CGDTest: A Testing Tool for Deep Neural Network

by

Guanting Pan

A research project
presented to the University of Waterloo
in fulfillment of the
research paper requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2023

Abstract

Over the past years, Deep Neural Networks (DNNs) have brought a transformative impact in various domains, including Artificial Intelligence (AI), Nature Language Processing (NLP), and Finance. Concomitant with this achievement, there have been numerous concerns related to the reliability and security issues associated with DNNs. Examples of security problems comprise various types of attacks, including but not limited to adversarial, data poisoning, and membership inference attacks. These problems become particularly acute when considering DNN in the context of mission-critical tasks such as Airborne Collision Avoidance Systems (ACAS) for aircraft. To address these issues, this report introduces a tester, CGDTest, to verify the robustness of the DNN. CGDTest is a novel approach in the research field of neural network verification and the first to apply the Gradient Descent (GD) method for verifying the robustness of DNNs. CGDTest aims to find a DNN input that satisfies user-specified constraints. Firstly, CGDTest parses user-specified constraints and converts them into a differentiable constraint loss function. Then, CGDTest employs gradient descent methods to modify the initialized DNN input, moving in the direction of decreased loss using the computed gradient value with respect to the input. If an input results in a constraint loss function of zero, it is considered a satisfiable assignment for the constraints. Otherwise, CGDTest outputs UNSAT (i.e., unsatisfiable). CGTest has achieved VNN-COMP compliance. VNN-COMP is an excellent platform for comparing the performance of various verification and testing tools over large real-world benchmarks, so VNN-COMP is an effective metric for evaluating verification and testing tools in handling real-world benchmarks.

Acknowledgements

I would like to express my most profound appreciation to the following mentors and friends, without whom this report would not have been possible:

I would like to extend my most heartfelt gratitude to my supervisor, Dr. Vijay Ganesh, for his continuous support, guidance, and encouragement throughout my research journey. His invaluable expertise, patience, and enthusiasm for research have been instrumental in shaping my growth as a researcher. Thank you so much, Professor Ganesh, for giving me the opportunity to pursue my Master's degree under your supervision.

I would like to express my gratitude to Mr. Vineel Nagisetty, who proposed the innovative idea for this project. His leadership, enthusiasm, and confidence in our team significantly contributed to its success. I am grateful for your guidance, Vineel, and thank you for giving me the opportunity to participate in this project.

I would like to express my gratitude to Mr. Piyush Jha, a Ph.D. student in our research group. Thank you for all the support you provided for this project, and for your excellent contributions. This has been my best collaboration throughout my entire academic journey.

I would like to express my appreciation to Mr. Joseph Scott, a Ph.D. candidate in our research group. Joe began guiding me when I first joined the group. Joe, thank you for all the inspiration and support; you are the one who helped me become a better programmer and researcher.

I would like to express my gratitude to Dr. Christopher Srinivasa for giving me the opportunity to undertake a MITACS internship at Borealis AI and for his mentorship and guidance throughout. Thank you for all the support during my internship.

I also want to thank my family for their love and all the support they provided.

Last but not least, I want to express my deepest appreciation to all my friends, including but not limited to Candice Wang, Clark Yu, Alex Wang, Richard Dong, and Joyce Zhao. They have been my best friends since freshman year. Thank you all for the mental support and for helping me through the darkest times.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The improvements in novel Machine Learning (ML) algorithms, coupled with the availability of more computational resources and data has contributed to the rise in the uses of ML, including image processing, intelligent assistant, and safety-critical systems in autonomous cars and drones [18]. Deep Neural Networks (DNN)s are one of the prominent ML techniques, and they have had a significant impact in many fields including AI, science, and engineering. However, they can be vulnerable to different types of attacks, such as adversarial and model stealing attacks [26]. These problems become particularly acute when considering using DNNs in the context of mission-critical tasks, such as ACAS systems for aircraft [2].

To address the above issues, several tools to check if DNNs exhibit vulnerability have been developed by leading researchers. In this report, I introduce a new testing algorithm that are scalable, expressible, and can be applied to a wide variety of DNN architectures, named CGDTest [22]. CGDTest is a new approach proposed by Vineel Nagisetty, who is a talented researcher from the Borealis AI. CGDTest tests DNN vulnerabilities that utilize Constrained Gradient Descent (CGD). CGDTest aims to verify the robustness issues such as adversarial robustness and bias in DNNs. CGD allows testing ML models for multiple types of vulnerabilities due to the fact that it allows constraints to express properties. Further, it is highly scalable. This approach is avant-garde, and the approaches can significantly impact the field of neural network verification/testing.

Comparing existing DNNs verification solvers is helpful in verifying the performance of developed algorithms. The International Verification of Neural Netorks Competition (VNNCOMP) is a good metric for testing the performance of a neural network verification tool. Participating in the VNN-COMP is the most intuitive way to make a comparison among all the algorithms because VNN-COMP is an annual competition that intends to attract researchers working on testing and verifying neural networks. VNN-COMP includes all the state-of-art testing and verification techniques, so it is a success marker to validate the performance of algorithms. The input of the neural network verification or testing problem consists of two parts: a neural network and its specification. The specification describes the assertions on the input and output of the DNN. The verification or testing algorithms can determine whether there exists one satisfying assignment such that the input and output constraints are satisfied. For example, giving a neural network M and specification (M(x) $<$ 5 AND x $>$ 0), the verification or testing algorithms aim to find an input x, such that (M(x) $<$ 5 AND x $>$ 0) is satisfied. Additionally, the DNN verification or testing algorithms can also be used to find an adversarial input for the DNN by setting the desired constraint.

As CGDTest is a scalable and expressible testing tool for evaluating the robustness of DNNs, we want to understand its potential impact on the field of neural network verification. So, participating in VNN-COMP can be a good metric for evaluating CGDTest. To ensure that CGDTest meets the VNN-COMP standards, we have developed and assessed the algorithm, making it a flexible tool for various applications. The evaluation will be based on the instances from VNN-COMP, such as ACAS-Xu which are machine learning models for air-to-air collision avoidance systems. Below is a list of some contributions I have made to this project:

Contributions:

1. **Adapting CGDTest to Meet VNN-COMP Standards**. I have participated in the implementation phase of CGDTest. The overall goal of this project is not only to implement the approach but also to make the approach more flexible in real-world applications. The implementation steps were designed to ensure that CGDTest has high accuracy and a low PAR-2 score. The PAR-2 score is the performance metric for the VNN-COMP, and the PAR-2 score of a solver on a benchmark is the wall-clock runtime if the benchmark is successfully solved; otherwise, it is twice the wall-clock runtime.

2. **Proposing ideas of enhancing CGDTest**. CGDTest has some limitations; for instance, it may fail to produce the desired result and return UNKNOWN for a verification problem. To address these issues, we propose several ideas aimed at enhancing CGDTest's accuracy when solving neural network verification problems.

Chapter 2

Background

2.1 Neural Networks

Neural Network (NN) is one of the most powerful techniques in the field of machine learning and has enabled significant advancements in various applications, such as natural language processing, fraud detection, and image recognition. NN algorithm is inspired by the structure and function of biological neurons in the human brain, and the neurons are the basic building blocks of an artificial neural networks (ANN) [1].

The neurons in an ANN consist of several key components, such as input from previous neurons, weights, and activation functions. Neurons are organized into layers, including input layers that receive data, hidden layers that perform the bulk of the processing, and output layers that produce the output score. All the layers are interconnected with weights, and the bias term plays a crucial role in fine-tuning the output of each neuron and enhancing the ANN's ability to learn data patterns.

