ and Neurify, the top performers, can verify less than half of the generated problems. Verifiers like BaB were unable to verify any problem derived from DAVE-2 because of the complexity of the seed problem. This point highlights the need

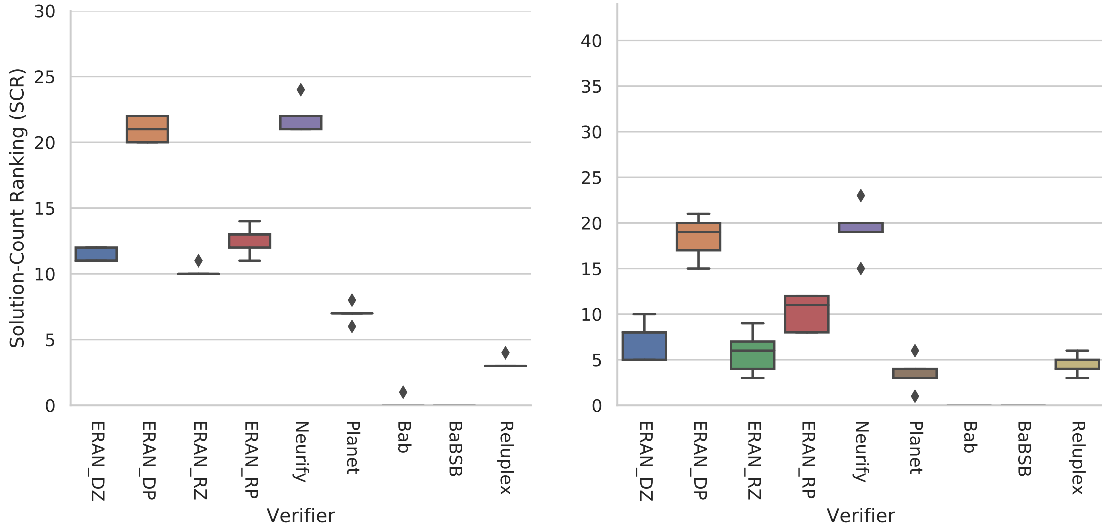
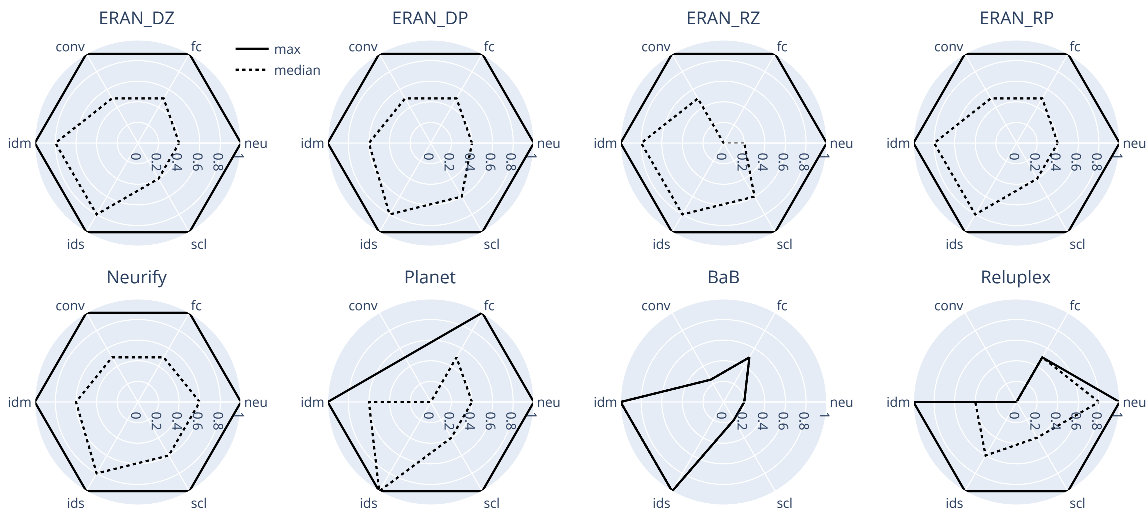


Fig. 5.1: SCR Score for Nine Verifiers on GDVB Benchmarks with $\text{MNIST}_{\text{ConvBig}}$ (left) and DAVE-2 (right) Seed Problems

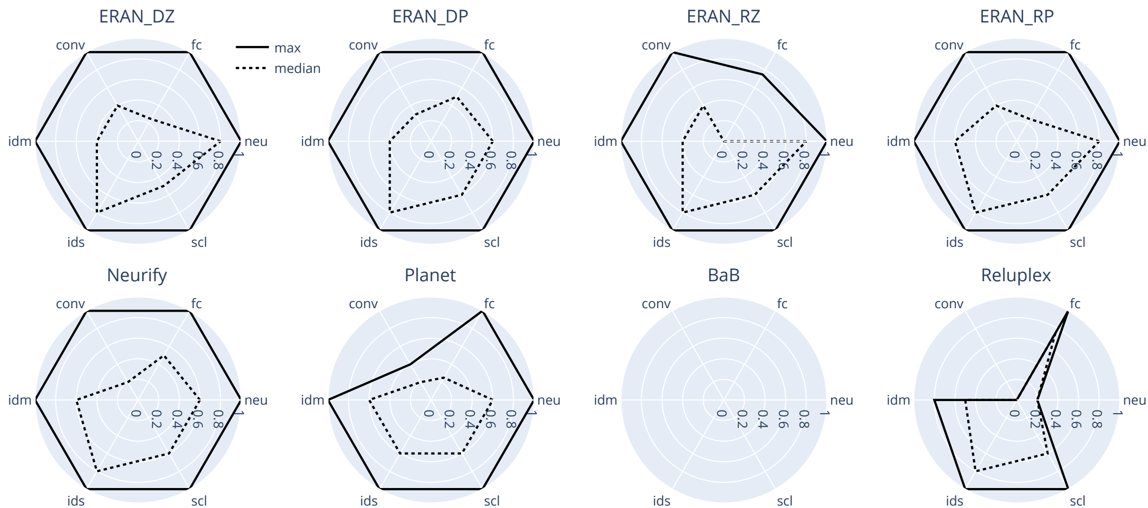
for benchmarks to evolve with networks that incorporate emerging technology, and also GDVB's ability to automatically generate a benchmark from different seeds to address that need.

Now, understanding the overall performance of a family of verifiers is useful, but it is likely just the first step for a researcher to understand under what conditions a verifier excels or struggles. When such conditions correspond to the factors manipulated by GDVB, then they are readily available for further analysis. One analysis may consist of simply plotting the data across its multiple dimensions. We do so in the form of radar charts for DAVE-2 in Fig. 5.2b and for $\text{MNIST}_{\text{ConvBig}}$ in Fig. 5.2a We do not plot BaBSB as its performance was identical to BaB. Since the observations we can gather from both networks are similar, we just discuss DAVE-2 in detail. Each chart includes six axes representing a factor scaled between 0 and 1. The solid lines link the maximum values across factors that were accurately verified while the dotted lines link the median values across factors.

The shape of the lines in the radar plots clearly shows that the **verification problems generated by GDVB reveal unique patterns across the verifiers**. For example, the ReLuplex plot indicates that it can do well-verifying networks with multiple fully connected (FC) layers but is challenged by larger networks (Neu) and those with convolutional layers (Conv). Comparing multiple



(a) $\text{MNIST}_{\text{ConvBig}}$ Artifact



(b) DAVE-2 Artifact

Fig. 5.2: Radar Plot with Maximum(solid) and Median(dotted) Values of the Two Artifacts

charts also reveals some interesting trade-offs. For example, for smaller networks with just fully connected layers, the medians seem to indicate that ReLuplex is better than Planet. However, when a network incorporates convolutional layers or a larger number of neurons, Planet appears to outperform ReLuplex.

Looking across charts can also pinpoint specific improvements resulting from tool extensions or

revisions. For example, the median line of $\text{ERAN}_{\text{RefineZono}}$ indicates that it was not as effective in handling verification problems with a larger number of layers as its predecessor $\text{ERAN}_{\text{DeepZono}}$; the same trend holds for the pair $\text{ERAN}_{\text{RefinePoly}}$ and $\text{ERAN}_{\text{DeepPoly}}$. We note that a more restrictive benchmark that is biased towards fewer fully connected layers might not reveal such differences.

GDVB offers the opportunity to investigate such differences even further by generating targeted verification problems for a subset of factors hypothesized to be culprits of those differences. For example, GDVB could generate additional verification problems with several fully connected layers between 60% and 80% of the total, while keeping the other factors constant, to refine the understanding of the differences between $\text{ERAN}_{\text{RefineZono}}$ and $\text{ERAN}_{\text{DeepZono}}$.

This study illustrates how GDVB benchmarks support the exploration of verifier performance, lowering the burden on researchers to manually prepare tens to hundreds of verification problems, and reducing the opportunities for bias.

5.2.3.2 GDVB and benchmark requirements R1-R3

As explained in Sect. 5, benchmarking in verification seeks to develop benchmarks that are: diverse; representative of real use cases; and reactive to new technologies. The previous sections have provided evidence of how, through its generative nature, GDVB is reactive to new advances in technology included in the seed network. We have also seen the high degree of parameterization GDVB offers including for setting a seed network from which realistic attributes are inherited in the generated verification problems. In this section, we want to illustrate how GDVB addresses the diversity requirement.

To depict diversity we use the parallel coordinate graph in Fig. 5.3. Each vertical line corresponds to a factor, and the markers in each vertical line correspond to an explored level. Each verification problem is a polyline that connects the factors' levels explored by it. The two sets of lines correspond to the verification problems included in the DAVE-2 benchmark published with Neurify, which is a downsized version of the full DAVE-2 network, and the benchmark produced by GDVB (DAVE-2, . . .). Each factor in the plot is normalized by dividing by the maximum value for the factor.

Fig. 5.3 shows that the Neurify's DAVE-2 has numerous neurons, inputs, and dimensions. Yet,

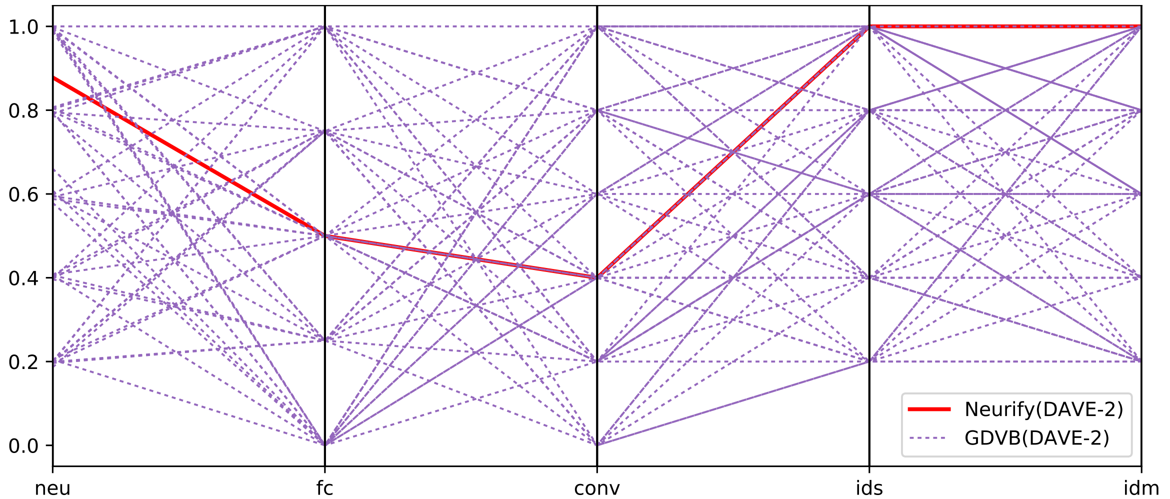


Fig. 5.3: Diversity Explored Across Factor Levels

it provides very limited coverage of all the factor levels that may affect verification performance. In contrast, GDVB provides a systematic exploration of the factor levels that can affect verifier performance making it much less biased – especially to the numbers of layers in the verification problems and the combination of those factor levels.

The parallel plot for GDVB benchmark with the $\text{MNIST}_{\text{ConvBig}}$ seed (not shown for space reasons), depicts a similar trend in terms of systematic exploration of diversity, but since $\text{MNIST}_{\text{ConvBig}}$ is simpler than DAVE-2, the generated benchmark is correspondingly simpler. This points to the need to identify representative and challenging seeds when parameterizing GDVB. GDVB is fully capable of accommodating factor levels that exceed 100% of a seed network, which is a means of pushing verifiers to the limits of their abilities.

We note that excluding factors or levels can yield a systematically generated benchmark that is unable to characterize differences between verifiers, or worse, misleads such a characterization by emphasizing certain factors while overlooking others. For example, not exploring different network sizes or exploring network sizes under 1000 neurons will render similar scores across many neural network verifiers that are differentiated by more comprehensive benchmarks. In applying GDVB, we suggest selecting as many factors as we know may matter, starting from a challenging seed problem, and incrementally refining the levels as needed to focus benchmark results to differentiate verifier

performance.

5.3 Conclusion

The increasing adoption of neural networks has led to a surge in research on verification techniques. Benchmarks to assess these emerging techniques, however, are costly to develop, often lack diversity, and do not represent the population of real evolving networks. To address this challenge, this chapter introduces GDVB, a framework for systematically generating NNV problems seeded in complex, real-world networks, ensuring that benchmarks are derived from real problems. GDVB is parameterizable by the factors that may influence verification performance and thereby supports scalable benchmarking. A preliminary study, using 9 verifiers, demonstrates how GDVB can support the assessment of the state-of-the-art.

The innovative method for generating benchmarks presented in this chapter serves as a source of inspiration for future research endeavors aimed at exploring the boundaries of verification performance in § 6. Additionally, it paves the way for the development of a sophisticated benchmark that surpasses the existing state-of-the-art benchmarks in VNN-COMP after four years in § 8.

Chapter 6

Adaptive Benchmark Generation for Neural Network Verification

As in other fields of verification, the comparative evaluation of techniques and tools serves as a driving force in scaling their performance. (refer to § 3.1) For the last 4 years, the annual neural network verification competition (VNN-COMP) has called for benchmarks that are "not so hard that none of the instances can be solved by any participant, but also not so easy that every tool can solve all of them [24]. Each instance of VNN-COMP [67, 7, 93, 23] has used benchmarks that are provided by verifier authors and has allowed verifiers to be tuned to benchmarks.

To avoid potential bias in benchmarking neural network verifiers, the GDVB approach (described in § 5) introduced the idea of systematically synthesizing benchmarks that are (i) diverse in difficulty, (ii) realistic to real-world use and that can (iii) evolve with advances in neural network verifier capabilities. GDVB identified 9 **factors** that could impact the difficulty of the verification problems, including the number of neurons (**neuron**), the number of layers (**layer**), and input dimension (**input dimension**), etc. It uses the concept of combinatorial interaction testing (CIT) [31] to generate the smallest possible benchmark that assures combinations of different scaling **levels** for each factor. The resulting benchmarks were used to characterize performance differences among 9

state-of-the-art verifiers in 2020, and more recently have been used to develop challenging benchmarks for state-of-the-art verifiers, as discussed in § 8.

In this chapter, we introduce AdaGDVB, an approach that enhances GDVB-based benchmark generation. It allows to automatically generate customized benchmarks for a given verifier. This is achieved by iteratively scaling the size of the verification problem until the verifier can no longer solve it. This happens across multiple dimensions of scaling and results in a benchmark that characterizes the *verification performance boundary* (VPB) of a verifier. Contrasting the VPB for two verifiers provides a new means of comparing their scalability. Besides, performance analysis focusing on benchmark problems on either side of the boundary can facilitate performance optimization.

In § 6.2, we illustrate two use cases for applying AdaGDVB. For competition holders, such as VNN-COMP, AdaGDVB can create adaptive benchmarks to showcase performance differences between verifiers using the computed VPB instead of just problems solved and solving time. For verifier developers, AdaGDVB can generate a benchmark that exposes the performance bottlenecks of their verifier and thereby facilitates algorithmic and implementation optimization.

The contribution of this chapter includes,

1. a novel approach to iteratively search for the VPB for a given verifier;
2. an open-source implementation of the main AdaGDVB framework;
3. and two use cases of the AdaGDVB approach.

6.1 Approach

The AdaGDVB method is based on the GDVB approach. The core concept involves utilizing the **factors** outlined in GDVB to create NNV instances that systematically vary in complexity. Subsequently, it employs an iterative binary search process to determine the given verifier's verification performance boundary, also known as VPB, as detailed in **Def.** 7.

Definition 7 *A neural network Verification Performance Boundary (VPB) is an NNV benchmark that contains a subset of solvable problems and another subset of unsolvable problems.*

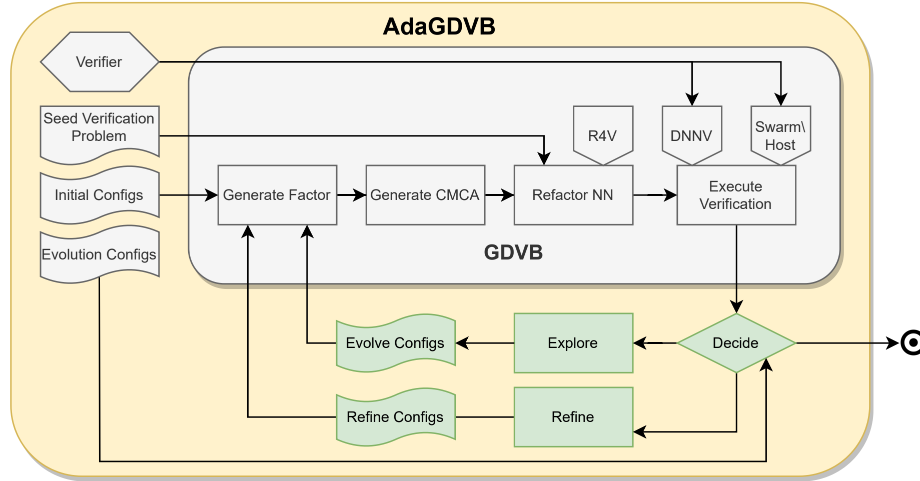


Fig. 6.1: Overview of AdaGDVB.

The VPB of a given verifier refers to a benchmark that encompasses both solvable and unsolvable problems. The VPB captures the decision boundary of the verifier's performance on the given verification problem. Considering the intricate nature of the NNV, which entails at least 9 **factors** that can influence the verifier's performance, the VPB becomes a non-trivial entity that may exhibit non-convexity and contain gaps. A trivial VPB consists of only two problems: the simplest and the most challenging verification problems. Conversely, a meaningful VPB comprises a limited number of benchmark problems that effectively capture the verifier's decision boundary, thereby minimizing the disparities between solvable and unsolvable problems, while delineating the verifier's behaviors.

6.1.1 Overview

The AdaGDVB framework is outlined in the Fig. 6.1. It builds upon the GDVB framework by incorporating an improved R4V (refer to § 5.1.6) library and introducing a verification framework, SwarmHost (refer to § 5.1.5), alongside the existing GDVB approach. The primary objective of GDVB is to create an NNV benchmark that systematically varies in complexity, execute the neural network verifier integrated with the verification framework, and evaluate the verification outcomes. The innovative AdaGDVB methodology consists of two main phases, namely **exploration** and

refinement. AdaGDVB aims to control the **factor-level** configurations of the next NNV benchmark based on the verifier's feedback of the current benchmark to continuously challenge the verifier until the VPB is identified.

6.1.2 The AdaGDVB Method

Alg. 2: The AdaGDVB Algorithm

Input : a seed verification problem $\langle \mathcal{N}_s, \phi_s \rangle$, a set of factors F, a set of initial levels for the selected factors L_0, their lower L^- and upper bounds L^+, refinement granularity g, a set of combinatorial constraints Γ, a neural Network verifier v, the maximum number of steps of the exploration(S_e) and refinement(S_r) phases

Output : An approximate verification boundary benchmark B with verification results R

```
1  t ← |F|
2  L ← L_0
3  P_u ← ∅, P_o ← ∅
   /* Exploration Phase                                    */
4  i ← 0
5  while i > S_e do
6      B ← GDVB(⟨N_s, φ_s⟩, F, L, Γ, t)
7      R ← Evaluate(B, v)
8      (P_u, P_o) ← UpdatePivots(R, L)
9      if (P_u ≠ ∅ ∧ P_o ≠ ∅) then
10         break
11     else
12         L ← ∅
13         forall f ∈ F do
14             L[f] ← Explore(L[f], f, P_u, P_o, L^-, L^+, R)
15     i ← i + 1
   /* Refinement Phase                                     */
16 i ← 0
17 while i < S_r do
18     L ← ∅
19     forall f ∈ F do
20         L[f] ← Refine(L[f], f, P_u, P_o, g)
21     B ← GDVB(⟨N_s, φ_s⟩, F, L, Γ, t)
22     R ← Evaluate(B, v)
23     i ← i + 1
24 return B, R
```

The goal of AdaGDVB is to characterize the boundary between verification problems that can be solved and those that cannot by a given verifier; we refer to this as the *verification performance boundary* (VPB). It has been shown in § 4, that verifiers are sensitive to as many as 9 independent factors (e.g., number of neurons, number of layers, and input dimensions, etc.), that define a problem and influence verifier performance. While in principle a VPB in this 9-dimensional space could be defined, in practice we focus on smaller subspaces. The user can select a subset of GDVB *factors*, and AdaGDVB searches the space of scaling *levels* of those factors. Conceptually, the VPB is a surface in the selected space of factors, but the AdaGDVB algorithm approximates the VPB with a benchmark consisting of verification problems on both sides of the boundary.

