From Counterexamples to Confidence: Advancing Neural Network Verification

Submitted in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Mohammad Afzal

Roll No. 194057001

Supervisors:

Prof. Ashutosh Gupta

Prof. S. Akshay

(Department of Computer Science & Engineering)

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

(2026)

Approval Sheet

This thesis entitled **From Counterexamples to Confidence: Advancing Neural Network Verification** by **Mohammad Afzal** is approved for the degree of **PhD**.

Examiners

Supervisor (s)

ASHUTOSH K. GUPTA

S. AKSHAY (Co-

Chairperson

Date : 19/Feb/2026

Place : IITB, Mumbai

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission.

I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

I declare that I have used the following AI tools and/or other similar tools:

Grammarly, Copilot, ChatGPT, and Perplexity

for the purpose of

gathering information on related works and copyediting

(Note: If required, the student may attach additional sheet)

Digital Signature
Mohammad Afzal (194057001)
19-Feb-26 10:29:52 PM

(Signature)

Mohammad Afzal

(Name of the student)

194057001

(Roll No.)

Date: *19-Feb-2026*

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

CERTIFICATE OF COURSE WORK

This is to certify that **Mohammad Afzal** (Roll No. 194057001) was admitted to the candidacy of Ph.D. degree on 05 Aug 2020, after successfully completing all the courses required for the Ph.D. programme. The details of the course work done are given below.

S.No	Course Code	Course Name	Credits
1	CS 621	Artificial Intelligence	6
2	CS 691	R & D Project	6
3	CS 766	Analysis of Concurrent Programs	6
4	CS 433	Automated Reasoning	6
5	CS 725	Foundations of Machine Learning	6
6	CS 765	Introduction to Blockchains, Crypto-currencies and Smart Contracts	6
7	CSS801	Seminar	4
8	HS 791	Communication Skills -I	PP
9	HS 792	Communication Skills -II	PP
10	GC 101	Gender in the workplace	PP
		Total Credits	**40**

IIT Bombay

Date: Dy. Registrar (Academic)

Dedicated to my beloved parents.

Acknowledgements

I would like to express my sincere gratitude to my advisors, Prof. Ashutosh Gupta and Prof. S. Akshay, for their continuous motivation and guidance throughout my PhD journey. Without their support and encouragement, this thesis would not have reached its final form.

I am also grateful to my Research Progress Committee members—Prof. Supratik Chakraborty, Prof. Krishna S., Prof. B. Srivathsan, and Dr Rahul Sharma—for their valuable feedback and constructive suggestions each year.

I also sincerely thank Prof. Deepak D'Souza and Prof. Priyanka Golia for thoroughly reviewing this thesis and for providing numerous detailed corrections and suggestions that greatly enhanced its quality.

My heartfelt thanks go to TCS Research for funding my PhD, and to my former and current research area heads, R. Venkatesh and Ravindra Metta, respectively, for providing me with the flexibility and support needed to pursue my PhD work alongside my professional responsibilities.

I would also like to thank my lab members at IIT Bombay and my colleagues at TCS Research, Pune, for their collaboration, encouragement, and the positive working environment they provided.

Finally, I owe my deepest gratitude to my parents, wife, daughter, and siblings, whose unwavering support and constant motivation have been a source of strength and inspiration throughout this journey.

Abstract

Given the pervasive use of neural networks in safety-critical systems, it is important to ensure that they are robust. Recent research has focused on the question of verifying whether networks do not alter their behavior under small perturbations in inputs. Verification of DNNs is an NP-complete problem, and its complexity grows exponentially with the number of non-linear activation units. To manage this, researchers often abstract activation units, but this leads to incomplete verification. To address this limitation, we proposed a method for systematically selecting activation units and removing their abstraction. We further extended this work to efficiently identify nonlinear activation units.

Next, we investigated the genuineness of bugs in neural network verification. We classified false positives/true positives for counterexamples and true negatives/false negatives for verified instances. Our study revealed that a significant number of counterexamples reported by state-of-the-art verifiers are false positives, caused by the limitations of existing robustness properties. To mitigate this, we proposed new properties that reduce the occurrence of false positives.

To make formal verification more intuitive and practical, we incorporated classification confidence into the verification process. Specifically, we bypass counterexamples with low confidence and focus on detecting high-confidence counterexamples. This required efficient abstraction of the softmax function, which computes confidence. Building on this, we proposed a general framework that transforms an arbitrary property into a set of layers and appends them to the underlying neural network. This framework enables verification of diverse properties without concern for their specific implementation or the verifier used.

We also explored the explainability of the machine learning systems in the context of reinforcement learning. Designing reward signals in reinforcement learning is often non-intuitive, difficult, and error-prone, hence, it can be advantageous to learn reward functions from expert demonstrations, which forms the basis of inverse reinforcement

learning (IRL). However, existing IRL approaches often lack explainability and produce opaque reward representations. To overcome this limitation, we proposed a method to learn reward functions while simultaneously providing explainability in the form of Linear Temporal Logic (LTL) formulas.

Contents

List of Figures

List of Tables

List of Abbreviations

DNN Deep Neural Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

CEGAR Counterexample-Guided Abstraction Refinement

MILP Mixed-Integer Linear Programming

IBP Interval Bound Propagation

SDP Semidefinite Programming

FP False Positives

TP True Positives

CONF Confidence

CNF Conjunctive Normal Form

DNF Disjunctive Normal Form

IRL Inverse Reinforcement Learning

NMRDP Non-Markovian Reward Decision Process

ILE Inverse Learning Error

LTL Linear Temporal Logic

Chapter 1

Introduction

Neural networks are being increasingly used in safety-critical systems such as autonomous vehicles, medical diagnosis, and speech recognition [1, 2, 3]. In autonomous vehicles, neural networks have become a core component in autonomous vehicle systems due to their ability to learn complex mappings directly from sensory data [1, 4]. For example, in the influential work [1], a convolutional neural network (CNN) was trained to map raw pixels from a front-facing camera directly to steering commands, enabling end-to-end learning for self-driving cars. Unlike traditional modular pipelines that separately perform lane detection, path planning, and control, this approach optimizes all steps jointly to improve overall performance. The trained model demonstrated robust driving behavior on highways, local roads, and in various weather conditions. This shows that neural networks can automatically learn meaningful road representations from minimal human supervision.

Neural networks have also shown remarkable success in the medical domain [2], where they are used to assist in disease diagnosis [5], medical imaging [6], and treatment planning [7].

It is important not only that such systems behave correctly in theory but also that they remain robust in practice. Unfortunately, even slight perturbations in the input can often cause neural networks to produce incorrect outputs. As shown in Figure 1.1, the seed image (Figure 1.1A) is correctly classified as “panda”, while a minor change in pixel values leads the network to misclassify it as “gibbon” with high confidence (Figure 1.1C). Such errors are difficult to identify, analyze, or debug, as neural networks consist of hundreds of thousands of non-linear nodes. This limitation poses a significant challenge to their deployment in safety-critical applications.

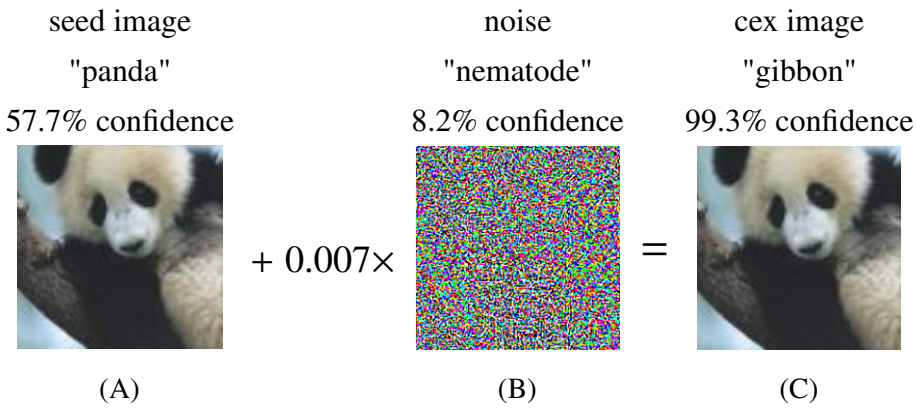


Figure 1.1: (A) shows the original (seed) image from the ImageNet [8] dataset. The GoogLeNet [9] network correctly classifies this image as "panda" with 57.7% confidence. (B) shows the noise pattern, which is classified as "nematode" with 8.2% confidence. Adding this noise, at a magnitude of 0.007, to the seed image produces the perturbed image in (C). Although visually indistinguishable from the original, the network misclassifies it as "gibbon" with 99.3% confidence. This figure is sourced from [10].

To check whether a network is robust against such imperceptible perturbations, researchers have developed various types of attacks. Some of the well-known ones include FGSM [10], PGD [11], Boundary [12], HopSkipJump [13], and DeepFool [14] attacks. The goal of these attacks is to generate adversarial examples, similar to the one shown in Figure 1.1C. However, these attack-based methods do not provide any guarantee on the absence of adversarial examples—if an attack fails to find one, it does not necessarily mean that such examples do not exist.

To address this problem, an entire line of research has emerged focusing on automatically proving (or disproving) the robustness of such networks. Since automatic verification of neural networks is NP-Complete [15], researchers use approximations in their methods. Classically, we may divide the methods into two classes, namely complete and incomplete. The methods [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] are complete. Since complete methods explore the exact state space, they suffer from scalability issues on large-scale networks. On the other hand, abstraction-based methods, e.g., [27], [28], [29], [30], [31], [32], [33] are sound and incomplete, because they over-approximate the state space, but they scale extremely well to large benchmarks. A representative method, DeepPoly [34], maintains and propagates upper and lower bound constraints using the so-called triangle

approximation. This is also sometimes called bound-propagation. Unsurprisingly, DeepPoly and other abstraction-based methods suffer from imprecision. Hence, the methods [35, 36, 37, 38, 39] refine the over-approximated state space to achieve completeness. In [35, 36, 39], the authors eliminate the spurious information (i.e., imprecision introduced by abstraction) by bisecting the input space on the guided dimension. In [38], which also works on top of DeepPoly [34], the authors remove the spurious region by doing the conjunction of each neuron's constraints with the negation of the robustness property and using an MILP (mixed integer linear programming) optimizer Gurobi [40] to refine the bounds of neurons. Another work that refines DeepPoly is kPoly [41], which considers a group of neurons at once to generate the constraints and compute the bounds of neurons. One issue with all these approaches is that refinement is not guided by previous information/runs and hence they suffer from scalability issues.

In this thesis, we develop an approach to effectively reduce imprecisions using counterexamples, following the counterexample-guided abstraction refinement paradigm. Our method addresses the limitations of abstraction-based neural network verification techniques such as DeepPoly [34], CROWN [33], and α-CROWN [42], which often suffer from over-approximation errors that lead to inconclusive verification results. Furthermore, we analyzed the limitations of the local robustness property in terms of false positives. We extended this work by incorporating classification sensitivity into the robustness property, enabling a more practical and meaningful analysis. In particular, we proposed confidence-based properties and introduced a unified framework for their verification. If a model is verified, one may still ask why a model behaves the way it does, since these models are not explicitly designed by humans but are instead learned from input-output pairs. Therefore, we also consider the problem of explainability in machine learning models, where the goal is to understand the reasoning behind the decisions of a model [43, 44]. In this context, we focus on the explainability of policies learned by reinforcement learning models. Our approach expresses the behavior of such policies in the form of *Linear Temporal Logic* (LTL) formulas, providing an interpretable description of the learned behavior.

We start with the counterexample-guided refinement approach. We develop an approach to identify the precise source of imprecision during abstraction. We present a *novel* counterexample-guided approach that can be applied alongside many abstraction techniques. As a specific case, we implement our technique on top of a basic abstraction framework provided by the tool DeepPoly and demonstrate how we can remove impreci-

sions in a targeted manner. The source of imprecision is identified in terms of neurons (also called *marked neurons*) whose abstractions lead to spurious counterexamples. Surprisingly, we are also able to verify several benchmark instances on which all leading tools fail [45].

The set of marked neurons is determined using MAXSAT queries (explained in Chapter 2) on each layer, starting from the first layer of the network up to the last layer, stopping once marked neurons are found. If the marked neurons lie in deeper layers, multiple MAXSAT queries are required to reach that layer. Thus, finding such marked neurons in this approach is computationally expensive. We extend this work to overcome the limitations by searching for marked neurons starting from the last layer and moving backward toward the first layer. This approach also employs MAXSAT queries. If the marked neurons are located in deeper layers, this method identifies them more quickly.

Most existing works [29, 34, 31, 33] either verify the specification or report a counter-example. However, the counterexamples found by state-of-the-art neural network verifiers are not always meaningful. In fact, a counterexample reported as a robustness violation by a neural network may not be considered a violation from the perspective of a domain expert. We refer to such cases as false positives. In this thesis, we propose a new approach to evaluate the robustness of neural networks by explicitly incorporating the view of the domain expert. We begin by adapting the notions of false positives (FP) and true positives (TP) to the context of robustness verification of neural networks. To approximate the domain expert, we employ an ensemble of classifiers.

We further extend the local robustness problem to incorporate the sensitivity of the classification decision (classification confidence). Intuitively, we bypass counterexamples with low confidence, since the network itself is not confident enough in its classification, it is more appropriate that such cases be monitored by a human authority. At the same time, this approach allows us to capture the high-confidence counterexamples. Confidence is defined using the well-known SOFTMAX function [46]. Incorporating confidence into the verification specification requires efficient handling of the highly non-linear SOFTMAX function. We also develop a unified framework capable of verifying a diverse set of properties, both confidence-based and non-confidence-based. Our approach constructs a neural network gadget from an arbitrary post-condition and appends it to the underlying network, thereby transforming the original post-condition into a simplified one.

For explainability, we learn a summary of the behaviors of the policies, which is deeply related to inverse reinforsement learning, where we learn the reward function from the

policies. In explainability, we go one step further and summarize the learned reward function into an interpretable formula. The key computational problem in *inverse reinforcement learning* (IRL) is to identify a reward signal implicit in the expert demonstrations. While effective, such reward signals often lack explainability and lead to opaque learning. To address this, we present a novel IRL method for eliciting declarative learning requirements in the GF fragment of *linear temporal logic* (LTL) from a set of traces provided by an expert policy. We implemented this approach as an open-source tool, QuantLearn, to perform logic-based non-Markovian IRL. Our results demonstrate the feasibility of the proposed approach in eliciting intuitive and interpretable LTL-based reward signals, even from noisy data.

1.1 Contributions

In this section, we outline the main contributions of this thesis. We begin by listing them as follows:

1. Using Counterexamples to Improve Robustness Verification in Neural Networks.
2. False Positives in Robustness Verification of Neural Networks.
3. Confidence-aware Robustness Properties of Neural Networks.
4. Verifying Rich Robustness Properties for Neural Networks.
5. Explainable Reward Learning in Inverse Reinforcement Learning.

We describe each of these contributions as follows:

1.1.1 Using Counterexamples to Improve Robustness Verification in Neural Networks

DeepPoly and other abstraction methods such as [29, 33] scale very well but remain imprecise due to over-approximation. In this work, we build upon the basic abstraction framework provided by DeepPoly and develop a novel refinement technique that is *counterexample-guided*, i.e., we use counterexamples generated from imprecisions during abstraction to guide the refinement process.

Counterexample-Guided Abstraction Refinement (CEGAR) [47, 48] is an iterative verification paradigm that alternates between constructing a coarse abstraction of a system and refining it whenever a discovered counterexample is found to be spurious. This

approach mitigates the state explosion problem while preserving soundness for safety properties. The process begins with an over-approximated state space. If a counterexample is found during analysis, we check its feasibility. If the counterexample is real, we report it, otherwise, we perform the refinement. During refinement, the imprecision introduced by over-approximation is reduced by removing abstraction from selected states. The selection of such states is guided by various techniques [47, 48, 49, 50]. After refinement, the analysis is repeated on the reduced over-approximated state space, and the CEGAR loop continues iteratively.

In software verification, CEGAR underlies landmark tools such as SLAM [51, 50, 52] and BLAST [49, 53]. These tools verify C programs by predicate abstraction [54], checking counterexample feasibility via path-sensitive reasoning [55], and using interpolation-based refinement [56] to discover new predicates when traces are spurious. Notably, the "lazy abstraction" [49] of BLAST refines only along relevant paths in an abstract reachability graph. This approach dramatically reduces effort, either producing a concrete counterexample or proving safety.

The above discussion highlights the significant advancements achieved through the CEGAR framework in program verification. In this work, we adapt the CEGAR approach for the formal verification of neural networks. Our main contributions are as follows:

We introduce two *maxsat-based* techniques to identify the cause of imprecision and spuriousness. Starting with an input where the abstraction is not verified (obtained using a MILP solver), we check whether the input generates a real counterexample that falsifies the property or a spurious counterexample, by executing the neural network. In the case of a spurious counterexample, we identify the neuron or set of neurons responsible for it. The first technique executes MAXSAT queries starting from the first layer and proceeding to the last layer, efficiently identifying the source of imprecision when it lies in the initial layers. The second technique executes the queries in the reverse order, from the last layers toward the initial layers, and is more efficient when the source of imprecision lies in deeper layers. Using these specially identified or *marked* neurons, we perform split and refine steps, ensuring that unlike earlier refinement methods, our method progresses at each iteration and eliminates spurious counterexamples. We further adapt the existing refinement framework built on ideas from MILP-methods and implement this as a counterexample guided abstraction refinement algorithm on top of DEEPPOLY. To enable this, we designed an MILP-based encoding for the MAXSAT queries, together with the corresponding algorithm,

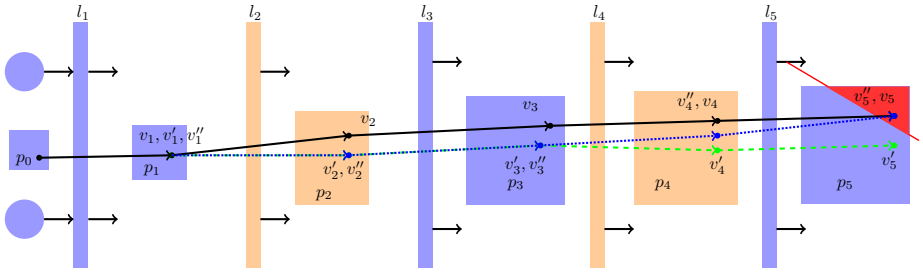


Figure 1.2: Pictorial representation of our approach on example in figure 4.1

and proved its correctness.

Consider the pictorial representation in Figure 1.2, where $l_1, l_2, \ldots, l_5$ represent the layer indexes. We consider separate layers for the affine and activation layers. We consider only ReLU activation in this work, so we also call the activation layer as ReLU layer. The space p_0 represents the constraints on the inputs and $p_1, p_2, \ldots, p_5$ represents the abstract space on each layer, generated by the abstract tool, like DeepPoly. The red triangle on the layer l_5 represents the negation of the property to be verified. We take all the abstract constraints with negation of the property and check for the satisfaction, if it is unsatisfied, then the property is verified otherwise, there may be a counterexample, the satisfying assignment. We check the satisfying assignment on the input layer, which is v_0, by simulating the neural network and checking if its corresponding output is violating the output property or not, if it violates the property then it is a counterexample otherwise its a spurious counterexample, which means in abstract domain it violates the property and in concrete execution it does not violate the property.

Let us say, the black solid line in Figure 1.2 represents the satisfying assignments in the abstract domain, also called abstract execution, $v_0, v_1, \ldots, v_5$, and the dashed green line represents the concrete execution of the satisfying input v_0. v_i' represents the concrete execution output at layer l_i.

Consider the pictorial representation in Figure 1.2, where $l_1, l_2, \ldots, l_5$ denote the layer indices. We consider separate layers for affine transformations and activation functions. Since this work focuses exclusively on the ReLU activation function, the activation layers are also referred to as ReLU layers. The space p_0 represents the set of constraints on the input, whereas $p_1, p_2, \ldots, p_5$ denote the abstract spaces corresponding to each layer, as generated by an abstraction based tool such as DeepPoly. The red triangle in layer l_5 represents the negation of the property to be verified.

To verify a property, we combine the abstract constraints with the negation of the property and check their satisfiability. If the resulting query is unsatisfiable, the property is verified. Otherwise, a satisfying assignment exists, representing a potential counter-example. We then evaluate this satisfying assignment at the input layer, denoted by v_0, by simulating the neural network to determine whether its corresponding output violates the property. If it does, the assignment is a genuine counterexample, otherwise, it is a spurious counterexample, meaning that it violates the property in the abstract domain but not in the concrete (actual) execution.