ANNs are designed as computational models used for solving complex problems. And a therorm called Universal Approximation theorem states that a feedforward ANN with a single hidden layer containing a finite number of neurons can approximate any continuous data pattern:

Universal Approximation Theorem [12]: Let $C(X, Y)$ represent the set of continuous functions from X to Y. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Then σ is not polynomial if and only if for every $n \in \mathbb{N}, m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n, f \in C(K, \mathbb{R}^m), \epsilon > 0$ there exist $k \in \mathbb{N}, A \in \mathbb{R}^{k*n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{m*k}$ such that

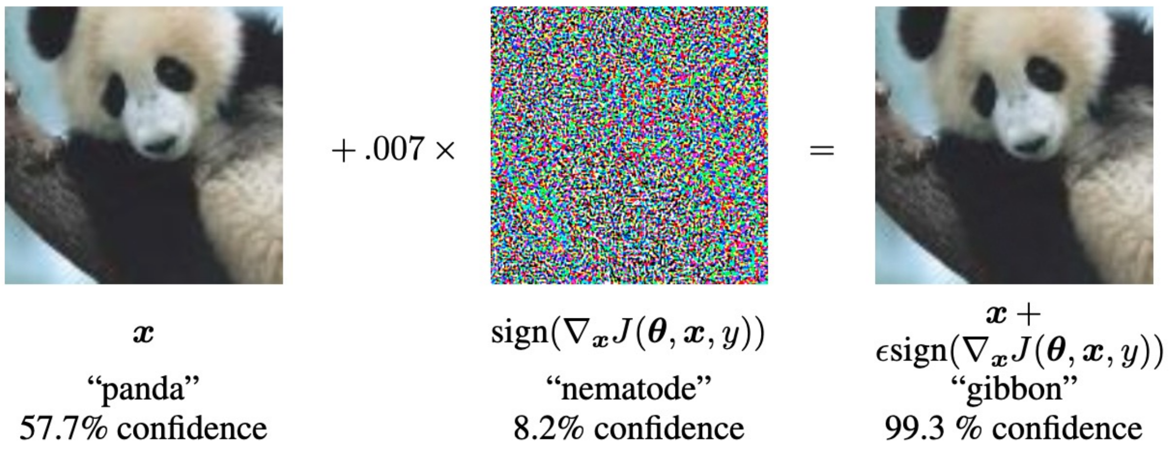
$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Figure 2.1: [8] Adversarial attacks can occur when a neural network is presented with a slightly perturbed input image. For example, a panda picture with a small perturbation added may be misclassified as a gibbon with high confidence by the neural network.

$$\sup_{\mathbf{X} \in k} ||f(\mathbf{X}) - g(\mathbf{X})|| < \epsilon$$

where

$$g(\mathbf{X}) = C \cdot (\sigma(A \cdot \mathbf{X} + \mathbf{b}))$$

Thus, an ANN can effectively capture intricate data patterns when the architecture is configured suitably.

2.2 Neural Network Verification

Concomitant with the success of DNNs, numerous concerns have emerged regarding their application in mission-critical tasks, such as self-driving cars, medical diagnosis, and finance. DNNs can produce undesirable behaviors under adversarial and model stealing attacks [26]. To address these issues, a dedicated research field called Neural Network Verification has emerged as one of the most effective approaches to validate user-specified properties of DNNs, ensuring their reliability and robustness.

When utilizing a neural network verification tool, users may want to validate a range of properties, such as robustness, correctness, fairness, monotonicity, and interpretability, to ensure the DNN's reliability, performance, and ethical use [35, 31, 24, 32, 29, 34, 33].

Robustness verification problems address the vulnerability of DNNs to small input perturbations that can lead to significant changes in the output. Robustness is a crucial property to verify in mission-critical applications, as adversarial examples can result in potentially harmful behavior [8]. Adversarial attacks employ these input perturbations to exploit DNN vulnerabilities. Refer to the provided figure 2.1 for an example of adversarial attacks. The primary objective of robustness verification is to guarantee that DNNs produce consistent and accurate outputs even with minor input perturbations.

Correctness verification problems address the accuracy of the DNN's output compared to the expected outcomes. A DNN is considered correct if it produces an output that aligns with the anticipated outcomes while adhering to a specified specification. For instance, consider verifying a DNN designed to diagnose lung cancer based on a CT image. In this scenario, the correctness problem focuses on ensuring that the DNN produces accurate and reliable diagnoses.

Fairness verification problems address the fairness of DNN's output based on specific sensitive attributes, such as gender, race, or age. A DNN is considered fair if it does not exhibit unintended biases related to sensitive attributes. For example, consider verifying a DNN designed to select candidates for employers; the fairness problem ensures that the DNN does not discriminate based on the applicants' race or gender.

Monotonicity verification problems address the specification that a DNN's output consistently increases or decreases concerning specific features. A DNN is considered to exhibit monotonicity if a change in the specified features' values results in a corresponding increase or decrease in the output. For instance, consider verifying a DNN used to predict house prices in a city; the monotonicity specification states that as the house size increases, the estimated house price should also increase.

Interpretability verification problems address the property of understanding the reasoning and decision-making processes of DNNs. DNNs are considered "black boxes" due to their complexity and the nonlinear relationships between input and output layers [17]. Therefore, having high interpretability can help build trust in a DNN's output, especially in mission-critical tasks such as medical, finance, and autonomous driving.

2.3 Neural Networks Verification Tools

In recent years, many neural network verification tools have been developed by leading AI researchers to check if DNNs satisfy specific specifications. However, not all tools employ the same techniques to address the verification problem. Some of them are based on satisfiability modulo theories (SMT) solvers, such as Marabou [15], while some of them encode the problem as a mixed-integer linear program (MILP) and use a MILP solver to find a solution, for example, Planet [6] and MIPVerify [27]. A tool known as the $\alpha, \beta-$Crown applies linear bound propagation and a branch-and-bound approach to solve the verification problem [35, 31, 24, 32, 29, 34, 33].

These tools can be broadly categorized into three major groups: testing, verification/analysis, and reachability. While testing tools do not give guarantees, both verification/analysis and reachability tools do verify networks and can give guarantees. However, I categorize verification/analysis and reachability tools separately due to the differences in their capabilities as explained below.

Testing methods search for inputs that exhibit model vulnerability to a given attack or property. Testing tools often use some form of gradient descent to search efficiently. Examples of these tools include the Fast Gradient Sign Method (FGSM) [15], the Carlini & Wagner (C & W) attack [5], and the Brendel & Bethge attack [4]. These attacks are efficient, extremely generalizable, and require nothing more than access to the model-under-test (and possibly an input to modify). However, they come at the expense of both expressibility and guarantees. Other standard testing methods include input fuzzing and genetic algorithms that attempt to do the same. Examples include DLFuzz [10] which utilizes neuron coverage statistics to guide search in the direction of greater neuron coverage to find adversarial examples. DLFuzz has difficulty scaling to industrial-sized networks and is limited to searching for adversarial examples.

Verification/Analysis tools encode the model-under-test symbolically and represent properties as formulas or constraints, with the aim of providing a certificate of guarantee (i.e., no violations of properties) or producing a counterexample. Several tools such as the ones by MIPVerify [27] encode DNNs and properties as MILP problems and use corresponding solvers to prove or disprove the existence of property-violating examples. They often restrict DNN activation functions and architectures to be piecewise-linear DNNs. On the other hand, tools like ReluPlex[13] and Marabou[15] are simplex-based methods that can handle arbitrary piecewise linear activation functions.

Finally, reachability tools define an allowable input region and propagate that region through the network, resulting in an output domain that encapsulates any possible reachable output given one of the input values. Via this method, one can verify robustness by limiting the input region to a shape around an example (such as a perturbation bound) and verifying that the reachable outputs do not include incorrect classifications. Examples of such tools include ERAN [23] and NNV [28]. Reachability tools can indeed verify networks and scale better than standard verification techniques.