As shown in Fig. 6.1, AdaGDVB comprises two phases: **exploration** and **refinement** – each of which performs a search of the space of verification problems. The primary objective of the exploration phase is to swiftly identify the approximate VPB with a minimal number of steps. Subsequently, the refinement phase is employed to enhance the granularity of the VPB, thereby facilitating a more comprehensive interpretation.

The **exploration** phase begins with an *initial configuration* setting for selected GDVB factors, describing the scaling levels for those factors($\boldsymbol{X}$), and a *seed verification problem*(s). In addition to these GDVB parameters, AdaGDVB takes a target *verifier* and an *evolution configuration*(e), which describes how the initial configuration should be scaled as the search proceeds. GDVB computes a covering array [31] based on the factor level settings. Then it uses the improved version of R4V to create a verification benchmark with one verification problem per covering array row. GDVB then uses SwarmHost or DNNV to execute the target DNN verifier and analyze the verification results (unsat, sat, unknown, error, or out of resource) which are used as feedback that drives the next round.

The **decide** step determines whether the performance boundary has been detected based on two *pivot* points: P_u and P_o. P_u underestimates the VPB as the *largest* neural network for all selected factors where all verification problems can be solved; P_o overestimates the VPB as the *smallest* neural network for all selected factors where no verification problems can be solved. If both pivots are found or the search boundary is reached, **decide** will transition to the **refinement** phase. Otherwise,

the **decide** step continues exploration by scaling the current factor-level bounds up or down based on the verification feedback and continues with another round. Once the VPB is found, the **refinement** phase uses the pivots to establish min/max scaling levels for each factor and generates problems evenly spaced within those levels to approximate the VPB.

The main algorithm of AdaGDVB is illustrated in Alg. 2, encompassing two distinct phases: **exploration** (Lines 1 to 15) and **refinement** (Lines 16 to 23). In the exploration phase, the algorithm begins by setting up the necessary variables in Lines 1 to 4. Line 1 determines the strength of the coverage, which is defined as the number of **factors**. Line 2 initializes the initial levels of the factors. Line 3 initializes the two pivots, while Line 4 initializes the iteration counter.

The exploration loop commences by generating a GDVB benchmark B in Line 6, followed by executing the verifiers and analyzing the results using the Evaluate method in Line 7. Line 8 calculates the two pivot points (P_u and P_o) for the current searching space. If both pivots are found (Line 9) or the maximum number of exploration steps is reached (Line 5), AdaGDVB transitions from the exploration phase to the refinement phase. Otherwise, the Explore method computes the factor-level configurations for the next iteration based on the verification results, and the algorithm continues looping at Line 5.

The refinement phase calculates a factor-level configuration using the Refine method with the given granularity g, as depicted in Lines 18 to 20. It then proceeds with the generation of the GDVB benchmark, verifier execution, and results evaluation using the Evaluate method from Line 21 to 22. The refinement phase continues until the maximum number of steps is reached in Line 17. If not, it loops back from Lines 17 to 23. Ultimately, the AdaGDVB algorithm returns the verification performance boundary B and the corresponding results of the verifier.

6.1.3 Examples

Example 6.1.1 Fig. 6.2 illustrates a conceptual example workflow of AdaGDVB. The initial GDVB factor-level settings contain two factors with two/three levels respectively, $\boldsymbol{X}_1 = \{1, 2, 3\}$ and $\boldsymbol{X}_2 = \{1, 2\}$. The **exploration** loop starts with these settings, and the subsequent steps evolve those levels by doubling them. The initial benchmark generates 6 problems shown in circles in the first

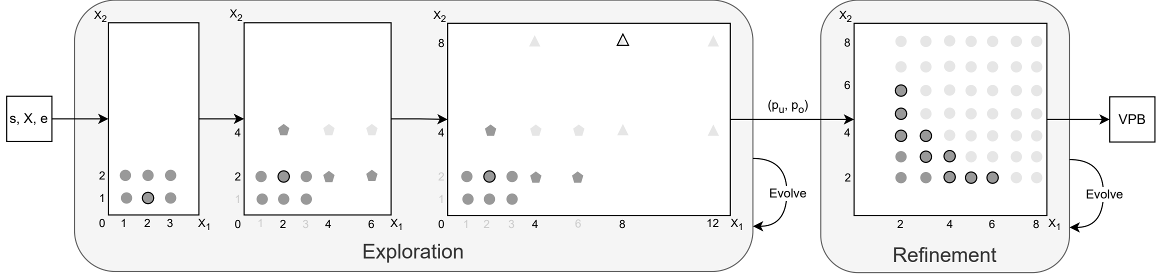


Fig. 6.2: Conceptual Workflow of AdaGDVB (Factors: $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ with exponential growth; Dark grey: solvable problems, light grey: unsolvable problems; Circles: step 1, pentagons: step 2, triangles: step 3; Bold items in exploration: the two pivot points, P_u and P_o, at current iteration. Bold items in refinement: the performance boundary on the solvable side.)

plane. The newly explored verification problems for iteration 2 and 3 are shown as pentagons and triangles, respectively. Problems filled with dark gray were solved by the verifier and those with light gray were not. After three iterations, the *verification performance boundary* is successfully identified and the **Decide** step moves on to the **refinement** loop. The refinement phase adds more resolution to the VPB to yield a benchmark that more clearly delineates the boundary.

Example 6.1.2 Fig. 6.3 illustrates the VPB exploration process for the α,β-CROWN verifier on the MNIST$_{3\times1024}$ artifact(refer to Tab. A.1). The chosen GDVB factors are **neuron** (X-axis) and **layer** (Y-axis). In the exploration phase, AdaGDVB explores 7 steps as shown in the series of rectangles moving upward and to the right. Each step comprises a GDVB benchmark (rectangle), with 9 networks (circles) and verification results for 5 verification properties shown as colored wedges of the circle – green (unsat) and blue (sat) denote solvable verification problems, while the rest are considered unsolvable. The result of this exploration phase identifies pivots that are the lower-left and upper-right corners of the refinement region in the Figure. The refinement phase fills in this region at a specified granularity to reveal the VPB as shown in Fig. 6.4a with a 2x granularity.

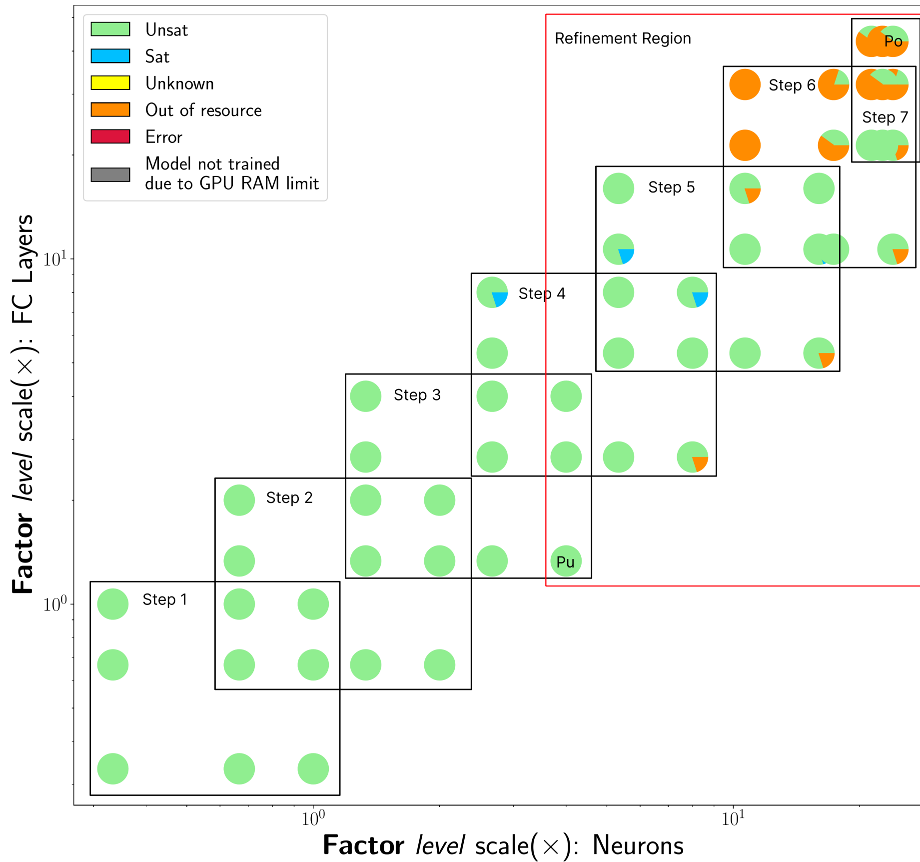


Fig. 6.3: AdaGDVB's Exploration Phase When Applied on α,β-CROWN Over the MNIST Network

6.1.4 Implementation

AdaGDVB is written in Python and is open source [1]. The tool can easily be extended to support new verifiers, e.g., a user can simply provide a Python wrapper including several lines of code to invoke the verifier and parse results. Moreover, it is designed to be run on a local computer or to distribute training and verification jobs to a cluster via the SLURM scheduler.

Relative to GDVB, AdaGDVB aims to minimize user effort – specifically the problem of choosing appropriate scaling levels for each factor. While automated, AdaGDVB is also highly configurable, allowing expert users to adjust multiple aspects of the search space and thereby control cost. For

[1]GitHub repository of AdaGDVB: `https://github.com/edwardxu0/AdaGDVB`

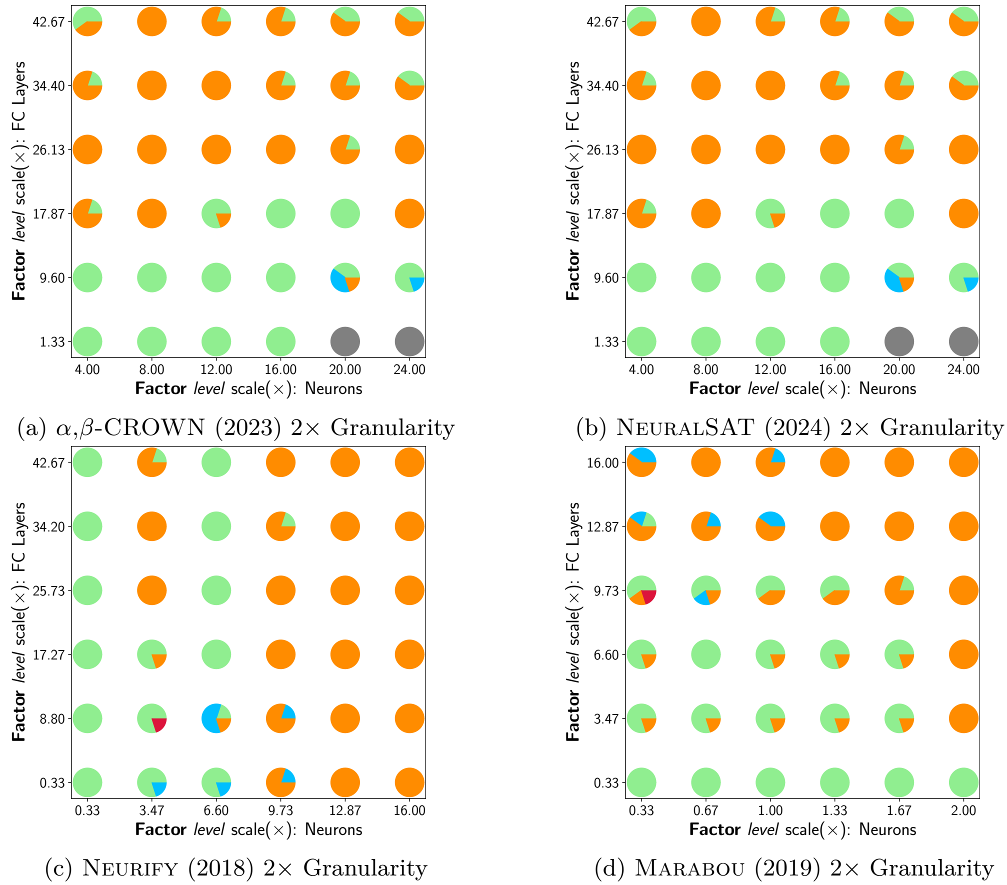
example, users can control various parameters including: the verification problem factors to be varied, scaling levels to be applied to those factors, scaling limits that define search space boundary, the maximum number of exploration iterations, the granularity of the benchmark generated in the refinement phase, and resource limits on verification jobs, such as timeout and memory limits. This allows AdaGDVB to be adopted easily and tuned for specific use cases.

6.2 Evaluation

We conducted studies to illustrate a pair of use cases for AdaGDVB. The studies use the same settings: a seed verification problem with the $\text{MNIST}_{3\times1024}$ (Tab. A.1) neural network of 3 fully connected layers with 1024 neurons each (3,072 neurons in total), and local robustness properties with an epsilon radius of 0.02 as was used in GDVB's evaluation. As discussed in § 4.1, the levels for GDVB factors define how to scale network parameters. We have three levels for scaling **neuron** $= \{\frac{1}{3}, \frac{2}{3}, 1\}$ and **layer** $= \{\frac{1}{3}, \frac{2}{3}, 1\}$, with upper bounds of **neuron** $= 42.67$ and **layer** $= 24$ to establish a bounded search space for AdaGDVB. Each network is verified on robustness properties with 5 different center points. For the verifiers, we select four representatives from various algorithmic families of different years, including Neurify (2018), Marabou (2019), α,β-CROWN (2023), and NeuralSAT (2024).

Using AdaGDVB to produce a benchmark requires executing numerous training and verification tasks. For the first use case discussed below, we executed more than 3,400 verification tasks each with a 600-second timeout – if run sequentially these would take more than 3 weeks of compute time. To compare VPB across different verifiers it is essential that AdaGDVB use the same hardware. The scale of this evaluation led us to choose to run verifiers on Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz CPUs with 16GB RAM using up to 6 cores and a 600-second timeout. While these CPUs are a few years old, we have access to several of them which allowed us to run verification jobs in parallel, bringing time for the study to under 1 day per verifier.

This choice does, however, present a concern when it comes to interpreting the computed VPB. Certain verifiers may be optimized to exploit different hardware resources, e.g., the computing power



(a) α,β-CROWN (2023) 2$\times$ Granularity

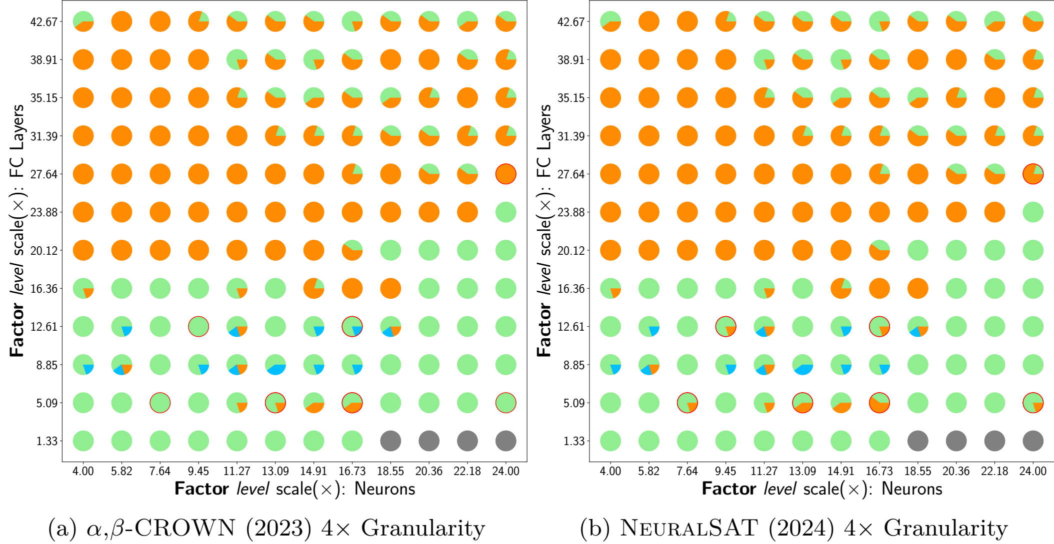
(b) NeuralSAT (2024) 2$\times$ Granularity

(c) Neurify (2018) 2$\times$ Granularity

(d) Marabou (2019) 2$\times$ Granularity

Fig. 6.4: AdaGDVB Generated Benchmarks (VPBs) for 4 Verifiers

of GPU/CPU and the memory size of RAM/VRAM. A specific choice of hardware may advantage some verifiers and limit the performance of others. To mitigate this threat, we compared the performance of two verifiers, α,β-CROWN and NeuralSAT, on the 6-core 16GB RAM CPU-only node – which we describe below – with the performance reported on a 64-core 128GB RAM node with a GPU [42]. Despite the hardware differences, the two evaluations found that the relative performance of these verifiers was very similar. This suggests that VPB computed with resource-constrained hardware still can provide an accurate relative characterization of verifier scalability.



(a) α,β-CROWN (2023) 4× Granularity (b) NeuralSAT (2024) 4× Granularity

Fig. 6.5: VPBs of α,β-CROWN and NeuralSAT with 4× Granularity

6.2.1 Use Case One: Adaptive benchmarks

Traditional DNN verification benchmarks evaluate verifiers over a fixed number of problems. The verifiers are ranked by the number of the problems solved and/or time, as in VNN-COMP [67, 7, 93]. AdaGDVB provides a new way to compare verifier performance by estimating the boundary between solvable and unsolvable problems. This directly measures the scalability of a verifier.

Fig. 6.4 depicts the VPBs of 4 verifiers which vary in the algorithms they use and when they were developed. The AdaGDVB exploration expands the scale of both layers and neurons based on a verifier's ability to solve problems, thereby characterizing their scalability. When a 2× refinement granularity is applied for α,β-CROWN and NeuralSAT, the resulting VPB for the generated MNIST benchmark is identical as shown in Fig. 6.4a and Fig. 6.4a. To further differentiate the two verifiers, we used a finer 4× refinement granularity to produce the benchmarks shown in Fig. 6.5a and Fig. 6.5b. The refined VPBs reveal that α,β-CROWN is slightly more capable. The differences are subtle, so we have highlighted them with red circles. The VPBs indicate that across the 720 verification problems in the benchmark, α,β-CROWN can falsify 1 more and prove 5 more than NeuralSAT, and NeuralSAT can prove 1 more than α,β-CROWN– a difference of less than

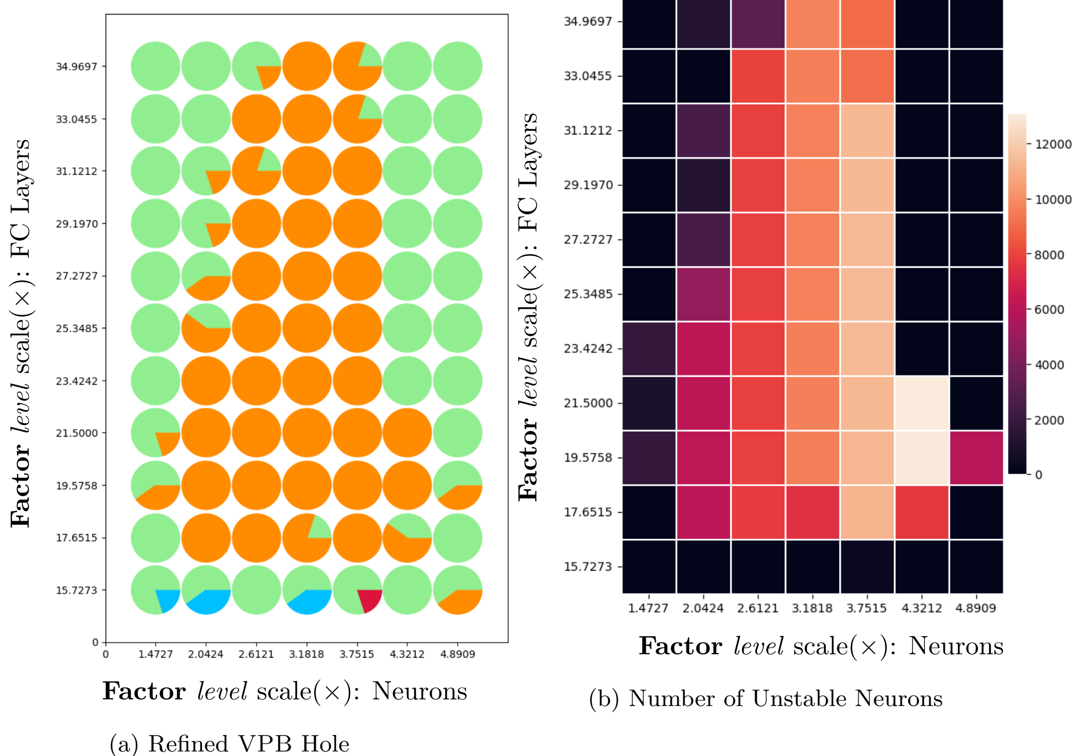
1%. This is consistent with the results of other published studies [41, 42], which show these verifiers performing comparably on benchmarks like this.