In Figure 1.2, the solid black line represents the sequence of satisfying assignments in the abstract domain, also called the abstract execution, denoted as $v_0, v_1, \ldots, v_5$. The dashed green line represents the concrete execution corresponding to the satisfying input v_0, where v_i' denotes the concrete output at layer l_i.

Although several approaches [57, 58, 59] identify important neurons, also referred to as *marked neurons*, by leveraging local information such as weights or gradients, none of the existing approaches provides a formal guarantee of progress. To address this limitation, we propose two variants for identifying marked neurons, namely, the *forward marking* and *backward marking* approaches. Both variants guarantee progress during the verification process. We briefly introduce these approaches here and discuss them in detail in Chapter 4.

Forward marking: In this approach, we begin marking from the initial layers and proceed towards deeper layers. Since affine layers do not introduce any approximation, we start from the first ReLU layer, denoted by l_2. The objective is to make the abstract execution point v_2 as close as possible to the corresponding concrete execution point v_2'. Here, closeness means that each element of the vector v_2 is equal to the corresponding element of v_2'. If all elements of v_2 become equal to the corresponding points of v_2', it implies that v_5 can still be reached from v_2'. In this case, the sequence $v_0, v_1', v_2', v_3, v_4, v_5$ remains a valid abstract execution in the abstract domain, represented by the dotted blue line in Figure 1.2. We then proceed to the next ReLU layer, l_4, and repeat the same process to align v_4 with v_4'. If, at this stage, some elements of v_4 do not match the corresponding elements in v_4', then the corresponding neurons are identified as marked neurons. Refining the abstraction from these neurons will eliminate the existing abstract counterexample.

This forward marking approach becomes inefficient when the marked neuron lies in the deeper layers, as it requires iterating through every layer from the initial one to that deep layer. To address this limitation, we introduce the *backward marking* variant, which

iterates from deeper layers towards the initial layers.

Backward marking: In this approach, we begin by checking at the last ReLU layer, l_4, whether v_5 is reachable from v'_4 in the abstract domain. To verify reachability, we transform the reachability query into a satisfiability query. If v_5 is not reachable, we move backward to the previous ReLU layer and check reachability from v'_2. If v_5 is reachable from v'_2, it implies that marked neurons exist between layers l_2 and l_4. If it is not reachable, we continue moving backward to earlier ReLU layers in the network.

We will discuss in Chapter 4, the processes of checking reachability and identifying marked neurons can be combined into a single query. It is also important to note that the output point v_5 is always non-reachable from the concrete point corresponding to the last ReLU layer, v'_4. Therefore, the backward marking process always begins from the second last layer.

We evaluated our tool against methods based on DeepPoly (kPoly [41] and deepSRGR [38]), the CEGAR-based cegar_nn [37], and state-of-the-art tools $\alpha\beta$-CROWN [60], Oval [61], marabou [62], and nnenum [63, 64] on a benchmark suite of 8496 instances. Our tool outperforms kPoly, deepSRGR, and cegar_nn, and also surpasses marabou and nnenum by about 150 and 619 benchmarks. While $\alpha\beta$-CROWN and Oval solve around 1000 more benchmarks, our tool solves 121 unique instances none of the four SOTA tools can handle, highlighting their complementary strengths.

1.1.2 False Positives in Robustness Verification of Neural Networks

In the above section, we discussed techniques to verify or falsify the local robustness property. This property states that for a given neural network and an input image, the network is considered locally robust if all images close to the given image are classified the same as the original, where closeness is defined using a distance metric such as L_1, L_2, or L_∞ norms. Recall that images close to the original but classified differently are often referred to as *counterexamples*. Existing research focuses on either finding such counterexamples or providing certification of their absence.

However, in doing so, existing works [29, 30, 33, 62, 60] often lose the focus on evaluating the *genuineness* of the counterexample. This leads to false positives, i.e., counterexamples that exist from the perspective of the underlying specification, but may not be considered genuine from a domain expert's perspective. In this work, our aim is to identify false and true positives and study their prevalence in commonly used image classification benchmarks.

(A) Pred-0 (B) Pred-6

Figure 1.3: Examples: (A) is the original image with label 0, correctly classified as 0, and (B) is the perturbed image, misclassified as label 6.

Consider the image in Figure 1.3A. This image is from the MNIST dataset [65] and has ground-truth labels of 0. The image on the right 1.3B is a counterexample generated by the state-of-the-art verifier $\alpha\beta$-CROWN [60] for the image labeled 0 and misclassified as 6. A domain expert may not consider these as actual *bugs* because the counterexample labeled as 6 also visually resembles a 6, and the network predicts it as either 0 or 6, which is reasonable. Since the network's predictions are visually justifiable, labeling them as counterexamples is, in fact, a false positive.

In this work, we recall and define the well-known notions of false positive (FP) and true positive (TP) in the context of local robustness verification of neural network image classifiers. Our modified definition requires a domain expert to classify the images and assign the ground truth labels, which we formally call an oracle. An important point to note is that in our setting, even an oracle may not always be able to classify the images since images may resemble many labels. Thus, it may assign a set of possible labels to an image. As a result, we may use the oracle in two possible ways: (1) to determine whether a counterexample produced by the verifier is a true or false positive by checking whether the classification of the counterexample belongs to one of the oracle-provided labels, and (2) to guide the verifier to search only for counterexamples that are not classified as one of the oracle-provided labels.

We evaluate the standard local robustness property with respect to false positives and true positives. Our analysis reveals approximately 12.86% false positives on the MNIST [65] dataset and 5.65% false positives on the CIFAR-10 dataset. To mitigate these false positives, we propose an ensemble-guided property, which successfully reduces the false positive rate from 12.86% to 3.22%. In summary, our approach and novel analysis show that the local robustness as is usually considered can be overtly conservative. And thus, verifiers are not able to declare a model trustworthy even when the model is actually

Figure 1.4: **Relaxed Robustness:** The network `convBigRELU-PGD.onnx` of Table 7.1, correctly classified the original image (A) as `horse` and (C) as `airplane` with high confidence. With an input perturbation of 16/255, we can find misclassified images (B) for `horse` and (D) for `airplane`, but with low confidence. In fact, it turns out all counterexamples have low confidence and hence verification succeeds under the relaxed robustness criterion with an 80% confidence threshold, while state-of-the-art verifiers would have declared this network non-robust.

trustworthy according to the domain experts.

1.1.3 Confidence-aware Robustness Properties of Neural Networks

As discussed in the previous sections, most existing works [26, 27, 30, 33] focus on *local robustness verification*, where, given an input image and its corresponding label, a neural network is required to correctly classify all images within a small neighborhood of that input. However, different applications may require different variants of robustness, as highlighted in [66]. Moreover, many existing robustness verification approaches regard answers returned by classifiers as binary, either a correct classification or a misclassification, ignoring the fact that classifiers provide a confidence in the counterexample in terms of the classification probability. The confidence is computed using the well-known Softmax function [46].

In this section, we introduce *confidence-based robustness properties*, which are more semantic and intuitive than the standard local robustness formulation. Since computing confidence involves the highly non-linear Softmax function, we employ suitable approximations to enable efficient analysis. We begin by motivating and defining several variants of confidence-based properties.

Relaxed Robustness: Let us first discuss a weaker notion of robustness, which we call *relaxed robustness*. The key idea is to ignore low-confidence counterexamples when determining the robustness of a network, as illustrated in Figure 1.4. Specifically, for all

images x close to a given image x^*, their classifications should remain the same unless the confidence of the network on x is less than a certain threshold. Intuitively, if a network is not confident about its prediction, the decision may or may not be correct, and such cases can be left for manual review by some authority. On the other hand, this notion of robustness highlights high-confidence counterexamples, cases where the network makes a confidently incorrect prediction, which indicates serious concerns about the reliability of the underlying neural network.

Strong Robustness: The second variant of robustness, which we call *strong robustness*, aims to capture cases where the confidence on the seed image is high, but drops below a certain threshold within an ϵ perturbation, regardless of whether the classification label changes. In other words, a significant drop in confidence itself indicates a weakness in the underlying network. This notion of strong robustness generalizes the concept introduced in [66]. In [66], the authors consider the logits values instead of the confidence values. However, using logits is challenging in practice because they can take any real value, making it difficult to fix a meaningful threshold. In contrast, our approach uses confidence values, which always lie between 0 and 100, making the choice of a threshold more intuitive and interpretable.

Smoothness: A third variant is *smoothness*, which corresponds to an instance of Lipschitz continuity in neural networks [67, 68, 69]. The work in [66] introduces the concept of Lipschitz robustness, which bounds logit values of each class using a Lipschitz constant. We propose a simplified version of Lipschitz robustness, where we require that, within an ϵ perturbation, the confidence of the network should not exhibit significant variations with respect to the seed image class.

1.1.4 Verifying Rich Robustness Properties for Neural Networks

Each of the properties discussed in the previous section gives rise to different constraints on the output layer of the network, which typically need to be handled separately and often require additional effort. In this section, we highlight the challenges associated with handling each property individually and present a solution to manage all properties in a unified manner, without incurring extra effort. Our approach is not limited to the properties discussed above but is also capable of accommodating a wide variety of robustness specifications.

The main challenge is whether such rich properties defined as complex constraints on the output layer can be directly encoded in state-of-the-art tools. In fact, as standardized

in VNN-COMP, robustness properties are specified in a special VNNLIB format as a set of constraints on input and output layers, which we also call pre- and post-conditions that a neural network must satisfy. Although VNNLIB supports arbitrary Boolean combinations of linear constraints, most state-of-the-art neural network verifiers are optimized for simpler post-conditions, which typically involve only disjunctions or conjunctions of linear atoms over the outputs. Thus, verifying properties with multi-layered conjunctions, disjunctions, or combinations of both poses significant challenges: (i) Modifying the verifier's code to handle complex post-conditions requires a deep understanding of its implementation, which can be challenging for users unfamiliar with the codebase, i.e. $\alpha\beta$-CROWN [60] (ii) The source code for some commercial verifiers may not be publicly available, restricting the ability to make modifications. (iii) Even when access is available, adapting the code for each new property format is time-consuming and prone to errors. (iv) Constraint-based tools, i.e., MARABOU [62], allow such properties to be encoded directly as constraints, but users are then limited to a particular solver only and lack to achieve the scalability. (v) Advanced techniques like Projected Gradient Descent (PGD) attacks, which are highly effective at finding counterexamples in neural networks, cannot always be applied directly due to the complexity of certain post-conditions.

To overcome these challenges, we propose a unifying framework that simplifies arbitrary post-conditions by appending a few additional layers to the neural network. This transformation converts the post-condition into the simplified form, such as $y \geq 0$ or $y > 0$, where y is the output of an added node in the neural network, while maintaining low error bounds. The transformation generates new layers that encode parts of the post-conditions and composes the outputs of these layers according to the Boolean operations in the post-conditions. We use ReLUs to model the Boolean operations in the post-conditions. The output of a sum of ReLUs can model conjunction or disjunction: if all inputs are negative, the output is zero; if any input is positive, the output is positive. Since we use ReLU operators to model all Boolean operations, conjunctions and disjunctions interpret input signals in opposite ways, i.e., for a conjunction, a negative input yields 1 and a positive input yields 0, while for a disjunction, a positive input yields 0 and a negative input yields 1. To enable the composition of conjunction outputs into disjunctions and vice versa, we introduce a novel technique that reverses the outputs using a flip operation, while maintaining low error bounds. Our method is inspired by the NP-Complete proof of hardness of the robustness problem of paper [20]. These simplified post-conditions can

be verified using any state-of-the-art verifier as a black box, eliminating the need for code modifications and ensuring seamless integration with existing tools.

We define a simple grammar on the post-conditions that unifies all the variants. The main idea then is that the post-condition is algorithmically transformed into additional layers. This allows the resulting network to be verified by any state-of-the-art neural network verifier. Our grammar allows us to capture existing notions of the properties from the literature, such as strong [66] and top-k robustness [70]. In our technique, instead of changing the encoding for each variant, we provide a generalized encoding for all properties that can be expressed using this grammar by adding additional layers to the neural network. Our encoding enables the use of state-of-the-art neural network verifiers like $\alpha\beta$-CROWN, PyRAT.

Our implementation encodes postconditions into additional neural network layers and then invokes a Neural network verification engine. For experiments, we selected the state-of-the-art tool $\alpha\beta$-CROWN [33, 71, 72, 42], a portfolio verifier, which has consistently ranked 1^{st} in VNN-COMP 2021-2024 [73, 74, 75, 76]. We conducted experiments on four different datasets: MNIST [65], CIFAR-10 [77], Traffic Sign Recognition (TSR) [78], and IMAGENET [8]. All networks used in the experiments, along with their properties (VNNLIB files), were taken from VNN-COMP 2021 to VNN-COMP 2024. The network sizes ranged from small architectures with 512 ReLUs to large networks with 11.16M ReLUs. Each neural network, along with its corresponding property (VNNLIB) file, represents a single benchmark. In total, we evaluated 8,870 benchmarks in our experiments.

Our experiments demonstrate that the method scales to the large VGG-7 network from VNN-COMP 2024. We are able to analyze all variants of properties with varying thresholds in a parametric manner. Notably, we observed that most benchmarks do not satisfy the strong robustness and smoothness properties mentioned in the previous section. To show the effectiveness of our proposed transformation technique, we compared our proposed layer-based encoding technique with the direct encoding of properties in a constraint-based state-of-the-art solver, such as MARABOU, and additionally compared our layer-based encoding using $\alpha\beta$-CROWN. We found that $\alpha\beta$-CROWN significantly outperforms MARABOU in both cases. Moreover, MARABOU with layer-based encoding performs better than MARABOU with direct encoding.

These findings show that our confidence-based properties and layer-based encoding enable the verification of diverse property variants. Moreover, the layer-based encoding

shows better scalability compared to direct encoding.

1.1.5 Explainable Reward Learning in Inverse Reinforcement Learning

Synthesizing explanations from demonstrations has emerged as an effective approach in scenarios where domain experts or performant agents can provide examples of both desirable and undesirable behaviors. One important embodiment of this form of learning is known as *inverse reinforcement learning* [79] (IRL), whereby an apprentice agent learns the reward function being optimized by a given expert policy or behavior. IRL is indispensable in settings where it is difficult or error-prone to explicate a reward signal that captures the underlying learning objective.

Interpretability and Explainability: Ng and Russell [80] present a convincing argument for preferring IRL over apprenticeship or imitation learning, emphasizing that learning a reward function—rather than policies—provides a more succinct, robust, and transferable representation of behavior. However, while inferring the objective function offers these advantages, scalar reward representations often lack interpretability. For instance, *why does an action yield different rewards in different states? What distinguishes an action with a reward of* 4.999 *from one with* 5.001*? How would the reward signal change for an agent with a lower discount factor (i.e., a more short-sighted one)?* Explaining an objective purely through scalar rewards is analogous to reading a program in assembly language—straightforward for machines but cumbersome for humans. This lack of human-readable structure (i.e., explainability) poses challenges for applying verification and validation methods to ensure the trustworthiness of learning-based system design. Furthermore, traditional IRL typically assumes that the reward to be learned is Markovian, an assumption that restricts the expressiveness of potential objectives [81, 82]. *In this work, we enhance the interpretability and explainability of inverse reinforcement learning by inferring logic-based objectives from demonstrations instead of scalar rewards.*

Linear Temporal Logic (LTL): We focus on a subset of Linear Temporal Logic (LTL) [83] as our specification language, primarily due to its succinctness [84, 85] and its wide applicability across multiple domains, including artificial intelligence [86, 87], formal methods [84, 88], control theory [89, 90], and machine learning [91]. In recent years, LTL has gained significant popularity [91, 92, 93] for expressing learning objectives in model-free reinforcement learning (RL), owing to its ability to naturally capture temporal dependencies and high-level task specifications in a compact and interpretable form.

The key computational problem for LTL-based IRL is as follows: given a pair $\mathcal{S} = (P, N)$ of samples consisting of positive traces P and negative traces N, both being sets of finite words, the task is to produce the highest-ranking LTL specification that is consistent with the sample. The ranking is determined by a user-defined notion of simplicity over LTL specifications. This formulation highlights the central challenge of induction: we must infer an LTL specification that captures behaviors over traces of potentially unbounded length, while having access only to a finite set of finite examples and counterexamples!

In this work, we define a quantitative semantics for evaluating the satisfaction of an LTL specification over a finite word, guided by the principle of parsimony in explanation. The complexity of an LTL formula can arise from two main sources: the complexity of its *temporal structure* (for instance, $\mathsf{G}p$ is simpler than $p \wedge \mathsf{X}p \wedge \mathsf{XX}p$ when explaining the sample $P = \{\{p\}\{p\}\{p\}\}$ and $N = \emptyset$), and the complexity of its *nesting structure* (for example, the formula p is simpler than $p \wedge \neg q$ when explaining the sample $P = \{p\}$ and $N = \emptyset$). We introduce two hyperparameters—temporal discounting (α) and nesting discounting (δ) to capture user preferences when balancing these sources of complexity. To mitigate overfitting, we restrict our attention to the GF fragment [94, 95] of LTL, where the temporal operators are limited to G and F.

We propose three optimization algorithms: *Constraint System Optimization*, *Compositional Ranking*, and *Hybrid Pattern Matching*, leveraging the state-of-the-art constraint solver Z3 [96] to address the problem of LTL formula learning. We establish the soundness and completeness of the *Constraint System Optimization* and *Hybrid Pattern Matching* algorithms, while the *Compositional Ranking* algorithm—though incomplete—achieves a speedup of up to three orders of magnitude compared to the other two. We have implemented these algorithms in an open-source tool and evaluated their effectiveness on randomized gridworld environments by computing the inverse learning error (ILE) for policies derived from rewards learned by our tool. Our results demonstrate competitive or superior performance compared to existing reward learning approaches.

We apply learning techniques to generate a reward function over an MDP defined as a grid world to obtain a non-Markovian reward decision process (NMRDP). After generating a randomized 10×10 grid environment labeled with propositional variables, we uniformly sample the grid, taking actions compatible with an input automaton. This ensures that the generated traces satisfy a given formula. We use the same LTL properties as the previous case. We allow the MDP to randomly simulate for at least 100 steps, after which

we wait for it to reach an accepting state. Through this method, we generated traces of length varying between 100 and 150, with 1000 positive and 1000 negative traces for each formula, amounting to a total trace length of at least 10^5 across all positive and negative inputs. However, for constraint system optimization and Traces2LTL [97], due to timeouts, a smaller subset was randomly selected from the traces. We computed the mean of inverse learning error (MeanILE) [98], which is a metric used in inverse reinforcement learning to quantify how far the policy induced by a learned reward deviates in value from the optimal policy of the expert on the same MDP. Our experiments demonstrate that the MeanILE computed by our methods is less than the same computed by Traces2LTL.

1.2 Structure of the Thesis

We introduce the basic notions and definitions in Chapter 2, followed by a discussion of related work in Chapter 3. In Chapter 4, we present our counterexample-guided frameworks for robustness verification. In Chapter 5, we analyze false positives and true positives and describe our approach to mitigating false positives. Chapter 6 introduces the definitions of different specifications based on the sensitivity of network decisions. In Chapter 7, we present a unified framework for verifying different variants of properties. In Chapter 8, we discuss our work on learning explainable rewards in inverse reinforcement learning using temporal logic specifications. Finally, Chapter 9 concludes the thesis and outlines future directions.

1.3 Publications

The following publications have resulted from the work presented in this thesis or are currently under submission:

- The work on counterexample-guided refinement for robustness verification was published in *ATVA 2023* [45].
- The work on identifying false positives in local robustness verification was published in *OVERLAY 2025*.
- The work on confidence-based properties and their verification within a unified framework has been accepted at *FM 2026* and is also available on arXiv [99].
- The work on explainable reward learning in inverse reinforcement learning was published in *AAMAS 2023* [100].

Chapter 2

Preliminaries

In this section, we present the basic mathematical notations used throughout this thesis. These notations facilitate the formulation of various neural network properties, proofs of theorems, and the description of algorithms. We begin with the fundamental concepts underlying neural networks by introducing their foundational building blocks, including neurons, their functionality, the composition of layers, and the overall architecture of neural networks. We formally define the key notions related to neural networks that are used throughout this thesis. In addition, we provide a formal definition of the local robustness property. To address different robustness properties, we employ the application programming interfaces (APIs) of the relevant solver. Accordingly, we describe the solver APIs used in this thesis.

2.1 Basic Mathematical Notations

The symbols $\mathbb{R}$ and $\mathbb{Z}$ denote the sets of real and integer numbers, respectively. Lowercase letters, such as x, are used to represent numerical variables. For instance, if a variable is of type $\mathbb{R}$, it serves as a placeholder for a real number, similarly, integer type variables represent integers. A binary variable is an integer variable that can take the value either 0 or 1.