Currently, all the three groups checking DNNs vulnerability are powerful in different contexts. Testing tools can scale to large DNN models, and reachability tools can verify various types of properties and are somewhat scalable. Verification tools are able to test for many kinds of properties. However, there are some limitations. Firstly, testing tools can only search for a limited number of vulnerabilities. Secondly, since the reachability methods verify robustness by limiting the input region to a region where the reachable outputs do not include incorrect classification, they require the vulnerabilities to be defined using geometric regions and can over-approximate their solution. Although the reachability tools have scalability issues, they are powerful approaches to verifying neural networks in some contexts (e.g., acas-xu). Thirdly, the verification tools may not be able to scale to large models.

A neural network verification tool typically requires two inputs: the DNN itself and the user-specified constraints relevant to the specification that needs to verify. And it will result in three possible outputs:

- **Satisfiable (SAT)**: A satisfiable assignment exists that meets the specified constraints.
- **Unsatisfiable (UNSAT)**: No assignment exists that satisfies the specified constraints.
- **UNKNOWN**: The tool fails to solve the problem.

2.4 Formulation of Robustness Verification

One of the most popular challenges in robustness verification is verifying the adversarial robustness of DNNs. Adversarial robustness problems address the property of DNNs maintaining stable performance, meaning that minor input perturbations will not result in significant spikes in the DNN's output[14]. Since DNNs are trained using a finite set of training data, they may not capture adversarial input during the training process. As a result, verifying their performance in such cases is essential.

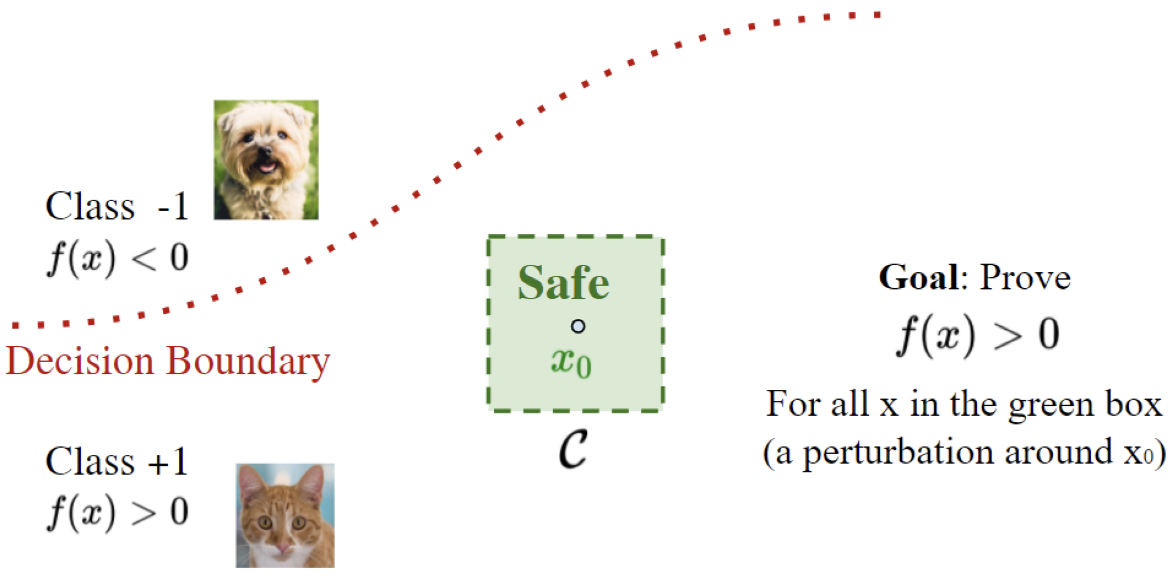

Figure 2.2: [35, 31, 24, 32, 29, 34, 33] The area above the decision boundary corresponds to a negative predictive score (i.e., a dog label), while the area below the decision boundary represents a positive predictive score (i.e., a cat label). The specification is verified by ensuring that the perturbation box remains below the decision boundary.

Definition 1[14] A DNN is $\delta - locally - robust$ at point X_0 if and only if

$$\forall X,\ ||X - X_0|| \leq \delta \ \rightarrow \ DNN(X) = DNN(X_0)$$

This definition states that for all input X in close proximity to X_0 (i.e., the neighbors of X_0), the DNN should result in the same outputs for both X and X_0. The value of δ refers to the "distance" between X and X_0; larger values of δ indicate better robustness. For instance, consider a classification neural network for dogs and cats. The network produces a predictive score for an input image; when the score is greater than zero, the network labels the image as a cat; and otherwise a dog. Suppose we want to verify the robustness of this network when given some perturbations to an image of a cat. The problem should be formulated as follows:

Example 1 Suppose an input X_0 and a perturbation set C that arround X_0, we need to verify:

$$DNN(X) > 0,\ \forall X \in C$$

This example states a specification that verifies the DNN can still correctly label the image as a cat when adding all the possible perturbations into the image. Figure 2.2 visualizes the example.

During the robustness verification process using a neural network verification tool, user-specified constraints are interpreted as the definitions of counterexamples for the property being verified. For instance, when verifying a binary neural network to ensure that it produces consistent results for an input and its perturbed variations, the user-specified constraints should be defined as counterexamples. The objective is to find a perturbation input that falsifies the result; if such perturbation input exists, the tool will output SAT, which means there is a counter-example that disproves the specification. Consequently, a property is considered "correct" if the specification is shown to be UNSAT, and a property is shown to be violated if a counterexample is found [21].

To use a neural network verification tool, users need to specify the constraints for the tool. The constraint file normally includes three parts: declaration of the input/output variables, input constraints, and output constraints. The input and output constraints should be represented as box constraints (i.e., variables bounded by upper and lower bounds). For instance, when setting up constraints for a binary classifier, the input constraints should be the defined perturbation set, and the output constraints should be the defined counter-example that falsifies the desired property.

Verifying local robustness can only demonstrate that a DNN is robust under a single specification. In this case, a verified DNN does not necessarily mean it won't be attacked. For example, a DNN may be robust in classifying a perturbed cat image but might not be robust in classifying a perturbed dog image. We need to establish different specifications to verify a DNN at a higher level.

2.5 VNNCOMP

The International Verification of Neural Networks Competition (VNN-COMP) is a platform that draws researchers in the field of neural network verification[21]. VNN-COMP is a highly recognized competition frequently used as a benchmark for testing and verifying algorithms within the community. Participating in the competition is an intuitive approach to comparing the performance of a developed tool with state-of-the-art tools.

Since a neural network verification tool typically takes a DNN and specified constraints as input, VNN-COMP uses a unified format for NNs called ONNX [3] and a unified format

Table 2.1: Overview of VNN-COMP benchmarks. [21]

Category	Benchmark	Application	Network Types
Complex	Carvana UNet	Image Segmentation	Complex UNet
	NN4Sys	Dataset Indexing& Cardinality Prediction	Complex (ReLU + Sigmoid)
CNN&ResNet	Cifar Bias Field	Image Classification	Conv. + ReLU
	Collins RUL CNN	Condition Based Maintenance	Conv. + ReLU
	Large ResNet	Image Classification	ResNet (Conv. + ReLU)
	Oval21	Image Classification	Conv. + ReLU
	SRI ResNet A/B	Image Classification	ResNet (Conv. + ReLU)
	VGGNet16	Image Classification	Conv. + ReLU + MaxPool
FC	MNIST FC	Image Classification	FC. + ReLU
	Reach Prob Density	Probability density estimation	FC. + ReLU
	RL Benchmarks	Reinforcement Learning	FC. + ReLU
	TLL Verifiy Bench	Two-Level Lattice NN	FC. + ReLU

for specified constraints called VNN-LIB [9]. In the context of VNN-COMP, a benchmark represents a combination of ONNX and VNN-LIB.