The VPBs of NEURIFY and MARABOU reveal a different story at a 2× refinement granularity. Neither scales as well across both factors, which is to be expected since they are older verifiers that do not incorporate the latest algorithmic techniques. It is interesting to note how the shape of their VPBs vary. NEURIFY scales well with layers while holding neurons at a low level, and MARABOU scales well with neurons while holding layers at a very low level. Understanding the relative strengths and weaknesses of verifiers as depicted by their VPB could help a user select a verifier for a given problem. For example, a network with 38 layers (3×12.87) and 29,890 neurons ($3,072 \times 9.73$), lies on the unsolvable side of the VPB for the older verifiers, but is comfortably on the solvable side of the VPB for the newer verifiers.

6.2.2 Use Case Two: Identifying Neural Network Verification Bottlenecks

The ability of ADAGDVB to compute per-verifier VPB allows one to identify potential performance anomalies. For the NEURIFY instance as shown in Fig. 6.4c, the row located at 25.73 shows that verification can be solved with either 1,024 or 20,275 neurons, but not with 10,650 neurons. ADAGDVB allows us to zoom in on that region to develop a finer benchmark that better delineates this region as shown in Fig. 6.6a. We instrumented a version of NEURIFY to collect a variety of different statistics during verification and ran it on this benchmark. We plotted the collected data in a grid corresponding to the benchmark to look for those that correspond to the boundary.

The heatmap in Fig. 6.6b, shows the number of ReLUs that had to be split during verification – these are referred to as *unstable neurons*. This finding concurs with the observation by Xiao et al. [135] that "the primary speed bottleneck of exact verification is the number of ReLUs the verifier has to branch on". This use case demonstrates how empirical studies using ADAGDVB can identify DNN verification scalability bottlenecks. This approach has allowed us to improve verifier performance both through training discussed in § 7 [139] and the addition of new optimizations introduced in § 8 [42].



(a) Refined VPB Hole

(b) Number of Unstable Neurons

Fig. 6.6: Confirming the **neuron stability** Bottleneck Using an Instrumented NEURIFY Verifier

6.3 Conclusion

This chapter extends the NNV benchmark generation tool GDVB to establish ADAGDVB, to iteratively generate adaptive benchmarks based on verifier performance to characterize a verifier's performance boundary. Findings from two use cases demonstrate that ADAGDVB can benefit both (i) competition holders – providing a new means to compare verifier performance by analyzing their performance boundaries; and (ii) DNN verification tool developers – providing a means to identify performance bottlenecks to drive further tool improvement. The identification of the **neuron stability** NNV performance bottleneck in § 6.2.2 has inspired our future research in improving verification performance by reducing the **neuron stability** in the training (refer to § 7) and verification (refer to § 8) processes.

Chapter 7

Increasing Neuron Stability to Scale Neural Network Verification

Neural network verification is challenging due to the high input dimension of models, the ever-growing complexity of network layers, the inherent non-linearity of learned function approximations, and the algorithmically complex methods required to formulate the verification problem [71]. Several approaches [46, 48, 106, 9] have been proposed to address the scalability issue, but as the results of recent neural network verifier competitions show scalability remains a challenge [67, 7, 93].

In § 6.2.2, it has been established that the number of stable neurons significantly influences the complexity of the verification problem. A higher number of stable neurons leads to a simpler problem, whereas a greater number of unstable neurons results in a more challenging problem. This correlation is logical, as unstable neurons contribute to the non-linear behavior of the neural network. Thus, there is a possibility to reduce neural network verification costs by reducing the number of unstable neurons, or in other words, increasing the number of stable neurons.

Several researchers have explored how neural networks can be defined to increase the number of stable neurons and thereby facilitate verification. For example, one can incorporate a loss term that uses an estimate of neuron stability to train a network that can be verified more efficiently [135].

Another training time approach identifies neurons that are likely to be stable and active and replaces them with linear functions [29], but this approach requires customization of the verifier to show performance improvement.

This chapter introduces OCTOPUS [1] a unified framework that trains neural networks with more stable neurons to reduce the cost of verification. It contains three algorithmic approaches to increase stability: RS Loss [135] incorporates a stability-oriented loss term, Bias Shaping is a novel training time method that only modifies bias parameters to increase stability, and Stable Pruning is a novel approach that adapts structural neural network pruning [118] to increase stability. These are paired with *stability estimation* algorithms that operate at training time to guide these methods towards increasing stability. We develop 4 estimators based on prior work: **NIP** [135], **SIP** [128, 127], **ALR** [142], and **ALRo** [143], and 2 novel estimators **SDD** and **SAD**.

Neuron instability can be a source of verification complexity for the two primary algorithmic approaches to neural network verification: abstraction-based methods and constraint-based methods Abstraction-based verifiers [129, 51, 8, 111, 110] overapproximate neuron behavior, but when the approximation is too coarse – due to unstable neurons – the approximations must be refined which can slow down verification. Constraint-based verifiers [68, 70, 45, 119] are challenged by the disjunctive nature of constraints that encode unstable neurons. Orthogonal to these approaches, branch and bound techniques [27, 129, 51] are also sensitive to neuron stability since they need to generate sub-problems for each of the active phases of unstable neurons. In our exploratory study, we evaluate the performance of verifiers that span several of these algorithmic approaches and that also constitute the state-of-the-art based on their performance in VNN-COMP 2022 [93]. This allows us to assess the extent to which increasing neuron stability can improve the state-of-the-art verifiers.

Whereas prior work studied individual methods for increasing neuron stability in combination with individual verifiers, in this chapter we conduct a broad exploratory study considering 18 different stabilizers paired with 3 state-of-the-art verifiers across neural networks for different datasets and comprising different architectures, on numerous challenging property specifications. Our primary finding is that stable training can significantly increase the number of verifications problem solved –

[1]The OCTOPUS approach introduced in this chapter is published in the TACAS2024 conference [139].

by as much as 5-fold – and significantly speed up verification – by as much as a factor of 14 – without compromising test accuracy or training time. Moreover, we find that if one is willing to tolerate a modest loss in test accuracy, then even greater improvement in verifier performance can be achieved.

The contributions of this chapter lie in a comprehensive evaluation of the potential for optimizing neural network verifier performance by increasing the number of stable neurons. More specifically,

- we adapt RS Loss with different stability estimators and evaluate its performance across multiple verifiers and benchmarks;
- we propose two novel approaches (Bias Shaping and Stable Pruning) to increase neuron stability and evaluate their performance across multiple verifiers and benchmarks;
- we integrate these state-of-the-art neuron stabilizers into an open-source framework that supports experimentation with stability optimization by the neural network verification research community; and
- we show empirically that the performance of state-of-the-art verifiers can be significantly enhanced using stable training methods. These contributions set the stage for further work on *training for verification* that aims to further characterize the best stable training strategy for a given verifier and verification problem.

7.1 Approach

7.1.1 Overview

The popularity of the rectified linear unit (ReLU) activation function, $z = \max(\hat{z}, 0)$, which allows for more efficient training and inference [56, 86], has led verification researchers to target networks using them. In this section, we illustrate how ReLU leads to exponential verification costs and how stabilization methods during neural network training can mitigate that cost, such as Stable Pruning.

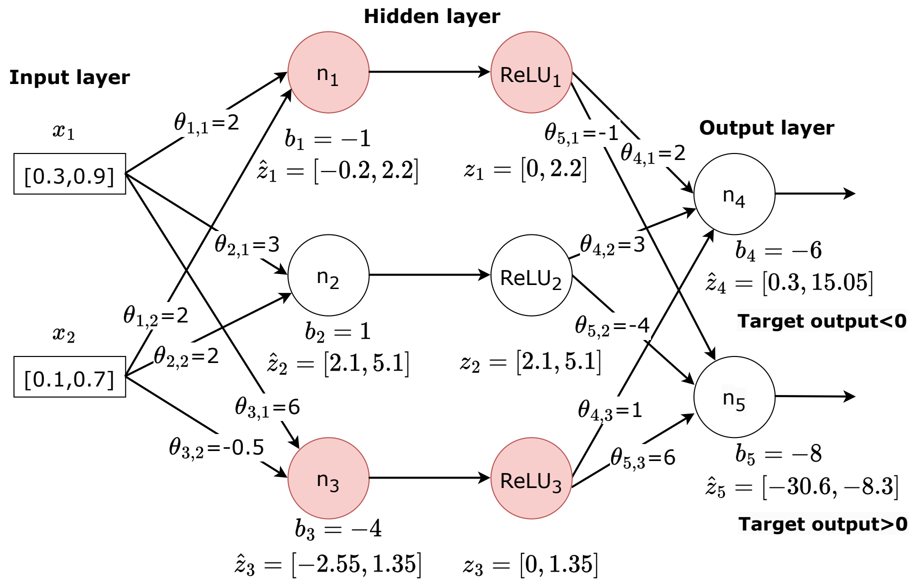


Fig. 7.1: A Small Original Neural Network with Various Stability of Neurons

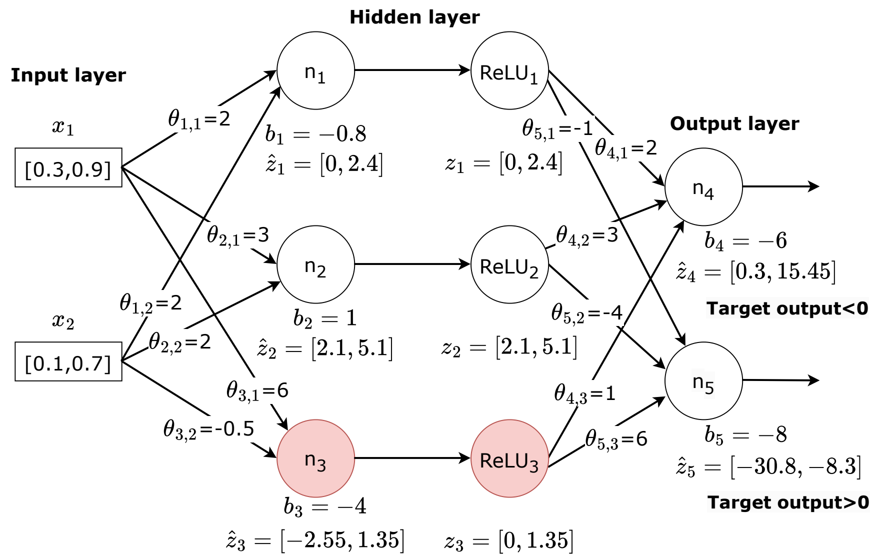


Fig. 7.2: Applying the Bias Shaping Method on the Original Neural Network

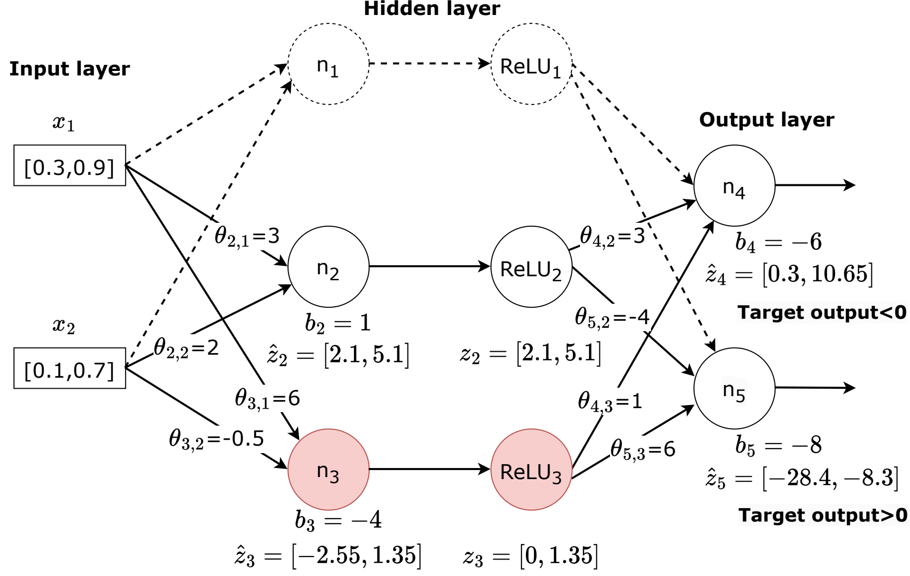


Fig. 7.3: Applying the Stable Pruning Method on the Original Neural Network

For a neural network with ReLU activation functions, $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$, comprised of k neurons, an inference, $\mathcal{N}(\boldsymbol{x})$, results in each neuron being either *active*, when $z = \max(\hat{z}, 0) = \hat{z}$, or *inactive*, when $z = \max(\hat{z}, 0) = 0$. The status of each neuron in a network during inference defines an activation pattern, $ap(\boldsymbol{x})$ – a Boolean vector of length k. Verifying a set of inputs, $\phi_{\boldsymbol{x}} \subseteq \mathbb{R}^n$, involves symbolically reasoning about the set of activation patterns, and the associated neuron outputs, for each $\boldsymbol{x} \in \phi_{\boldsymbol{x}}$. In the worst case, there are 2^k possible activation patterns which lead to the exponential complexity of ReLU verification [68].

For a given set of inputs, $\phi_{\boldsymbol{x}}$, a neuron, n_i, is *stable* and active if $\forall \boldsymbol{x} \in \phi_{\boldsymbol{x}} : ap(\boldsymbol{x})[i]$, and stable and inactive if $\forall \boldsymbol{x} \in \phi_{\boldsymbol{x}} : \neg ap(\boldsymbol{x})[i]$. A neuron's stability is dependent on the computation performed by its cone of influence [18] taking into account both $\phi_{\boldsymbol{x}}$ and the behavior of neurons on which n_i depends on. In Fig. 7.1, consider verification of a local robustness property centered at $\boldsymbol{x} = (0.6, 0.4)$ with a radius of $\epsilon = 0.3$ – so $\phi_{\boldsymbol{x}} = [0.3, 0.9] \times [0.1, 0.7]$. For such inputs, a single neuron, n_2, is stable – its pre-activation values are all positive, $\hat{z}_2 = [2.1, 5.1]$.

As defined in § 7.1, a set of techniques that aim to estimate which neurons are unstable during

training and then bias the training process to stabilize them. Fig. 7.2 and Fig. 7.3 show the application of one pair of those techniques to the original network and property. More specifically, the **NIP** estimator propagates interval approximations of neuron pre-activation values to estimate whether they are stable, and then the Bias Shaping or Stable Pruning technique stabilizes the neurons that are stable and inactive. During training this method estimates the pre-activation value for n_1 to be $\hat{z}_1 = [-0.2, 2.2]$ which is nearly stable. Both Bias Shaping and Stable Pruning rank the neurons based on the distance they need to be modified to be stable; for $\hat{z}_1$ that distance is 0.2.

For Bias Shaping, we designed a novel approach to stabilize the unstable neurons by directly manipulating the bias term of the unstable neurons. As shown in Fig. 7.2, Bias Shaping adds 0.2 to the bias term of n_1 so that its pre-activation value becomes in the stable active phase of the ReLU function, and hence n_1 becomes a stable neuron. As for Stable Pruning, we adapt the iterative pruning approach of DropNet [118] to use this ranking. The intuition is that when a neuron is nearly stable it can be removed and in subsequent training, the parameters of the remaining neurons will adapt to compensate and preserve accuracy [52]. As illustrated in Fig. 7.3, by setting the weights and bias of n_1 to 0, this neuron is softly "removed" from the network, and hence the neuron becomes stable. The reduction of unstable neurons will eventually reduce verification costs.

This section presents the two novel neuron stabilization methods: Bias Shaping and Stable Pruning, as well as six different stability estimators. Alg. 3 shows the general training iterations for a neural network with stabilizers (pairs of stabilization method, $\mathcal{A}$, and stability estimator, $\mathcal{B}$). The conventional neural network training process of a mini-batch is shown in Line 2. Stabilizers are applied at every sth mini-batch (line 3). Line 4 determines each neuron's stability estimation by calculating their boundaries, $\hat{Z}$, using different estimators described in §7.1.2. Lastly, Line 5 applies the main stabilization algorithms, e.g. Bias Shaping (Alg. 4) and Stable Pruning (Alg. 5).

Alg. 3: Training with Stabilizers

Input : neural network $\mathcal{N}$, data loader D, stabilization method $\mathcal{A}$, stability estimator $\mathcal{B}$, ratio i, and step s

Output : stabilized network $\mathcal{N}'$

```
1 for j, (X, Y) in D do
2     Train_Mini-Batch(N, X, Y);
3     if j ≡ 0 (mod s) then
4         Ẑ ← Estimate_Stability(B, N);
5         N' ← Stablize(A, N, Ẑ, i);
6 return N'
```

7.1.2 Neuron Stability Estimation

The neural network training process is performed on the data samples, while the verification process seeks to prove certain properties on an effectively unbounded set of inputs. Hence, there exists a gap between the two stages, since a neuron that is stable on the training dataset is not guaranteed to also be stable based on the set of values described by the precondition of the verification problem. Guiding the training process to produce neural networks with more stable neurons in the verification stage requires reducing this gap. This is achieved by estimating neuron stability over a broader set of values representative of those encountered during the verification process, and then stabilizing the unstable neurons.

We identify two general categories of neuron stability estimators that can be calculated during the training phase: **Sampled[S]** and **Reachability[R]** estimators. The sampled estimators consider a finite set of sampled data gathered directly or inferred from the training dataset. The reachability estimators operate on set propagations that generalize the training dataset. The six neuron stability

estimators are defined as follows:

$$\mathcal{B}(D) = \{x | x = \beta(x') \wedge x' \sim D\}$$

where $\beta \in \{\mathbf{SDD}, \mathbf{SAD}, \mathbf{NIP}, \mathbf{SIP}, \mathbf{ALR}, \mathbf{ALRo}\}$ and D is the network training dataset distribution.

The **SDD** (Sampled Dataset Distribution[**S**]) estimator uses the training mini-batch samples directly and takes advantage of the training process's forward propagations to determine whether neurons are stable. The **SAD** (Sampled Adjacent Distribution[**S**]) estimator samples from the robustness radii of the training mini-batch and runs extra forward propagations on the adjacent examples to determine the stability of neurons. The **NIP** (Naive Interval Propagation[**R**]) [135] estimator generates a set of intervals based on the mini-batch samples and the given robustness radii. However, instead of propagating exact samples, it propagates the intervals through the network. The **SIP** (Symbolic Interval Propagation[**R**]) [128, 127] extends **NIP** by using symbolic intervals instead of concrete intervals when propagating through the network. The symbolic intervals are concretized whenever neuron stability needs to be evaluated. The **ALR** and **ALRo** (Auto_LiRPA[**R**]) [142, 143] estimators further improve **SIP** by applying more precise but computationally expensive over-approximation constraints and parameterizing upper and lower bounds of hidden neurons to optimize objectives concerning the property of interest. **ALRo** applies the α optimization [143] when compared to the base approach. Note that although many of these approaches were developed for other uses, the integration of them to induce stable neurons during training is novel.