In a similar manner, we define vectors and matrices. Bold lowercase letters, such as $\mathbf{x}$, denote vectors, with individual components represented as $\mathbf{x}[i]$ or $\mathbf{x}_i$ for the i^{th} element. An n-dimensional real-valued vector is denoted by $\mathbf{x} \in \mathbb{R}^n$, where $\mathbb{R}^n$ represents the n-dimensional real vector space. Similarly, bold uppercase letters, such as $\mathbf{A} \in \mathbb{R}^{m \times n}$, denote matrices, with elements $\mathbf{A}[i, j]$ referring to the entry in the i^{th} row and j^{th} column. The

notation $\mathbb{R}^{m\times n}$ represents the space of real matrices with m rows and n columns, and thus the dimensions of a matrix $\mathbf{A}$ are $m \times n$. Once clear from the context, we also use non-bold letters x and A to represent vectors and matrices, respectively.

Using the above conventions, we define a linear expression as

$$LinExpr = \left\{ w_0 + \sum_i w_i x_i \;\middle|\; w_i \in \mathbb{R} \text{ and } x_i \text{ is a real variable} \right\}.$$

Intuitively, a linear expression is a linear combination of real variables. Based on *LinExpr*, we define the set of linear constraints as

$$LinConstr = \{\ expr\ \ \text{op}\ \ 0 \;\middle|\; expr \in LinExpr \wedge op \in \{\leq, =\}\}.$$

We further define a Boolean combination of *LinConstr* as the result of combining multiple *LinConstr* instances using Boolean operators: logical conjunction ($\wedge$), logical disjunction ($\vee$), and logical negation ($\neg$). A predicate is any Boolean combination of elements in *LinConstr*.

Given two functions $f : X \to Y$ and $g : Y \to Z$, the composition $g \circ f$ denotes a function from X to Z obtained by applying f first and then g. Formally, for any $x \in X$, we have $(g \circ f)(x) = g(f(x))$. This notation captures the sequential application of two mappings, where the output of the first function serves as the input to the second.

2.2 Neural Networks

Neural Networks (NNs) are computational models inspired by the interconnected structure of neurons in the human brain [46]. They are designed to recognize patterns, approximate functions, and adaptively learn from data. A neural network processes information by successively transforming inputs through multiple layers, each composed of interconnected computational units called neurons.

For instance, consider a target function f^*. In the case of classification, this function maps an input $\boldsymbol{x}$ to a category label y, i.e., $y = f^*(\boldsymbol{x})$. A neural network aims to approximate this mapping by defining $y = f(\boldsymbol{x}; \theta)$, where θ denotes the set of learnable parameters. The learning corresponds to adjusting θ such that $f(\boldsymbol{x}; \theta)$ closely approximates f^*.

The strength of neural networks lies in their ability to model complex nonlinear relationships by hierarchically composing simple mathematical transformations. Modern applications of neural networks encompass diverse domains, including computer vision [46,

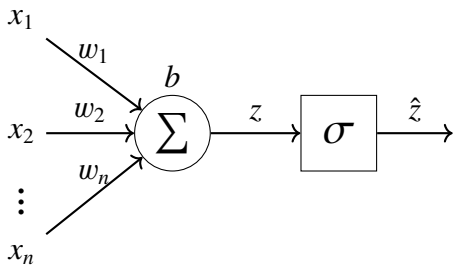


Figure 2.1: A schematic representation of a single neuron. The symbol $\sum$ denotes the weighted sum of the inputs, while σ represents the nonlinear activation function. Here, $w_1, w_2, \ldots, w_n$ are the weights associated with the inputs, and b denotes the bias term.

101], natural language processing [102], robotics [103], and safety-critical decision-making systems [1, 2, 3].

We present the fundamental architecture and building blocks of neural networks as follows:

2.2.1 Neurons

The fundamental computation unit of a neural network is a neuron (or node) [46], which is inspired by the biological neuron. An artificial neuron receives multiple input signals, weights them, adds a bias, and transforms the result through a non-linear activation function.

The mathematical computation of a neuron is expressed as follows:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

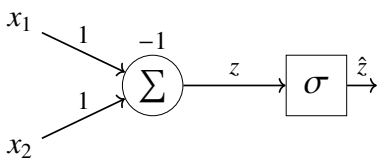
$$\hat{z} = \sigma(z)$$

The above computation is also illustrated in Figure 2.1. In this formulation, z denotes the weighted sum of the inputs (i.e., the linear combination of input features), also referred to as an affine node, while $\hat{z}$ represents the final output obtained after applying a nonlinear activation function. The inputs $x_1, x_2, \ldots, x_n$ are collectively referred to as features.

For illustration, consider the logical AND function $\hat{z} = \text{AND}(x_1, x_2)$, as shown in Table 2.2A. The output of this function is 1 when both inputs are 1, and 0 otherwise. Let x_1 and x_2 be binary variables. We simulate this function using a single neuron, as depicted in Figure 2.2B. The weights corresponding to x_1 and x_2 are both set to 1, and the bias

x_1	x_1	$\hat{z}$
0	0	0
0	1	0
1	0	0
1	1	1

(A) Truth table for the logical AND operation



(B) Neuron representation of logical AND

Figure 2.2: Neuron example

term is set to -1. The activation function $\hat{z} = \sigma(z)$ is chosen as the well-known ReLU function [104, 101], defined as $\hat{z} = \max(0, z)$.

Here, z represents the weighted sum of the inputs x_1 and x_2, i.e., $z = x_1 + x_2 - 1$. From Table 2.2A, it can be observed that $\hat{z} = 1$ only when both inputs are 1. The chosen activation function thus correctly reproduces the AND behavior: when both inputs are 1, $z = 1$ and $\hat{z} = 1$; in all other cases, $\hat{z} = 0$.

The choice of activation functions and their properties will be discussed in more detail later. For now, it is important to note that the weights w_i and bias b are learnable parameters, adjusted during training to optimize the network's performance.

2.2.2 Layers

Neurons are organized in layers, such that all neurons in the current layer receive signals from the neurons of the previous layer, perform computations, apply nonlinear activation functions, and forward the signals to the next layer. Figure 2.3 illustrates this layered organization. The input layer, which precedes the first hidden layer, propagates the input as it is. The hidden layers perform computations, followed by nonlinear activations, and forward the signals to the output layer. Each neuron has input weights to compute the weighted sum of the input signals. We represent the weights of all neurons in a layer in matrix form, and the biases are similarly stored in a matrix. For simplicity, we assume no activation functions on the output layer. Under our assumptions, there can be any number of hidden layers, but there is only one input layer and one output layer.

2.2.3 Neural Network Architectures

The arrangement of layers, the number of neurons in each layer, and the choice of activation functions define the architecture of a neural network. The architecture shown in Figure 2.3 represents a feedforward neural network, where signals propagate only to

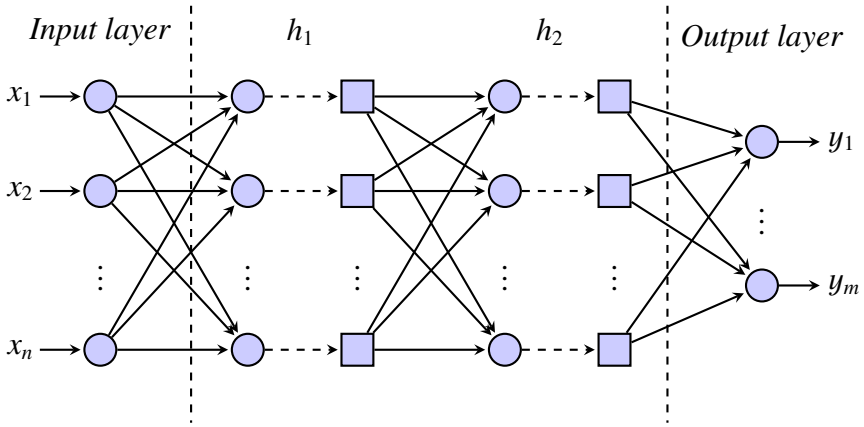


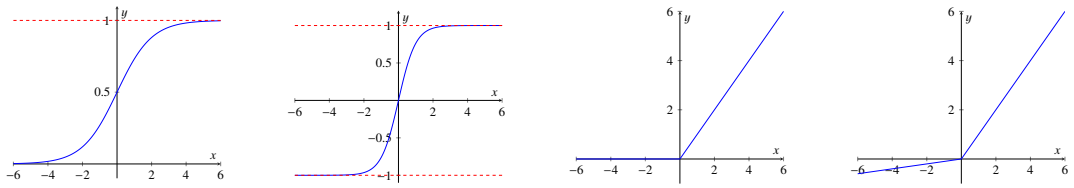
Figure 2.3: A schematic representation of a neural network. The network consists of one input layer, one output layer, and two hidden layers. It takes an input vector $\boldsymbol{x}$ of n dimensions and produces an output vector $\boldsymbol{y}$ of m dimensions. The neurons in each hidden layer use the activation functions discussed in Figure 2.1. The circular nodes denote the affine nodes, while the square nodes denote the activation nodes.

the next layer. Feedforward networks are suitable for relatively simple tasks. In some cases, the hidden layers perform convolutions on the outputs of preceding layers, followed by activation functions, such networks are called convolutional neural networks (CNNs), which are particularly effective for vision-based tasks. When a network contains many hidden layers, it is referred to as a deep neural network (DNN). Some well-known DNN architectures, which are also CNNs, include AlexNet [105], VGGNet [106], InceptionNet [107], and EfficientNet [108]. This thesis focuses on the verification of DNNs and CNNs. Additionally, in some architectures, the output of a layer can be fed back as input to the same layer, such networks are known as recurrent neural networks (RNNs), which are beyond the scope of this thesis.

2.2.4 Learning Parameters

As mentioned earlier, the function $f(\boldsymbol{x}, \theta)$ represents the neural network, while $f^*(\boldsymbol{x})$ denotes the target function. The target function f^* is not given explicitly; instead, we have access to a set of input–output pairs. The goal of parameter learning algorithms is to determine θ such that $f(\boldsymbol{x}, \theta)$ accurately learns the mapping between these pairs and generalizes well to similar inputs.

To achieve this, at the output layer, the discrepancy between the predicted output and the ground truth is quantified using a loss function. Let L denote the loss function.



(A) Sigmoid (B) Tanh (C) ReLU (D) Leaky ReLU

Figure 2.4: Activation functions

Gradients of L with respect to the parameters θ, i.e., $\frac{dL}{d\theta}$, are computed and used to update θ. This process is known as gradient-based learning [109, 110]. A necessary condition for applying the gradient descent algorithm is that the neural network must be differentiable, allowing precise computation of $\frac{dL}{d\theta}$.

2.2.5 Activation Functions

Activation functions are essential for the generalization of neural networks. Let us consider removing all activation functions from the network shown in Figure 2.3. In this case, all remaining layers are affine (weighted sum) layers, which can be merged into a single layer by multiplying the weight matrices. This means that no matter how many layers the network has, without activation functions, it effectively behaves as a single layer, leading to reduced generalization. Therefore, activation functions are the heart of the neural networks. On the other hand, activation functions also pose the main challenge in the analysis of neural networks, as we discuss later. There are many activation functions; some of the commonly used ones are listed below. Let us assume activation is represented as $y = \sigma(x)$:

Sigmoid: The sigmoid activation [111, 101] function is defined as

$$y = \frac{1}{1 + e^{-x}},$$

as illustrated in Figure 2.4A. The Sigmoid function maps the input value to the range $(0, 1)$ and is particularly suitable for binary classification tasks where the network must assign the input to one of two categories.

Tanh: The hyperbolic tangent activation [101, 46] function is given by

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}},$$

as shown in Figure 2.4B. The Tanh function maps the input to the range $(-1, 1)$.

Although the above activation functions are commonly used in output layers for classification, they suffer from the vanishing gradient problem [46]. The ReLU activation function mitigates this issue to some extent.

ReLU: The Rectified Linear Unit activation [104, 101] function is defined as

$$y = \max(0, x),$$

as shown in Figure 2.4C. It returns the input x when $x > 0$ and 0 otherwise. This function is also referred to as a piecewise linear activation function, since it comprises two linear regions based on the sign of the input. A drawback of ReLU is the dying neuron problem: when $x < 0$, the gradient remains zero, potentially causing certain neurons to stop learning.

LeakyReLU: The LeakyReLU activation [112, 101] function modifies ReLU to address the dying neuron issue and is defined as

$$y = \max(\alpha x, x),$$

where α is a positive slope for negative input values. An example with $\alpha = 0.1$ is shown in Figure 2.4D. Compared to ReLU, LeakyReLU introduces a slightly higher computational cost and requires tuning of the slope hyperparameter α.

Softmax: Let $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$ denote the output vector of a neural network. The Softmax function [46, 101] computes a probability distribution over all classes. For a given class y_c, it is defined as

$$\text{Softmax}((y_1, \ldots, y_m), c) = \frac{e^{y_c}}{\sum_{i=1}^{m} e^{y_i}}.$$

Intuitively, the Sigmoid function provides a probability distribution for binary classification tasks, whereas the Softmax function extends this concept to multiclass classification, i.e., when there are more than two possible classes.

In a neural network used for classification, each output neuron corresponds to a distinct class, and its output y_i represents the *logit* value for that class. The Softmax function transforms these logits into a normalized probability distribution across all classes. This distribution can be further scaled to the range $[0, 100]$ to represent the classification confidence [113, 114], defined as

$$\text{Conf}((y_1, \ldots, y_m), c) = 100 \cdot \text{Softmax}((y_1, \ldots, y_m), c). \tag{2.1}$$

2.3 Formal Definition of Neural Network

We now combine all the submodules defined above and establish the notational framework required to formalize the theorems, proofs, and auxiliary lemmas presented throughout this thesis. The preceding discussion provided the necessary intuition for understanding neural networks, this section introduces a rigorous formalization of those concepts. To this end, inputs, activations, and parameters are considered as elements of appropriately typed vector spaces, while layer-wise transformations are modeled as compositions of functions.

Definition 1. *A neural network* $N = (Neurons, Layers, Edges, W, B, Type)$ *is a 6-tuple, where*

- $Neurons$ *is the set of neurons in N,*
- $Layers = \{l_0, ..., l_k\}$ *is an indexed partition of* $Neurons$*,*
- $Edges \subseteq \bigcup_{i=1}^{k} l_{i-1} \times l_i$ *is a set of edges linking neurons on consecutive layers,*
- $W : Edges \mapsto \mathbb{R}$ *is a weight function on edges,*
- $B : Neurons \mapsto \mathbb{R}$ *is a bias function on neurons,*
- $Type : Layers \mapsto \{\text{Affine}, \text{Activation}\}$ *defines type of neurons on each layer.*

A neural network is a collection of layers $l_0, l_1, l_2, ...l_k$, where k represents the number of layers. Each layer contains neurons that are also indexed, with n_{ij} denoting the jth neuron of layer l_i. We call l_0 and l_k the *input* and *output layers* respectively, and all other layers as *hidden layers*. In our presentation, we assume separate layers for the activation functions, hence each layer can either be Affine or Activation layer. The definition of W and B applies only to the Affine layer. Without loss of generality, we assume that the output layer is an Affine layer, as we can always append an identity Affine layer. Layers $l_1, l_3, l_5, ..., l_k$ are the Affine layers, and $l_2, l_4, l_6, ..., l_{k-1}$ are the Activation layers. If $Type_i = \text{Activation}$, then $|l_{i-1}| = |l_i|$. We extend the weight function from edges to layers using matrix $W_i \in \mathbb{R}^{|l_i| \times |l_{i-1}|}$ that represents the weight for layer l_i, s.t.,

$$W_i[t_1, t_2] = \begin{cases} W(e) & e = (n_{(i-1)t_2}, n_{it_1}) \in Edges, \\ 0 & \text{otherwise.} \end{cases}$$

We also write matrix $B_i \in \mathbb{R}^{|l_i|\times 1}$ to denote the bias matrix for layer l_i. The entry $B_i[t, 0] = B(n_{it})$, where $n_{it} \in Neurons$.

To define the semantics of N, we will use vectors $val_i = [val_{i1}, val_{i2}, ...val_{i|l_i|}]$ that represent the values of each neuron in the layer l_i. Let f_i be a function that computes the output vector of values at layer i using the values at layer $i-1$ as $val_i = f_i(val_{i-1})$. For each type of the layer, the functions are defined as follows: if $Type_i = \textsc{Affine}$, then $f_i(val_{i-1}) = W_i * val_{i-1} + B_i$; if $Type_i = \textsc{Activation}$, then $f_i(val_{i-1})_j = \sigma_i(val_{(i-1)j})$, where σ_i is an activation function for layer l_i.

The semantics of a neural network N is a function (we abuse notation and also denote this function as N) which takes an input, an $|l_0|$-dimensional vector of reals and gives as output an $|l_k|$-dimensional vector of reals, as a composition of functions $f_k \circ ... \circ f_1$. Thus, for an input $v \in \mathbb{R}^{|l_0|}$, we write its value computed by N at layer i as $val_i^v = f_i \circ ... \circ f_1(v)$, also the value of the j^{th} in i^{th} layer is represented by val_{ij}^v. We represent the output of the network as $N(v)$, which corresponds to val_k^v, and $N(v)[j]$ denotes the value of the j^{th} neuron in the output layer. The final decision $\hat{N}$ of the neural network is determined using the $\textsc{argmax}$ [101] function, which returns the index of the maximum output value: $\hat{N}(v) = \textsc{argmax}(N(v))$.

2.3.1 Formal Verification

Formal verification is a rigorous mathematical approach used to prove or disprove the correctness of systems with respect to a given specification. Originating from the field of software and hardware verification [115, 116, 117], it was developed to ensure that critical systems—such as operating systems, compilers, or safety-critical software behave as intended under all possible conditions. This is achieved by modeling both the system and its specifications in a formal logic and using automated reasoning techniques like model checking [117], theorem proving [118], or satisfiability solving [119] to exhaustively verify properties such as safety, liveness, or correctness.

In contrast to testing [120], which validates a system by executing it on a finite set of test cases, formal verification provides mathematical guarantees about the behavior of the system across all possible inputs. Testing can reveal the presence of bugs but cannot prove their absence, whereas verification can provide such proof within given assumptions or bounds. Similarly, while static analysis [121] inspects code or models without execution to detect potential errors, i.e., data flow violations, it generally relies on conservative approximations and cannot offer the same level of soundness and completeness as formal

verification. Static analysis is often used for scalability and early error detection, whereas formal verification emphasizes sound proofs of correctness.

The adoption of formal verification in neural network analysis [15, 28, 122] stems from its potential to provide guarantees in domains where empirical testing can fail to capture corner cases. Neural networks, particularly deep learning models, are highly non-linear and complex, making their behavior difficult to interpret and validate solely through empirical testing. Formal verification techniques applied to neural networks aim to mathematically prove properties such as robustness against adversarial perturbations, fairness, thereby extending the foundational principles of software verification into the domain of machine learning systems.

2.3.2 Verification Query

A neural network verification query aligns closely with the logical foundation of Hoare triples [123] from classical program verification. In software verification, a Hoare triple is written as $\{P\}\, C\, \{Q\}$, where C denotes a program, P is a precondition describing assumptions on the input state, and Q is a postcondition describing the desired property of the output state. The triple asserts that if the program C starts in any state satisfying P, then after execution, it will terminate in a state satisfying Q. Equivalently, the validity condition can be expressed as

$$\forall s,\ P(s) \implies Q(C(s)).$$

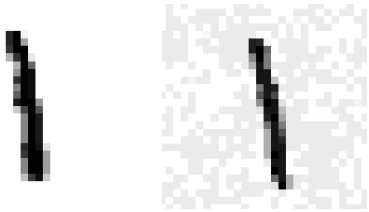
Where $P(s)$ denotes that s satisfies the condition P.

Analogously, in neural network verification, a *verification query* is defined as a triple $\langle N, P, Q\rangle$, where N is the neural network, P is a predicate over inputs, and Q is a predicate over outputs. The query holds if, for every input x such that $x \models P$, it follows that $N(x) \models Q$, i.e.,

$$\forall x,\ P(x) \implies Q(N(x)).$$

Here, N plays the role of the program C, the predicate P specifies the admissible input set, and Q specifies the desired output property. If the query does not hold, there exists a counterexample x such that $x \models P$ and $N(x) \not\models Q$, which corresponds to a concrete violation of the Hoare-style specification.

Intuitively, the verification query $\langle N, P, Q\rangle$ holds when, for all inputs x satisfying the input predicate P, the corresponding output $N(x)$ also satisfies the predicate Q. Otherwise,



(A) Pred-1 (B) Pred-3

Figure 2.5: Examples: (A) is original image, and (B) is perturbed image

any input x violating this implication serves as a witness of property violation. In this thesis, we refer to P and Q as the pre-condition and post-condition, respectively. A simplified post-condition is $y \geq 0$ or $y > 0$, where y denotes the output of a neural network with a single output node. Let us see how the local robustness definition can be written as a verification query in the above form.