The final score of a participation tool is the sum of all benchmark scores, and each benchmark score is the number of points achieved based on the Instance Score:

- Property Proven: 10 points;
- counterexample found: 10 points;
- Incorrect result: -100 points;

and Time Bonus:

- The fastest tool for each solved instance: 2 points;
- The second fastest tool: 1 points

To assess the performance of neural network verification tools, VNN-COMP employs a variety of real-life benchmarks. You can find a detailed information of these benchmarks in the Table 2.1.

2.6 Converting User-specified Constraints to Loss

In the context of VNN-COMP, the constraint language is composed of Boolean combinations of term comparisons, where each term either represents a constant or a differentiable function defined across the inputs, intermediate layers, and outputs of the DNNs. To convert the user-sepecified constraints into differentiable loss functions, some designed mappings need to be performed by following the approach introduced in Probabilistic Soft Logic [16]. These mappings utilize the *Lukasiewicz t-norm* and its corresponding *co-norm* [22]. The mappings are designed to be precise at extreme values (i.e., "completely true" or "completely false"); when a variable assignment results in a constraint loss of zero, that assignment is considered to be the satisfiable assignment for the specified constraints.

The transformation rules for boolean combination of constraints are listed as follows:

- $L(\phi^{'} \wedge \phi^{''}) := L(\phi^{'}) + L(\phi^{''})$
- $L(\phi^{'} \vee \phi^{''}) := L(\phi^{'}) * L(\phi^{''})$

The transformation rules for comparison terms as follows:

- $L(t \leq t^{'}) := max(t - t^{'}, 0)$
- $L(t \neq t^{'}) := \epsilon * [t = t^{'}]$
- $L(t = t^{'}) := L(t \leq t^{'} \wedge t^{'} \leq t)$
- $L(t < t^{'}) := L(t \leq t^{'} \wedge t^{'} \neq t)$

If a constraint contains negation, the rules is listed below:

- $L(\neg(t = t^{'})) := L(t \neq t^{'})$
- $L(\neg(t \leq t^{'})) := L(t^{'} < t)$
- $L(\neg(t \neq t^{'})) := L(t = t^{'})$
- $L(\neg(t < t^{'})) := L(t^{'} \leq t)$
- $L(\neg(\phi^{'} \wedge \phi^{''})) := L(\neg\phi^{'} \vee \neg\phi^{''})$
- $L(\neg(\phi^{'} \vee \phi^{''})) := L(\neg\phi^{'} \wedge \neg\phi^{''})$

The above rules follow a list of properties:

- The converted loss value is non-negative
- Larger the loss value, greater the number of violations
- A loss value of zero represents all the constraints are satisfied
- Sub-differentiable (i.e., differentiable almost everywhere)

2.7 Constraint Gradient Descent

Algorithm 1 The CGD Algorithm

```
Require: DNN $M$, Label $y$, Weight $\alpha$, Constraint $\varphi$
Ensure: DNN Input $x$ s.t. $(argmax(M(x)) = y) \wedge \varphi(x)$
 1: $\varphi' =$ ConstraintLoss($\varphi$)              //Convert Constraint to Loss
 2: initialize $x$
 3: while True do
 4:     if $(argmax(M(x)) = y)$ AND $\varphi'(x) = 0$ then
 5:         break                                    //Stopping criteria
 6:     end if
 7:     $l_1 = \varphi'(x)$                              //compute constraint loss
 8:     $l_2 =$ Loss($M(x), y$)                          //compute misclassification loss
 9:     $\nabla x_1 = \frac{\partial l_1}{\partial x}$                                //gradient value
10:     $\nabla x_2 = \frac{\partial l_2}{\partial x}$                                //gradient value
11:     $x = x - \alpha * (\nabla x_1 + \nabla x_2)$       //Alter x in a direction of decreased loss
12: end while
13: return $x$
```

The Constrained Gradient Descent (CGD) algorithm was introduced to tackle challenges in neural network verification[22]. CGD improves gradient descent (GD) techniques by integrating the capability to specify logical constraints for evaluating DNNs against specific properties. These properties are not generally supported by the most popular testing, verification, or reachability tools. CGD utilizes the converted constraint loss and minimizes the constraint loss values using a GD method (Please refer to Algorithm 1)[22].

First, CGD maps the specified constraint φ into a constraint loss function $\varphi^{'}$ (line 1). The constraint loss function $\varphi^{'}$ has the property that it is equal to 0 if and only if all the constraints are satisfied. An input x to the DNN is randomly initialized as a starting point (line 2). The process then enters a loop that terminates when the constraints are satisfied (i.e., the constraint loss equals 0) (lines 4-6). If x meets the criteria in line 4, the loop will break and return x as the counterexample for the specification (line 13). If x does not meet the criteria, both constraint loss (line 7) and classification loss (line 8) are calculated, followed by computing the gradient values of both losses concerning x (lines 9-10). Finally, the gradient values are combined to adjust x slightly using a small α, moving in the direction of decreased loss (line 11). The loop ensures the process continues until either x satisfies the criteria or a timeout is reached.

The gradient values for constraint gradient loss and misclassification loss (Lines 9-10) play roles in the CGD algorithm to find global minima. If a global minimum for both losses does not exist, there is no possible solution for the verification problem. Moreover, CGD may not always be able to find a solution even if a solution exists[22]. This represents the most significant limitation of CGDTest, which arises because the losses may be non-convex, causing the GD algorithm to get stuck in a local minimum potentially.

2.8 CGDTest

The concept of CGDTest[22], proposed by Vineel Nagisetty, introduces an approach that enables users to integrate logical constraints within the loss function of a gradient descent (GD) algorithm (Refer Section 2.6 for more details), and the objective to generate deep neural network (DNN) inputs that satisfy user-defined specifications. CGDTest effectively addresses the concerns of model developers:

- **Scalability**: A tool that scales to industrial size DNNs
- **Generality**: A tool that can be applied to most of the architecture of DNNs
- **Expressibility**: Users can specify specification they want.

CGDTest takes ONNX (i.e., DNN) and VNN-LIB (i.e., user-specified constraints) as inputs. It utilizes a pre-processing step to transform these user-specified constraints into a constraint loss function while converting the ONNX into a PyTorch model.Then the tool proceeds to the main process detailed in Section 2.7. Please refer to the architecture diagram of CGDTest in Figure 2.3.

A proposed work called DL2[7] also converts user-specified constraints into differentiable loss as proposed for CGDTest; however, DL2 has benefits for training neural networks. We