7.1.3 Bias Shaping

To increase the number of stable neurons in the neural network, we adapt training to ensure the same polarity of lower and upper bounds of neuron pre-activation values. In Equation 2.1, the pre-activations of the current ReLU function are controlled by the parameters of the neural network and the post-activations of the previous layer. The weighted-sum term depends on the weights, bias, and the post-activations of the previous layer. The pre-activation values can be easily manipulated by changing the bias term. We refer to this as Bias Shaping, as described in Alg. 4.

Alg. 4: BIAS SHAPING

Input : neural network $\mathcal{N}$, stability estimation boundaries $\hat{Z}$, ratio i

Output : stabilized network $\mathcal{N}'$

1 $\underline{\hat{Z}}, \bar{\hat{Z}} \leftarrow$ GET_BOUNDS($\hat{Z}$) ;

2 $N_u \leftarrow \{n_i$ in $\mathcal{N}$ where $\underline{\hat{z}}_i < 0 \wedge \bar{\hat{z}}_i > 0\}$;

3 $Z_u \leftarrow \{$MIN$(-\underline{\hat{z}_{n_i}}, \overline{\hat{z}_{n_i}})$ where $n_i \in N_u\}$;

4 $\gamma \leftarrow$ SORT$(Z_u)[|\hat{Z}| \times i]$;

5 **for** n_i *in* N_u **do**

6   **if** $(\overline{\hat{z}_i} < \gamma) \wedge (\overline{\hat{z}_i} < -\underline{\hat{z}_i})$ **then**

7         $n_i.b \leftarrow n_i.b - \overline{\hat{z}_i}$;

8   **else if** $-\underline{\hat{z}_i} < \gamma$ **then**

9         $n_i.b \leftarrow n_i.b - \underline{\hat{z}_i}$;

10 $\mathcal{N}' \leftarrow$ LOAD_PARAMETERS(N_u);

11 **return** $\mathcal{N}'$

Instead of using just the native pre-activation of the mini-batch samples, the stability estimators are applied to further close the gap between neuron stability during training and verification. Alg. 4 takes the set of stability estimations for all neurons, $\hat{Z} = [\hat{z}_1, \hat{z}_2, ..., \hat{z}_m]$, the neural network $\mathcal{N}$ with m neurons $(n_1, n_2, ..., n_m)$, and the stabilization ratio i as inputs. Given the stability estimation $\hat{z}$ for a neuron, $\underline{\hat{z}}$ and $\bar{\hat{z}}$ denote the lower and upper bounds respectively. Line 1 calculates the lower($\underline{\hat{Z}}$) and upper($\bar{\hat{Z}}$) bounds of the estimation $\hat{Z}$ for all neurons. Using those bounds, the algorithm first finds the unstable neurons of the input network (line 2). Next, those neurons are ranked based on the distance of their according pre-activation estimations(MIN of lower $\underline{\hat{z}}$ and upper $\bar{\hat{z}}$ bounds) to zero(lines 5 - 9), and the smallest subset of neurons will be selected for shaping if their distances are less than an adaptive threshold γ (lines 3, 4) – any unstable neurons with smaller pre-activation estimation distance below the threshold γ will be stabilized. Note that the number of selections is controlled by the stabilization ratio parameter i – a percentage of neurons would be shaped at a

time. Each neuron's bias term of the subset is modified by (a) shifting left by the value of the upper bound if the upper bound is closer to zero (line 7); or (b) shifting right by the absolute value of lower bound if the lower bound is closer to zero (line 9). As a result, the stabilized network is created by loading the new parameters at line 10.

7.1.4 Stable Pruning

Inspired by the DropNet [118] approach, we developed a new pruning method to reduce unstable neurons, named STABLE PRUNING as shown in Alg. 5. It uses iterative structured pruning to modify the global weight matrix by selectively masking neurons. Its novel criteria target unstable neurons for masking. STABLE PRUNING sets weight and bias to zero to softly "remove" the neuron from the network, allowing back-propagation to recover accuracy loss in subsequent training.

Alg. 5: STABLE PRUNING

Input : neural network $\mathcal{N}$, stability estimation boundaries $\hat{Z}$, ratio i

Output : stabilized network $\mathcal{N}'$

1. $m = \{1\}^{|n|}$;
2. $\underline{\hat{Z}}, \bar{\hat{Z}} \leftarrow$ GET_BOUNDS($\hat{Z}$) ;
3. $m[\bar{\hat{Z}} \leq 0] \leftarrow 0$;
4. $Z'_u =$ SORT($\bar{\hat{Z}} > 0$);
5. $\gamma = Z_u[|Z'_u| \times i]$;
6. $m[\hat{Z} < \gamma] \leftarrow 0$;
7. $\mathcal{N}' \leftarrow \mathcal{N} \bigodot m$;
8. **return** $\mathcal{N}'$

When the lower bound $\underline{\hat{z}}$ is greater than 0, although the neuron is stable-active, it cannot be pruned without changing the network's behavior, as the ReLU function is treated as an identity function. When $\bar{\hat{z}}$ is less than 0, the ReLU function is treated as a zero-function, and this neuron

can be removed safely (line 3). To prune unstable neurons with minimal effects on network behavior, Stable Pruning ranks the unstable neurons by the distance between $\bar{\hat{z}}$ and 0, from smallest to largest (line 4), and a subset of neurons (also controlled by the ratio parameter, i) will be selected for pruning if their distances are less than an adaptive threshold γ (line 5). Initially, all neurons are enabled in the mask, m, (line 1) and those that fall below the threshold are updated to be removed from the network (line 6). Finally, the stabilized network is generated by applying the pruning mask on the network (line 7).

7.1.5 Implementation

We implemented all of the above techniques, including: **SDD**, **SAD**, **NIP**, **SIP**, **ALR**, **ALRo**, RS Loss (§3.4), Bias Shaping (§7.1.3), and Stable Pruning (§7.1.4), into the OCTOPUS framework. OCTOPUS allows training neural networks with stabilizer methods and stability estimators, including their combinations. It can be easily applied to different datasets and network architectures and presents a rich hyper-parameter space that can be tuned by hand or algorithmically, e.g., by search methods. RS Loss [135] is reimplemented to support all the additional neuron stability estimators. The **SIP** estimator uses the Symbolic Interval Analysis Library developed in [127], and the **ALR** and **ALRo** estimators integrate the Auto_LiRPA Library [143]. OCTOPUS also allows combinations of various neuron stabilizers and estimators, i.e., training with multiple stabilizers sequentially or simultaneously. The framework is built for ease of extension to adopt new techniques and is available at both FigShare [138] and GitHub.[2]

7.2 Evaluation

We explore two research questions to understand how stabilizers can be beneficial for NNV:

- **RQ1.** How effective are the stabilizers in increasing the proportion of stable neurons?
- **RQ2.** How effective are stabilizers in enhancing NNV performance?

[2]OCTOPUS GitHub link: `https://github.com/edwardxu0/octopus`

Tab. 7.1: Experimental Parameter Space

Parameters	**Choices**
Architectures	**M2**: $\text{MNIST}_{2\times256}$ (FC(256)×2), **M6**: $\text{MNIST}_{6\times256}$ (FC(256)×6), **C3**: CIFAR_{2020} (Conv(32,5,2), Conv(128,4,2), FC(250))
Verifiers	α,β-CROWN, MN-BaB, Nnenum
Properties	[0,1,. . . ,9]
Epsilon Radii	**M2** & **M6**:[12e-3, 14e-3, 16e-3, 18e-3, 20e-3] **C3**:[18e-4, 20e-4,22e-4, 24e-4, 26e-4]
Stabilization Methods	Baseline, **B**ias Shaping, **R**S Loss, Stable **P**runing
Stability Estimators	**SDD**, **SAD**, **NIP**, **SIP**, **ALR**, **ALRo**
Seeds	[0,1,2,3,4]

7.2.1 Study Design

To answer these questions, we design a broad study considering different neural network architectures, specifications, and verifiers. Tab. 7.1 shows the full experimental parameter space we consider across the research questions.

The annual VNN-COMP NNV competition [67, 7, 93] provides a range of benchmarks with standard network and property formats to evaluate state-of-the-art verifiers. These benchmarks cover a variety of network architectures and activation functions. This *architectural variety* evaluates verifiers' applicability across a range of network graph operations, e.g. ResNets with skip connections, max-pooling layers, non-linear activation, and domain-specific networks. Benchmarks also vary in *scale* with some having large numbers of layers, neurons, and parameters under the assumption that this will yield challenging benchmarks.

We conducted an exploratory study of the VNN-COMP 2022 benchmarks and found that 1156 of 1288 (89%) could be solved within 30 seconds. Nearly all the solved problems were proven (UNSAT) with coarse over-approximation or falsified (SAT) with adversarial attacks. Such benchmarks do not exhibit the exponential complexity that is inherent in NNV [68]. To address this limitation, we designed a set of benchmarks that are better suited to assessing NNV algorithm performance.

Selecting Networks A retrospective analysis of VNN-COMP benchmarks determined that small weakly-regularized networks exhibit exponential complexity and medium-sized with large numbers

of neurons are hard to scale for precise methods, such as branch and bound [24]. Of course, large weekly-regularized networks with large numbers of neurons are even harder, but it was found that these incur significant memory requirements which makes experimentation challenging, e.g., due to hardware limitations. Based on this analysis, we focus on three small and medium-sized networks with traditional network architectures, i.e. **M2**, **M6**, and **C3** as shown in Tab. 7.1, selected from the VNN-COMP 2022 benchmarks, since these proved capable of forcing verifier algorithms to cope with exponential complexity.

Selecting Properties Rather than focusing on a variety of structurally distinct property specifications, we exploit the fact that general reachability properties can be reduced to local robustness properties [105]. This allows us to vary the verification problem difficulty by controlling the robustness property's epsilon-radius. Conceptually, we know that verification problems with sufficiently small (large) radii will be verified (falsified) – a radius of 0 is trivially verified and a radius comprising the full input domain requires that a network produce a constant output. Verifier developers have incorporated techniques, like applying adversarial attacks and using coarse overapproximations, to quickly handle such cases [8, 129]. To sidestep these verification fast paths and exercise the core verification algorithms in our study, we select epsilon values for properties as follows.

As discussed in § 4.2, for each network, we conducted a preliminary study with varying radii to assess the difficulty of the verification problems. We observe the trend that small epsilon leads to uniformly verified problems and large epsilon to uniformly falsified problems.

Our strategy for selecting harder verification properties is to choose a sample of radii around the point where the number of verified and falsified problems crossover, e.g., 0.018 in Fig. 4.2 for MN-BaB. We choose the crossover point of the best verifier who solved the most problems to design the radii shown in Tab. 7.1. This leads to a balance in verification ground truth between SAT and UNSAT answers, and these more challenging problems force the underlying algorithms to more precisely model network behavior, e.g., splitting unstable neurons into branch and bound cases.

Selecting Verifiers Unlike other research that focuses on improving the performance of a single verifier with a single customized pruning technique [29, 135, 151], our goal is to explore how the space

of stabilization strategies impact a range of verification approaches. Towards this goal, we select the three best-performing verifiers from VNN-COMP 2022 [93] that were available: α,β-CROWN, MN-BaB, and Nnenum [3]. Improving the performance of these verifiers will extend the state-of-the-art in scalable NNV.

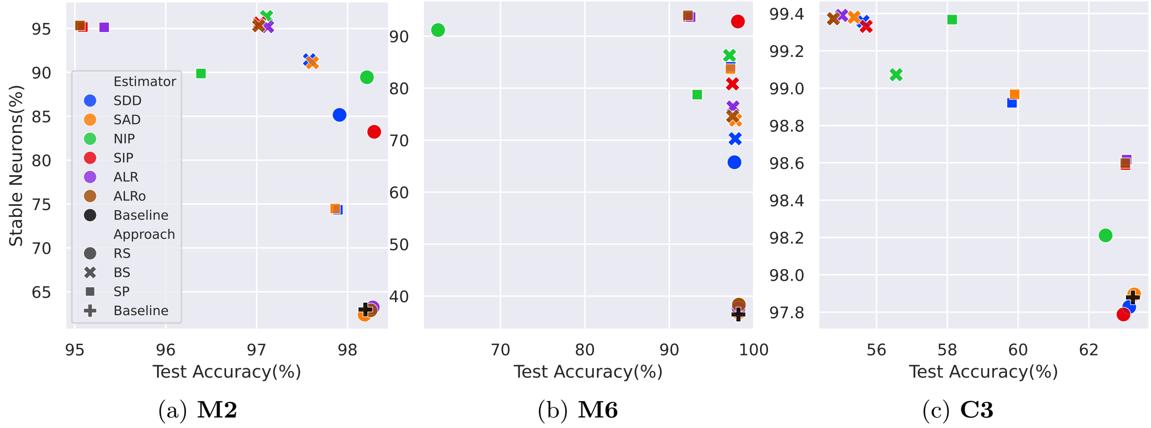
Network Training Stabilizers are incorporated into training, so we use a baseline (Baseline) trained without any stabilizers using the Adam optimizer with a 10^{-3} learning rate and 0.99 decay for 20 epochs. All stabilizers are customizable with hyperparameters, as described in § 3.4 and § 7.1. We use the well-tuned parameter for RS Loss introduced in [135], and perform a binary search of the parameter space for Bias Shaping and Stable Pruning. To elaborate, RS Loss uses always-active scheduling with 10^{-4} weight parameter; Bias Shaping uses interval scheduling activated every 5/25/50 mini-batches and adjusts 2%/5%/5% of unstable neurons each time it is applied for the **M2**/**M6**/**C3** architectures, respectively; Stable Pruning undertakes an interval scheduling that is activated for every 5/50/50 mini-batches with a pruning ratio of 2%/5%/5%, respectively. The resulting neural networks with the largest test accuracy of the last five epochs are selected for verification. To account for stochasticity in training, we train each network 5 times and report the mean data for each.

These choices for the space of experiments yield a total of 1,215 training tasks and 36,450 verification tasks. Each training task is run with one GTX 1080 Ti GPU with 11G VRAM. Each verification task is run with 8GB of memory on one core of the Intel Xeon Gold 6130 CPU @ 2.10GHz with a timeout of 300 seconds. The total CPU time spent on training and verification across our experiments is 1858 and 1052 hours, respectively.

7.2.2 RQ1: Stabilizing Neurons

Stabilizers aim to linearize a portion of the behavior encoded by ReLU activation across the set of computations activated for a property precondition. In this experiment, we directly measure this by recording the percentage of neurons that are stable during verification. We also record model test accuracy to understand the trade-offs of the stabilization methods and stability estimators. Existing

[3]Verinet performed well in the competition, but it required a custom solver that is not freely available.

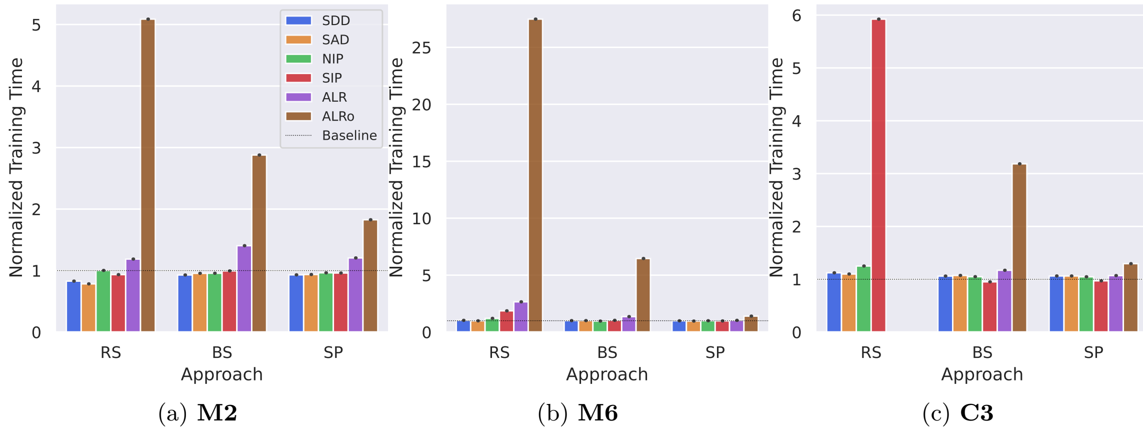


(a) **M2** (b) **M6** (c) **C3**

Fig. 7.4: Stable Neurons(%) vs. Test Accuracy(%) per Model

verifiers do not record the number of stable neurons, so we modified an open-source neural network verifier, NeuralSAT [41], to record the number of stable neurons computed during verification.

Fig. 7.4 presents the average test accuracy and the average number of stable neurons computed across the five training seeds for the three architectures across the stabilizers in the benchmark, as described in §7.2.1. The black ✚ sign indicates the Baseline (Baseline), the ● sign represents RS Loss (RS), ✖ means the Bias Shaping (BS) method, and ■ is Stable Pruning (SP). Six different colors denote the different stability estimators. Across all three architectures, most techniques can increase the number of stable neurons, but some of the techniques lead to a loss in test accuracy. For the **M2** architecture, RS Loss with NIP can significantly increase the number of stable neurons by more than 26 percentage points without compromising accuracy. For **M6**, RS Loss yields an even greater increase of 55 percentage points but in combination with the SIP estimator. For the Convolutional **C3** network, a very high percentage of neurons are already stable, so only marginal improvement can be achieved. Here the Stable Pruning method performs best while preserving accuracy, but it only yields a percentage point increase. For all the architectures, if one is willing to sacrifice a degree of accuracy, then further increases in stability can be achieved. For example, for **M2** Bias Shaping can achieve an additional 7 percentage point increase in stable neurons at the cost of just over 1 percentage point in test accuracy.



(a) **M2** (b) **M6** (c) **C3**

Fig. 7.5: Normalized Training Time

Incorporating stabilization in training can increase training time. Fig. 7.5 shows the average training time of the three models normalized to the Baseline. For all three models, the clear outlier in terms of cost is the **ALRo** estimator when used with RS Loss, which incurs up to more than a 25-fold increase in training time. This overhead even prevents RS Loss from practically training with **ALR** and **ALRo** on the **C3** architecture. RS Loss also experiences large cost with paired with **SIP** on the **C3** architecture. The overhead of most of the other estimators is negligible, including those that yielded significant increases in stable neurons.

> **RQ1 Findings** Across the study there are combinations of stabilization methods and stability estimators that are capable of increasing the number of stable neurons, in many cases substantially, without compromising test accuracy or training time.

7.2.3 RQ2: Enhancing Verification

RQ1 demonstrates the ability of stabilizers to increase the number of stable neurons across a space of verification problems. This question explores whether those increases lead to improvements in verifier performance. To assess the generalization of the stabilizers to variations of neural network properties, we verify 50 local robustness properties per trained network, pairing 10 center points with each of the 5 epsilon radii. We run the three selected state-of-the-art verifiers on each problem.

We measure two metrics to assess verification performance: (1) the number of problems, i.e., the network, center-point, and radii combination, each verifier can solve, i.e., produce either an SAT or UNSAT result, and (2) the time taken to solve those problems. Note that our metrics exclude runs that produce errors, exceed a 300-second timeout, or an 8GB memory bound. These metrics are standard for assessing verifier performance and while sometimes they are aggregated, as in PAR2 [140], we keep them separate here to explore them independently.