2.4 Local Robustness

As discussed in Chapter 1, neural networks are increasingly being used in safety-critical applications. However, they are also vulnerable to small perturbations in the input, meaning that even minor changes can cause the network to make incorrect decisions. As shown in Figure 2.5A, an image from the MNIST dataset [65], is correctly classified by the network as the digit 1. After introducing small modifications to the image, the altered version shown in Figure 2.5B is misclassified by the same network as the digit 3, even though the modified image still visually appears to represent the digit 1.

We formally define a specification, also referred to as a property, to capture such bugs or counterexamples in neural networks, or to verify that the network is free of such bugs around a particular image. In the literature [29, 34, 41, 33, 42, 124], this property is known as the *local robustness property*.

Definition 2. *For a neural network N, an input x, and an input perturbation value ϵ, N is* locally robust *if for every x' such that $dist(x', x) \leq \epsilon$, we have $\hat{N}(x') = \hat{N}(x)$.*

The above property ensures that a neural network's classification remains unchanged under small perturbations ϵ of the input. Here, ϵ is a user-defined parameter and *dist* is a distance metric, often taken in some L_p norm, where $p \in \{1, 2, \infty\}$. An input x' is

considered a counterexample if $dist(x', x) \leq \epsilon$ and $\hat{N}(x') \neq \hat{N}(x)$. The predicate

$$P = \bigwedge_{i=1}^{|I_0|} x^*[i] - \epsilon \leq x_{0i} \leq x^*[i] + \epsilon,$$

where $x^*[i]$ denotes the value of the i^{th} pixel in the image x^*, and represents the condition $\texttt{dist}(x^*, x) \leq \epsilon$ under the L_∞ norm. Similarly,

$$Q = \bigwedge_{\substack{i=1,\\ i \neq \hat{N}(x^*)}}^{m} N(x)[\hat{N}(x^*)] > N(x)[i]$$

expresses that the neuron corresponding to the correct class $\hat{N}(x^*)$ has a value greater than that of any other output neuron's value.

2.5 Solvers

To check whether $\langle N, P, Q \rangle$ holds, we formally use the function checkSAT in our algorithms, which determines the satisfiability of the given formula. Additionally, we also use the function maxSat in our algorithms. Both functions operate on quantifier-free formulae as input and return either SAT or UNSAT as output, depending on whether the formula is satisfiable or not.

The function checkSAT takes a logical formula as input and determines whether it is satisfiable. If the formula is unsatisfiable, the function returns UNSAT. Otherwise, if the formula is satisfiable, it returns SAT. In addition, we use the function getModel, which maps each variable to its corresponding satisfying value. The getModel function is invoked when checkSAT returns SAT, in order to extract the model, i.e., the satisfying assignment, for the variables involved.

The function maxSat takes two arguments as input: hardConstr and softConstr. The hardConstr is a Boolean formula of constraints, and softConstr is a set of constraints. The function maxSat satisfies the maximum number of constraints in softConstr while satisfying the hardConstr. The function maxSat returns SAT with the set of constraints satisfied in softConstr, or returns UNSAT if hardConstr fails to satisfy.

We can use either Mixed-Integer Linear Programming (MILP) solvers or Satisfiability Modulo Theories (SMT) solvers to use the above APIs. MILP solvers are optimization tools designed to handle problems that involve both continuous and discrete (integer or binary) variables subject to linear constraints and a linear objective function. These solvers, such

as CPLEX [125], Gurobi [40], and SCIP [126], use a combination of linear programming relaxation [127], branch-and-bound [128], cutting-plane [129], etc, techniques to efficiently explore the solution space. MILP solvers are particularly effective for problems that require finding an optimal configuration under linear relationships, such as resource allocation, scheduling, and network optimization. Their strength lies in producing globally optimal solutions for well-formulated linear problems, though the computational cost can grow exponentially with problem size and integer complexity.

SMT solvers, such as Z3 [96], CVC5 [130], and Yices [131], extend propositional satisfiability (SAT) solving by adding background theories like arithmetic [132], bit-vectors [133], arrays [134], and real numbers [119]. SMT solvers aim to determine whether logical formulas are satisfiable under these theories. They use a combination of SAT-solving techniques and theory-specific reasoning, often via the DPLL(T) framework [132]. SMT solvers are widely used in formal verification, model checking, and symbolic execution, where reasoning about logical constraints over variables is essential. Unlike MILP solvers, which focus on optimization, SMT solvers focus on satisfiability—checking if a model exists that satisfies given constraints.

Both MILP and SMT solvers handle constraints over variables, their primary goals and underlying methods differ. MILP solvers focus on optimization of linear objectives under linear constraints. SMT solvers, on the other hand, focus on logical satisfiability under rich theories, often involving non-linear or symbolic reasoning. MILP solvers rely heavily on numerical methods, whereas SMT solvers rely on logical inference. The work [122] shows that MILP-based solvers perform better than SMT-based solvers (e.g., Z3 [96]) in neural network verification tasks. This may be because SMT solvers answer queries with infinite precision, while MILP solvers operate with finite precision. For scalability, we also utilize an MILP solver, specifically, the Gurobi (v9.1) [40] optimizer to implement the checkSAT, getModel, and maxSat methods.

The nature of both the tools is to find a satisfying assignment rather than proving the properties. To determine whether $\langle N, P, Q \rangle$ holds, we check the satisfiability of the negated post-condition $\neg Q$. If $\neg Q$ is unsatisfiable, then the property $\langle N, P, Q \rangle$ holds, otherwise, if $\neg Q$ is satisfiable, $\langle N, P, Q \rangle$ does not hold, and the corresponding satisfying assignment provides a witness or counterexample to the violation.

Intuitively, $\neg Q$ encodes a misclassification condition. To detect such counterexamples, a solver checks whether both P and $\neg Q$ can be satisfied simultaneously. The verification

query is satisfiable if and only if the neural network N is not locally robust at x^* with respect to the perturbation bound ϵ.

Chapter 3

Related Work

In this thesis, we consider the problem of effective and efficient verification of neural networks. Our main focus includes the counterexample-guided abstraction refinement (CEGAR) techniques for neural network verification, as well as various robustness properties of neural networks. Since the area is vast and has been extensively studied in recent times, we focus on some of the recent advances in the closely related areas. We first present various adversarial attacks that are used to generate adversarial examples in Section 3.1. Next, we present various incomplete and complete techniques for neural network verification in Section 3.2.1 and Section 3.2.2, respectively. The complete techniques are supported by various branching and refinement heuristics that are discussed in Section 3.2.3 and Section 3.2.4. Next, we present various related works that take confidence values into account for robustness verification in Section 3.3. Next, we discuss the works that add layers in the neural networks to encode various postconditions in Section 3.4. Finally, we present the works that aim to learn robust neural networks during training in Section 3.5. In addition to the works related to neural network verification, we would also like to refer the comprehensive survey papers [135, 136]. Our methods have built upon prior techniques in hardware and software verification, for which we refer readers to the survey [115, 116, 117, 119]. We leave the related work on our explainability technique for reinforcement learning to Chapter 8.

3.1 Generating adversarial examples

Adversarial examples are inputs to machine learning models that have been intentionally perturbed to cause the model to make a mistake. These perturbations are often impercept-

ible to humans but can lead to significant misclassifications by the model. The generation of adversarial examples has been a topic of extensive research, with various techniques developed to create such examples.

The fast gradient sign method (FGSM), introduced by [10], is one of the earliest and simplest techniques for generating adversarial examples for neural networks. FGSM operates in a white-box setting by computing the gradient of the loss with respect to the input image and perturbing the input in the direction of the sign of this gradient, scaled by a small parameter α. Formally, the adversarial image x_{adv} is generated as $x_{adv} = x + \alpha \cdot \text{sign}(\nabla_x J(\theta, x, y))$, where J denotes the loss function, θ represents the network parameters, x is the original input, y is the true label, ∇_x is the gradient with respect to the input, and α is the step size. This one-step method is known for its computational efficiency and for demonstrating that small, human-imperceptible perturbations can cause significant misclassifications. The projected gradient descent (PGD) attack [11] generalizes this approach by applying multiple small steps of gradient ascent on the loss function, constrained within a predefined perturbation norm. After each iteration, the perturbed input is projected back onto the allowed ϵ-distance around the original input to maintain boundedness.

Introduced in [14], DeepFool is a more sophisticated white-box attack designed to compute the minimal perturbation required to change the classification decision of a neural network. DeepFool iteratively perturbs the input image in the direction that most rapidly moves it toward the decision boundary of the neural network, effectively applying perturbations orthogonal to this boundary. It assumes that the classifier behaves approximately linearly in the local neighborhood of the input and projects the input in each iteration to the nearest point on the decision boundary. The process continues until the prediction of the neural network changes, ensuring that the obtained adversarial perturbation is nearly minimal. This attack is particularly valuable for robustness evaluation, as it exposes the geometric vulnerability of the classifier and serves as a strong benchmark for assessing model robustness.

Introduced in [12], the boundary attack is a black-box adversarial attack that requires access only to the predicted class labels of the target model. It starts with a large, misclassified perturbation and iteratively refines it by reducing the perturbation magnitude while preserving misclassification. The objective is to make the perturbed image as close as possible to the original input while maintaining misclassification. Unlike gradient-based

methods, it does not rely on gradients or logit values and instead depends solely on the prediction of the network. This characteristic makes the Boundary Attack particularly useful for evaluating the robustness of deployed models with restricted internal access, demonstrating the vulnerability of black-box classifiers to adversarial manipulation.

Introduced in [13], the HopSkipJump attack can operate in either a white-box or black-box setting, depending on the available model access. The attack consists of two main phases: boundary search and gradient update. It begins with a highly perturbed misclassified image and performs a binary search toward the original image until it reaches the decision boundary. Once the boundary is identified, the algorithm minimizes the perturbation distance between the original and adversarial images by updating the input in the direction of the estimated gradient. In the white-box setting, this gradient is computed using backpropagation, whereas in the black-box setting, it is estimated through repeated model evaluations on multiple inputs. This flexible design allows the HopSkipJump attack to operate effectively even under restricted access conditions.

3.2 Neural network local robustness verification

Since adversarial example generation techniques are not exhaustive, researchers have developed formal verification techniques that can provide guarantees on the absence of adversarial examples within a specified neighborhood of an input. These techniques can be broadly classified into incomplete and complete methods.

3.2.1 Incomplete Techniques

DeepPoly [34] maintains both lower and upper linear constraints, as well as lower and upper bounds, for every neuron in the network. It combines the interval (box) abstract domain [121] with the polyhedral abstract domain [137], thereby balancing precision and efficiency. CROWN [33] uses the same underlying abstraction principle as DeepPoly, with the main distinction being in implementation: CROWN is optimized for GPU acceleration, whereas DeepPoly primarily runs on CPU. A detailed explanation of DeepPoly is provided in Section 4.3. The work [138] further improved the DeepPoly by advancing the technique for parallelization and acceleration on GPU, called GPUPoly. kPoly [41] considers a group of k neurons at once to generate the constraints and compute the bounds of neurons. Further improvements are made in a tool called PRIMA [139], which also performs multi-neuron convex relaxations of nonlinear layers. Their algorithms have polynomial complexity, yield fewer constraints, and minimize precision loss. The work [140] also

provides tighter convex relaxations by jointly relaxing multiple neurons at once. To enable scalability, the authors introduce two polynomial-time algorithms: OptC2V, an LP-based verifier that yields tight bounds, and FastC2V, a faster propagation-based verifier. Notably, FastC2V generalizes several existing bound propagation techniques, including Fast-Lin [31], DeepPoly, and CROWN, providing a unified and more expressive framework for convex outer approximations. The approach deepSRGR [38] removes the spurious region by taking the constraints of each neuron with the negation of the property and using the MILP solvers, more specifically, Gurobi [40] optimizer to tighten the bounds of each neuron. They need to call the MILP solver repeatedly for each neuron, so it is not scalable. The work [29] employs the Zonotope [141] abstract domain to soundly approximate the behavior of neural networks. A Zonotope represents sets as affine forms with a center and a collection of generator noise symbols, which enables it to track linear correlations between variables rather than treating them as independent interval bounds. A follow-up work by the same authors, RefineZono [30], refines the abstractions generated by DeepZ using an MILP solver, resulting in tighter bounds but remaining incomplete. The framework AI^2 [28] is recognized as the first scalable abstract-interpretation-based technique capable of verifying neural networks with up to 53K neurons. It combines the Box [121] and Zonotope abstract domains to propagate abstract bounds layer by layer through the network. This approach is specifically designed for networks employing ReLU and MaxPool activation functions.

Some researchers observed that the abstractions of activation functions have choices, which can impact the precision of the verification, and introduced LP-based techniques to optimize the abstraction of activation functions. α-CROWN [42] optimizes the bounds of neural network neurons by jointly optimizing the slopes (α values) of the constraints associated with each neuron. The approach is further extended into a complete verification technique by performing branching on selected neurons in batches and bounding each branch using the incomplete verification tool based on linear relaxation perturbation analysis (LiRPA) [71]. LiRPA is a library for automatically deriving and computing bounds, which underlies methods such as CROWN and DeepPoly. The key contribution is a generalization of LiRPA from feed-forward neural networks to arbitrary computational graphs defined in PyTorch. A dual approach to scalable verification of deep networks [27] reformulates the verification problem as an optimization task aimed at identifying the largest possible violation of a given verification property. The optimization problem is

solved using Lagrangian relaxation [142], which yields bounds on the worst-case violation of the property. The technique operates in an anytime fashion, meaning it can be interrupted at any stage while still providing valid bounds. The method in [31] provides a certified lower bound on the minimum adversarial distortion. However, it only focuses on ReLU activation functions. The authors propose two techniques—Fast-Lin and Fast-Lip. Fast-Lin derives certified bounds by applying per-neuron upper and lower linear relaxations of the ReLU function, while Fast-Lip estimates robustness by computing an upper bound on the local Lipschitz constant of the network's logit outputs. The method in [18] computes the output range of a neural network using a combination of gradient-based local search and linear programming (LP) formulation. The local search is used to escape local minima, while an LP solver is employed to verify the global minima, ensuring accurate estimation of the output bounds.

In an orthogonal direction, the approach [143] reduces the size of a neural network by merging similar neurons. The similarity between neurons is determined using clustering techniques. Specifically, a set of input samples is propagated through the network layer-by-layer. At each layer, clustering is applied to group neurons with similar activation patterns, and neurons belonging to the same cluster are merged. The authors also provide an upper bound on the abstraction error introduced by this merging procedure. This line of work is commonly referred to as *semantic abstraction-based merging*. In contrast, the work of [37] also merges multiple neurons into a single neuron but provides explicit over-approximation guarantees on the resulting abstraction. Similarly, [144, 145] merges neurons while maintaining bounds on the weights and ensuring sound over-approximation guarantees. These approaches are typically known as *syntactic abstraction-based merging*. Furthermore, the recent work [146] combines both semantic and syntactic abstraction principles to merge neurons in a unified framework.

3.2.2 Complete Techniques

Incomplete techniques can provide sound guarantees, but may fail to verify certain properties due to their inherent approximations. To address this limitation, researchers have developed complete verification techniques that can definitively determine whether a property holds for a neural network or not. These methods typically involve implicitly or explicitly splitting the problem into smaller subproblems and applying the above abstractions in each of the subproblems.

ReLUplex [20] is a decision procedure for verifying neural networks with ReLU ac-

tivation functions. It extends the classical simplex algorithm by incorporating symbolic constraints that capture the piecewise-linear nature of ReLU. The method begins by relaxing all ReLU constraints and incrementally reinstates them during the solving process based on heuristic guidance. This incremental refinement allows ReLUplex to efficiently explore the feasible region while maintaining the correctness of the verification outcome. The work [22] combines SAT solving with linear programming and uses a linear approximation of the overall behavior of the network to verify feed-forward neural networks with piecewise-linear activation functions, such as ReLU and Maxpool.

MILP-based techniques have been widely adopted for neural network verification due to their ability to provide exact solutions for piecewise-linear networks. MIPVerify [122] introduces a MILP-based formulation for neural networks with ReLU and Max activation functions. The approach relies on using incomplete verification methods to compute sound bounds, followed by an exact MILP encoding of the network constraints. The authors demonstrate that, with these bounds in place, MILP-based solvers can significantly outperform SMT-based solvers such as ReLUplex [20], particularly in terms of scalability. The paper [16] introduces an MILP-based method for deciding reachability in feed-forward ReLU neural networks by formally reducing the reachability question to the feasibility of a mixed-integer linear program (MILP). The paper [17] discusses the peculiarities of MILP encodings for neural networks containing ReLU and Maxpool functions. The authors also study the impact of bound tightening on the solving time and demonstrate that applying a bound tightening algorithm significantly reduces the computation time. The work [19] computes the maximum perturbation bound that a network can tolerate, i.e., the largest perturbation under which the network continues to classify the input into the same class. The authors formulate this problem as an MILP optimization problem, which can be solved using standard MILP solvers.

Most complete verification methods are based on either SMT or MILP solving and are limited to CPU execution. β-CROWN [24] accelerates the verification process on GPUs by extending the CROWN bound-propagation framework. It introduces optimizable per-neuron multipliers, denoted as β, to encode branching constraints in the branch-and-bound verification procedure, thereby improving both efficiency and scalability. When a neuron is split (e.g., enforcing a ReLU to be either active or inactive), instead of solving a separate LP to enforce the constraint, the method incorporates the split through backward linear relaxation with Lagrangian multipliers. This achieves bounds that are as tight

as those produced by LP-based methods, while retaining GPU efficiency. As a result, β-CROWN enables scalable and complete verification for networks that are otherwise difficult to verify using traditional LP-based approaches. The work [26] introduces GCP-CROWN, a generalized bound propagation method that integrates arbitrary cutting plane constraints into neural network verification. Unlike existing approaches, it supports relaxed integer variables and enables the use of cutting planes generated by mixed integer programming (MIP) solvers to tighten convex relaxations. By combining GPU-accelerated bound propagation with CPU-based MIP solvers in parallel, this method achieves strong verification performance.

3.2.3 Branching heuristics

In addition to the above techniques, there has been significant research on developing effective branching strategies to enhance the efficiency of complete verification methods. Branch and Bound (BaB) is a widely used complete verification technique that systematically explores the search space by dividing the verification problem into smaller subproblems, either by branching on the input space or on hidden neurons, and bounding each subproblem to prove the desired property. The efficiency of BaB heavily depends on the branching strategy used to select the next subproblem for exploration. Many state-of-the-art verifiers, including $\alpha\beta$-CROWN, employ different BaB strategies to determine the most promising subproblem to branch on. Several works [35, 57] have proposed various branching strategies to improve the efficiency of BaB in neural network verification. Broadly, there are two types of branching approaches: branching on the input space [57] and branching on activation functions [57, 147]. We discuss some of the well-known strategies below.

The early work [20] utilizes an SMT core to perform branching on ReLU activations. The method implemented in Neurify [35] computes a gradient-based score for each ReLU and selects the one with the highest score for branching. The work [57] proposes several branching strategies within the BaB framework, including both input-space and activation-function branching. For input-space branching, they introduce BaB and BaBSB, the former splits based on gradient information, while the latter bisects the input dimension along the longest edge. Both approaches employ an LP solver to compute bounds for each branch. For activation-function branching, they propose ReLUBaB, which prioritizes unresolved ReLU nodes from the first activation layer, and BaBSR, which assigns a score to each ReLU based on its current bounds to determine branching priority.

The work [147] employs two graph neural networks (GNNs) that treat the neural network under verification as a graph input and perform forward and backward passes through the GNN layers. One GNN is trained to mimic the behavior of a strong branching heuristic, while the other computes a feasible dual solution of the convex relaxation, providing a valid lower bound during verification. The work [148] employs symbolic IBP to compute bounds and determine the split node by evaluating the indirect effect that a split has on the relaxation of its successor nodes. To further enhance performance, the splitting constraints are encoded as linear programming (LP) constraints, allowing tighter bound computation. The work [58] proposes a branching heuristic called BaB-FSB, which selects branching ReLU nodes by evaluating the impact of a split on both child subproblems using gradient information. The method employs Lagrangian relaxation to compute bounds in the dual space, thereby improving the efficiency and precision of the branch-and-bound verification process. Finally, the work [149] first applies an LP relaxation and invokes an LP solver to obtain the optimal solution. Based on this solution, it selects the ReLU neuron to branch on using a solution-aware branching technique. Additionally, the method employs an MILP solver after each split to tighten the bounds of individual neurons.