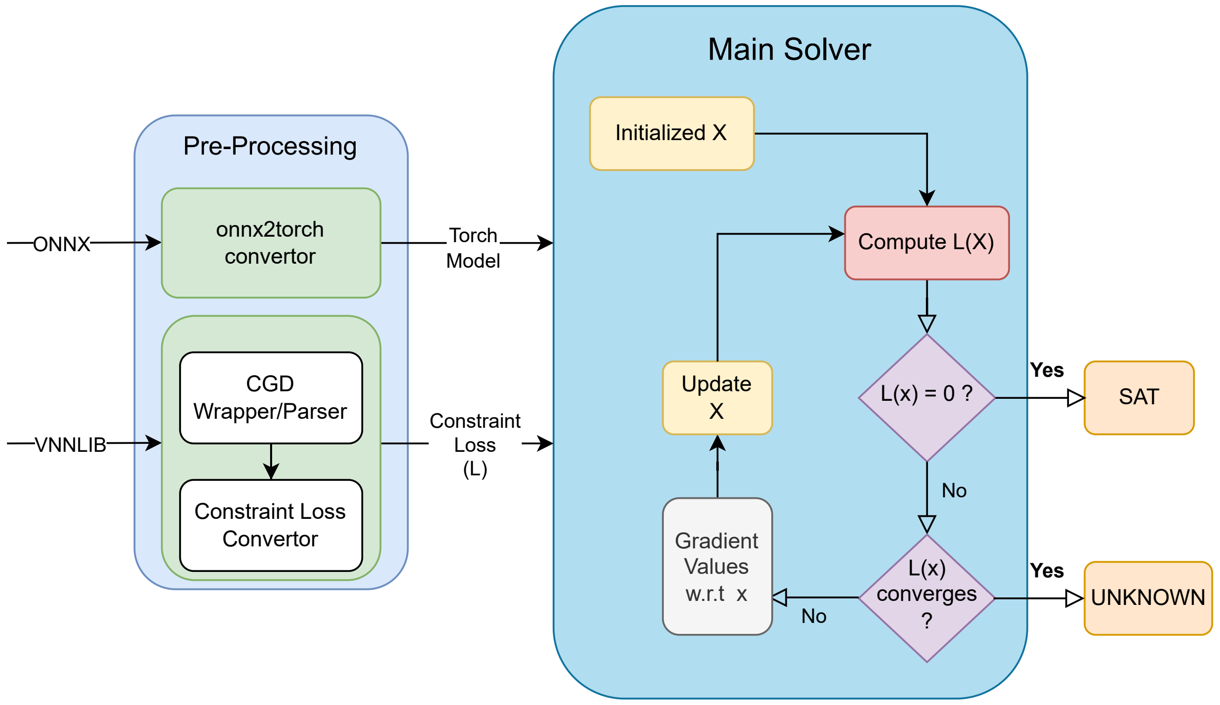

Figure 2.3: The architecture diagram of CGDTest. CGDTest first converts the ONNX and VNN-LIB into torch model and constraint loss function, then enters the main solver that describes in section 2.7

use the idea in the verification/testing of neural networks. CGDTest also seeks different gradient descent solvers (e.g., PGD and GA solvers); we will investigate more solvers and find more potential combinations of solvers and optimization techniques.

CGDTest is designed to test whether a neural network satisfies a given constraint. We assume that this tool is used by the model developer or a model validator with white-box access to the model. The constraints can encapsulate various types of properties, such as vulnerability to fairness, bias and robustness [20, 19]. Note that, if an adversary uses the CGDTest tool, he/she would require white-box access to the model. However, this can be circumvented by model-stealing attacks, where an adversary is able to create a local copy of a target model using only black-box access, and adversarially attack the local copy using white-box methods (such as CGDTest) and use those to attack the target model successfully.

Chapter 3

CGDTest Implementation

I joined the team responsible for implementing the CGDTest and ensuring its compliance with VNN-COMP standards. The goal of the implementation was to enhance the flexibility of CGDTest in both VNN-COMP and real-world applications. We implemented CGDTest tool using Pytorch. We implemented CGDTest using Pytorch 1.110 in Python 3.8. In this section, I include my contributions to the implementation.

3.1 Pre-processing in CGDTest

We incorporated a pre-processing step into CGDTest (please refer to Figure 1). The pre-processing engine plays a crucial role in parsing input DNNs and input constraints (VNNLIB).

Parsing ONNX. We employed a Python tool called *onnx2torch* to parse ONNX models into PyTorch models. However, we encountered some bugs in this tool when parsing Convolutional Neural Networks (CNNs). After debugging and refining the tool, we ensured it could parse all types of DNNs in the VNN-COMP.

Parsing VNN-LIB. VNN-LIB is the standard for neural network verification benchmarks. This standard is built upon the ONNX model description format and the Satisfiability Modulo Theories Library (SMT-LIB) format for the property specification [9]. To parse the VNN-LIB constraint format, we incorporated a wrapper that encapsulates all the constraints and translates them into a language that the CGDTest parser can understand. Furthermore, we included a parser that can convert CGDTest's constraints into a constraint loss function.

3.2 VNN-COMP Compliant

The organizers of VNN-COMP evaluated all participating tools using Amazon Web Services (AWS). We ran our tool on the permitted instance, the m5.16xlarge, with 64 vCPUs and 256 GB RAM. We developed scripts to enable CGDTest evaluation on AWS and ensured that CGDTest could run all the benchmarks in the competition. To ensure that CGDTest can run successfully on AWS, we conducted numerous tests on both GPU and CPU using Compute Canada. CGDTest is capable of verifying DNNs on either a CPU or GPU by utilizing a flag in the command line.

Chapter 4

Experiments Result

CGDTest has been evaluated by participating in VNN-COMP. In this section, the VNN-COMP results are presented [21]. The benchmarks used for evaluation are described in Section 2.5. According to VNN-COMP rules [21], participants are not required to run all the provided benchmarks. On the other hand, they can choose a minimum of two benchmarks to evaluate their tool. All tools are scored according to the rules described in Section 2.5, and their ranking is based on the total scores.

In this report, cactus plots are utilized to compare the performance of all participants. The X-axis represents the number of benchmarks the tools solved, while the Y-axis indicates the Wallclock Runtime required for the tools to solve a problem. For each solver, we sorted the runtime in ascending order for each solver. In a cactus plot, lower Wallclock Runtimes signify a faster solver for a verification problem. Therefore, the solver is better if the line is closer to the bottom right.

This section will first introduce the overall scores achieved by CGDTest when running all the provided benchmarks. Secondly, it will discuss the results of CGDTest for the benchmarks in which CGDTest stood second place, including *cifar100-tinyimagenet-resnet*, *cifar-biasfield*, and *sri-resnet-a*.