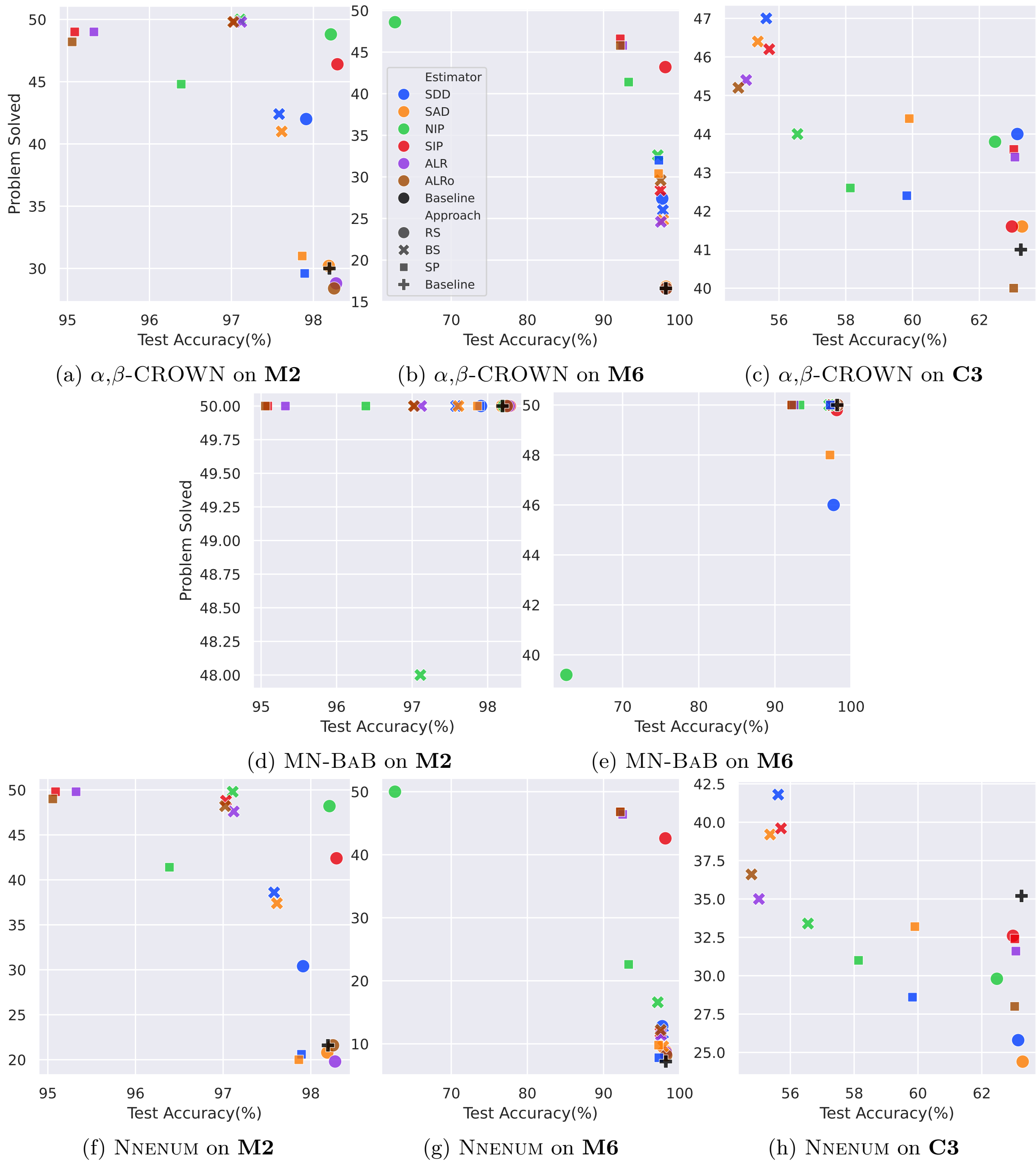


Fig. 7.6: Solved Verification Problems vs. Test Accuracy(%)

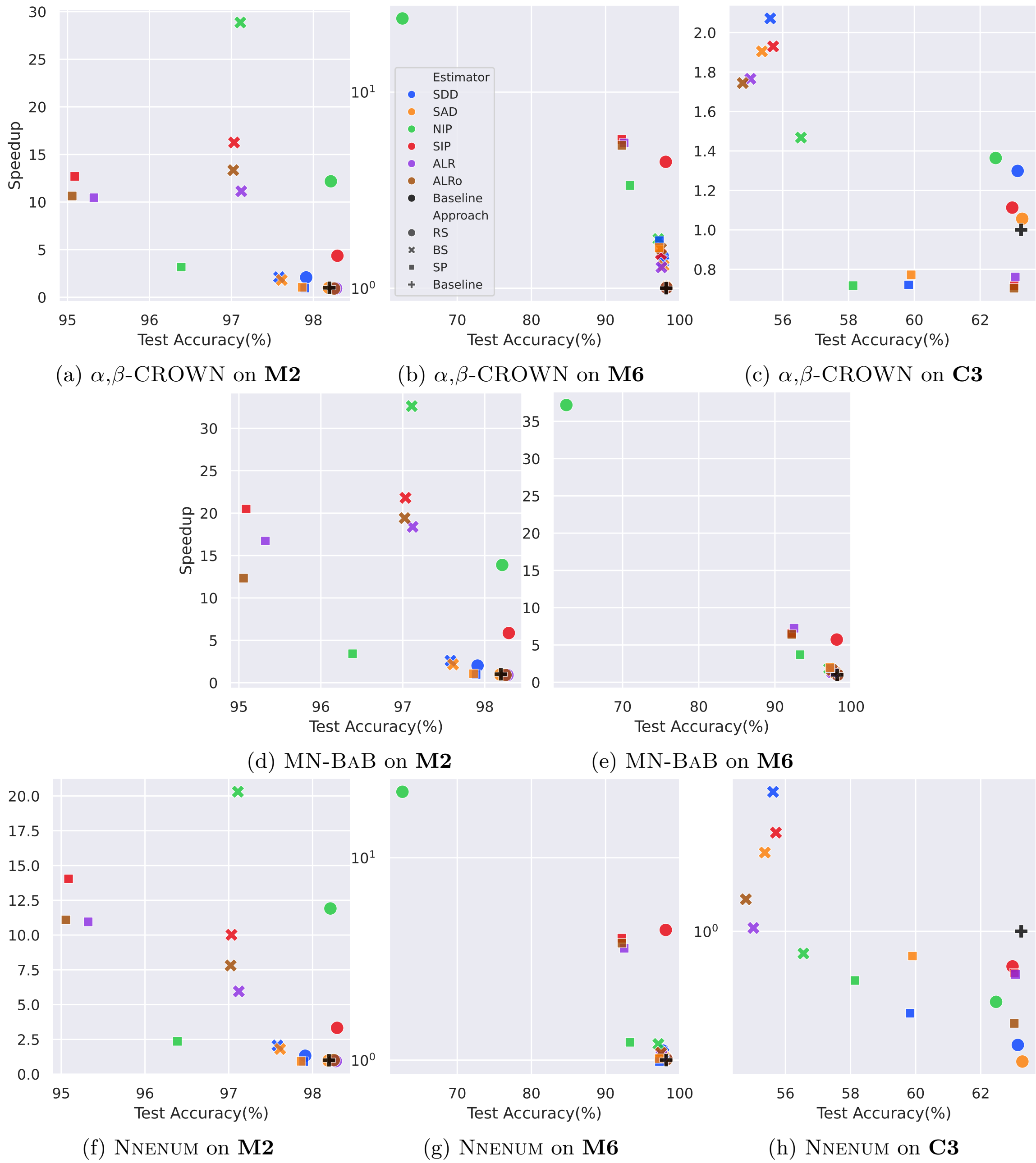
Fig. 7.6 shows eight plots of the number of verification problems solved versus test accuracy across

the three architectures using three of the verifiers. The trends in these plots are largely consistent with the findings of RQ1 - when more neurons are stable the verifiers are more effective in solving problems. RS Loss, with different estimators, increases the number of problems solved by factors up to 5.92 for these verifier network combinations without sacrificing test accuracy. As in RQ1, further performance improvements are possible by sacrificing accuracy. For example, on **M2** α,β-CROWN can improve by a factor of 1.67 using Bias Shaping with a reduction of 1 percentage point in accuracy.

The trends shown here for α,β-CROWN and Nnenum are similar, but MN-BaB exhibited different performance. For **M2** and **M6**, the baseline technique was able to solve all 50 problems, so there is no opportunity for improvement, while almost all the stabilizers can maintain the 50 problems solved. Note that the implementation of MN-BaB doesn't support the **C3** architecture. While the number of problems does not change for MN-BaB with stabilization as we discuss below its runtime is reduced.

Fig. 7.7 plots the verification time speedup over Baseline against test accuracy for eight verifier network pairs. We observe a similar trend to what was observed for the number of neurons stabilized and the number of verification problems solved − stabilization can speed up verification without compromising test accuracy. For MN-BaB on **M2** while the number of problems solved did not change, using RS Loss with **NIP** yielded a factor of 14 speedup. For **M6** we see a speedup of up to a factor of 5 with Nnenum and for **C3** more modest speedups for α,β-CROWN. The MN-BaB plot also shows, as observed above, that further speedups − greater than 30 fold − can be achieved if one compromises accuracy by about 1 percentage point.

RQ2 Findings Stabilizing neurons during training can substantially increase the number of problems solved and reduce the time required to solve them by state-of-the-art neural network verifiers without compromising test accuracy. Further improvement in verifier performance can be achieved with a small sacrifice in test accuracy.



(a) α,β-CROWN on **M2** (b) α,β-CROWN on **M6** (c) α,β-CROWN on **C3**

(d) MN-BaB on **M2** (e) MN-BaB on **M6**

(f) Nnenum on **M2** (g) Nnenum on **M6** (h) Nnenum on **C3**

Fig. 7.7: Verification Time Speedup vs. Test Accuracy(%)

7.2.4 Discussion

The data show a significant degree of variability in the effectiveness of particular stable training approaches with verifiers and verification problems. Broadly speaking RS Loss seems to perform well when one is unwilling to sacrifice test accuracy, but the best estimator varies depending on the verifier and problem – with **SDD**, **NIP**, and **SIP** yielding the best performance. The main drawbacks of RS Loss is much more memory intensive and doesn't scale to larger neural networks and more precise stability estimators, as shown in Fig. 7.5. For the large Convolutional network, Stable Pruning also performs well without compromising test accuracy. We believe this to be consistent with the broader results from the field of structured pruning [52, 118], where it has been found that large networks tend to be over-parameterized and can thus accommodate significant pruning without compromising accuracy. While the study shows that many of the methods can yield benefits, we believe that it also demonstrates that certain stabilization approaches, e.g., RS Loss with **ALRo**, are too costly for use in practice. Further study should focus on how to select the best stable training approach, and its hyper-parameters, to yield the best improvement for a given verifier and class of verification problems. We believe it will be fruitful to develop such *training for verification* approaches in concert with algorithmic and engineering improvements to verification algorithms.

7.2.5 Threats to Validity

The chief threats to internal validity relate to whether the collection of test accuracy, stable neurons, verification problems solved, and verification time were accurate. We tested the accuracy of all stabilizer-trained networks, cross-checked problem solutions across verifiers, and thoroughly tested our instrumentation of NeuralSAT for recording neuron stability. Regarding external validity, while our study was scoped to manage experimental costs, it spanned: 3 verifiers, 3 network architectures, 50 property specifications, and 5 seeds. We used fixed sets of training and stabilizer parameters per neural network architecture, which potentially underestimated the benefit that might be observed by customizing parameters. While broadening the study further would be a valuable direction for future

work, the scope of the study is sufficient to support the finding that stabilizers can enhance NNV across a breadth of contexts.

7.3 Conclusion

Verifying neural networks is a challenging task due to their high computational complexity. In this Chapter, we propose two novel approaches Bias Shaping and Stable Pruning, to enhance the scalability of neural network verifiers by inducing more stable neurons during the training process. In addition, we designed six neuron stability estimators to drive stability-oriented training. Across a significant study, we found that focusing on stability yields a viable method to achieve training for verification that can significantly improve the ability to solve problems and speed up state-of-the-art verifiers.

Chapter 8

Harnessing Neuron Stability to Improve Verification

During the past six years, researchers have developed a range of techniques for verifying neural network properties formulated as pre/post-condition specifications that can be rendered in a canonical form [105]. Many dozens of neural network verifiers have been reported in the literature, and the yearly VNN-COMP competition has documented advances in the capabilities of such techniques [67, 7, 93, 23, 24].

Despite those advances, as with traditional software verification, NNV suffers from exponential worst-case complexity [68]. The exponential complexity comes from the non-linearity of the neural networks, i.e., the *activation patterns* of the ReLU activation functions regarding the input. While this complexity seems daunting, history has shown that despite the worst-case exponential growth of verification problems, like propositional satisfiability (SAT) [34], it is possible to solve very large problem instances with sophisticated algorithmic techniques [19].

Modern SAT solvers aim to determine if there exists an assignment of truth values to propositional variables that satisfies a given set of logical constraints. They are based on the classic Davis-Putnam-Logemann-Loveland (DPLL) algorithm [36] which searches the space of assignments by alternating

between *deciding* how to extend a partial assignment – by choosing a variable and a truth value for it – and identify additional assignments that are *implied* by that decision. State-of-the-art solvers also incorporate a plethora of optimizations like Conflict-Driven Clause Learning (CDCL) to short-circuit later portions of the search [149], heuristics to restart search with learned clauses [17], and parallel exploration of variable assignments [79]. Modern satisfiability modulo theory (SMT) solvers combine DPLL with theory-specific symbolic deduction methods that adapt and integrate with CDCL to form DPLL(T), where T stands for theory [95].

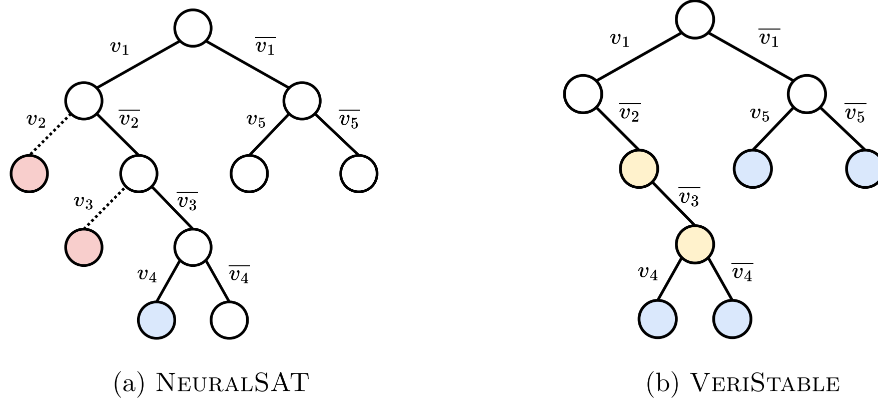
Most prior work on NNV either used SMT to discharge sub-problems formed by search of the space of activation patterns [68, 69, 64], applied forms of abstract interpretation to approximate the disjunctive neuron behavior [110, 111, 112, 127, 6, 109, 129, 143, 53, 119, 62, 22, 51], or combined these approaches [70, 45].

The success of these methods inspired recent work that adapts DPLL(T) to NNV by incorporating an abstraction-based theory solver [41] to realize the NeuralSAT verifier. In NeuralSAT, propositional variables encode whether a neuron is active or inactive, and additional constraints encode the weighted sums for each neuron input. As illustrated on the left of Fig. 8.1, NeuralSAT searches the space of activation patterns for a neural network; here v_i and $\overline{v_i}$ denote that the ith neuron is active or inactive, respectively, and a path in the tree is a partial activation pattern. As we discuss in § 3.3.1, NeuralSAT's contribution lies in combining DPLL(T) with a custom theory solver, that uses abstraction, to determine whether a partial activation pattern implies the specified property or implies conflict clauses that can prune subsequent search through CDCL.

In this chapter, we introduce the VeriStable [1] approach to further extend DPLL(T)-based NNV in two significant ways: to reduce the number of unstable neurons uncovered in § 6.2.2, and to adapt parallelism and restart heuristics from traditional SAT solving.

First, we propose a method for computing, from a partial activation pattern, a set of neurons that must be either active or inactive – such a neuron is said to be *stable*. Stable neurons eliminate the need for deciding their activation status later in the search and thereby lead to combinatorial reduction in the search. Unlike prior work [135, 146, 83, 29] as discussed in § 3.3, which seeks to modify the

[1] The VeriStable approach introduced in this chapter is published in the FSE2024 conference [42]. This chapter is joint work with fellow George Mason University(GMU) PhD student Hai Doung.



(a) NeuralSAT (b) VeriStable

Fig. 8.1: The Tree of Activation Patterns Computed by NeuralSAT (left) and VeriStable (right) at Corresponding Points during a Verification Run

network to create neurons that are stable for inputs described by the specification precondition, our approach (1) does not modify the network being verified and (2) detects neurons that are stable relative to subsets of the precondition. Our method can be thought of as *state-sensitive neuron stabilization*, where the state is a partial activation pattern encoding a subset of the precondition. Fig. 8.1 depicts how after v_1 is decided our method, VeriStable, stabilizes two neurons to be stable and inactive – shown in yellow – which eliminates the need to search their active branches – shown in red – as required by NeuralSAT. In this depiction, v_1 constitutes the state relative to which v_2 and v_3 are determined to be stable.

Second, we adapt parallelization techniques and restart heuristics from propositional SAT solvers to target the problem of NNV. Fig. 8.1 depicts how NeuralSAT's search frontier is a single state – shown in blue ($\{v_4, \overline{v_4}, v_5, \overline{v_5}\}$) – and how VeriStable can expand a broader frontier and do so in parallel. As depicted, stabilization and parallelization are synergistic in that the former reduces the tree width, which allows the latter to process a larger percentage of the tree.

While we developed these methods in the context of DPLL(T), these conceptual contributions are broadly applicable to any NNV approach that performs a search of the space of activation patterns and splits the search based on the activation status of neurons, such as [129, 6]. We implement the methods in the VeriStable framework and demonstrate empirically that each of the methods it incorporates leads it to outperform NeuralSAT, that the combination of all methods leads

to a 12-fold increase in the ability to solve verification problems, and that it establishes a new state-of-the-art in NNV compared with the top performers in the most recent neural network verifier competition [93].

The key contributions of this chapter lie in:

- a novel approach that computes state-sensitive neuron stability to eliminate the need for neuron splitting in NNV;
- adaptation of advanced SAT optimizations into a DPLL(T)-based verification algorithm;
- evaluation results using a new challenging NNV benchmark, as well as existing benchmarks, that demonstrate a 12-fold performance improvement and that VeriStable establishes the state-of-the-art in neural network verifier performance; and
- release of an open source implementation of VeriStable[2] accepting verification problems in standard formats to promote the application of NNV and comparative evaluation.

8.1 The VeriStable Approach

Alg. 6 shows the VeriStable algorithm, which takes as input the formula α representing the ReLU-based neural network N and the formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property to be proved. Internally, VeriStable checks the satisfiability of the formula

$$\alpha \wedge \phi_{in} \wedge \overline{\phi_{out}}. \tag{8.1}$$

VeriStable returns `unsat` if the formula unsatisfiable, indicating that ϕ is a valid property of N, and `sat` if it is satisfiable, indicating the ϕ is not a valid property of N.

[2] `https://github.com/dynaroars/neuralsat`

Alg. 6: The VeriStable Algorithm.

```
Input  : neural network α, property φ_in ⇒ φ_out, parallel factors n and k
Output : unsat if the property is valid and sat otherwise
1  clauses ← BooleanAbstraction(α)
2  while true do
3      assignments ← [(∅, ∅)] // initialize empty assignment and igraph
4      while true do // main DPLL loop
           // select n assignments (activation patterns) and corresponding igraphs
5          [(σ_1, igraph_1), ..., (σ_n, igraph_n)] ← Select(assignments, n)
           // process n assignments in parallel
6          parfor (σ_i, igraph_i) in [(σ_1, igraph_1), ..., (σ_n, igraph_n)] do
7              is_conflict ← true
8              if BCP(clauses, σ_i, igraph_i) then
9                  if StabilizeCondition() then // stabilize with condition
10                     Stabilize(α, φ_in, φ_out, σ_i, k) // stabilize k neurons
11                 if Deduce(σ_i, α, φ_in, φ_out) then
12                     (is_sat, v_i) ← Decide(α, φ_in, φ_out, σ_i) // decision heuristic
13                     if is_sat then
14                         return sat // consistent and complete assignment
15                     assignments ← assignments ∪ {(σ_i ∧ v_i, igraph_i) ; (σ_i ∧ ¬v_i, igraph_i)}
16                     is_conflict ← false // no conflict
17             if is_conflict then
18                 clauses ← clauses ∪ AnalyzeConflict(igraph_i) // learn conflict clauses
19         if length(assignments) ≡ 0 then // check unsat
20             return unsat // no more assignment to be processed
21         if Restart() then // check restart heuristic
22             break // restart occurs
```

8.1.1 DPLL(T)-based Neural Network Verification

VeriStable uses a DPLL(T)-based algorithm to check unsatisfiability. The algorithm consists of a Boolean abstraction, standard DPLL components, and a theory solver (T-solver) that is specific to the verification of ReLU neural networks.

8.1.1.1 Boolean Representation

BooleanAbstraction(Alg. 6, line 1) encodes the NNV problem into a Boolean constraint to be solved. This step creates Boolean variables to represent the *activation status* of hidden neurons in the neural network. VeriStable also forms a set of initial clauses, ensuring that each status variable is either T (active) or F (inactive).

8.1.1.2 The DPLL search

VeriStable iteratively searches for an assignment satisfying the clauses. Throughout it maintains several state variables including: clauses, a set of *clauses* consisting of the initial activation clauses and learned conflict clauses; σ, a *truth assignment* mapping status variables to truth values which encodes a partial activation pattern; and *igraph*, an *implication graph* used for analyzing conflicts.