3.2.4 Refinement-Based Techniques

In addition to branching strategies, several works have focused on developing effective refinement techniques to enhance the precision of neural network verification. These methods typically start with an abstraction of the network and iteratively refine it based on the verification results. We discuss some of the well-known refinement techniques below.

In the work [35], the authors propose a symbolic interval analysis technique to compute output bounds of neural networks. When the method fails to prove a given property, it performs refinement by bisecting the input ranges. The selection of the input variable for bisection is guided by the gradient of the network output with respect to the input, allowing the refinement process to focus on the most influential input dimensions. This work was further improved in [36], where the authors combine linear relaxation with symbolic interval analysis to compute tighter output bounds. Additionally, they extend the refinement process to hidden nodes, thereby enhancing verification precision. This paper [150] synergistically combines gradient-based optimization for counterexample search with abstraction-based proof search to achieve a sound and complete decision procedure. Furthermore, it employs a data-driven approach to learn a verification policy that guides the abstract interpretation process during proof search, thereby improving efficiency

and scalability. The work [151] first performs linear relaxation abstraction for each non-linear activation in the neural network. It then applies several refinement heuristics, such as randomly selecting neurons, refining based on approximation error, using layer-biased strategies that prioritize earlier layers, and introducing a novel technique that considers the influence of hidden neurons on the output neurons to guide the refinement process effectively. The work [152] presents an abstraction-refinement framework specifically designed for verifying convolutional neural networks (CNNs). In the abstraction phase, convolutional connections are initially removed while preserving an over-approximation of the network's behavior. During the refinement phase, the convolutional connections are incrementally reintroduced to progressively tighten the approximation and improve verification precision.

The work [37] presents a counterexample-guided abstraction-refinement framework for neural network verification. The approach starts by constructing a simplified abstract model through neuron merging, where the merging strategy ensures an over-approximation of the original network. If verification on the abstract model yields inconclusive results (i.e., produces a spurious counterexample), the framework refines the abstraction by splitting the merged neurons back into smaller groups or individual neurons. This refinement process does not necessarily eliminate the spurious counterexample in a single step; instead, it progressively refines the abstraction through iterative neuron splitting. The work [153] employs two abstraction strategies—merging neurons and removing neurons, while preserving an over-approximation of the network's behavior. During the refinement phase, they apply two complementary strategies, splitting neurons and recovering previously removed neurons based on the dependency graph. The work [154] enhances the CEGAR-based approach by reducing the number of verification calls. The key intuition is that, in the initial stages, the abstraction is typically very coarse, therefore, instead of directly invoking the verification query, the method first employs adversarial attack techniques to quickly search for counterexamples. Only if these attacks fail to find any counterexample, the verification query is invoked, thereby improving efficiency.

This thesis also includes works based on classification confidence and layer-based encoding. In the following two sections, we discuss related work from the perspective of formal methods.

3.3 Confidence-based analysis

The work [114] introduced SOFTMAX-based confidence and its verification in the global robustness setting. The key intuition behind using confidence is to avoid boundary cases in the input space. They approximate confidence by computing bounds on the output based on the input bounds, achieved by first approximating the SOFTMAX function using a SIGMOID function and then further approximating the SIGMOID itself. The work [113] introduces convex bounds on the SOFTMAX function, providing both lower and upper bounds compatible with convex optimization. However, both techniques result in highly complex constraints that can only be handled by constraint-based solvers and scale only to networks with a few hundred parameters. In contrast, our approximation allows the use of any solver and scales efficiently to large networks, including ImageNet-1k [8] classifiers. The work [155] employs confidence for the equivalence verification of two neural networks. Similar to [114], they use confidence to avoid boundary cases and apply a similar approximation technique by comparing the logit value of the expected class with the logit values of the remaining classes. The work [156] probabilistically estimates the global robustness of neural networks by sampling input points in regions where the model exhibits high-confidence predictions and performing local robustness analysis around these sampled inputs to approximate the overall robustness measure.

3.4 Layer-based encoding

The paper [15] also introduces a conversion from a 3-SAT formula to neural networks to prove the NP-completeness of the verification problem. They assume that input variable values are either close to 0 or 1, and that the property formula is expressed in conjunctive normal form (CNF). In contrast, our method operates directly on linear real arithmetic inputs and supports arbitrary formulas of any size or structure. The work [157] appends user-specified requirements as an additional layer to the neural network and trains the network to satisfy these requirements; however, their approach is limited to propositional formulas and is not applied to post-training robustness verification. The framework [158] simplifies neural networks by statically replacing unsupported operations with supported ones, thereby enabling the underlying verifier to process them. It also decomposes complex properties into smaller sub-properties that can be individually verified by the backend verifier.

3.5 Provable Defenses Against Adversarial Examples

In addition to verification techniques, researchers have also focused on the *correct-by-construction* design of robust neural networks during training. These approaches aim to train neural networks that are provably robust against adversarial perturbations within a specified norm-bound. In this section, we discuss some of the well-known methods in this direction.

The works [32, 159, 160, 161, 162] introduce methods for training neural networks that are *provably* robust to adversarial perturbations. Similar to adversarial training, these methods consider a neighborhood around each training input. However, instead of sampling adversarial examples, they construct a convex outer polytope around the input and propagate this convex region layer by layer by relaxing the non-linear activation functions. The bounds at the output layer are then used to compute a certified robustness loss, and backpropagation updates the network parameters accordingly. Propagating a convex region for every training example is computationally expensive, which limits the scalability of these methods. To address this, the works [163, 164] demonstrate the effectiveness of simple Interval Bound Propagation (IBP) techniques, which scale to larger models. Although IBP generally produces weaker bounds than polytope-based relaxations, the authors show that with suitable loss formulations and hyperparameter scheduling, the network can be trained in a way that tightens the IBP bounds during training. The work [165] combines interval bound propagation (IBP) as a forward bound propagation method with CROWN [33] as a backward bound propagation technique to compute tight bounds for training verifiably robust neural networks. Their approach is GPU-accelerated and scales efficiently to large networks. The work [166] proposes a more efficient approach for propagating convex regions based on semidefinite programming (SDP). In contrast to earlier relaxation techniques, which often yield loose bounds, the SDP-based relaxation is significantly tighter and provides stronger robustness guarantees. The work [39] introduces an abstraction-refinement framework for training neural networks that are provably correct with respect to specified properties. Similar to provable robustness training, it constructs a loss function over the output region; however, the loss is also influenced by the input region and by abstractions over hidden neurons. To mitigate the impact of these abstractions on the loss, the method performs refinement by bisecting the input range, thereby improving the precision of the training process.

In the following chapters, we will present an abstraction–refinement–based complete

verification method that guarantees progress in each iteration by leveraging the bounds computed by incomplete verifiers. We will analyze the counterexamples generated by different verifiers, discussed earlier, to identify false positives. Furthermore, we will propose a family of local robustness properties that make diverse use of confidence values, as compared to the works discussed earlier. In addition, we will introduce an approximation scheme that enables the development of an efficient confidence-aware robustness verification tool.

Chapter 4

Using Counterexamples to Improve Robustness Verification in Neural Networks

The robustness verification of neural networks has become a critical area of research, particularly for safety-critical applications where ensuring the reliability of neural networks is paramount. We will begin by presenting our counterexample-guided abstraction refinement (CEGAR)-based technique to improve the precision of abstraction-based robustness verification methods of DNNs. We focus on the class of abstraction-based methods that use convex polyhedra to over-approximate the behavior of neural networks, such as DeepPoly [29] and kPoly [41]. Earlier refinement techniques for these methods have relied on heuristics to identify neurons for refinement, without guarantees of progress [24, 25, 57, 58, 59]. In contrast, our CEGAR-based technique systematically marks neurons responsible for spurious counterexamples and guarantees progress in each refinement iteration. We use MILP-based maxsat queries to mark the neurons, and we present two variations of our technique that differ in the order of the analysis of neurons.

Our technique outperforms, to the best of our knowledge, all existing refinement strategies based on DeepPoly. Moreover, we identify a class of benchmarks coming from adversarially trained networks, where state-of-the-art tools do not work well because of the ineffectiveness of certain preprocessing steps (e.g., PGD attack [11]). Finally, our implementation is able to verify several benchmarks that are beyond the reach of state-of-the-art tools such as $\alpha\beta$-CROWN [60] and marabou [62].

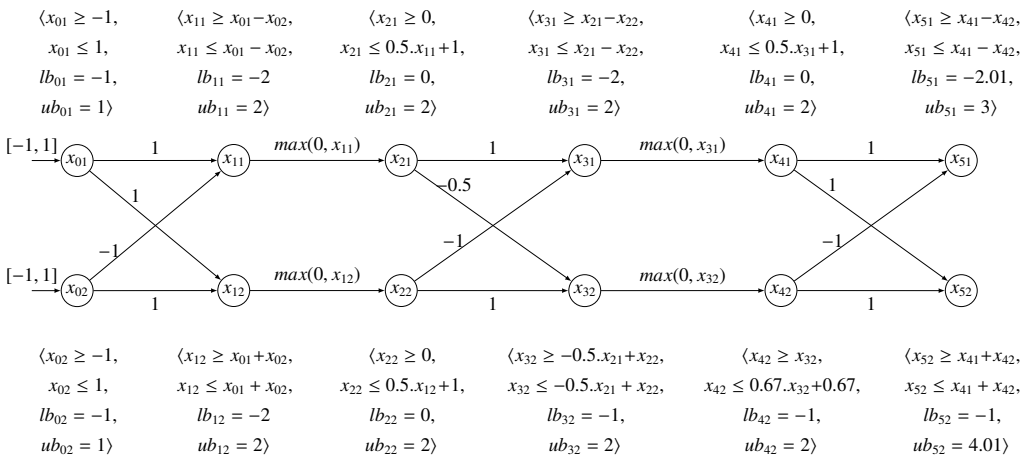


Figure 4.1: Example of a neural network

In this chapter, we first present a motivating example to illustrate our refinement technique and its variations in Section 4.1. We then formally describe the problem and provide an overview of the DeepPoly abstraction method, which our method iteratively refines in Sections 4.2 and 4.3. Next, in Section 4.4, we detail our CEGAR-based algorithms, including the two variations for identifying marked neurons. We also discuss theoretical results comparing the two variations of our technique. Finally, we present experimental results in Section 4.5, demonstrating the effectiveness of our approach, followed by a conclusion summarizing our contributions and potential future work in Section 4.6.

4.1 Motivating Example

Consider the neural network depicted in Figure 4.1, which consists of one input layer, two hidden layers, and one output layer. Each hidden layer is divided into two sub-layers: an Affine layer followed by a ReLU layer. This results in a total of six layers, as illustrated in Figure 4.1. Each layer contains two neurons. The weight of each edge is annotated on the edge itself, and the bias of every neuron is set to 0. Our verification goal is to ensure that, for all inputs $x_{01}, x_{02} \in [-1, 1]$, the outputs satisfy the condition $x_{51} \leq x_{52}$. Our approach extends DeepPoly [34]. DeepPoly maintains one upper and one lower constraint and an upper and lower bound for each neuron. For a neuron of the affine layer, the upper and lower constraints are the same, which is the weighted sum of the input neurons, i.e., x_{11}'s upper and lower constraints are $x_{01} - x_{02}$. For an activation neuron, the upper and lower

expression is computed using triangle approximation [34]. To verify the property $x_{51} \leq x_{52}$, DeepPoly creates a new expression $x_{61} = x_{51} - x_{52}$ and computes the upper bound of x_{61}. The upper bound of x_{61} should not be greater than 0. DeepPoly computes the upper bound of x_{61} by back substituting the expression of x_{51} and x_{52} from the previous layer. They continue back substituting until only input layer variables are left. The process of back substitution is shown in Equation 4.1. After back substitution, the upper bound of x_{61} is computed as 2, which is greater than 0, hence, the DeepPoly fails to verify the property.

$$
\begin{aligned}
x_{61} &\leq x_{51} - x_{52} & \qquad x_{61} &\leq x_{21} - 2x_{22} \\
x_{61} &\leq x_{41} - x_{42} - x_{41} - x_{42} & \qquad x_{61} &\leq 0.5x_{11} + 1 \\
x_{61} &\leq -2x_{42} & \qquad x_{61} &\leq 0.5x_{01} + 0.5x_{02} + 1 \\
x_{61} &\leq -2x_{32} & \qquad x_{61} &\leq 2 \\
x_{61} &\leq -2(-.5x_{21} + x_{22})
\end{aligned}
\tag{4.1}
$$

There are two main reasons for the failure of DeepPoly. First, it cannot maintain the full correlation between neurons. In this example, if neuron $x_{12} = 2$, then neuron x_{11} must always be 0, since x_{12} attains the value 2 only when $x_{01} = 1$ and $x_{02} = 1$, which implies $x_{11} = 0$. However, during the DeepPoly analysis, x_{11} and x_{12} may fail to simultaneously attain the values 0 and 2, respectively. Second, it uses triangle approximation on ReLU neurons. We take the conjunction of upper and lower expressions of each neuron with the negation of the property, as shown in Equation 4.2, and use the MILP solver to check satisfiability, thus addressing the first issue.

$$
\begin{aligned}
-1 &\leq x_{01} \leq 1 & \qquad -1 &\leq x_{02} \leq 1 \\
x_{01} - x_{02} &\leq x_{11} \leq x_{01} - x_{02} & \qquad x_{01} + x_{02} &\leq x_{12} \leq x_{01} + x_{02} \\
0 &\leq x_{21} \leq 0.5x_{11} + 1 & \qquad 0 &\leq x_{22} \leq 0.5x_{12} + 1 \\
x_{21} - x_{22} &\leq x_{31} \leq x_{21} - x_{22} & \qquad -0.5x_{21} + x_{22} &\leq x_{32} \leq -0.5x_{21} + x_{22} \\
0 &\leq x_{41} \leq 0.5x_{31} + 1 & \qquad x_{32} &\leq x_{42} \leq 0.67x_{32} + 0.67 \\
x_{41} - x_{42} &\leq x_{51} \leq x_{41} - x_{42} & \qquad x_{41} + x_{42} &\leq x_{52} \leq x_{41} + x_{42} \\
x_{51} &> x_{52} \text{ (negation of property)}
\end{aligned}
\tag{4.2}
$$

The second issue can be resolved either by splitting the bound at zero of the affine node or by using the exact encoding [122], instead of the triangle approximation. But both solutions increase the problem size exponentially in terms of ReLU neurons, and this results in a huge blowup if we repair every neuron of the network.

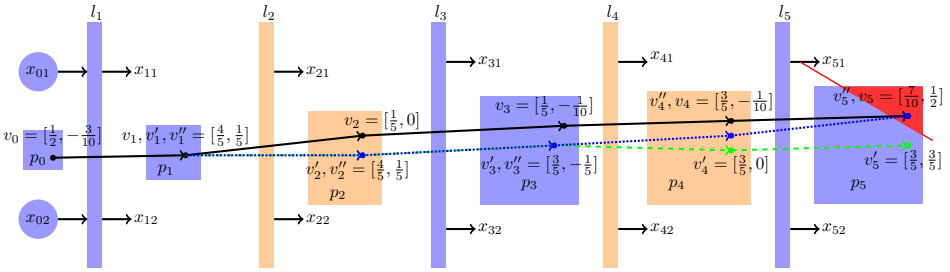


Figure 4.2: Pictorial representation of our approach on example in figure 4.1

So, the main hurdle toward efficiency is to find the set of important neurons (we call these *marked neurons*), and only repair these. For this, we crucially use the satisfying assignment obtained from the MILP solver. For instance, a possible satisfying assignment of Equation 4.2 is in Equation 4.3.

$$\begin{aligned} x_{01} = \frac{1}{2},\ x_{02} = -\frac{3}{10},\ x_{11} = \frac{4}{5},\ x_{12} = \frac{1}{5},\ x_{21} = \frac{1}{5},\ x_{22} = 0, \\ x_{31} = \frac{1}{5},\ x_{32} = -\frac{1}{10},\ x_{41} = \frac{3}{5},\ x_{42} = -\frac{1}{10},\ x_{51} = \frac{7}{10},\ x_{52} = \frac{1}{2} \end{aligned} \tag{4.3}$$

We execute the neural network with the inputs $x_{01} = \frac{1}{2}, x_{02} = -\frac{3}{10}$ and get the values on each neuron as shown in Equation 4.4.

$$\begin{aligned} x'_{01} = \frac{1}{2},\ x'_{02} = -\frac{3}{10},\ x'_{11} = \frac{4}{5},\ x'_{12} = \frac{1}{5},\ x'_{21} = \frac{4}{5},\ x'_{22} = \frac{1}{5}, \\ x'_{31} = \frac{3}{5},\ x'_{32} = -\frac{1}{5},\ x'_{41} = \frac{3}{5},\ x'_{42} = 0,\ x'_{51} = \frac{3}{5},\ x'_{52} = \frac{3}{5} \end{aligned} \tag{4.4}$$

Then we observe that the output values $x'_{51} = \frac{3}{5}, x'_{52} = \frac{3}{5}$ satisfy the property, so, the input $x_{01} = \frac{1}{2}, x_{02} = -\frac{3}{10}$ is a spurious counterexample. The question is to identify the neuron whose abstraction leads to this imprecision and remove the abstraction from those neurons.

Although several approaches [57, 58, 59] identify important neurons by leveraging local information such as weights or gradients, in this work, we utilize spurious counterexamples to determine important neurons. We identify important neurons starting from the initial layers—a technique we refer to as *forward marking*. However, this approach can become inefficient when the important neurons are located in deeper layers. To address this limitation, we propose a method that identifies important neurons starting from the last layers, which we refer to as *backward marking*.

4.1.1 Forward Marking

Let us first discuss the forward marking method. To illustrate our method of identifying the neurons whose abstraction leads to imprecision, let us refer to Figure 4.2. In the figure, p_i represents the abstract constraint space in layer l_i, while the solid black line denotes the spurious counterexample depicted in Equation 4.3. On the other hand, the dashed green line represents the exact execution of the input point of the spurious counterexample, as denoted by Equation 4.4.

The objective is to make the solid black line as close as possible to the dashed green line from the first layer to the last layer, while keeping the first and last points fixed, i.e., $x_{01} = \frac{1}{2}$, $x_{02} = -\frac{3}{10}$, and $x_{51} = \frac{7}{10}$, $x_{52} = \frac{1}{2}$. If, at a given layer, the black point coincides exactly with the green point, then that layer does not generate a spurious counterexample. Otherwise, we identify the components of the black point that differ from the corresponding components of the green point and mark the associated neurons for further refinement. The closest line to achieving this goal is represented by the dotted blue line, which is also the abstract execution but exhibits the highest closeness to the exact execution of the spurious counterexample. In this context, v_i refers to the vector of values of neurons in layer l_i of the solid black line, while v_i' and v_i'' represent the vectors of the dashed green and dotted blue lines, respectively.

The green and black points are the same for the input layer, i.e., $[\frac{1}{2}, -\frac{3}{10}]$. On the first affine layer, l_1, the black point v_1 is the same as the green point v_1' since the affine layer does not introduce any spurious information. For l_2, we try to make v_2'' close to v_2', such that v_2'' reaches to the v_5. We do that by encoding them as soft constraints (i.e., $\{x_{21} = \frac{4}{5}, x_{22} = \frac{1}{5}\}$) while maintaining that the rest of the hard constraints are satisfied (see Equation 4.5), e.g., input points $v_0 = v_0''$ and output points $v_5 = v_5''$ remain the same. We mark the neurons of the layer where the dotted blue line starts diverging from the dashed green line, i.e., l_4. The divergence we find by the maxSat query. If maxSat returns all the soft constraints as satisfied, it means the blue point becomes equal to the green point. If maxSat returns partial soft constraints as satisfied, we mark the neurons whose soft constraints are not satisfied. In our example, maxSat returns soft constraints $\{x_{21} = \frac{4}{5}, x_{22} = \frac{1}{5}\}$ are satisfied, which means the blue point becomes equal to the green point, and no marked neuron is found in layer l_2. Since no marked neurons are found in l_2 and the next layer l_3 is an affine layer, which does not introduce any imprecision. We move to layer l_4 and run the maxSat query with soft constraints $\{x_{41} = \frac{3}{5}, x_{42} = 0\}$ and hard constraints reduced as in

Equation 4.6. The soft constraint $\{x_{41} = \frac{3}{5}\}$ is satisfied, and the soft constraint of $\{x_{42} = 0\}$ could not be satisfied, so we mark x_{42}. We optimize the dotted blue line to be close to the dashed green line while also resulting in as few marked neurons as possible.

$$\begin{gathered}
x_{01} = \frac{1}{2} \wedge x_{02} = -\frac{3}{10} \\
x_{11} = x_{01} - x_{02} \wedge x_{12} = x_{01} + x_{02} \\
0 \leq x_{21} \leq 0.5x_{11} + 1 \wedge 0 \leq x_{22} \leq 0.5x_{12} + 1 \\
x_{31} = x_{21} - x_{22} \wedge x_{32} = -0.5x_{21} + x_{22} \\
0 \leq x_{41} \leq 0.5x_{31} + 1 \wedge \leq x_{42} \leq 0.67x_{32} + 0.67 \\
x_{51} = x_{41} - x_{42} \wedge x_{52} = x_{41} + x_{42} \\
x_{51} = \frac{7}{10} \wedge x_{52} = \frac{1}{2}
\end{gathered} \tag{4.5}$$

Once we have x_{42} as the marked neuron, we use an *MILP-based approach*, and add the exact encoding [122] of the marked neuron x_{42}, in addition to the constraints in Equation 4.2. and check the satisfiability, now it becomes UNSAT, hence, the property verified.

$$\begin{gathered}
x_{01} = \frac{1}{2} \wedge x_{02} = -\frac{3}{10} \wedge x_{11} = \frac{4}{5} \wedge x_{12} = \frac{1}{5} \wedge x_{21} = \frac{4}{5} \wedge x_{22} = \frac{1}{5} \wedge x_{31} = \frac{3}{5} \wedge x_{32} = -\frac{1}{5} \\
0 \leq x_{41} \leq 0.5x_{31} + 1 \wedge \leq x_{42} \leq 0.67x_{32} + 0.67 \wedge x_{51} = x_{41} - x_{42} \wedge x_{52} = x_{41} + x_{42} \\
x_{51} = \frac{7}{10} \wedge x_{52} = \frac{1}{2}
\end{gathered} \tag{4.6}$$

This approach requires two MAXSAT calls, both involving the full network's constraints. In the next subsection, we show that the same marked neurons can be obtained in a more efficient manner.