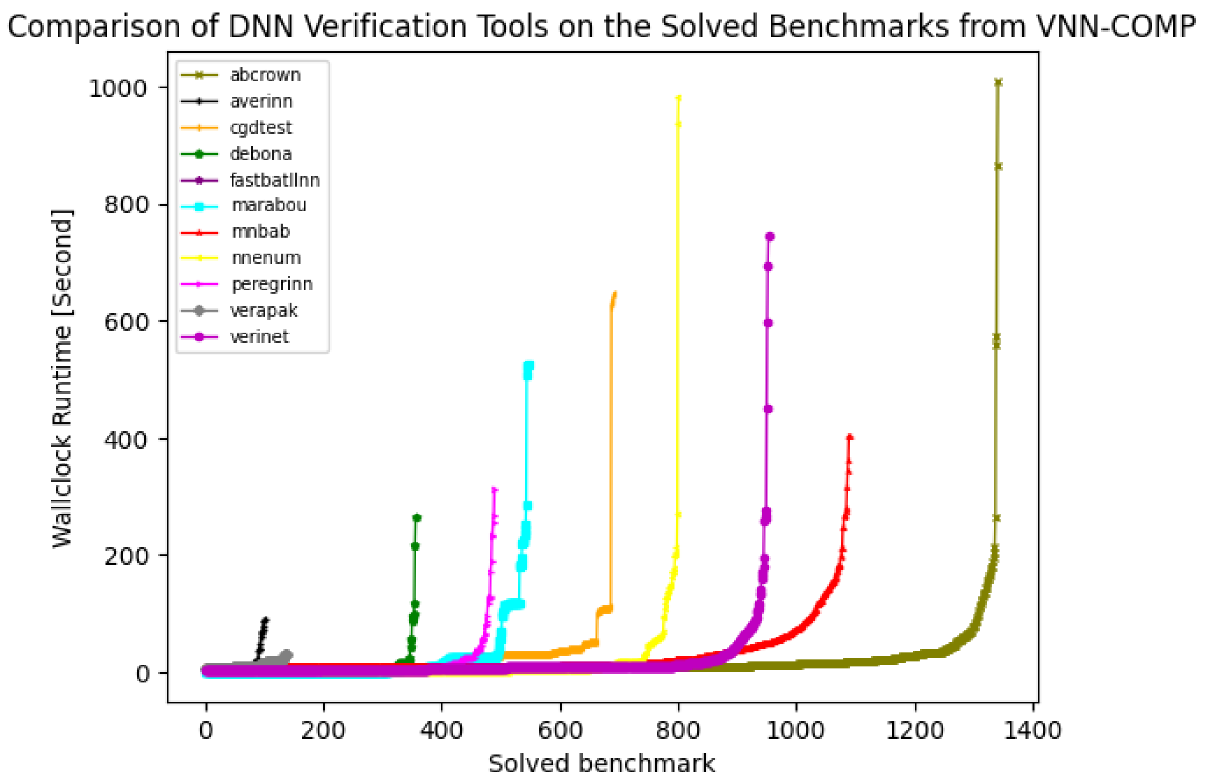

Figure 4.1: Catus plots for the tools in VNN-COMP

4.1 Total Score

CGDTest stood fifth overall among the 11 participating tools (Please see Table 4.1 for the rank). The scoring rule is detailed in section 2.5.

Figure 4.1 presents a cactus plot that compares the performance among all participants. CGDTest is depicted by an orange line. The plot also proves that CGDTest ranked fifth overall. It demonstrates that CGDTest successfully solved approximately 700 benchmarks, while the winner (i.e., $\alpha, \beta-$ Crown) solved more than 1300 benchmarks.

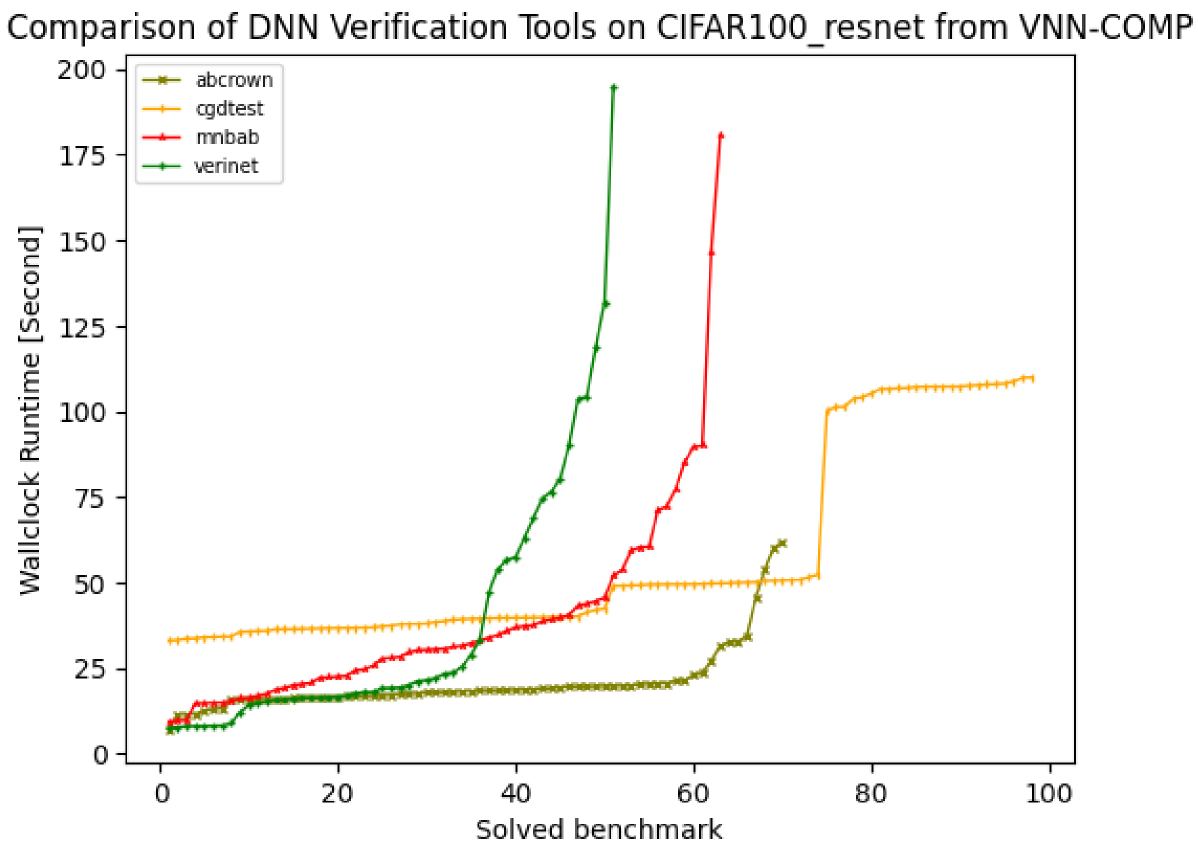

Figure 4.2: Catus plots for the tools in solving *cifar100-tinyimagenet-resnet*

4.2 Results on *cifar100-tinyimagenet-resnet*

CGDTest ranked second in solving the benchmark called *cifar100-tinyimagenet-resnet*. By referring to the cactus plot in Figure 4.2, CGDTest successfully solved the most benchmarks when addressing *cifar100-tinyimagenet-resnet*. However, as referenced in Table 4.2, CGDTest placed second due to producing three incorrect results, which led to a score penalty.

Please note that only four tools are included in this benchmark set because the other tools either do not support this benchmark or fail to solve the problems.