Decide (Alg. 6, line 12) chooses an unassigned variable and assigns it a random truth value. Assignments from Decide are essentially guesses that can be wrong, which degrades performance. The purpose of BCP, Deduce, and Stabilize – which are discussed below – is to eliminate unassigned variables so that Decide has fewer choices.

BooleanConstraintPropagation or BCP (Alg. 6, line 8) detects *unit clauses* from constraints representing the current assignment and clauses and infers values for variables in these clauses. A unit clause is a clause that has a single unassigned literal. For example, after the decision $a \mapsto F$, BCP determines that the clause $a \vee b$ becomes unit, and infers that $b \mapsto T$. Internally, VeriStable uses an *implication graph* [13] to represent the current assignment and the reason for each BCP implication.

AnalyzeConflict (Alg. 6, line 18) processes an implication graph with a conflict to learn a new *clause* that explains the conflict. The algorithm traverses the implication graph backward, starting from the conflicting node, while constructing a new clause through a series of resolution steps. AnalyzeConflict aims to obtain an *asserting* clause, which is a clause that will result a BCP implication. These are added to clauses so that they can block further searches from encountering an instance of the conflict.

These are standard components in DPLL-based algorithms including modern SAT/SMT solvers and NeuralSAT. As shown in Fig. 3.1, DPLL also has backtracking, which allows the algorithm to go back to an incorrect assignment decision and choose the correct one instead. However, as will be described in subsubsection 8.1.2.2, the VeriStable parallel DPLL(T) does not require backtracking because it has optimistically considered both the correct and incorrect assignments simultaneously.

8.1.1.3 Theory Solver

VeriStable's Theory or T-solver (Alg. 6, lines 9-16) consists of two parts: stabilization and deduction.

Deduce (Alg. 6, line 11) checks the feasibility of the neural network constraints represented by the current propositional variable assignment. This component is shared with NeuralSAT and it leverages specific information from the neural network problem, including input and output properties, for efficient feasibility checking. Specifically, it obtains neuron bounds using the polytope abstraction[143] and performs infeasibility checking to detect conflicts.

The second part of the theory solver, which is specific to VeriStable, implements stabilization and is described next.

8.1.2 Improvements in VeriStable

We now describe neuron stability, parallel search, and restart. In §8.3.1 and §7.2.3 we present ablation studies demonstrating the performance of these ideas individually and in combination.

8.1.2.1 Neuron Stability

The key idea in using neuron stability is that if we can determine that a neuron is stable, we can assign the exact truth value for the corresponding Boolean variable instead of having to guess. This has a similar effect as BCP – reducing mistaken assignments by Decide – but it operates at the theory level not the propositional Boolean level.

Stabilization involves the solution of a mixed integer linear program (MILP) system [119]:

$$
\begin{aligned}
&\text{(a)} \quad z^{(i)} = W^{(i)} \hat{z}^{(i-i)} + b^{(i)};\\
&\text{(b)} \quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&\text{(c)} \quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0;\\
&\text{(d)} \quad a_j^{(i)} \in \{0,1\};\\
&\text{(e)} \quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned}
\tag{8.2}
$$

Alg. 7: The STABILIZE Function

Input : neural network α, property $\phi_{in} \Rightarrow \phi_{out}$, current assignment σ, number of neurons for stabilization k

Output : Tighten bounds for variables **not** in σ (unassigned variables)

```
1  model ← CreateMILP(α, φ_in, φ_out, σ) // create model (Eq. 8.2) with current
      assignment
2  [v_1, ..., v_m] ← GetUnassignedVariable(σ) // get all m current unassigned
      variables
3  [v'_1, ..., v'_m] ← Sort([v_1, ..., v_m]) // prioritize tightening order
4  [v'_1, ..., v'_k] ← Select([v'_1, ..., v'_m], k) // select top-k unassigned variables, k ≤ m
   // stabilize k neurons in parallel
5  parfor v_i in [v'_1, ..., v'_k] do
6     if (v_i.lower + v_i.upper) ≥ 0 then // lower is closer to 0 than upper, optimize
       lower first
7        Maximize(model, v_i.lower) // tighten lower bound of v_i
8        if v_i.lower < 0 then // still unstable
9           Minimize(model, v_i.upper) // tighten upper bound of v_i
10    else // upper is closer to 0 than lower, optimize upper first
11       Minimize(model, v_i.upper) // tighten upper bound of v_i
12       if v_i.upper > 0 then // still unstable
13          Maximize(model, v_i.lower) // tighten lower bound of v_i
```

where x is input, y is output, and $z^{(i)}$, $\hat{z}^{(i)}$, $W^{(i)}$, and $b^{(i)}$ are the pre-activation, post-activation, weight, and bias vectors for layer i. The equations encode the semantics of a neural network as follows: (a) defines the affine transformation computing the pre-activation value for a neuron in terms of outputs in the preceding layer; (b) defines the inputs and outputs in terms of the adjacent hidden layers; (c) asserts that post-activation values are non-negative and no less than pre-activation values; (d) defines that the neuron activation status indicator variables are either 0 or 1; and (e) defines constraints on the upper, $u_j^{(i)}$, and lower, $l_j^{(i)}$, bounds of the pre-activation value of the jth neuron in the ith layer. Deactivating a neuron, $a_j^{(i)} = 0$, simplifies the first of the (e) constraints to $\hat{z}_j^{(i)} \leq 0$, and activating a neuron simplifies the second to $\hat{z}_j^{(i)} \leq z_j^{(i)}$, which is consistent with the semantics of $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

Alg. 7 describes STABILIZE solves this equation system. First, a MILP problem is created from the current assignment, the neural network, and the property of interest using formulation in Equation 8.2.

Note that the neuron lower ($l_j^{(i)}$) and upper bounds ($u_j^{(i)}$) can be quickly computed by polytope abstraction.

Next, it collects a list of all unassigned variables which are candidates being stabilized (line 2). In general, there are too many unassigned neurons, so STABILIZE restricts consideration to k candidates. Because each neuron has a different impact on abstraction precision we prioritize the candidates. In STABILIZE, neurons are prioritized based on their interval boundaries (line 3) with a preference for neurons with either lower or upper bounds that are closer to zero. The intuition is that neurons with bounds close to zero are more likely to become stable after tightening.

We then select the top-k (line 4) candidates and seek to further tighten their interval bounds. The order of optimizing bounds of select neurons is decided by its boundaries, e.g., if the lower bound is closer to zero than the upper bound then the lower bound would be optimized first. These optimization processes, i.e., MAXIMIZE (line 7 or line 13) and MINIMIZE (line 9 or line 11), are performed by an external LP solver (e.g., Gurobi [60]).

Note that the work in [119] uses the MILP system in Eq. 8.2 to encode the entire verification problem and thus is limited to the encodings of small networks that can be handled by an LP solver. In contrast, VERISTABLE creates this system based on the current assignment, which has significantly fewer constraints. Moreover, we only use the computed bounds of hidden neurons from this system, and thus even if it cannot be solved, VERISTABLE will continue.

8.1.2.2 Parallelism

The DPLL(T) process in VERISTABLE is designed as a tree-search problem where each internal node encodes an *activation pattern* defined by the variable assignments from the root. To parallelize DPLL(T), we adopt a beam search-like strategy that combines distributed search from Distributed Tree Search (DTS) algorithm [49] and Divide and Conquer (DNC) [79] paradigms for splitting the search space into disjoint subspaces that can be solved independently. At every step of the search algorithm, we select up to n nodes of the DPLL(T) search tree to create a beam of width n. This splits (like DNC) the search into n subproblems that are independently processed. Each subproblem extends the tree by a depth of 1.

Our approach simplifies the more general DNC scheme, since the n bodies of the **parfor** on line 6 of Fig. 6 are roughly load balanced. While this is a limited form of parallelism, it sidesteps one of the major roadblocks to DPLL parallelism – the need to efficiently synchronize across load-imbalanced subproblems [79, 80].

In addition to raw speedup due to multiprocessing, parallelism accelerates the sharing of information across search subspaces, in particular learned clause information for DPLL. In VeriStable, we only generate independent subproblems which eliminates the need to coordinate their solution. When all subproblems are complete, their conflicts are accumulated, Fig. 6 line 18, to inform the next round of search. As we show in §6.2, the engineering of this form of parallelism in DPLL(T) leads to substantial performance improvement.

As with any stochastic algorithm, VeriStable would perform poorly if it gets into a subspace of the search that does not quickly lead to a solution, e.g., due to choosing a bad sequence of neurons to split [51, 129, 37]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [57] to avoid being stuck in such a *local optima*.

VeriStable uses a simple restart heuristic that triggers a restart when either the number of processed assignments (nodes) exceeds a pre-defined number or the number of remaining assignments that need to be checked exceeds a pre-defined threshold. After a restart, VeriStable avoids using the same decision order of previous runs (i.e., it would use a different sequence of neuron splittings). It also resets all internal information except the learned conflict clauses, which are kept and reused as these are *facts* about the given constraint system. This allows a restarted search to quickly prune parts of the space of assignments. Although restarting may seem like an engineering aspect, it plays a crucial role in stochastic algorithms like VeriStable and helps reduce verification time for challenging problems as shown in §8.3.1.

8.1.3 Implementation

VeriStable is written in Python, and uses Gurobi [60] for LP solving and bounds tightening, and the LiRPA abstraction library [143] for approximation. Currently, VeriStable supports feedforward (FNN), convolutional (CNN), and Residual Learning Architecture (ResNet) neural networks that use

Tab. 8.1: Benchmark Instances. U: `unsat`, S: `sat`, ?: `unknown`.

Benchmarks	**Networks**		**Per Network**		**Tasks**	
	Type	Networks	Neurons	Parameters	Properties	Instances (U/S/?)
ACAS Xu	FNN	45	300	13305	10	139/47/0
MNISTFC	FNN	3	0.5-1.5K	269-532K	90	56/23/11
CIFAR2020	CNN	3	17-62K	2.1-2.5M	203	149/43/11
RESNET_A/B	CNN+ResNet	2	11K	354K	144	49/23/72
MNIST_GDVB	FNN	38	0.7-5.1K	0.2-3.0M	16	51/0/39
Total		**91**			**463**	**444/136**/133

ReLU. VeriStable supports the standard specification formats ONNX [4] for neural networks and VNN-LIB [38] for properties. These formats are standard and are supported by state-of-the-art NNV tools, which enable comparative evaluation. The VeriStable implementation is open-source and can be retrieved from `https://github.com/dynaroars/neuralsat`.

8.2 Experimental Design

Our goals are to understand how incorporating stabilization and other DPLL(T) optimizations allows for scaling of NNV. We focus our evaluation on three research questions as follows:

- **RQ1**: How does stabilization impact the performance of DPLL(T)-based NNV?
- **RQ2**: How do VeriStable optimizations improve performance in isolation and combination?
- **RQ3**: How does VeriStable compare to state-of-the-art neural network verifiers?

8.2.1 Selection of NNV Benchmarks

To gain insights into the performance improvements of VeriStable we require benchmarks that force the algorithm to search a non-trivial portion of the space of activation patterns. It is well-known that SAT problems can be very easy to solve regardless of their size or whether they are satisfiable or unsatisfiable [55]. The same is true for NNV problems. The organizers of the first three neural network verifier competitions remark on the need for benchmarks that are "not so easy that every tool can solve all of them" to assess verifier performance [24].

To accomplish this task, we utilize the systematic NNV benchmark generator GDVB, which is outlined in § 5, and implement the epsilon controlling approach detailed in § 4.2. In this experiment, we used the $\text{MNIST}_{3\times1024}$ (refer to Tab. A.1) network with 3 layers as the seed network, each with 1024 neurons, and generated 38 different neural networks that cover combinations of parameter variations. We leverage the fact that local robustness properties are a pseudo-canonical form for pre-post condition specifications [105] and use GDVB to generate 16 properties with varying radii and center points. Next we run two state-of-the-art verifiers: CROWN and MN-BaB, for each of the $38 * 16 = 608$ combinations of neural network and property with a small timeout of 200 seconds. Any problem that could be solved within that timeout was removed from the benchmark as "too easy". This resulted in 90 verification problems that not only are more computationally challenging than benchmarks used in other studies, e.g., [93], but also exhibit significant architectural diversity. We use this **MNIST_GDVB** benchmark for RQ1 and RQ2 to study the variation in performance on challenging problems.

For RQ3 we use five VNN-COMP standard benchmarks in addition to MNIST_GDVB. These benchmarks, shown in Tab. 8.1, consist of 91 networks, spanning multiple types and architectures of layers, and 463 safety and robustness properties. The **Per Network** column gives the size of each network (**neurons** are the numbers of hidden neurons and **parameters** are the numbers of weights and biases). For example, each FNN in ACAS Xu has 5 inputs, 6 hidden layers (each with 50 neurons), 5 outputs, and thus has 300 neurons (6×50) and 13305 parameters ($5\times50\times50+2\times50\times5+6\times50+5$).

In total, we have 713 problem instances (an instance is the verification task of a property of a network). Among these instances, 444 are known to be `unsat` (U), 136 are `sat` (S), and 133 are unknown (?) because no existing verifiers, in this study or VNN-COMP, can solve them. We exclude unknown instances from our study because they do not contribute to our evaluation or comparison to other tools.

The six benchmarks are as follows. **ACAS Xu** consists of 45 FNNs to issue turn advisories to aircraft to avoid collisions. Each FNN has 5 inputs (speed, distance, etc). We use all 10 safety properties as specified in [68] and VNN-COMP, where properties 1–4 are used on 45 networks and properties 5–10 are used on a single network. **MNISTFC** consists of 3 FNNs for handwritten digit

recognition and 30 robustness properties. Each FNN has 28x28 inputs representing a handwritten image. **CIFAR2020** has 3 CNNs for object detection and 203 robustness properties (each CNN has a set of different properties). Each network uses 3x32x32 RGB input images. For **RESNET_A/B**, each benchmark has only one network with the same architecture and 72 robustness properties. Each network uses 3x32x32 RGB input images.

8.2.2 Selection of Neural Network Verifiers Baselines

For RQ1 we compare VeriStable to **NeuralSAT** [41] which is the only DPLL(T) neural network verifier available. NeuralSAT is recent and did not participate in VNN-COMP. However, it has been shown to have good performance for feedforward networks. RQ2 compares different configurations of VeriStable to each other. For RQ3, we selected four well-known neural network verifiers as baselines for comparison in addition to NeuralSAT. **CROWN** [129, 147] employs multiple abstractions and algorithms for efficient analysis, e.g., input splitting for networks with small input dimensions and parallel Branch-and-Bound [27] (BaB) otherwise. **MN-BaB** [51], the successor of ERAN [111, 109], uses multiple abstractions and BaB. **Marabou** [70, 69], the successor of the Reluplex work, is a simplex-based solver that employs a parallel Split-and-Conquer (SnC) [132] search and uses polytope abstraction [111] and LP-based bound tightening. **Nnenum** [8] combines optimizations such as parallel case splitting and multiple levels of abstractions, e.g., three types of zonotopes with imagestar/starset [122].

These four tools competed in VNN-COMP [93] and were among the very top performers. For example, CROWN is the winner for MNISTFC and also the overall winner, MN-BaB ranked 3rd on MNISTFC and second overall, and Nnenum was the only one that can solve all instances in ACAS Xu and was 4th overall. Marabou ranked 6th on MNISTFC and 7th overall.

8.2.3 Experimental Setup

Our experiments were run on a Linux machine with an AMD Threadripper 64-core 4.2GHZ CPU, 128GB RAM, and an NVIDIA GeForce RTX 4090 GPU with 24 GB VRAM. All tools use multi-processing (even external tools/libraries including Gurobi, LiRPA, and Pytorch are multi-thread).

CROWN, MN-BaB, NeuralSAT, and VeriStable leverage GPU processing for abstraction.

To conduct a fair evaluation, we reuse the benchmarks and installation/run-scripts available from VNN-COMP[3]. These scripts were tailored by the developers of each verifier to maximize performance on each benchmark. VNN-COMP uses varying runtimes for each problem instance ranging from 30 seconds to more than 20 minutes. The competition also uses several different Amazon AWS instances with different configurations (e.g., CPU, GPU, RAM) to run the tools. Thus, we experimented with timeouts on our machine and settled on 900 seconds per instance which allowed the verifiers to achieve similar scoring performance reported in VNN-COMP'22.

8.3 Results and Analysis

We discuss the metrics for each question, present experimental results, and interpret those results to answer the research questions.

8.3.1 RQ1: Benefit of Stabilization

We focus here on the benefit of stabilization on DPLL(T)-based NNV as implemented in NeuralSAT. We use the 51 challenging verification problems in the MNIST_GDVB benchmark with a time limit of 900 seconds to explore performance and measure the number of problems solved and the time to solve problems as metrics. Tab. 8.2 and Fig. 8.2 shows the performance differences between various VeriStable optimization settings vs. the original NeuralSAT tool. For example, "N" stands for the base case (NeuralSAT), "P" enables Parallelism, "R" enables Restarts, and "S" enables Stabilization.

The first row in Tab. 8.2 and the black dashed line of Fig. 8.2 presents data on NeuralSAT. The plot shows the problems solved within the 900-second timeout for each technique sorted by runtime from fastest to slowest; problems that timeout are not shown on the plot. Enabling only stabilization in VeriStable yields the data indicated with an "S": the second row and yellow lines, respectively. We observe a 50% increase in the number of problems solved with stabilization. The

[3]`https://github.com/ChristopherBrix/vnncomp2022_benchmarks`

Tab. 8.2: Problems Solved and Solving Time of NeuralSAT vs. VeriStable with Various Optimization Settings

Tool	Setting	#Solved	Avg. Time
NeuralSAT	-	4	867.35
VeriStable	S	6	833.25
	P	14	713.21
	P+R	17	741.00
	P+S	38	430.60
	P+S+R	48	330.46

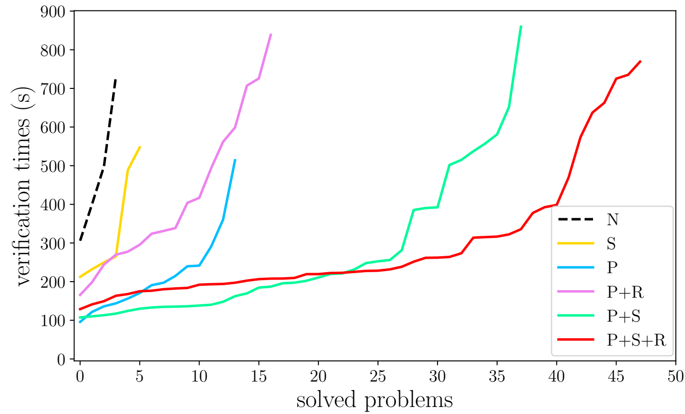
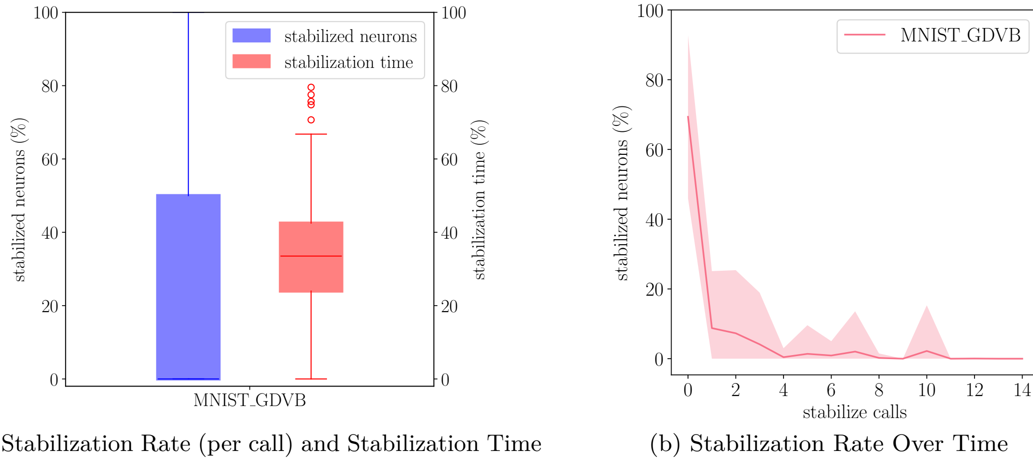


Fig. 8.2: Problems solved of NeuralSAT vs. VeriStable with various optimization settings

average times show a modest reduction of about 4%, but since NeuralSAT or "S" solved just a few benchmarks the average is swamped by the time taken by problems that timeout – at 900 seconds. Comparing the dashed and yellow lines in Fig. 8.2 shows that for the solved problems "S" reduces verification time significantly, e.g., on the first problem from just over 300 seconds to just over 200 seconds. Stabilization alone improves performance, but it has a much more significant benefit in combination with other optimizations.