4.1.2 Backward Marking

In this approach, we start finding mark neurons from the last layer to the first layer, intuitively giving priority to the deeper layer first. Since the output layer (l_5) is an affine layer, it does not introduce any imprecision. The layer l_4 is an activation layer, so we start finding marked neurons from layer l_4. Since v_0 is a spurious counterexample, and v'_3, v'_4 are reachable from v_0. We check if v_5 is reachable from v'_3 by checking the satisfiability of Equation 4.7, its satisfiability implies v_5 is reachable from v'_3. If v_5, which violates the property, is reachable from v'_3, which is an exact execution of the neural network on input

v_0, then some neurons from layer l_4 to l_5 contribute to the spurious counterexample. The layer l_5 is an affine layer which does not introduce any imprecision, so the neuron must be from layer l_4. We build the soft constraints in layer l_4, which are $\{x_{41} = \frac{3}{5}, x_{42} = 0\}$, the hard constraints are in Equation 4.7. The soft constraint $\{x_{41} = \frac{3}{5}\}$ is satisfied, whereas $\{x_{42} = 0\}$ is not satisfied. Therefore, we mark x_{42} as a neuron for further refinement.

$$\begin{gathered} x_{31} = \frac{3}{5} \wedge x_{32} = -\frac{1}{5} \\ 0 \leq x_{41} \leq 0.5x_{31} + 1 \wedge \leq x_{42} \leq 0.67x_{32} + 0.67 \\ x_{51} = x_{41} - x_{42} \wedge x_{52} = x_{41} + x_{42} \\ x_{51} = \frac{7}{10} \wedge x_{52} = \frac{1}{2} \end{gathered} \tag{4.7}$$

Although it appears that two solver calls are needed, one to check reachability and another for MaxSat, in practice, a single MaxSat call suffices for both purposes. Specifically, if the MaxSat call returns UNSAT, then v_5 is not reachable; otherwise, it is reachable.

In this example, the *forward marking* approach required two MaxSat calls with the full constraint system, whereas the *backward marking* approach required only a single MaxSat call with approximately half of the constraints. This demonstrates the efficiency of the *backward marking* approach.

We highlight this improvement using the *backward marking* approach in this example, though it is not always more efficient in general. Intuitively, if the marked neurons exist in the early layers, the *forward marking* approach is more suitable, whereas if they exist in the deeper layers, the *backward marking* approach is preferable.

4.2 Problem Statement

The typical abstraction-refinement-based approaches use heuristics to identify the refinements and do not guarantee progress in each iteration [24, 25, 57, 58, 59]. Here, we investigate whether a refinement-based approach can *ensure progress*, i.e., the guaranteed elimination of spurious counterexamples, thereby combining effectiveness both in theory and in practice.

4.3 DeepPoly

We develop our abstract refinement approaches on top of an abstraction-based method DeepPoly [34], which uses a combination of well-understood polyhedra [137] and box [121] abstract domains. The abstraction maintains upper and lower linear expressions as well as

upper and lower bounds for each neuron. The variables appearing in the upper and lower expressions are only from the predecessor layer. Formally, we define the abstraction as follows.

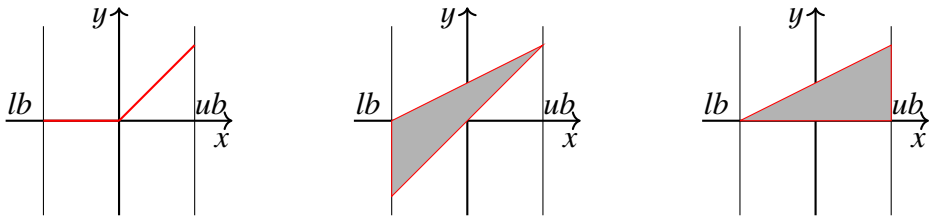
Definition 3. *For a neuron n, an abstract constraint* $A(n) = (lb, ub, lexpr, uexpr)$ *is a tuple, where* $lb \in \mathbb{R}$ *is the lower bound on the value of n,* $ub \in \mathbb{R}$ *is the upper bound on the value of n,* $lexpr \in LinExpr$ *is the expression for the lower bound, and* $uexpr \in LinExpr$ *is the expression for the upper bound.*

In DeepPoly, we compute the abstraction A as follows.

- If $Type_i$ = Affine, we set $A(x_{ij}).lexpr := A(x_{ij}).uexpr := \sum_{t=1}^{|l_{i-1}|} W_i[j,t] * x_{(i-1)t} + B_i[j,0]$. We compute $A(x_{ij}).lb$ and $A(x_{ij}).ub$ by back substituting the variables in $A(x_{ij}).lexpr$ and $A(x_{ij}).uexpr$, respectively, up to the input layer. Since P of the verification query has lower and upper bounds of the input layer, we can compute the bounds for x_{ij}. Consider the neuron x_{12} in Figure 4.1. Both its upper and lower constraints are the same, represented by the expression $x_{01} + x_{02}$. To compute the upper bound of x_{12}, we substitute the upper bounds of x_{01} and x_{02}, which are both 1. Consequently, the upper bound is calculated as $1 + 1$, which evaluates to 2. Similarly, for the lower bound of x_{12}, we substitute the lower bounds of x_{01} and x_{02}, which are both -1. Thus, the lower bound is computed as -2.

- If $Type_i$ = ReLU and y = ReLU(x), where x is a neuron in l_{i-1} and y is a neuron in l_i, we consider the following three cases:

  1. If $A(x).lb \geq 0$ then ReLU is in active phase and $A(y).lexpr := A(y).uexpr := x$, and $A(y).lb := A(x).lb$ and $A(y).ub := A(x).ub$
  2. If $A(x).ub \leq 0$ then ReLU is in passive phase and $A(y).lexpr := A(y).uexpr := 0$, and $A(y).lb := A(y).ub := 0$.
  3. If $A(x).lb < 0$ and $A(x).ub > 0$, the behavior of ReLU is uncertain, and we need to apply over-approximation. We set $A(y).uexpr := u(x-l)/(u-l)$, where $u = A(x).ub$ and $l = A(x).lb$. And $A(y).lexpr := \lambda.x$, where $\lambda \in \{0, 1\}$. We can choose any value of λ dynamically. We compute $A(y).lb$ and $A(y).ub$ by doing the back-substitution similar to the Affine layer's neuron.



(a) (b) (c)

Figure 4.3: (a) ReLU activation function, (b) lower approximation with $\lambda = 1$, (c) lower approximation with $\lambda = 0$. The upper approximation remains the same in both cases (b) and (c).

The value of λ in DeepPoly is chosen to minimize the over-approximation. Figure 4.3(a) illustrates the ReLU function $y = \max(0, x)$, while Figures 4.3(b) and 4.3(c) depict the two choices of λ, namely 1 and 0, respectively. The CROWN method [33] is similar to DeepPoly, differing only in implementation details. The work α-CROWN [42] optimizes the value of λ within the range $[0, 1]$ to further reduce the over-approximation introduced by abstraction. Our approach is applicable to all techniques for which the bounds of all neurons can be obtained.

The constraints for an Affine neuron are exact because it is just an Affine transformation of input neurons. The constraints for a ReLU neuron are also exact if the ReLU is either in the active or passive phase. The constraints for ReLU are over-approximated if the behavior of ReLU is uncertain. Although we may compute exact constraints for this case, but the constraints will be an arbitrary polyhedron, which are expensive to compute. The DeepPoly abstraction finds a balance between precision and efficiency.

4.4 Algorithms

We present our main counterexample-guided refinement process in Section 4.4.1, which consists of two key components: the refinement method (isVerify) and the marking mechanisms. The refinement method is described in Section 4.4.2, while the marking mechanisms, which have two variations (getMarkedNeuronsForward and getMarkedNeuronsBackward), are detailed in Section 4.4.3. Both algorithms for obtaining the marked neurons rely on a maxSat query, described in Section 4.4.4. Section 4.4.5 provides the proofs of progress and termination of our algorithms.

4.4.1 The top-level algorithm

We start by describing Algorithm 1, which presents the top-level flow of our approach. The algorithm takes as input a verification query $\langle N, P, Q \rangle$, where N is a neural network, and P and Q are predicates over the input and output layers, respectively. It returns SUCCESS if the verification is successful; otherwise, it returns a counterexample to the query. The algorithm relies on supporting procedures: GETMARKEDNEURONSFORWARD, GETMARKEDNEURONSBACKWARD, and ISVERIFIED (described subsequently). The first two are used to obtain the marked neurons for refining the abstraction, while the last checks the validity of the verification query after refinement. The first line of the algorithm performs preprocessing steps similar to those in state-of-the-art tools (e.g., $\alpha\beta$-CROWN). These preprocessing steps are optional and include the Projected Gradient Descent (PGD) attack [11], followed by CROWN [33], which is an incomplete verification method conceptually similar to DeepPoly but differing primarily in implementation details. The PGD attack is a technique for generating counterexamples. It works by iteratively updating the perturbation in the direction of the gradient of the loss function with respect to the input data, while constraining the perturbation magnitude within a predefined limit. If PGD fails to find a counterexample, CROWN is then applied to compute over-approximated bounds. Line 4 of Algorithm 1 generates all the abstract constraints by using DEEPPOLY, as described in Section 4.3. For a node $n_{ij} \in N.neurons$, the abstract constraints consist of the lower and upper constraints as well as the lower and upper bounds. Let $A.lc_i = \bigwedge_{j=1}^{|l_i|} A(n_{ij}).lexpr \leq x_{ij} \leq A(n_{ij}).uexpr$, which is a conjunction of upper and lower constraints of each neuron of layer l_i with respect to abstract constraint A. The *lexpr* and *uexpr* for any neuron of a layer contain variables only from the previous layer's neurons, hence $A.lc_i$ contains the variables from layers l_{i-1} and l_i. If the preprocessing steps in line 1 are applied, then DEEPPOLY generates the *lexpr* and *uexpr* for RELU neurons as per the triangle approximation. In this case, we may return a counter-example and stop, or use these bounds without performing any back-substitution.

At line 5, we initialize the variable *marked* to the empty set of neurons. At the next line, we iterate in a while loop until either we verify the query or find a counterexample. At line 7, we call ISVERIFIED with the verification query, abstraction A, and the set of marked neurons. In this verification step, the behavior of the marked neurons is encoded exactly, as detailed in Section 4.4.2. The call either returns that the query is verified or returns an abstract counterexample, which is defined as follows.

Definition 4. *A sequence of value vectors* $v_0, v_1, ..., v_k$ *is an* abstract execution *of abstract*

Algorithm 1 A CEGAR-based approach of neural network verification

Input: A verification problem $\langle N, P, Q\rangle$

Output: Verified or Counterexample

1: cex, $bounds = preprocessing(N, P, Q)$
2: **if** cex is not None **then**
3: **return** Failed(cex) ▷ cex is a counter example
4: A := DeepPoly($N, P, bounds$) ▷ use DeepPoly to generate abstract constraints.
5: $marked := \{\}$
6: **while** True **do**
7: result=isVerified($\langle N, P, Q\rangle$, A, $marked$)
8: **if** result = CEX($v_0, v_1...v_k$) **then**
9: **if** $N(v_0) \models \neg Q$ **then**
10: **return** Failed(v_0) ▷ v_0 is a cex
11: **else**
12: **if** isForwardMarking **then**
13: $markedNt$:= getMarkedNeuronsForward(N,A,$marked$, $v_0, v_1...v_k$)
14: **else**
15: $markedNt$:= getMarkedNeuronsBackward(N,A,$marked$, $v_0, v_1...v_k$)
16: $marked := marked \cup markedNt$
17: **else**
18: **return** verified

constraint A if $v_0 \models lc_0$ and $v_{i-1}, v_i \models A.lc_i$ for each $i \in [1, k]$. An abstract execution $v_0, ..., v_k$ is an abstract counterexample *if $v_k \models \neg Q$.*

If these algorithms return verified, we are done, otherwise, we analyze the abstract counterexample $CEX(v_0, ..., v_k)$. The abstract counterexample $CEX(v_0, ..., v_k)$ may or may not be a real counterexample, so we first check at line 9 if executing the neural network N on input v_0 violates the predicate Q. If yes, we report input v_0 as a counterexample, for which the verification query is not true. Otherwise, we declare the abstract counterexample to be spurious. We call getMarkedNeuronsForward or getMarkedNeuronsBackward, depending on the isForwardMarking flag, to analyze the counterexample and return the cause of spuriousness, which is a set of neurons *markedNt*. We add the new set *markedNt* to the old set *marked* and iterate our loop with the new set of marked neurons. The flag is-

Algorithm 2 Verify $\langle N, P, Q\rangle$ with abstraction A

Name: isVerified

Input: Verification query $\langle N, P, Q\rangle$, abstract constraint A, $marked \subseteq N.neurons$

Output: verified or an abstract counterexample.

1: $constr := P \wedge (\bigwedge_{i=1}^{k} A.lc_i) \wedge \neg Q$
2: $constr{:=}constr \wedge (\bigwedge_{n \in marked} exactConstr(n))$ ▷ as in Equation 4.8
3: isSat = checkSat(constr)
4: **if** isSat **then**
5: m := getModel(constr)
6: **return** CEX($m(x_0)$, ..., $m(x_k)$) ▷ Abstract counter example where x_i is a vector of variables in layer l_i
7: **else**
8: **return** verified

ForwardMarking is user defined parameter used to choose to run either forward marking or backward marking methods. Now let us present isVerified, getMarkedNeuronsForward, and getMarkedNeuronsBackward in detail.

4.4.2 Verifying query under marked neurons

In Algorithm 2, we present the implementation of isVerified, which takes the verification query, the DeepPoly abstraction A, and a set of marked neurons as input. At line 1, we construct constraints *contr* that encode the executions that satisfy abstraction A at every step. At line 2, we also include constraints in *constr* that encode the exact behavior of the marked neurons. The following is the encoding of the exact behavior [122] of neuron n_{ij}.

$$\begin{aligned} exactConstr(n_{ij}) := \; & x_{(i-1)j} \leq x_{ij} \leq x_{(i-1)j} - A(n_{(i-1)j}).lb * (1-a) \wedge \\ & 0 \leq x_{ij} \leq A(n_{(i-1)j}).ub * a \wedge a \in \{0,1\} \end{aligned} \tag{4.8}$$

where a is a fresh binary variable for each neuron, which can take values either 0 or 1.

Lemma 1. *The constraints in Equation 4.8 correctly capture the behavior of a ReLU node* $x_{ij} = \text{ReLU}(0, x_{(i-1)j})$.

Proof. The behavior of a ReLU node is as follows: if $x_{(i-1)j} \geq 0$, then $x_{ij} = x_{(i-1)j}$ (active phase), otherwise, if $x_{(i-1)j} \leq 0$, then $x_{ij} = 0$ (passive phase).

In Equation 4.8, $a = 1$ represents the active phase and $a = 0$ represents the passive phase.

1. **Case** $a = 1$**:** The constraints in (4.8) reduce to $x_{(i-1)j} \leq x_{ij} \leq x_{(i-1)j} \ \wedge \ 0 \leq x_{ij} \leq A(n_{(i-1)j}).ub$. This implies $x_{ij} = x_{(i-1)j}$ and $x_{ij} \in [0, A(n_{(i-1)j}).ub]$, which corresponds to the active phase.

2. **Case** $a = 0$**:** The constraints in (4.8) reduce to $x_{(i-1)j} \leq x_{ij} \leq x_{(i-1)j} - A(n_{(i-1)j}).lb \ \wedge \ 0 \leq x_{ij} \leq 0$. This implies $x_{ij} = 0 \ \wedge \ x_{(i-1)j} \leq 0 \ \wedge \ 0 \leq x_{(i-1)j} - A(n_{(i-1)j}).lb$. The condition $0 \leq x_{(i-1)j} - A(n_{(i-1)j}).lb$ holds trivially because $A(n_{(i-1)j}).lb$ is a lower bound of $x_{(i-1)j}$. Thus, Equation (4.8) simplifies to $x_{ij} = 0 \ \wedge \ x_{(i-1)j} \leq 0$, which corresponds to the passive phase of ReLU.

□

At line 3, we call a solver to find a satisfying assignment of the constraints. If *constr* is satisfiable, we get a model *m*. From the model *m*, we extract an abstract counterexample and return it. If *constr* is unsatisfiable, we return that the query is verified.

4.4.3 Maxsat-based approach to find the marked neurons

In this section, we find the important neurons whose approximation contributes to the spurious counterexamples most, and call them *markedNeurons*. We present two methods to find the *markedNeurons*, getMarkedNeuronsForward and getMarkedNeuronsBackward, both based on the maxSat approach. In getMarkedNeuronsForward, we find *markedNeurons* from initial layers to last layers, giving priority to the starting layers, while getMarkedNeuronsBackward, we find *markedNeurons* from last layers to the initial layers, giving priority to the last layers.

In our abstract constraints, we encode Affine neurons exactly, but over-approximate ReLU neurons. We identify a set of marked neurons whose exact encoding will eliminate the counterexample in the future analysis. As we defined earlier, let $val_i^{v_0}$ represent the value vector on layer l_i, if we execute the neural network on input v_0. Let us say $v_0, v_1, ..., v_k$ is an abstract spurious counterexample. We iteratively modify the counterexample such that its values coincide with $val_i^{v_0}$. Initially, $val_0^{v_0}$ is equal to v_0. Since we encode the affine layer exactly in $A.lc_i$, the following theorem helps us explain our method in detail.

Theorem 1. *Let* $v_0, v_1, ...v_k$ *be an abstract execution. For all* $1 \leq i \leq k$*, if* $Type_i =$ Affine *and* $val_{i-1}^{v_0} = v_{i-1}$*, then* $val_i^{v_0} = v_i$*.*

Algorithm 3 Marked neurons from counterexample

Name: getMarkedNeuronsForward

Input: Neural network N, abstract constraint A, $marked \subseteq N.neurons$, and abstract counterexample $(v_0, v_1 ... v_k)$

Output: New marked neurons.

Let $val_{ij}^{v_0}$ be the value of n_{ij}, when v_0 is input of N.
for $i = 1$ to k **do** ▷ inputLayer excluded
  if l_i is ReLU layer **then**
    $constr := \bigwedge_{t=i}^{k} A.lc_t$
    $constr := constr \wedge (\bigwedge_{n \in marked} exactConstr(n))$ ▷ as in Eq 4.8
    $constr := constr \wedge \bigwedge_{j=1}^{|l_{i-1}|}(x_{(i-1)j} = val_{(i-1)j}^{v_0})$
    $constr := constr \wedge \bigwedge_{j=1}^{|l_k|}(x_{kj} = v_{kj})$
    $softConstrs := \cup_{j=1}^{|l_i|}(x_{ij} = val_{ij}^{v_0})$
    $res, softsatSet =$ maxSat$(constr, softConstrs)$ ▷ res always SAT
    $newMarked := \{n_{ij} | 1 \leq j \leq |l_i| \wedge (x_{ij} = val_{ij}^{v_0}) \notin softsatSet\}$
    if $newMarked$ is empty **then**
      continue
    else
      return $newMarked$

Marked neurons in forward manner

In Algorithm 3, we present getMarkedNeuronsForward, which analyzes an abstract spurious counterexample. By the Theorem 1, v_1 and $val_1^{v_1}$ are also equal. The core idea of our algorithm is to find v_2' as close as possible to $val_2^{v_0}$, such that $v_0, v_1, v_2', ...v_{k-1}', v_k$ becomes an abstract spurious counterexample. We measure closeness by the number of elements of v_2' that are equal to the corresponding element of the vector $val_2^{v_0}$.