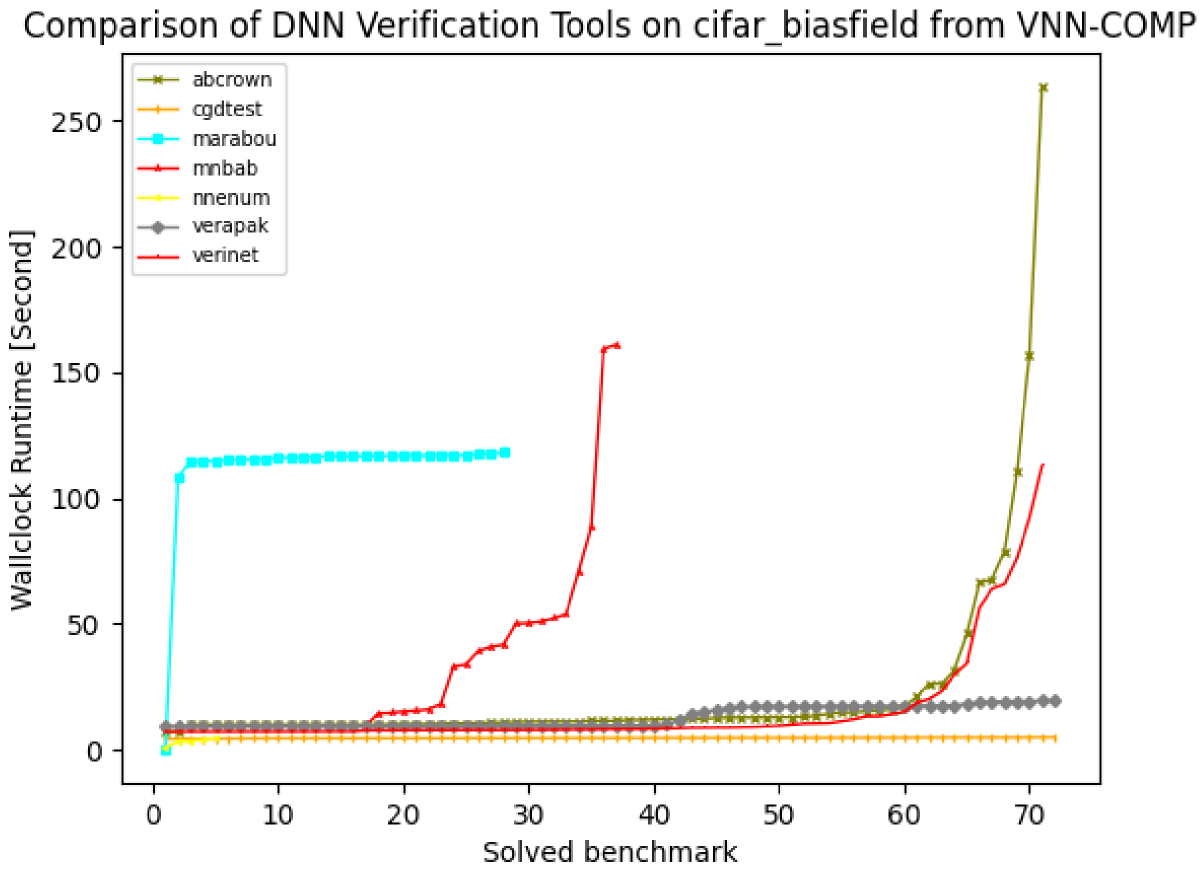

Figure 4.3: Catus plots for the tools in solving *cifar-biasfield*

4.3 Results on *cifar-biasfield*

CGDTest ranked second in solving the benchmark called *cifar-biasfield*. By referring to the cactus plot in Figure 4.3, CGDTest successfully solved the most benchmarks when addressing *cifar-biasfield*. Furthermore, CGDTest is the most efficient tool in this benchmark set, as its line is positioned at the bottom right. However, as referenced in Table 4.3, CGDTest placed second due to producing one incorrect result, which led to a score penalty.

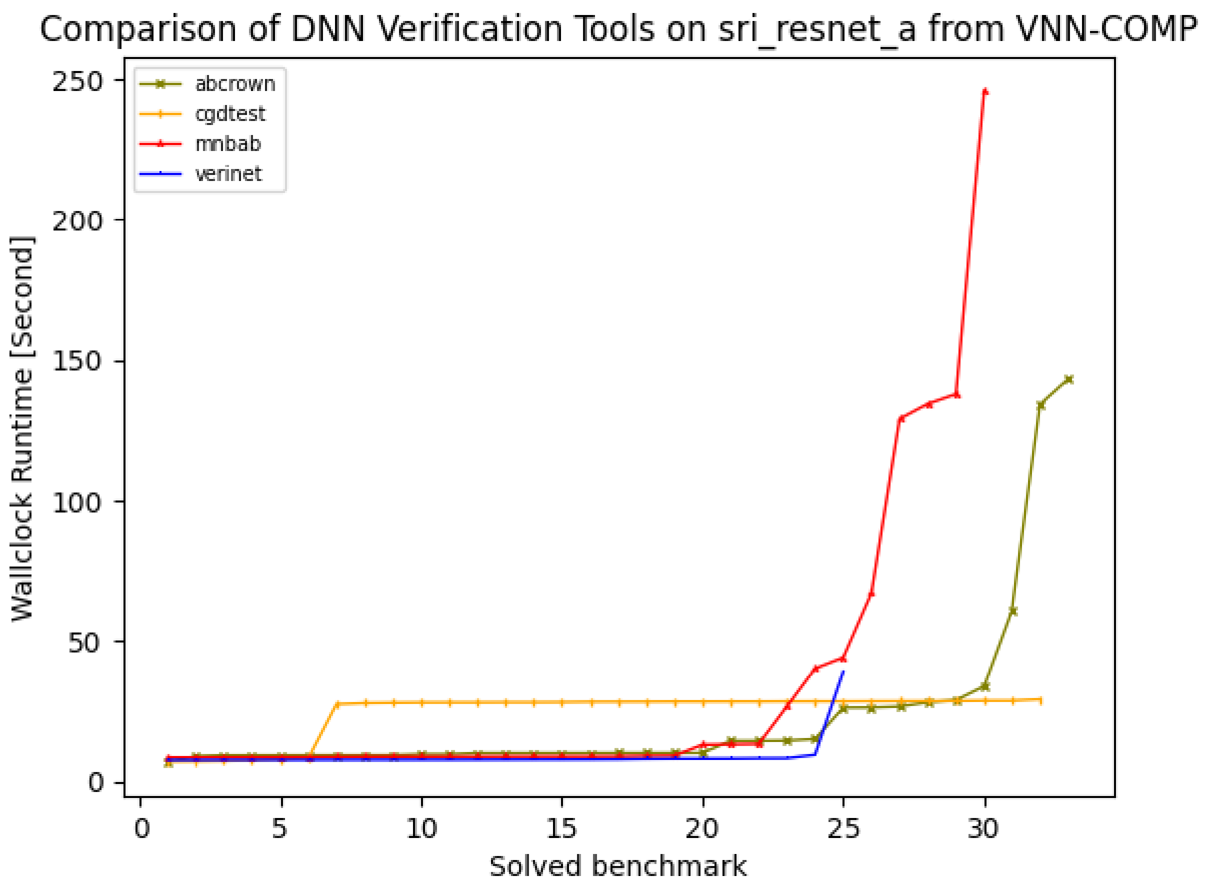

Figure 4.4: Catus plots for the tools in solving *sri-resnet-a*

4.4 Results on *sri-resnet-a*

CGDTest ranked second in solving the benchmark called *sri-resnet-a*. By referring to the cactus plot in Figure 4.4, CGDTest successfully solved the most benchmarks when addressing *sri-resnet-a*. Furthermore, CGDTest is the most efficient tool in this benchmark set, as its line is positioned at the bottom right. As indicated in Table 4.4, CGDTest's score is very close to the first place, and CGDTest has the highest number of verified benchmarks in this benchmark set.

Table 4.1: VNN-COMP Results

Rank	Tool	Score
1	$\alpha, \beta-$ Crown	1274.9
2	MN-BAB	1017.5
3	Verinet	892.4
4	Nnenum	534.0
5	**CGDTest**	408.4
6	Peregrinn	399.0
7	Marabou	372.2
8	Debona	222.9
9	Fastbatlnn	100.0
10	Verapak	98.2
11	Averinn	29.1

Table 4.2: Benchmark *cifar100-tinyimagenet-resnet*

Rank	Tool	Verified	Falsified	Fastest	Penalty	Score
1	$\alpha, \beta-$ Crown	69	0	56	0	813
2	**CGDTest**	95	0	28	3	725
3	MN-BAB	60	3	10	0	673
4	Verinet	48	3	6	0	540

Table 4.3: Benchmark *cifar-biasfield*

Rank	Tool	Verified	Falsified	Fastest	Penalty	Score
1	$\alpha, \beta-$ Crown	69	1	1	0	736
2	**CGDTest**	71	0	55	1	731
3	Verinet	69	1	0	0	721
4	Verapak	71	0	0	1	635
5	MN-BAB	36	1	17	0	404
6	Marabou	27	0	0	0	270
7	NNenum	4	0	0	0	43

Table 4.4: Benchmark *sri-resnet-a*

Rank	Tool	Verified	Falsified	Fastest	Penalty	Score
1	$\alpha, \beta-$ Crown	20	12	7	0	356
2	**CGDTest**	26	6	14	0	352
3	MN-BAB	18	12	19	0	343
4	Verinet	12	12	4	0	248

Chapter 5

Limitations, Future Work and Conclusion

5.1 Limitations

A current limitation of CGDTest is that it is an incomplete tool. An incomplete solver is an algorithm that may not always produce a satisfiable output for constraints, even if one does exist. However, incomplete solvers are particularly useful for large-size models, since they are more efficient than complete solvers. In the context of CGDTest, it can only guarantee that the SAT output is indeed satisfiable, but it cannot guarantee a UNSAT output in the current phase. This is because GD methods may not always reach a global minimum, as they sometimes get stuck at local minima. Therefore, we cannot conclude that a problem is UNSAT when the minimum is greater than zero, as there is a probability that a global minimum of 0 exists. However, if a DNN's input results in a constraint loss of zero, then CGDTest can guarantee SAT.