We collected data to understand how frequently neurons could be stabilized and at what cost. Fig. 8.3a plots the percentage of neurons that are stabilized across the MNIST_GDVB benchmark, on the left axis, and the percentage of verification time taken up by stabilization, on the right axis. This aggregated data shows that stabilization can incur a non-trivial share of verification time, but



(a) Stabilization Rate (per call) and Stabilization Time

(b) Stabilization Rate Over Time

Fig. 8.3: Stabilization Cost and Effectiveness during Verification

as the data in Tab. 8.2 and Fig. 8.2 showed despite this overhead the overall verification time is reduced for solved problems.

We can also observe that while the mean number of stabilized neurons is low, the variance is quite high which indicates a degree of effectiveness in reducing the combinatorics in subsequent searches. We dug into the stabilization data further to try to understand this variance. Fig. 8.3b plots the mean – red line – and standard deviation – shaded region – of the number of stabilized neurons over time during verification; recall from line 9 of Alg. 6 that stabilization is selectively enabled during search. Stabilization is effective early in the search and less so as it progresses. This makes sense since line line 3 in Alg. 7 prioritizes neurons for stabilization. This is desirable because it encourages stabilization at the beginning of the search which leads to a greater combinatorial reduction in the search and a consequent improvement in its scalability.

RQ1 Findings: Stabilization improves the number of problems solved and reduces verification time. It does so by trading overhead to compute stable neurons to linearize parts of the search of the space of activation patterns. Moreover, it pushes this linearization to the top of the search tree to yield greater combinatorial reduction.

8.3.2 RQ2: Optimization Ablation Study

We used the same benchmark as in RQ1, but here we focus primarily on the benefits and interactions among the optimizations in VeriStable. We omit the use of restart on its own since it is intended to function in concert with parallelization. Both “S” and “P” improve the number of problems solved and reduce cost relative to the NeuralSAT baseline, but parallelism yields greater improvements. When parallelism is combined with restart we see that the number of problems solved increases, but the average time increases slightly. The reason for this is that for the 3 additional benchmarks that could be solved the verification process had conducted a partial search of the space of activation patterns before restarts and the cost of that search is added to the cost of the successful post-restart search.

Perhaps most noteworthy is the data on parallelism in combination with stabilization. We see a significant jump in the number of solved problems relative to both “S” and “P” – a 6.3 fold and 2.7 fold increase, respectively. As illustrated in Fig. 8.1 this combination is synergistic because stabilization creates a *narrower* tree within which the parallel *beam* can make more rapid progress. Adding in restart yields the best performance in terms of both problems solved – 12 fold increase NeuralSAT – and solve time – 2.6 fold decrease.

Fig. 8.2 shows the trend in verification solve times for each optimization combination across the benchmarks. One can observe that adding more optimizations improves performance both by the fact that the plots are lower and extend further to the right. For example, extending “P” to “P+S” shows lower solve times for the first 17 problems – the one’s “P” could solve – and that 38 of the 51 benchmark problems are solved. Extending “P+S” to the full set of optimizations exhibits what appears to be a degradation in performance for the first 23 problems solved and this is likely because, as explained above, restart forces some re-exploration of the search. However, the benefit of restarting shows in the ability to significantly reduce verification time for 25 of the 48 problems solved by “P+S+R”.

Tab. 8.3: A **Verifier**'s Rank (#) is Based on its VNN-COMP Score (**S**) on a Benchmark. (For each benchmark, the number of problems verified (**V**) and falsified (**F**) are shown)

Verifier	ACAS Xu				MNISTFC				CIFAR2020				RESNET_A/B				MNIST_GDVB				Overall			
	#	**S**	**V**	**F**	**#**	**S**	**V**	**F**	**#**	**S**	**V**	**F**	**#**	**S**	**V**	**F**	**#**	**S**	**V**	**F**	**#**	**S**	**V**	**F**
VeriStable	**1**	**1437**	**139**	**47**	2	573	55	**23**	**1**	**1533**	**149**	**43**	**1**	**513**	**49**	**23**	**1**	**480**	**48**	0	**1**	**4536**	**440**	**136**
CROWN	3	1436	**139**	46	**1**	**582**	**56**	22	2	1522	148	42	**1**	**513**	**49**	**23**	2	400	40	0	2	4453	432	133
NeuralSAT	5	1417	137	**47**	4	383	36	**23**	4	1522	148	42	3	483	46	**23**	4	40	4	0	3	3845	371	135
MN-BaB	6	1097	105	**47**	5	370	36	10	3	1486	145	36	4	363	34	**23**	3	200	20	0	4	3516	340	116
Nnenum	**1**	**1437**	**139**	**47**	3	403	39	13	5	518	50	18	-	-	-	-	-	-	-	-	5	2358	228	78
Marabou	4	1426	138	46	6	370	35	20	-	-	-	-	-	-	-	-	-	-	-	-	6	1796	173	66

RQ2 Findings: Each of the VeriStable optimizations improves on the performance of the baseline DPLL(T)-based neural network verifier. Moreover, combinations of the optimizations appear to operate synergistically to increase performance beyond their additive benefits. When running VeriStable, enabling all optimizations appears to be the best choice.

8.3.3 RQ3: Comparison with State-of-the-Art Neural Network Verifiers

In this section, we evaluate VeriStable relative to a set of 5 baseline neural network verifiers across a broader benchmark that reflects the problems used in VNN-COMP [93]. For metrics, we adopt the scoring system proposed for VNN-COMP 2023 which seeks to balance the relative difficulty of verifying a problem versus falsifying it and to account for the possibility that verifiers report erroneous results. More specifically, for each benchmark instance, a verifier scores 10 points if it correctly verifies an instance, 1 point if it correctly falsifies an instance, 0 points if it cannot solve (e.g., times out, has errors, or returns `unknown`), and -150 points if it gives incorrect results[4]. This scoring emphasizes a technique's ability to correctly verify problems[5].

Tab. 8.3 shows the results of all six tools. Since the magnitude of the score is not easily interpreted, since it depends on the size of the benchmark, we report the **Rank** of each tool using the VNN-COMP score for each benchmark as well as the overall rank. Tools that do not work on a benchmark are not shown under that benchmark (e.g., Marabou reports errors for all CIFAR2020 problems, Nnenum and Marabou cannot solve any instances of MNIST_GDVB). The last two columns break down the

[4]We note that all of the verifiers in our study gave correct results on the considered benchmarks.

[5]We dropped the extra 2 bonus points for the fastest verifiers in the VNN-COMP'22 scoring system because VNN-COMP has removed this time bonus as they found it did not make a difference in scoring

number of problems each verifier was able to **Verify** or **Falsify**.

On 5 of the 6 benchmarks, and overall, VeriStable ranks at the top, tying with other verifiers on the ACAS Xu and RESNET benchmarks. Recall that these benchmarks vary significantly in the number of neurons and parameters, with the ACAS Xu models being modestly sized and the CIFAR models being the largest, and VeriStable is the best on both ends of the scale spectrum. VeriStable ranks second on the MNISTFC benchmark to CROWN both solve the same number of problems, but CROWN verifies a problem that VeriStable does not lead to its higher score. The MNIST_GDVB benchmark varies in size from being comparable to the smallest MNISTFC network to larger than the largest MNISTFC network. Still, a key distinguishing feature of the benchmark is the filtration of *easy* problems. Whereas MNISTFC includes 23 problems that can be falsified, MNIST_GDVB has none, yet VeriStable performs better on these harder problems.

While not a factor in our evaluation, we note that several baseline verifiers require hyperparameter tuning. For example, the run-script of CROWN for VNN-COMP customizes 10 parameters per *each* benchmark to optimize its performance[6]. In contrast, when run with all optimizations enabled, which we recommend based on RQ2's findings, VeriStable has two parameters: the degree of parallelism, n, and the number of neurons to attempt to stabilize, k. In these experiments, we fixed these at $k = 64$ and $n = 4000$ for all benchmarks, which we believe is evidence that developers can more easily apply VeriStable to new benchmarks while achieving good performance.

RQ3 Findings: VeriStable ranks at the top of a set of baseline neural network verifiers that were shown to be the best performers in a recent NNV competition [93]. It performs well on smaller problems like ACAS Xu, where techniques with sophisticated abstract domains like Nnenum work well. It performs well on larger problems like CIFAR2020, where techniques like Nnenum fail to solve problems and even highly optimized abstraction-based methods like CROWN fall short. It performs well on challenging problems like MNIST_GDVB, forcing verifiers to analyze the combinatorially sized space of activation patterns to verify problems.

[6]For MNIST_GDVB, the default configuration of CROWN performed poorly, so we adopted the configuration used for MNISTFC which gave good results for MNIST_GDVB.

8.4 Threats to Validity

Regarding threats to internal validity, we built off the existing code base of NeuralSAT, thereby leveraging that team's efforts to validate their implementation. We used assertions in almost every function of our implementation to check the correctness properties flowing from our algorithms, e.g., that lower and upper bounds are properly ordered. Those assertions were enabled during our rigorous testing process that ran all of the VNN-COMP benchmarks through our implementation, where we confirmed the expected results.

We selected VNN-COMP benchmarks to promote comparability and enhance external validity. Those benchmarks were developed by other researchers to express verification problems for neural networks, e.g., ACAS Xu is a collision avoidance prediction network for small aerial drones. Based on our own experience and the experience of the VNN-COMP organizers, who found that some of the VNN-COMP benchmarks were *too easy*, we developed a new benchmark, MNIST_GDVB. That benchmark was developed using an approach that guarantees a form of systematic diversity across the networks and specifications that comprise the benchmark. We plan to continue to push for the development of benchmarks that reflect the challenges of NNV, but in this work, we believe our benchmarks are broader and more challenging than prior work.

Regarding construct validity, we used standard metrics, like number of problems solved and VNN-COMP score, that have been widely used [140, 93]. This makes comparing our results to prior work easier and allows researchers familiar with the metrics to interpret our results easily. Moreover, the metrics lead to a natural interpretation that permits answering the research questions, e.g., a verifier that solves more problems has better performance.

8.5 Discussion

In addition to scalability, which is the focus of VeriStable, there are two other common challenges in NNV: identifying valuable correctness properties of neural networks and developing a formal notation to encode them.

8.5.1 Property Specifications

This paper focuses on improving the verification of specifications formulated using sets of half-space polytopes – each specified as the conjunction of cutting planes – where one set defines the pre-condition, ϕ_{in}, and another the post-condition, ϕ_{out}. While it does not address the pragmatics of expressing meaningful domain-specific specifications, we note that this is an active area of work [121, 54]. For example, the work in [121] allows one to define domain-specific masking and transformation operations to localize perturbations to a region within an input image and within a range of values that remain on the data distribution. For example, the color of vehicles in a scene does not impact the predicted steering angle for an end-to-end driving model. We note, however, that such properties are amenable to verification with tools like VeriStable.

This paper leverages the fact that an arbitrary half-space polytope specification can be expressed as a local robustness property [105]. This means that we can evaluate verification scalability improvements by only considering local robustness specifications and these results will be informative about the verifier performance on a much broader class of specifications, like those in [121].

8.5.2 Specification Format

As mentioned, half-space polytope allows for a general class of specifications to be checked, but for high-dimensional input spaces it can be inconvenient to write specifications using notations like the standard VNN-LIB format [38]. For example, a well-studied class of specifications expresses local robustness properties of the form: $\forall x : \forall p \in [0, \epsilon] : N(x) = N(x \pm p)$. Expressing such a specification for MNIST requires choosing an input image, x, and a maximum perturbation, p, as defined by the robustness radius, ϵ. In VNN-LIB such a specification would would be more than 1500 lines long since each dimension of the 784 input must be constrained from above and below. Moreover, a separate specification must be produced for each input image and radius.

To address these pragmatic challenges, Shriver et al. developed the DNNV toolkit [104], which consists of a parametric Python-embedded DSL to express such specifications concisely, e.g., just 10

lines of code for the aforementioned MNIST specification[7]. Moreover, DNNV allows specifications to be written in a form that is independent of model input dimension, as above, and translated to VNN-LIB for verification with VeriStable.

8.6 Conclusion

As the need for formal analysis increases when more neural networks are being deployed in safety-critical areas, the NNV field has received great attention in recent years. In this chapter, we introduce VeriStable, a ReLU-based NNV tool that integrates an advanced DPLL(T) search technique in SAT solving with the concept of neuron stability to significantly reduce the search space of NNV. Our evaluation confirms the effectiveness of VeriStable, which establishes a new state-of-the-art in NNVs compared to the performances in the recent NNV competition.

[7] `https://github.com/dlshriver/DNNV`

Chapter 9

Conclusion & Future Work

9.1 Conclusion

Due to the increasing deployment of neural network models in various domains, researchers have become increasingly concerned with ensuring their correct behaviors, particularly in safety-critical areas. Consequently, the field of neural network verification has experienced significant advancements over the past six years. However, the emergence of approximately 10 new verifiers annually necessitates thorough empirical analysis to comprehend the variations in their performance. Regrettably, despite the diligent efforts of the NNV community, the growth rate of neural networks surpasses that of the verifiers. Hence, there is an urgent need to enhance the performance of verifiers to effectively scale up to real-world applications. This dissertation presents two significant contributions to the field of neural network verification: automated approaches in helping with the investigation into the performance characteristics of neural network verifiers and the subsequent enhancement of their scalability.

In the initial phase, we devised innovative methodologies to comprehend the performance characteristics of neural network verifiers through extensive empirical studies. Our first step in NNV performance analysis involves identifying nine influential factors that impact verification performance. Additionally, we developed strategies to balance SAT/UNSAT instances and eliminate easy instances,

as mentioned in Chapter 4. Chapter 5 introduces a groundbreaking approach that systematically generates diverse NNV benchmarks by leveraging the concept of combinatorial interaction testing from traditional software engineering. This approach enables the synthesis of unbiased benchmark instances that encompass a wide range of neural network architectures and verification property specifications, that closely represent real-world problems. Expanding upon the ideas presented in Chapter 5, Chapter 6 extends the approach to iteratively generate benchmarks, aiming to explore the verification performance boundary for a given verifier. This not only offers a fresh perspective for comparing the performance disparities between verifiers but also serves as an effective means to investigate potential bottlenecks. By employing this methodology, we successfully identified the "neuron stability" bottleneck, which significantly impacts the scalability of the verifier when dealing with larger models.

In the subsequent phase, we aim to enhance the scalability of the state-of-the-art verifiers by addressing the performance bottleneck known as "neuron stability" in both the training and verification procedures. In Chapter 7, we have developed two innovative methods to guide the training process of neural networks, resulting in networks with fewer unstable neurons. This, in turn, reduces the necessity for extensive case-splitting in the verifiers, thereby minimizing costs and increasing their scalability. Our evaluation demonstrates that these methods not only effectively increase the number of problems solved but also significantly reduce the time required for solving them. In Chapter 8, we further enhance an existing verification method by incorporating an optimizer that reduces the occurrence of unstable neurons during the verification process. This optimization leads to a significant reduction in the search space for the verification problem. In conjunction with the restart and parallelism optimizations, our verifier establishes a new state-of-the-art in the NNV competition

The research conducted in this dissertation has resulted in multiple publications [106, 140, 139, 42], in addition to the creation of six open-source research software artifacts, namely R4V [107], GDVB [141], SwarmHost [136], AdaGDVB [137], OCTOPUS [138], and VeriStable [40]. The GDVB and OCTOPUS artifacts have been endorsed by conferences as being available, functional, and reusable. Our research tools have received 12 stars and are actively under development to be advocated to help more researchers advance the NNV field.

9.2 Future Work

In this section, we elaborate on our vision for a compilation of forthcoming projects stemming from the research findings presented in this dissertation.

9.2.1 NNV Benchmark Generation

In Chapter 4, it has been demonstrated that 9 factors may impact the performance of the verifier. Our first opportunity is to continue to explore additional potential factors that can influence the verification performance. Furthermore, both GDVB and AdaGDVB employ the R4V tool for synthesizing neural networks, which supports a limited number of network architectures. We plan to continue enhancing R4V to introduce greater diversity in the network architectures of GDVB and AdaGDVB, with larger and more complicated networks like Residual networks, VAEs, GANs, or even large language models (LLM).

In Chapter 4, we formulated a technique to balance between SAT/UNSAT and filtering out "easy" verification instances. This method was then utilized in Chapter 8 to create the MNIST_GDVB benchmark. Notably, the MNIST_GDVB benchmark has proven to be more effective in distinguishing the state-of-the-art verifiers compared to the most recent benchmarks in VNN-COMP. We intend to incorporate this as an automated feature within GDVB, enabling the generation of more meaningful and challenging VNN benchmarks. Additionally, we aim to contribute multiple benchmarks produced by both GDVB and AdaGDVB to support the forthcoming VNN-COMP, thereby enriching it with a wider range of diverse and challenging benchmarks.

We envision another use case for AdaGDVB in VNN-COMP. Using a collection of verifiers, including top performers from the prior year, organizers can compute benchmarks that characterize the aggregate VPB of that collection. This would, by construction, exclude problems that they could all solve as well as those that none could solve – meeting the complexity requirement of VNN-COMP organizers. Moreover, this process could be repeated each year to allow the benchmark to track progress in the field by becoming progressively more challenging over time. Multiple such benchmarks, using different seed networks defined perhaps by competition participants, would provide a means of

automatically generating a relatively unbiased and evolving benchmark to help drive advances in the field of NNV.

9.2.2 Training Verifiable Neural Networks

Chapter 7 presents evidence that the implementation of stabilization techniques can result in neural networks that significantly improve their verification speed while maintaining a high level of model accuracy. Additionally, we have plans to delve deeper into this area of research. Initially, we have devised a set of direct optimizations for OCTOPUS, including: (1) extending our methods to accommodate real-world large neural network architectures; (2) exploring automated approaches to fine-tune hyper-parameters for improved performance; (3) further enhancing the performance of stabilizers while minimizing any potential trade-offs in accuracy; (4) investigating the applicability of combining multiple stabilizers; and finally, (5) studying verification algorithms to gain insights on how to tailor stabilizers for maximum benefit.

In addition, we anticipate more open opportunities, including training for property-preserving networks [43]. We believe that for neural networks to preserve certain property specifications, it is required to integrate them directly as objectives during the training phase. Conversely, we assume that the verification process inherently involves the analysis of property-preserving sets of data when analyzing the neural network. Consequently, it is feasible to utilize such data in the network training procedure to train property-preserving networks.

9.2.3 Enhancing the Scalability of the Neural Network Verifiers

In Chapter 8, we present the VeriStable verifier, which is an enhanced version of the existing DPLL(T) verifier called NeuralSAT. This enhancement significantly improves the performance of NeuralSAT, making it a cutting-edge neural network verifier. There are numerous opportunities available to further enhance the performance of VeriStable. One such opportunity is to extend the capability of neuron stabilization to other non-DPLL-based NNV techniques. Additionally, we can explore new decision heuristics based on neuron stabilization specifically designed for DPLL(T)-based tools. Furthermore, VeriStable benefits from utilizing DPLL with implication graphs, which

provides it with a built-in mechanism to verify its results. This is achieved by utilizing these graphs and conflicting clauses to generate resolution graphs/proofs and UNSAT cores as proofs of unsatisfiability [3, 74, 150].

Furthermore, we aim to support for broader network architectures. Specifically, we are keen on expanding the tool towards domain-specific applications (e.g., autonomous driving, medical devices, etc.) and additional types of neural networks like Graph Neural Networks, Transformers, and LLMs, etc. Additionally, we are interested in supporting more complex property specifications, including temporal properties, metamorphic properties, and so forth.

Appendix A

Neural Networks Artifacts and Verifiers

A.1 Neural Network Datasets

This section describes all the neural network training datasets that are used in the thesis.

The MNIST dataset(Modified National Institute of Standards and Technology) [39] is the most recognized dataset for neural network research. It contains a set of 70K labeled instances. Each instance contains a handwritten digit image of size 28*28 and a label in the form of an integer number of 10 classes that describes the ground truth of the image from '0' to '9'. The complete set is divided into 60K training and 10K testing instances.

The CIFAR datasets(Canadian Institute For Advanced Research) [73] is also a popular dataset. The two variations CIFAR-10 and CIFAR-100 are created from subsets of the 80 million tiny image dataset. The CIFAR-10 dataset contains 50K training images and 10K test images of 10 classes, whereas the CIFAR-100 contains 5K training images and 1K test images of 100 classes. The images in the datasets are randomly selected and evenly distributed among the 10/100 different classes.