1. If v_2' is equal to $val_2^{v_0}$ then v_3' will also become equal to $val_3^{v_0}$ due to Theorem 1. Now we move on to the next ReLU layer l_4 and try to find a similar point v_4', such that $v_0, v_1, v_2, v_3, v_4''...v_{k-1}'', v_k$ is an abstract spurious counterexample. We repeat this process until the following case occurs.

2. If at some i, we can not make v_i' equal to $val_i^{v_0}$ then we collect the neurons whose values are different in v_i' and $val_i^{v_0}$. We call them marked neurons.

In the algorithm, the above procedure is implemented using a MaxSat solver. The loop at Line 2 iterates over the ReLU layers. Line 4 constructs the abstract constraints from layer i onwards, while Line 5 encodes the exact behavior of the marked neurons. Lines 6 and 7 ensure that the neurons in layer l_{i-1} take values equal to $val_{i-1}^{v_0}$, and that the execution terminates at v_k. At Line 8, the soft constraints are constructed, encoding that x_{ij} is equal to $val_i^{v_0}$. At Line 9, the MaxSat solver is invoked. This call always finds a satisfying assignment since the hard constraints are guaranteed to be satisfiable. Additionally, the solver returns a subset $softsatSet$ of the soft constraints that can be satisfied simultaneously. At Line 10, we check which soft constraints are missing from $softsatSet$; the corresponding neurons are then added to $newMarked$. If $newMarked$ is empty, we have successfully found a spurious abstract counterexample from $val_i^{v_0}$, and the algorithm proceeds to the next layer. Otherwise, the new set of marked neurons is returned.

In our motivating example in Section 4.1, the abstract counterexample obtained is given in Equation 4.3. In the first iteration of the `for` loop, the constraints in Equation 4.5 represent the abstract constraints from layer l_1 onwards, while the first and last lines in (4.5) ensure that the endpoints of the abstract counterexample remain fixed. The soft constraints are $\{x_5 = \frac{4}{5},\ x_6 = \frac{1}{5}\}$. The MaxSat call satisfies all the hard and soft constraints, and therefore, no neuron is marked in layer l_1. In the second iteration of the loop at Line 2, we move to the second ReLU layer. In this iteration, the hard constraints (Lines 4–7) are expressed in Equation 4.6, while the soft constraints are $\{x_9 = \frac{3}{5},\ x_{10} = 0\}$. The MaxSat call satisfies only the soft constraint $x_9 = \frac{3}{5}$, leaving x_{10} unsatisfied. Consequently, x_{10} is marked as a neuron in Line 10.

Marked neurons in backward manner

In Algorithm 4, we present the method getMarkedNeuronsBackward that analyzes the spurious counterexample in a backward manner. Suppose the spurious counterexample is $v_0, v_1, \ldots, v_k$, and $val_0^{v_0}, val_1^{v_0}, \ldots, v_k^{v_0}$ is the exact execution by simulating the neural network on input v_0. The core idea is to check whether v_k is reachable from the exact execution point $val_{i-1}^{v_0}$. If it is, then the sequence

$$val_0^{v_0},\ val_1^{v_0},\ \ldots,\ val_{i-1}^{v_0},\ v_i',\ v_{i+1}',\ \ldots,\ v_k$$

forms an abstract counterexample, where $v_0 = val_0^{v_0}$. In this case, we only need to analyze layers $l_{i-1}, l_i, \ldots, l_k$ to determine the marked neurons. Since our assumption is that the output layer l_k is always an affine layer, l_{k-1} is a ReLU layer, and l_{k-2} is an affine layer. We

Algorithm 4 Marked neurons in backward manner from the counterexample

Name: GETMARKEDNEURONSBACKWARD

Input: Neural network N, abstract constraint A, $marked \subseteq N.neurons$, and abstract counterexample $(v_0, v_1 ... v_k)$

Output: New marked neurons.

1: Let $val_{ij}^{v_0}$ be the value of n_{ij}, when v_0 is input of N.
2: **for** $i = k - 1$ to 1 **do** ▷ inputLayer excluded
3: **if** l_i is RELU layer **then**
4: $constr := \bigwedge_{t=i}^{k} A.lc_t$
5: $constr{:=}constr \wedge (\bigwedge_{n \in marked} exactConstr(n))$ ▷ as in Eq 4.8
6: $constr{:=}constr \wedge \bigwedge_{j=1}^{|l_{i-1}|}(x_{(i-1)j}{=}val_{(i-1)j}^{v_0})$
7: $constr := constr \wedge \bigwedge_{j=1}^{|l_k|}(x_{kj} = v_{kj})$
8: $softConstrs := \cup_{j=1}^{|l_i|}(x_{ij} = val_{ij}^{v_0})$
9: $res, softsatSet{=}$ MAXSAT$(constr, softConstrs)$
10: **if** res is SAT **then**
11: $newMarked := \{n_{ij} | 1 \leq j \leq |l_i| \wedge (x_{ij} = val_{ij}^{v_0}) \notin softsatSet\}$
12: **return** $newMarked$

start from $i = k - 1$ in a backward manner with the following two steps:

1. Check if v_k is reachable from $val_{i-1}^{v_0}$ (lines 4-10). If yes, then go to the next step; otherwise, continue to the previous ReLU layer ($i = i - 2$).

2. Find the marked neurons in layer l_i (lines 8-11).

Lemma 6 ensures that there must exist a layer l_{i-1} from which v_k will be reachable from $val_i^{v_0}$, and marked neurons exist only in layer l_i. In Algorithm 4, the above scenario is implemented using the MAXSAT solver, which we have defined in the next section.

We begin from the second-last layer, i.e., the final RELU layer l_{k-1}, and construct the abstract constraints at Line 4. Line 5 encodes the exact behavior of the marked neurons. In the first iteration, the set of marked neurons is empty. Lines 6 and 7 encode the starting point from which reachability is checked and the endpoint at which reachability is verified. The soft constraints are added at Line 8 to identify potential marked neurons. At Line 9, the MAXSAT solver is invoked, implemented via MILP in Algorithm 5. The MAXSAT solver guarantees satisfiability of all hard constraints and maximizes the number of satisfied soft

constraints. Finally, at Line 11, neurons corresponding to unsatisfied soft constraints are added to the set of marked neurons.

The last three lines in Equation 4.2 (except for the negation of the property) in our motivating example in Section 4.1 represent the abstract constraints. The starting point from which reachability is checked at Line 6 is given in our motivating example as:

$$x_7 = \tfrac{3}{5} \ \wedge \ x_8 = -\tfrac{1}{5}.$$

The point v_k for which reachability is checked is represented by the following values in our motivating example:

$$x_{11} = \tfrac{7}{10} \ \wedge \ x_{12} = \tfrac{1}{2}.$$

The hard constraints in Lines 4–7 correspond to Equation 4.7 in the motivating example. We then add the following soft constraints to identify the marked neurons:

$$\{\, x_9 = \tfrac{3}{5}, \ x_{10} = 0 \,\}.$$

Here, neuron x_{10} is marked, since its corresponding soft constraint $x_{10} = 0$ is not satisfied.

In the getMarkedNeuronsForward method, the initial hard constraints in the Max-SAT call include the constraints of the full network, and these constraints are gradually reduced in each iteration. In contrast, the getMarkedNeuronsBackward method begins with very few hard constraints to optimize, which then increase as the loop progresses. Both techniques have their advantages: if the marked neurons exist in the early layers, getMarkedNeuronsForward identifies them efficiently, whereas if the marked neurons exist in the deeper layers, then getMarkedNeuronsBackward is more effective.

4.4.4 MaxSAT Encoding in MILP

In this section, we present the maxSat encoding in MILP. The maxSat function is used in Algorithms 3 and 4 to find the marked neurons from the abstract counterexample. The maxSat function takes two arguments: *hard constraints* and *soft constraints*, and returns the satisfiability result of the hard constraints (SAT or UNSAT) along with the subset of *soft constraints* that are additionally satisfied. We can exploit the abstract constraints generated by DeepPoly to encode the soft constraints in the MILP.

Suppose we want to encode the soft constraints $\{x_{ij} = val_{ij}\}$. Since these are soft constraints, we attempt to satisfy them without affecting the satisfiability of the hard constraints. Equation 4.9 shows the encoding, where b_{ij} is a binary variable that takes values in $\{0, 1\}$. Intuitively, x_{ij} is the corresponding variable of the neuron n_{ij}.

$$\texttt{softConstrEqual}(x_{ij}, val_{ij}, b_{ij}) := x_{ij} - val_{ij} \leq (1 - b_{ij})(A(n_{ij}).\text{ub} - A(n_{ij}).\text{lb}) \wedge \\ x_{ij} - val_{ij} \geq -(1 - b_{ij})(A(n_{ij}).\text{ub} - A(n_{ij}).\text{lb}) \tag{4.9}$$

Lemma 2. *For a softConstrEqual*$(x_{ij}, val_{ij}, b_{ij})$*, if* $b_{ij} = 1$ *then* $x_{ij} = val_{ij}$*, otherwise, the constraints will always be satisfied.*

Proof. If $b_{ij} = 1$, then the constraints in Equation 4.9 reduce to $0 \leq x_{ij} - val_{ij} \leq 0$, which implies $x_{ij} - val_{ij} = 0$, i.e., $x_{ij} = val_{ij}$. If $b_{ij} = 0$, then the constraint is relaxed to $-(A(n_{ij}).\text{ub} - A(n_{ij}).\text{lb}) \leq x_{ij} - val_{ij} \leq A(n_{ij}).\text{ub} - A(n_{ij}).\text{lb}$, which is satisfied because $|x_{ij} - val_{ij}|$ is bounded by the difference between the bounds from the abstract state, which is ensured by DeepPoly. □

Algorithm 5 presents the maxSat function, which is invoked from Algorithms 3 and 4. The maxSat function takes two arguments: *hard constraints* and *soft constraints*, and returns the satisfiability result of the hard constraints (SAT or UNSAT) along with the subset of *soft constraints* that are additionally satisfied. Line 1 encodes the soft constraints using Equation 4.9. For each neuron n in the soft constraints, we introduce a binary variable b_n. By Lemma 2, setting $b_n = 1$ enforces satisfaction of the corresponding soft constraint, whereas $b_n = 0$ relaxes it. Since our goal is to satisfy as many soft constraints as possible, we aim to maximize the number of binary variables set to 1. Thus, we construct an objective function in Line 2 by summing all such binary variables. At Line 3, the function `maximize(constrs, obj)` maximizes the objective function obj while satisfying all constraints *constrs*. This function returns the satisfiability result of *constrs* and the optimal value of obj, the latter being valid only when the result is SAT. Finally, at Lines 7–9, we identify which soft constraints are satisfied by checking the assignments of the corresponding binary variables. We then return the subset of soft constraints for which the associated binary variable is set to 1, indicating that these constraints are satisfied in the optimal solution.

4.4.5 Proofs of progress and termination

A refinement strategy needs to ensure that the algorithm terminates and may also ensure that it does not repeat the same spurious counterexample in the future iterations of Algorithm 1, which is called the progress property of the strategy. Not all refinement strategies ensure these properties. For example, the refinement strategies in [57, 58, 59] do

Algorithm 5 MaxSAT

Name: MaxSAT

Input: $\langle hard\_constraints, soft\_constraints \rangle$

Output: $\langle res, soft\_sat\_constraints \rangle$

1: $constrs := (\bigwedge_{x_{ij}=val_{ij} \in soft\_constraints}$ `softConstrEqual`$(x_{ij}, val_{ij}, b_n))$
2: $obj = \sum_{n \in soft\_nts} b_n$
3: isSat, objVal = $maximize(hard\_constraints \wedge constrs, obj)$
4: $soft\_sat\_constraints = \{\}$
5: **if** isSat **then**
6: m := getModel(constrs)
7: **for** (n, v) in $(soft\_nts, soft\_vals)$ **do**
8: **if** $m(b_n) = 1$ **then**
9: $soft\_sat\_constraints = \{n = v\}$
10: **return** isSat, $soft\_sat\_constraints$

not ensure that the same spurious counterexample is not repeated in future iterations. In this section, we prove that our refinement strategy satisfies these properties.

Let us suppose the algorithms getMarkedNeuronsForward or getMarkedNeurons-Backward get the abstract spurious counterexample $v_0, v_1, ...v_k$ and returns marked neurons in some iteration of the while loop, say i^{th}-iteration. The call to maxSat at line 10 declares that $constr \wedge softsatSet$ is satisfiable. We can extract an abstract spurious counterexample from a model of $constr \wedge softsatSet$. Let m be the model. Let the abstract spurious counterexample be $cex = val_0^{v_0}, ..., val_{i-1}^{v_0}, v'_i, ..., v'_{k-1}, v_k$. Before the iteration i, cex follows the execution of N on input v_0. After the iteration i, we use the model to construct cex, i.e., $m(x_i) = v'_i$.

Lemma 3. *In the rest of run of Algorithm 1, i.e., future iterations of the while loop,* IsVerified *will not return the abstract spurious counterexample cex again.*

Proof. For $n_{ij} \in newMarked$, the maxSat query ensures that $val_{ij}^{v_0} \neq v'_{ij}$ in both getMarked-NeuronsForward and getMarkedNeuronsBackward. If we have the same counterexample again in the future, then input of n_{ij} will be $val_{(i-1)j}^{v_0}$. Since we will have exact encoding for n_{ij}, the output will be $val_{ij}^{v_0}$, which contradicts the earlier inequality. □

The above lemma ensures progress. Next, we turn to termination of the algorithm,

which we prove using the following two lemmas.

Lemma 4. *In every refinement iteration* getMarkedNeuronsForward *returns a non-empty set of marked neurons.*

Proof. By the definition of an abstract spurious counterexample, we have $v_k \models \neg Q$ and $val_k^{v_0} \models Q$. Suppose the method getMarkedNeuronsForward returns an empty set of marked neurons. Then, by the soft constraints at Line 10 and constraints at Line 7 of Algorithm 3, the following constraints hold:

$$\bigwedge_{i=0}^{k} \bigwedge_{j=1}^{|l_i|} x_{ij} = val_{ij}^{v_0} \ \wedge \ \bigwedge_{j=1}^{|l_k|} x_{kj} = v_{kj}.$$

The above constraints imply $\bigwedge_{j=1}^{|l_k|} val_{kj}^{v_0} = v_{kj}$, which further implies $v_k = val_k^{v_0}$. However, this leads to a contradiction since $v_k \models \neg Q$ and $val_k^{v_0} \models Q$. □

Lemma 5. *In every refinement iteration* getMarkedNeuronsForward *returns marked neurons, which were not marked in previous iterations.*

Proof. We will show that if a neuron n_{ij} is marked in the t^{th} iteration, then n_{ij} will not be marked again in any iteration greater than t. Consider an iteration $t' > t$. If the marked neurons come from a layer other than l_i, then n_{ij} cannot be part of it, since n_{ij} belongs to layer l_i. Now consider the case where the marked neurons are from layer l_i in the t'^{th} iteration. Since we have made n_{ij} exact in Line 5 of Algorithm 3, by Lemma 1 the following equality holds:

$$v'_{ij} = val_{ij}^{v_0}.$$

According to the criterion for soft constraints at Line 10, we have:

$$newMarked = \{n_{ir} \mid v'_{ir} \neq val_{ir}^{v_0} \wedge 1 \leq r \leq |l_i|\}.$$

From the above two conditions, we can conclude that $n_{ij} \notin newMarked$. □

We also need to prove that the backward marking algorithm also satisfies the progress property.

Lemma 6. *In every refinement iteration,* getMarkedNeuronsBackward *returns a non-empty set of marked neurons.*

Proof. We prove this by establishing the following two claims:

- **Claim-1:** There must exist a relu layer l_i for which line 10 of Algorithm 4 is satisfied. Suppose no such layer exists, then the following constraints form in the last iteration of the loop at line 2, these constraints are not satisfied.

$$\bigwedge_{j=1}^{|l_1|} x_{1j} = val_{1j}^{v_0} \wedge \bigwedge_{t=2}^{k} A.lc_t \wedge \bigwedge_{j=1}^{|l_k|} (x_{kj} = v_{kj}) \wedge \bigwedge_{n \in marked} exactConstr(n)$$

  Since l_1 is an affine layer and affine layers impose exact constraints, we also have:

$$\bigwedge_{j=1}^{|l_0|} x_{0j} = val_{0j}^{v_0}$$

  By definition, $val_0^{v_0} = v_0$ and $v_0 \models P$. Given that $v_0, v_1, \ldots, v_k$ form an abstract spurious counterexample, and that the same constraints are applied to abstract counterexamples in lines 4-5, the values $v_1, v_2, \ldots, v_k$ will also be satisfing assignement of the above constraint, which leads to a contradiction.

- **Claim-2:** The non-empty set of neurons is from the same layer l_i.
  Let $constrain_i$ denote the constraints for layer l_i as defined in lines 4-7 of Algorithm 4. Assume, for contradiction, that there are no marked neurons in l_i; then the following holds:

$$\bigwedge_{j=1}^{|l_i|} x_{ij} = val_{ij}^{v_0}$$

  Since l_{i+1} is an affine layer, by Theorem 1, we also have:

$$\bigwedge_{j=1}^{|l_{i+1}|} x_{(i+1)j} = val_{(i+1)j}^{v_0}$$

  From the above two equalities, we conclude that $constrain_{i+2}$ holds. However, from the previous iteration of the loop at line 2, we know that $constrain_{i+2}$ is unsatisfiable, which leads to a contradiction.

By the above two claims, we conclude that there must exist a layer l_{i+1} for which a non-empty set of marked neurons is returned. □

Lemma 7. *In every refinement iteration* getMarkedNeuronsBackward *returns marked neurons, which were not marked in previous iterations.*

Proof. We will show that if a neuron n_{ij} is marked in the t^{th} iteration, then n_{ij} will not be marked again in any iteration greater than t. Consider an iteration $t' > t$. If the marked neurons come from a layer other than l_i, then n_{ij} cannot be part of it, since n_{ij} belongs to layer l_i. Now consider the case where the marked neurons are from layer l_i in the t'^{th} iteration. Since we have made n_{ij} exact in Line 5 of Algorithm 4, by Lemma 1 the following equality holds:

$$v'_{ij} = val^{v_0}_{ij}.$$

According to the criterion for soft constraints at Line 11, we have:

$$newMarked = \{n_{ir} \mid v'_{ir} \neq val^{v_0}_{ir} \wedge 1 \leq r \leq |l_i|\}.$$

From the above two conditions, we can conclude that $n_{ij} \notin newMarked$. □

Theorem 2. *Algorithm 1 always terminates.*

Proof. Lemmas 4, 5, 6, and 7 imply that in every iteration, getMarkedNeuronsForward and getMarkedNeuronsBackward return a non-empty set of unmarked neurons, which will then be marked. In the worst case, the algorithm will mark all the neurons in the network and encode them with exact behavior. Thus, we conclude that Algorithm 1 always terminates. □

In this section, we have proved the progress and termination guarantees of our algorithms. Our methods, getMarkedNeuronsForward and getMarkedNeuronsBackward, select the important neurons (marked neurons) guided by the spurious counterexample. Encoding these marked neurons guarantees the removal of the spurious counterexample. In the worst case, we may end up selecting all neurons as marked neurons in order to encode the exact behavior, which demonstrates the completeness of our approach.

4.5 Experiments

We have implemented our approach in a prototype and compared it against three categories of existing techniques: (i) DeepPoly [34] and its refinements kPoly [41], deepSRGR [38]; (ii) counterexample-guided refinement (CEGAR)-based technique [37]; and (iii) state-of-the-art tools such as $\alpha\beta$-CROWN [33, 24, 25, 167, 122], marabou [21], nnenum [63, 64], and Oval [168, 57, 169, 170, 171, 58].

Since our tool is built upon DeepPoly, we first compare it with DeepPoly to establish a baseline. We also compare its performance with kPoly and deepSRGR, two other refinement tools based on DeepPoly, although their refinement strategies differ significantly from ours. Our findings show that our tool performs competitively compared to the DeepPoly-based refinement approaches. As our approach is based on the CEGAR framework, we further compare it with the CEGAR-based tool cegar_nn [37]. Finally, we evaluate our tool against state-of-the-art verifiers, and our results demonstrate that it can solve a number of unique benchmark instances that are beyond the reach of current state-of-the-art tools.