CGDTest was the only incomplete tool participating in the VNN-COMP. Since the VNN-COMP does not have a category for incomplete tools, we employed an ”estimate” approach to enable CGDTest to produce UNSAT results. We dynamically define an ϵ value for each benchmark, and if the loss minimum found by the GD method is greater than ϵ, it outputs UNSAT. If the loss minimum found by the GD method is less than ϵ but not equal to zero, it outputs UNKNOWN. The limitation of this approach is that it may potentially produce incorrect results, which could result in penalties from the competition.

We proposed ideas to boost CGDTest to a higher level to address the above limitations (Section 5.2, 5.3 and 5.4).

5.2 Dividing Input Search Domain

To prevent the GD method from moving in the direction of local minima, we propose the idea of dividing the input search domain into a set of sub-domains. This idea is inspired by the divide-and-conquer algorithm [25]. Since the GD method in CGDTest aims to search for a DNN input that results in the minimum loss, dividing the search domain into a set of sub-domains allows each minimum (including global and local minima) to be placed in different sub-domains. We can then apply the GD method to each sub-domain to find all the minima for all the sub-domains. The smallest minimum from all the sub-domains will be treated as the global minimum. We believe that this approach will help avoid the GD method from getting stuck at a local minimum.

We already have a prototype for this idea. The current splitting heuristic is to choose the input variable in a DNN with the largest range and then split the box constraints for this variable into a set of box sub-constraints (where the combined sub-constraints should equal the original one). This splitting heuristic is inspired by a paper on parallelization techniques for neural network verification[30]. However, in our experiments, this prototype does not significantly improve the solving of verification problems. As a result, further tuning or new splitting heuristics may need to be applied to enhance this approach.

5.3 Utilizing MILP to Break Out of Local Minima

Another idea is to help the GD method break out of local minima. In the current version of CGDTest, we utilize random noise to prevent the GD method from getting stuck at local minima. We propose a new approach to help the GD method in breaking out of local minima, which involves using a MILP solver, such as Gurobi[11]. When the GD method is detected that stucking at a minimum during the process, we can encode the problem into a MILP using the current search input. The objective function is to minimize the constraint loss function. We then use Gurobi to find a satisfiable assignment that results in a lower loss. If Gurobi finds a satisfiable DNN input that leads to a lower loss, we can apply the GD method again, using that DNN input found by Gurobi as a starting point. The challenge of this idea is that we need to encode and include a large Torch model within a Gurobi model, which may slow down Gurobi's performance.

5.4 Integrate CGDTest into an Algorithm Selection

We are currently working on integrating CGDTest into an algorithm selection tool for neural network verification called Goose. Goose is another neural network verification tool from our research team, which is led by Mr. Joseph Scott. Goose aims to select the optimal DNN verification tool for a given DNN, which is decided by the Algorithm Selection Engine (ASE). Goose is a complete verification tool, and the ASE includes state-of-the-art verification tools. In experiments using the VNN-COMP benchmarks, Goose achieved a higher score than the VNN-COMP winner. Based on the results of CGDTest from VNN-COMP, we observed that CGDTest was the fastest at solving some of the benchmarks in VNN-COMP. We believe integrating CGDTest into Goose's ASE can improve Goose's performance.

5.5 Conclusion

To enhance the security of Deep Neural Networks (DNNs) in mission-critical applications, we implemented a novel testing tool for neural network verification, called CGDTest. CGDTest takes user-specified constraints and transforms them into a differentiable loss function named the constraint loss function. By employing the gradient descent method, CGDTest could find a DNN input that minimizes the loss function. When an input results in a loss value of zero, it signifies a satisfiable assignment for the constraints.

CGDTest was submitted to VNN-COMP, where it stood fifth place out of 11 participants. Furthermore, it achieved second place in several benchmarks within the competition. However, CGDTest has its limitations, as it is an incomplete solver and may not always produce a satisfiable assignment for the constraints. Several ideas for enhancing CGDTest have been proposed, and we believe that these approaches will elevate CGDTest's performance to a higher level.

References

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2018.

[2] Luis E. Alvarez, Ian Jessen, Michael P. Owen, Joshua Silbermann, and Paul Wood. Acas sxu: Robust decentralized detect and avoid for small unmanned aircraft systems. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–9, 2019.

[3] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[4] Wieland Brendel, Jonas Rauber, Matthias Kümmerer, Ivan Ustyuzhaninov, and Matthias Bethge. Accurate, reliable and fast robustness evaluation. *Advances in neural information processing systems*, 32, 2019.

[5] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

[6] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks, 2017.

[7] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. Dl2: training and querying neural networks with logic. In *International Conference on Machine Learning*, pages 1931–1941. PMLR, 2019.

[8] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[9] Dario Guidotti, Stefano Demarchi, Armando Tacchella, and Luca Pulina. The Verification of Neural Networks Library (VNN-LIB). `www.vnnlib.org`, 2019.

[10] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, oct 2018.

[11] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[12] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[13] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 97–117. Springer, 2017.

[14] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Towards proving the adversarial robustness of deep neural networks. In Lukas Bulwahn, Maryam Kamali, and Sven Linker, editors, *Proceedings First Workshop on Formal Verification of Autonomous Vehicles, FVAV@iFM 2017, Turin, Italy, 19th September 2017*, volume 257 of *EPTCS*, pages 19–26, 2017.

[15] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pages 443–452. Springer, 2019.

[16] Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *NIPS 2012*, 2012.

[17] Zachary C. Lipton. The mythos of model interpretability, 2017.

[18] Guido Manfredi and Yannick Jestin. An introduction to acas xu and the challenges ahead. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2016.

[19] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning, 2022.

[20] Andy Michel, Sumit Kumar Jha, and Rickard Ewetz. A survey on the vulnerability of deep neural networks against adversarial attacks. *Progress in Artificial Intelligence*, pages 1–11, 2022.

[21] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. The third international verification of neural networks competition (vnn-comp 2022): Summary and results, 2023.

[22] Vineel Nagisetty, Laura Graves, Guanting Pan, Piyush Jha, and Vijay Ganesh. Cgdtest: A constrained gradient descent algorithm for testing neural networks, 2023.

[23] Anian Ruoss, Maximilian Baader, Mislav Balunović, and Martin Vechev. Efficient certification of spatial robustness, 2021.

[24] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. A convex relaxation barrier to tight robustness verification of neural networks. *Advances in Neural Information Processing Systems*, 32:9835–9846, 2019.

[25] Douglas R Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[26] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S Yu, Lifang He, and Bo Li. Adversarial attack and defense on graph data: A survey. *arXiv preprint arXiv:1812.10528*, 2018.

[27] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming, 2019.

[28] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T Johnson. Nnv: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, pages 3–17. Springer, 2020.

[29] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *Advances in Neural Information Processing Systems*, 34, 2021.

[30] Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Ahmed Irfan, Kyle Julian, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. Parallelization techniques for verifying neural networks, 2020.

[31] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33, 2020.

[32] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. Fast and Complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In *International Conference on Learning Representations*, 2021.

[33] Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. General cutting planes for bound-propagation-based neural network verification. *Advances in Neural Information Processing Systems*, 2022.

[34] Huan Zhang, Shiqi Wang, Kaidi Xu, Yihan Wang, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. A branch and bound framework for stronger adversarial attacks of ReLU networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pages 26591–26604, 2022.

[35] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems*, 31:4939–4948, 2018.