The self-driving car projects [77] offered by Udacity encompass various tasks, and one of the

Tab. A.1: The Complete Set of Neural Network Architectures

Dataset	**Structure**	**Neurons**	**Parameters**
$\text{MNIST}_{2\times256}$	$\text{FC}(256)\times 2$	512	268,800
$\text{MNIST}_{6\times256}$	$\text{FC}(256)\times 6$	1,536	530,944
$\text{MNIST}_{3\times1024}$	$\text{FC}(1,024)\times 3$	3,072	2,902,528
CIFAR_{2020}	Conv(32,5,2), Conv(128,4,2), FC(250)	49,412	2,133,736
$\text{MNIST}_{\text{ConvBig}}$	Conv(32,3,1), Conv(32,4,2), Conv(64,3,1), Conv(64,4,2), FC(512), FC(512)	48,074	1,974,762
$\text{CIFAR}_{\text{ConvBig}}$	Conv(3,3,1), Conv(32,3,1), Conv(32,4,2), Conv(64,3,1), Conv(64,4,2), FC(512), FC(512)	65,546	2,466,870
DAVE-2	Conv(24,5,2), Conv(36,5,2), Conv(48,5,2), Conv(64,3,1), Conv(64,3,1), FC(100), FC(50), FC(10)	82,669	2,116,983

prominent challenges involves predicting the steering angle of the autonomous vehicle. Within this dataset, there are 117K colored images depicting the road ahead, with clearly marked lane lines. Each image is accompanied by a label indicating the corresponding steering angle of the vehicle in that particular scenario. The objective is to train neural networks to accurately forecast the appropriate steering angles based on the visual information provided by the images. The Dave-2 network [97] is an example of an architecture specifically designed to operate on this dataset.

A.2 Neural Network Architectures

The thesis incorporates seven fundamental neural network architectures, which are detailed in the Tab. A.1. These include three Fully connected networks designed for the MNIST dataset: $\text{MNIST}_{6\times256}$, $\text{MNIST}_{2\times256}$, and $\text{MNIST}_{3\times1024}$, and a single Convolutional MNIST network: $\text{MNIST}_{\text{ConvBig}}$; two CIFAR neural networks: $\text{CIFAR}_{\text{ConvBig}}$ and CIFAR_{2020}, as well as a DAVE-2 network tailored for the Udacity Driving dataset. The neural architectures exhibit a range of neuron counts, spanning from 512 to over 82K, and parameter quantities varying from 268K to 2B. The assortment of architectures and their respective sizes offer a diverse array of empirical investigations within the thesis.

Tab. A.2: The Complete Set of Neural Network Verifiers

Verifier	**Category**	**Key Techniques**
$\text{ERAN}_{\text{DeepZono}}$	Reachability	Zonotope
$\text{ERAN}_{\text{DeepPoly}}$	Reachability	Polytope
$\text{ERAN}_{\text{RefineZono}}$	Reachability	Zonotope
$\text{ERAN}_{\text{RefinePoly}}$	Reachability	Polytope
Planet	Search-Optimization	SMT
ReLuplex	Search-Optimization	SMT
BaB	Search-Optimization	SMT
BaBSB	Search-Optimization	SMT
Neurify	Optimization	Interval
Nnenum	Reachability	Star Set
Marabou	Search-Optimization	SMT
α,β-CROWN	Optimization	LP
MN-BaB	Reachability	Polytope
NeuralSAT	Search-Optimization	LP

A.3 Neural Network Verifiers

The thesis includes a comprehensive collection of 14 neural network verifiers, as outlined in Tab. A.2. These verifiers can be broadly categorized into three groups: reachability, search-optimization, and optimization. Additionally, the study incorporates six different underlying key techniques, namely Zonotope, Polytope, Interval, Stat Set abstract domains, SMT solvers, and Linear Programming (LP). For a more detailed understanding of the verification techniques, please consult § 3.3. This diverse set of 14 verifiers encompasses a wide range of algorithmic families and underlying techniques, providing a solid foundation for the experimental evaluations conducted throughout the thesis.

Bibliography

[1] A. Albarghouthi. Introduction to neural network verification. *arXiv preprint arXiv:2109.10317*, 2021.

[2] G. Amendola, F. Ricca, and M. Truszczynski. A generator of hard 2QBF formulas and ASP programs. In *16th International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

[3] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 16–30. Springer, 2008.

[4] J. Bai, F. Lu, and K. Zhang. ONNX Open neural network exchange, 2023.

[5] S. Bak. Execution-guided overapproximation (ego) for improving scalability of neural network verification. In *Proc. 3rd Int. Workshop on Verification of Neural Networks (VNN)*, 2020.

[6] S. Bak. nnenum: Verification of relu neural networks with optimized abstraction refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer, 2021.

[7] 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.

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

[9] T. Baluta, Z. L. Chua, K. S. Meel, and P. Saxena. Scalable quantitative verification for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 312–323. IEEE, 2021.

[10] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[11] C. Barrett, M. Deters, L. De Moura, A. Oliveras, and A. Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[12] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 13:14, 2010.

[13] C. W. Barrett. Decision Procedures: An Algorithmic Point of View. *J. Autom. Reason.*, 51(4):453–456, 2013.

[14] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring neural net robustness with constraints. *arXiv preprint arXiv:1605.07262*, 2016.

[15] D. Beyer. Competition on Software Verification.

[16] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019.

[17] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[18] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a powerpc- microprocessor using symbolic model checking without bdds. In *Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11*, pages 60–71. Springer, 1999.

[19] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[20] A. Biere and M. Preiner. Hardware Model Checking Competition.

[21] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[22] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener. Efficient verification of relu-based neural networks via dependency analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34(04), pages 3291–3299, 2020.

[23] C. Brix, S. Bak, C. Liu, and T. T. Johnson. The fourth international verification of neural networks competition (vnn-comp 2023): Summary and results. *arXiv preprint arXiv:2312.16760*, 2023.

[24] C. Brix, M. N. Müller, S. Bak, T. T. Johnson, and C. Liu. First three years of the international verification of neural networks competition (vnn-comp). *International Journal on Software Tools for Technology Transfer*, pages 1–11, 2023.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] R. Bunel, A. De Palma, A. Desmaison, K. Dvijotham, P. Kohli, P. Torr, and M. P. Kumar. Lagrangian decomposition for neural network verification. In *Conference on Uncertainty in Artificial Intelligence*, pages 370–379. PMLR, 2020.

[27] R. Bunel, P. Mudigonda, I. Turkaslan, P. Torr, J. Lu, and P. Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.

[28] R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar. A unified view of piecewise linear neural network verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 4795–4804, 2018.

[29] T. Chen, H. Zhang, Z. Zhang, S. Chang, S. Liu, P.-Y. Chen, and Z. Wang. Linearity grafting: Relaxed neuron pruning helps certifiable robustness. In *International Conference on Machine Learning*, pages 3760–3772. PMLR, 2022.

[30] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[31] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, 2007.

[32] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, Sep 2008.

[33] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *25th International Conference on Software Engineering*, pages 38–48, May 2003.

[34] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[35] M. Das, R. Ray, S. K. Mohalik, and A. Banerjee. Fast falsification of neural networks using property directed testing. *arXiv preprint arXiv:2104.12418*, 2021.

[36] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[37] A. De Palma, R. Bunel, A. Desmaison, K. Dvijotham, P. Kohli, P. H. Torr, and M. P. Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

[38] S. Demarchi, D. Guidotti, L. Pulina, and A. Tacchella. Supporting standardization of neural networks verification with vnn-lib and coconet. In *Proceedings of the 6th Workshop on Formal*, volume 16, pages 47–58, 2023.

[39] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[40] H. Duong, L. Li, D. Xu, T. Nguyen, and M. Dwyer. **NeuralSAT**: A DPLL(T) framework for verifying deep neural networks. `https://github.com/dynaroars/neuralsat`, 2023.

[41] H. Duong, T. Nguyen, and M. Dwyer. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266*, 2024.

[42] H. Duong, D. Xu, T. Nguyen, and M. Dwyer. Harnessing Neuron Stability to Improve DNN Verification. *Proceedings of the ACM on Software Engineering (PACMSE)*, FSE, 2024.

[43] K. Dvijotham, S. Gowal, R. Stanforth, R. Arandjelovic, B. O'Donoghue, J. Uesato, and P. Kohli. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265*, 2018.

[44] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. In *UAI*, volume 1(2), page 3, 2018.

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

[46] Y. Y. Elboher, J. Gottschlich, and G. Katz. An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer, 2020.

[47] M. Fazlyab, M. Morari, and G. J. Pappas. Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming. *IEEE Transactions on Automatic Control*, 2020.

[48] C. Feng, Z. Chen, W. Hong, H. Yu, W. Dong, and J. Wang. Boosting the robustness verification of dnn by identifying the achilles's heel. *arXiv preprint arXiv:1811.07108*, 2018.

[49] C. Ferguson and R. E. Korf. Distributed tree search and its application to Alpha-Beta Pruning. In *AAAI*, volume 88, pages 128–132, 1988.

[50] J. Ferlez, H. Khedr, and Y. Shoukry. Fast BATLLNN: fast box analysis of two-level lattice neural networks. In *Proceedings of the 25th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2022.

[51] C. Ferrari, M. N. Müller, N. Jovanovic, and M. T. Vechev. Complete verification via multi-neuron relaxation guided branch-and-bound. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[52] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[53] 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 *IEEE Symposium on Security and Privacy*, pages 3–18, May 2018.

[54] C. Geng, N. Le, X. Xu, Z. Wang, A. Gurfinkel, and X. Si. Towards reliable neural specifications. In *International Conference on Machine Learning*, pages 11196–11212. PMLR, 2023.

[55] I. P. Gent and T. Walsh. The SAT phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN, 1994.

[56] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[57] C. P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[58] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[59] X. Guo, W. Wan, Z. Zhang, M. Zhang, F. Song, and X. Wen. Eager falsification for accelerating robustness verification of deep neural networks. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 345–356. IEEE, 2021.

[60] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[61] L. Hadarean, A. Hyvarinen, A. Niemetz, and G. Reger. 14th International Satisfiability Modulo Theories Competition.

[62] P. Henriksen and A. Lomuscio. Efficient neural network verification via adaptive refinement and adversarial search. In *ECAI 2020*, pages 2513–2520. IOS Press, 2020.

[63] M. Heule, M. Järvisalo, M. Suda, M. Iser, and T. Balyo. The International Satisfiability Competitions.

[64] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.

[65] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[66] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices*, 47(6):77–88, 2012.

[67] T. T. Johnson and C. Liu. VNN-COMP2020 report.

[68] 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*, pages 97–117. Springer, 2017.

[69] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, 60(1):87–116, 2022.

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

[71] M. I. Khedher, H. Ibn-Khedher, and M. Hadji. Dynamic and scalable deep neural network verification algorithm. In *ICAART (2)*, pages 1122–1130, 2021.

[72] H. Khedr, J. Ferlez, and Y. Shoukry. Effective formal verification of neural networks using the geometry of linear regions. *ArXiv, abs/2006.10864*, 2020.

[73] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images, 2009.

[74] D. Kroening and O. Strichman. *Decision procedures.* Springer, 2016.

[75] T. Kropf. Benchmark-circuits for hardware-verification. In *International Conference on Theorem Provers in Circuit Design*, pages 1–12, 1994.

[76] R. Kuhn and R. Kacker. Automated Combinatorial Testing for Software.

[77] S. Kulshrestha. Udacity open source self-driving car. `https://github.com/udacity/self-driving-car`, 2024.

[78] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals. CNFgen: A generator of crafted benchmarks. In *Theory and Applications of Satisfiability Testing*, pages 464–473, 2017.

[79] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. PaInleSS: a framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*, pages 233–250. Springer, 2017.

[80] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I 25*, pages 135–151. Springer, 2019.

[81] W. Leeson and M. B. Dwyer. Algorithm selection for software verification using graph neural networks. *ACM Transactions on Software Engineering and Methodology*, 2021.

[82] J. Li, J. Liu, P. Yang, L. Chen, X. Huang, and L. Zhang. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *International Static Analysis Symposium*, pages 296–319. Springer, 2019.

[83] Z. Li, T. Chen, L. Li, B. Li, and Z. Wang. Can pruning improve certified robustness of neural networks? *arXiv preprint arXiv:2206.07311*, 2022.

[84] W. Lin, Z. Yang, X. Chen, Q. Zhao, X. Li, Z. Liu, and J. He. Robustness verification of classification deep neural networks via linear programming. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11418–11427, 2019.

[85] 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*, 4(3-4):244–404, 2021.

[86] R. Livni, S. Shalev-Shwartz, and O. Shamir. On the computational efficiency of training neural networks. *Advances in neural information processing systems*, 27, 2014.

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

[88] A. Loquercio, A. I. Maqueda, C. R. D. Blanco, and D. Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 2018.

[89] J. Lu and M. P. Kumar. Neural network branching for neural network verification. *arXiv preprint arXiv:1912.01329*, 2019.

[90] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai. Bottleneck detection using statistical intervention analysis. In A. Clemm, L. Z. Granville, and R. Stadler, editors, *Managing Virtualization of Networks and Services*, pages 122–134, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[91] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[92] C. Müller, F. Serre, G. Singh, M. Püschel, and M. Vechev. Scaling polyhedral neural network verification on gpus. *Proceedings of Machine Learning and Systems*, 3:733–746, 2021.

[93] 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.

[94] M. N. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–33, 2022.

[95] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[96] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*, pages 237–246. IEEE, 2013.

[97] T. Onishi, T. Motoyoshi, Y. Suga, H. Mori, and T. Ogata. End-to-end learning method for self-driving cars with trajectory recovery using a path-following function. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–8. IEEE, 2019.

[98] OpenAI. Introducing ChatGPT.

[99] M. Paul, F. Chen, B. W. Larsen, J. Frankle, S. Ganguli, and G. K. Dziugaite. Unmasking the lottery ticket hypothesis: What's encoded in a winning ticket's mask? In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[100] R. Pelánek. BEEM: benchmarks for explicit model checkers. In *International SPIN Workshop on Model Checking of Software*, pages 263–267, 2007.

[101] A. Raghunathan, J. Steinhardt, and P. Liang. Certified defenses against adversarial examples. *arXiv preprint arXiv:1801.09344*, 2018.

[102] J. Schoenen, M. Lenaerts, and E. Bastings. High-dose riboflavin as a prophylactic treatment of migraine: Results of an open pilot study. *Cephalalgia*, 14(5):328–329, 1994.

[103] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281, 2015.

[104] D. Shriver, S. Elbaum, and M. B. Dwyer. DNNV: A framework for deep neural network verification. In A. Silva and K. R. M. Leino, editors, *Computer Aided Verification*, pages 137–150, Cham, 2021. Springer International Publishing.

[105] D. Shriver, S. Elbaum, and M. B. Dwyer. Reducing DNN properties to enable falsification with adversarial attacks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 275–287. IEEE, 2021.

[106] D. Shriver, D. Xu, S. Elbaum, and M. B. Dwyer. Refactoring neural networks for verification. *arXiv preprint arXiv:1908.08026*, 2019.

[107] D. Shriver, D. Xu, S. Elbaum, and M. B. Dwyer. **R4V**: Refactoring for verification. `https://github.com/edwardxu0/r4v`, 2019.

[108] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[109] G. Singh, R. Ganvir, M. Püschel, and M. Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32:15098–15109, 2019.

[110] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. Effective robustness certification. *NeurIPS*, 1(4):6, 2018.

[111] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[112] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev. Boosting robustness certification of neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.* OpenReview.net, 2019.

[113] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield. Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4241–4247, Sep 2017.

[114] M. Sordo. Introduction to neural networks in healthcare. *Open clinical: Knowledge management for medical care*, 2002.

[115] M. Subramaniyan, A. Skoogh, A. S. Muhammad, J. Bokrantz, B. Johansson, and C. Roser. A generic hierarchical clustering approach for detecting bottlenecks in manufacturing. *Journal of Manufacturing Systems*, 55:143–158, 2020.

[116] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[117] G. Sutcliffe and C. Suttner. The TPTP problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[118] C. M. J. Tan and M. Motani. Dropnet: Reducing neural network complexity via iterative pruning. In *International Conference on Machine Learning*, pages 9356–9366. PMLR, 2020.

[119] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[120] F. Toledo, D. Shriver, S. Elbaum, and M. B. Dwyer. Distribution models for falsification and verification of DNNs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–329. IEEE, 2021.

[121] F. Toledo, D. Shriver, S. Elbaum, and M. B. Dwyer. Deeper notions of correctness in image-based dnns: Lifting properties from pixel to entities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2122–2126, 2023.

[122] H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. Star-based reachability analysis of deep neural networks. In *International symposium on formal methods*, pages 670–686. Springer, 2019.

[123] H.-D. Tran, N. Pal, P. Musau, D. M. Lopez, N. Hamilton, X. Yang, S. Bak, and T. T. Johnson. Robustness verification of semantic segmentation neural networks using relaxed reachability. In *International Conference on Computer Aided Verification*, pages 263–286. Springer, 2021.

[124] H.-D. Tran, X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2020.

[125] R. E. Uhrig. Use of neural networks in nuclear power plants. *ISA Transactions*, 32(2):139–145, 1993.

[126] A. Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing*, pages 317–328, 2011.

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

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

[129] 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 neural network robustness verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[130] L. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, D. Boning, and I. Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.

[131] E. Wong and Z. Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, pages 5286–5295. PMLR, 2018.

[132] H. Wu, A. Ozdemir, A. Zeljic, K. Julian, A. Irfan, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization techniques for verifying neural networks. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, volume 1, pages 128–137. TU Wien Academic Press, 2020.

[133] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys*, 55(5):1–37, 2022.

[134] W. Xiang, H.-D. Tran, and T. T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5777–5783, 2018.

[135] K. Y. Xiao, V. Tjeng, N. M. Shafiullah, and A. Madry. Training for faster adversarial robustness verification via inducing relu stability. *arXiv preprint arXiv:1809.03008*, 2018.

[136] D. Xu. [The **SwarmHost** Framework] A Unified Framework for Neural Network Verification, 2 2024.

[137] D. Xu, H. Duong, M. B. Dwyer, and T. Nguyen. [The **AdaGDVB** Framework] Adaptive Benchmark Generation for DNN Verification, 2 2024.

[138] D. Xu, N. J. Mozumder, H. Duong, and M. B. Dwyer. [The **OCTOPUS** Framework] Training for Verification: Increasing Neuron Stability to Scale DNN Verification(cav aec artifact evaluated[consistent, complete, well documented, easy to reuse]), 1 2024.

[139] D. Xu, N. J. Mozumder, H. Duong, and M. B. Dwyer. Training for verification: Increasing neuron stability to scale DNN verification. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 24–44, Cham, 2024. Springer Nature Switzerland.

[140] D. Xu, D. Shriver, M. B. Dwyer, and S. Elbaum. Systematic generation of diverse benchmarks for DNN verification. In *International Conference on Computer Aided Verification*, pages 97–121. Springer, 2020.

[141] D. Xu, D. Shriver, M. B. Dwyer, and S. Elbaum. [The **GDVB** Framework] Systematic Generation of Diverse Benchmarks for DNN Verification, 2 2024.

[142] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33, 2020.

[143] 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. In *International Conference on Learning Representations*, 2021.

[144] J. You, H. Wu, C. Barrett, R. Ramanujan, and J. Leskovec. G2SAT: Learning to generate SAT formulas. In *Advances in Neural Information Processing Systems*, pages 10552–10563, 2019.

[145] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 199–208. IEEE, 2012.

[146] 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.

[147] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Z. Kolter. General cutting planes for bound-propagation-based neural network verification. *arXiv preprint arXiv:2208.05740*, 2022.

[148] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. Efficient neural network robustness certification with general activation functions. *arXiv preprint arXiv:1811.00866*, 2018.

[149] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, pages 279–285. IEEE, 2001.

[150] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 880–885. IEEE, 2003.

[151] L. Zhangheng, T. Chen, L. Li, B. Li, and Z. Wang. Can pruning improve certified robustness of neural networks? *Transactions on Machine Learning Research*, 2022.

[152] Z. Zhong, Y. Tian, and B. Ray. Understanding local robustness of deep neural networks under natural variations. In *Fundamental Approaches to Software Engineering: 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 24*, pages 313–337. Springer International Publishing, 2021.