Furthermore, we conducted a comparison of performance across different ϵ values for all the tools evaluated. We also extended this analysis to specifically focus on adversarially trained networks and observed a significant improvement in performance. Additionally, we performed a detailed comparison with $\alpha\beta$-CROWN, employing the same preprocessing steps used by the $\alpha\beta$-CROWN tool.

The tool $\alpha\beta$-CROWN has consistently achieved first place in all editions of VNN-COMP (The International Verification of Neural Networks Competition) from 2021 to 2024 [73, 74, 75, 76]. The tool marabou secured second place in VNN-COMP 2024 and also performed well in earlier editions of the competition. The tool nnenum achieved fourth place in VNN-COMP 2024. We also compared our approach with Oval, which was considered state-of-the-art in VNN-COMP 2021. It is worth noting that both $\alpha\beta$-CROWN and Oval employ a portfolio of different algorithms and optimizations to enhance performance.[1]

4.5.1 Implementation

We have implemented our techniques in a tool named drefine, developed in the C++ programming language. The marking methods getMarkedNeuronsForward and getMarkedNeuronsBackward can be selected via a command-line argument `-backprop-marking`. If this parameter is set to `false`, then drefine_f runs; otherwise, drefine_b is executed. The default value of this parameter is `false`. For clarity, throughout this paper, we refer to the analysis using getMarkedNeuronsForward as drefine_f, and the analysis using getMarkedNeuronsBackward as drefine_b. The tool is available at https://github.com/afzalmohd/VeriNN/tree/master/deep_refine. Our approach builds upon DeepPoly, for which we have also provided a C++ implementation. For satisfiab-

[1] We could not compare with PyRAT, ranked second in VNN-COMP 2024, as it is a closed-source tool.

ility checking and solving MaxSat queries, we utilize the C++ interface of the Gurobi optimizer [40].

4.5.2 Benchmarks

We use the MNIST [65] dataset to check the effectiveness of our tool and comparisons. We use 11 different fully connected feedforward neural networks with ReLU activation, as shown in Table 4.1. These benchmarks are taken from DeepPoly's paper [34]. The input and output dimensions of each network are 784 and 10, respectively. The authors of DeepPoly used projected gradient descent (PGD) [11] and DiffAI [172] for adversarial training. The PGD attack is a widely used adversarial method that iteratively perturbs an input by ascending the gradient to maximize the model loss. After each step, the perturbation is projected back onto a constrained norm ball to ensure that the modifications remain imperceptible. DiffAI methods, which leverage diffusion models to generate or denoise data, enhance the robustness of AI systems against adversarial attacks. By learning to reconstruct clean inputs from their noisy counterparts, these methods make models less susceptible to adversarial manipulations. Table 4.1 contains the defended network i.e., trained with adversarial training, as well as the undefended network. The last column of Table 4.1 shows how the defended networks were trained.

The predicate P on the input layer is constructed using the input image $\boldsymbol{im}$ and a user-defined parameter ϵ. Each pixel of $\boldsymbol{im}$ is first normalized to the interval $[0, 1]$. We then define

$$P = \bigwedge_{i=1}^{|l_0|} \Big(\max\{0,\ im(i) - \epsilon\} \leq x_{0i} \leq \min\{1,\ im(i) + \epsilon\} \Big)$$

ensuring that the lower and upper bounds for each pixel do not exceed the valid input range. Intuitively, each pixel $im(i)$ can vary within $[im(i) - \epsilon,\ im(i) + \epsilon]$, but is always constrained to remain within $[0, 1]$.

The predicate Q on the output layer is defined based on the network's output. Suppose the predicted label of $\boldsymbol{im}$ by the network N is y; then we set

$$Q = \bigwedge_{i=1,\, i \neq y}^{|l_k|} \Big(x_{ki} < x_{ky} \Big)$$

Since the prediction of a neural network is given by an Argmax function, which selects the index of the largest value, this post-condition ensures that the value corresponding to the correct class remains the largest. Therefore, no misclassification occurs as long as Q holds.

One query instance $\langle N, P, Q \rangle$ is created for one network, one image, and one epsilon value. In our evaluation, we took 11 different networks, 8 different epsilons, and 100 different images. The total number of instances is 8800. However, there are 304 instances for which the network's predicted label differs from the image's actual label. We avoided such instances and considered a total of 8496 benchmark instances.

Neural Network	# Hidden Layers	# Activation Units	Defensive Training
3×50	2	110	None
3×100	2	210	None
5×100	4	410	None
6×100	5	510	DiffAI
9×100	8	810	None
6×200	5	1010	None
9×200	8	1610	None
6×500	6	3000	None
6×500	6	3000	PGD, $\epsilon = 0.1$
6×500	6	3000	PGD, $\epsilon = 0.3$
4×1024	3	3072	None

Table 4.1: Neural network architectures and defensive training methods used in our experiments.

4.5.3 Results

We conducted the experiments on two different machines. Since DeepPoly and kPoly are lightweight techniques, the experiments reported in Table 4.2 and Table 4.4 were performed on a machine equipped with 64GB RAM, 2.20 GHz Intel(R) Xeon(R) CPU E5-2660 v2 processor running CentOS Linux 7. All other experiments were carried out on a machine with two AMD EPYC 7742 64-Core processors (2.25 GHz, 256 hardware threads in total) and 660GB RAM. To ensure a fair comparison across tools, we restricted each experiment to a single CPU and enforced a timeout of 2000 seconds per instance.

We begin by comparing our method with the most closely related tools, namely kPoly and deepSRGR, which are improvements built on top of DeepPoly. Next, we consider the cegar-based technique, represented by cegar_nn. We then extend the comparison to state-of-the-art solvers, including $\alpha\beta$-CROWN, marabou, nnenum, and Oval. In addition, we

Solved \ Unsolved	DeepPoly	kPoly	deepSRGR	cegar_nn	drefine_f	drefine_b	drefine_f ∪ drefine_b	total
DeepPoly	0	0	0	3708	0	0	0	4633
kPoly	156	0	114	3760	14	14	9	4789
deepSRGR	54	2	0	3736	0	0	0	4687
cegar_nn	713	609	687	0	417	408	387	1624
drefine_f	688	546	634	4114	0	58	0	5321
drefine_b	747	605	693	4173	117	0	0	5380
drefine_f ∪ drefine_b	805	658	751	4231	117	58	0	5438

Table 4.2: Pairwise comparison of tools, e.g. entry on row kPoly and column DeepPoly represents 156 benchmark instances on which kPoly verified but DeepPoly fails. The green row highlights the number of solved benchmark instances by our tools, and not others, while the red column is the opposite.

analyze the performance of these tools as the perturbation parameter ϵ varies. A separate evaluation is conducted on benchmarks containing adversarially trained networks. We further provide a detailed analysis with respect to $\alpha\beta$-CROWN.

Comparison with the most related techniques

In this subsection, we consider the techniques DeepPoly, kPoly, and deepSRGR for comparison with our approaches. We include DeepPoly because it forms the basis of our technique, and both kPoly and deepSRGR refine DeepPoly in a manner similar to ours. We used vanilla DeepPoly (i.e., DeepPoly without the preprocessing step in line 4 of Algorithm 1) to generate the abstract constraints of benchmark instances.

As shown in Table 4.2, our technique outperforms all related approaches by solving approximately 700 additional benchmarks. In particular, we solve around 4231 benchmarks that are out of reach for the cegar_nn method, whereas cegar_nn solves 387 benchmarks that our technique cannot handle. Notably, cegar_nn is also a CEGAR-based approach. Furthermore, we solve approximately 750 benchmarks that are beyond the reach of DeepPoly, kPoly, and deepSRGR, while none of these techniques solve any benchmark instances that our approach cannot solve.

Since DeepPoly, kPoly, and deepSRGR report only verified instances, whereas our tool can report both verified instances and counterexamples, we additionally compare these techniques against the verified instances produced by our approach. These results

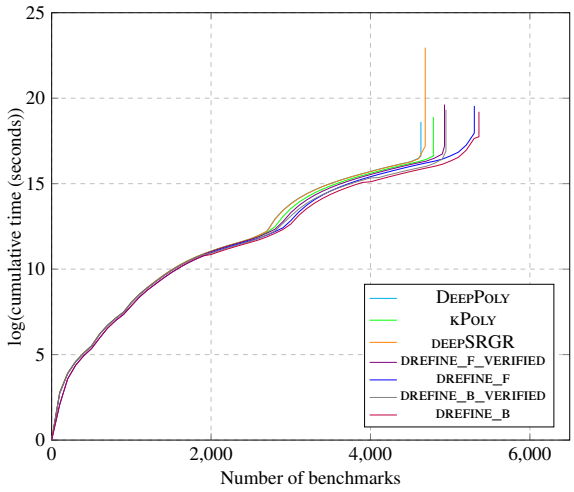


Figure 4.4: Cactus plot comparing related techniques. The x-axis indicates the number of benchmarks solved, ordered by increasing solution time. The y-axis shows the cumulative solving time (in seconds) on a logarithmic scale- a tick value of 15 on the y-axis corresponds to 2^{15} seconds.

are captured by dRefine_F_Verified and dRefine_B_Verified in the cactus plot shown in Figure 4.4. We observe that when verification succeeds, DeepPoly and kPoly are often more efficient, which is expected since these techniques rely solely on abstraction, whereas our approach performs abstraction followed by iterative refinement. Despite this additional overhead, our technique remains more efficient than deepSRGR.

Comparison with cegar based techniques

cegar_nn [37] is a tool that also uses counterexample guided refinement. But the abstraction used is quite different from DeepPoly. This tool reduces the size of the network by merging similar neurons, such that they maintain the overapproximation. In the refinement process, it identifies the most important neurons, guided by the spurious counterexample, and selectively splits them back.

Table 4.2 shows that cegar_nn solves only 19.11% of the total benchmark instances, whereas our techniques dRefine_F and dRefine_B solve 62.70% and 63.38%, respectively. Although cegar_nn solves significantly fewer benchmarks overall, it is pertinent to note that it solves many unique instances, as seen from the cegar_nn row in Table 4.2.

Solved \ Unsolved	OVAL	$\alpha\beta$-CROWN	MARABOU	NNENUM	DREFINE_F	DREFINE_B	DREFINE_F ∪ DREFINE_B	TOTAL
OVAL	0	30	1471	1438	1051	1019	967	6195
$\alpha\beta$-CROWN	96	0	1535	1487	1112	1073	1023	6261
MARABOU	511	509	0	2105	608	521	513	5235
NNENUM	9	8	576	0	50	28	11	4766
DREFINE_F	183	178	700	521	0	61	0	5327
DREFINE_B	209	197	671	557	119	0	0	5385
DREFINE_F ∪ DREFINE_B	218	208	724	691	119	61	0	5446

Table 4.3: Pairwise comparison of tools, e.g. entry on row MARABOU and column OVAL represents 1005 benchmark instances on which MARABOU verified but OVAL fails. The green row highlights the number of solved benchmark instances by our tools, and not others, while the red column is the opposite.

Comparison with state-of-the-art solvers

The tools $\alpha\beta$-CROWN and OVAL use several algorithms that are highly optimized and use several techniques. The authors of $\alpha\beta$-CROWN implement the techniques [33, 24, 25, 167, 122], and the authors of OVAL implement [168, 57, 169, 170, 170, 171, 58]. The authors of MARABOU implement the technique [21, 20]. Table 4.3 shows that $\alpha\beta$-CROWN and OVAL solve about 1000 more benchmarks (out of 8496) than our tool, while we outperform MARABOU and NNENUM by a significant margin.

Although $\alpha\beta$-CROWN and OVAL outperform our tool overall, we identified around 178 and 197 benchmark instances where $\alpha\beta$-CROWN fails and our tools (DREFINE_F and DREFINE_B, respectively) succeed, and around 183 and 209 instances where OVAL fails and our tools succeed. Moreover, there are 700 and 671 benchmarks where MARABOU fails but are successfully handled by DREFINE_F and DREFINE_B, respectively. Further, both our techniques, DREFINE_F and DREFINE_B, outperform the tool NNENUM by approximately 561 and 619 benchmarks, respectively. Additionally, DREFINE_F and DREFINE_B solve 521 and 557 benchmarks that are not solved by NNENUM, whereas NNENUM solves only 50 and 28 benchmarks not solved by DREFINE_F and DREFINE_B, respectively, see Table 4.3.

In total, we solve 113 and 121 unique benchmarks using DREFINE_F and DREFINE_B, respectively, where all four state-of-the-art tools ($\alpha\beta$-CROWN, OVAL, MARABOU, and NNENUM) fail. The number of uniquely solved benchmarks increases to 132 when combining the res-

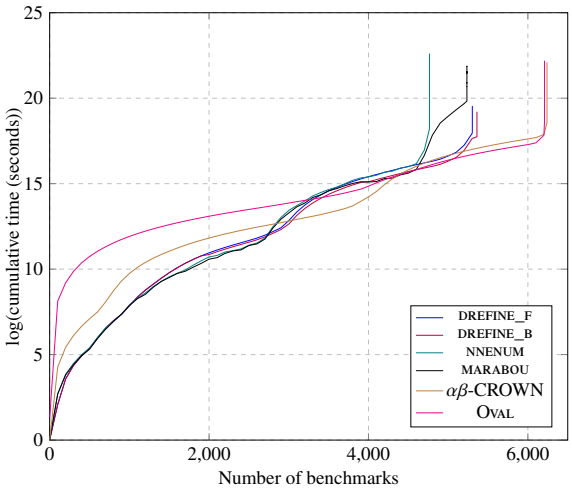


Figure 4.5: Cactus plot comparing states of the arts. The x-axis indicates the number of benchmarks solved, ordered by increasing solution time. The y-axis shows the cumulative solving time (in seconds) on a logarithmic scale- a tick value of 15 on the y-axis corresponds to 2^{15} seconds.

ults of both DREFINE_F and DREFINE_B. Thus, we believe that these tools are truly orthogonal in their strengths and could potentially be combined for greater effectiveness.

Although we solve additional benchmarks that are beyond the reach of state-of-the-art verifiers, there is a trade-off between accuracy and efficiency. We further compare efficiency with state-of-the-art tools using a cactus plot, as shown in Figure 4.5. Our approach is more efficient than NNENUM and MARABOU, while $\alpha\beta$-CROWN and OVAL outperform our tool.

Epsilon vs. performance

As a sanity check, we analyzed the effect of perturbation size on the performance of the tools. Figure 4.6 presents the comparison of the fractional success rate of the tools as ϵ increases from 0.005 to 0.05. At $\epsilon = 0.005$, the performance of all the tools is almost identical, except for CEGAR_NN. As ϵ increases, the success rate of most tools consistently decreases, with the exception of CEGAR_NN. In this setting, our approach outperforms DEEPPOLY, KPOLY, DEEPSRGR, CEGAR_NN, and NNENUM, while $\alpha\beta$-CROWN and OVAL achieve better results. MARABOU performs better than our method only when $\epsilon \geq 0.04$.

All tools, except CEGAR_NN, struggle in terms of efficiency as the value of the input perturbation ϵ increases. This decline in performance is likely due to the expansion of the

Solved \ Unsolved	DeepPoly	kPoly	deepSRGR	cegar_nn	dreFine_F	dreFine_B	dreFine_F ∪ dreFine_B	total
DeepPoly	0	0	0	1215	0	0	0	1626
kPoly	9	0	9	1221	2	2	2	1635
deepSRGR	2	2	0	1215	0	0	0	1628
cegar_nn	15	10	13	0	7	0	0	426
dreFine_F	168	161	166	1375	0	5	0	1794
dreFine_B	187	180	185	1387	24	0	0	1813
dreFine_F ∪ dreFine_B	192	185	190	1392	24	5	0	1818

Table 4.4: Pairwise comparison of tools on adversarially trained networks

Solved \ Unsolved	Oval	$\alpha\beta$-CROWN	marabou	nnenum	dreFine_F	dreFine_B	dreFine_F ∪ dreFine_B	total
Oval	0	11	619	599	214	199	195	1935
$\alpha\beta$-CROWN	1	0	611	588	205	192	188	1925
marabou	55	57	0	-	64	62	61	1371
nnenum	5	4	55	0	11	4	0	1341
dreFine_F	75	76	496	519	0	5	0	1796
dreFine_B	79	82	517	536	24	0	0	1815
dreFine_F ∪ dreFine_B	80	83	523	540	24	5	0	1820

Table 4.5: Pairwise comparison of tools on adversarially trained networks

search space with larger ϵ values. However, cegar_nn remains agnostic to the value of ϵ, which may explain its stable performance across varying perturbation levels.

Comparison with adversarially trained networks

The networks considered for evaluation in this study are the ones corresponding to the 4st, 9th, and 10th rows of Table 4.1. These networks have been trained using adversarial techniques, where adversarial examples were generated using standard methods such as PGD/DiffAI, and the network was subsequently trained on these adversarial examples to enhance its robustness. Table 4.4 presents a pairwise comparison of verifiers on these adversarially trained networks, encompassing a total of 2223 benchmark instances. Our approach demonstrates significant superiority over DeepPoly, kPoly, deepSRGR, and cegar_nn in terms of performance on these benchmarks.

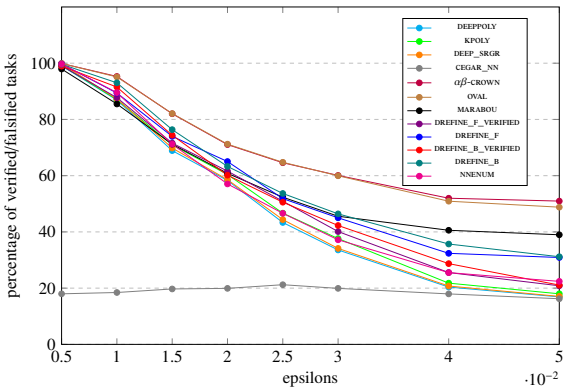


Figure 4.6: Size of input perturbation (epsilon) vs. percentage of solved instances

Table 4.5 presents the pairwise comparison with state-of-the-art verifiers. Although $\alpha\beta$-CROWN and OVAL outperform both of our approaches by approximately 130 benchmarks, our methods still solve 80 and 83 benchmarks (as shown in the last row of Table 4.5) that remain unsolved by OVAL and $\alpha\beta$-CROWN, respectively. Out of a total of 2223 benchmarks in this category, this constitutes a notable subset that our approach is able to address.

Detailed comparison with αβ-CROWN with same preprocessing

To evaluate the effectiveness of our approach, we conducted the experiments on the instances where refinement was applied. Specifically, we selected the benchmarks from the $\alpha\beta$-CROWN experiments where refinement [24] was actually used, including those that resulted in timeouts. We excluded all benchmarks that were already solved by preprocessing techniques, namely PGD (Projected Gradient Descent) attack or CROWN [33], which is an incomplete verification method. The comparison was then performed on the remaining benchmarks. The PGD attack generates counterexamples by iteratively updating the perturbation in the direction of the gradient of the loss function with respect to the input, while constraining the perturbation to remain within a predefined bound. If PGD fails, CROWN is applied to compute over-approximated bounds.

After preprocessing, the total number of benchmarks was reduced to 2362, which were the benchmarks that were not solved by the preprocessing steps. Table 4.6 shows the detailed comparison among the tools $\alpha\beta$-CROWN, DREFINE_F, and DREFINE_B. The explanation of this table follows the same format as Table 4.2. The (i, j) entry indicates the number of benchmarks solved by tool i but not by tool j. For example, the cell value 208 represents the number of benchmarks solved by $\alpha\beta$-CROWN but not by DREFINE_F. Out of

Tool	$\alpha\beta$-CROWN	DREFINE_F	DREFINE_B	DREFINE_F ∪ DREFINE_B	TOTAL
$\alpha\beta$-CROWN	0	208	202	201	629
DREFINE_F	152	0	3	0	573
DREFINE_B	155	12	0	0	582
DREFINE_F ∪ DREFINE_B	157	12	3	0	585

Table 4.6: Comparison with $\alpha\beta$-CROWN, DREFINE_F, and DREFINE_B on the filtered benchmarks. The (i, j) entry indicates the number of benchmarks solved by tool i but not by tool j. For example, the cell value 208 represents the number of benchmarks solved by $\alpha\beta$-CROWN but not by DREFINE_F.

the 2363 benchmarks, $\alpha\beta$-CROWN was able to verify 629, while DREFINE_F verified 573. Notably, DREFINE_F solved 152 benchmarks that were not solved by $\alpha\beta$-CROWN, whereas $\alpha\beta$-CROWN solved 208 benchmarks that were not solved by DREFINE_F. A similar trend is observed for DREFINE_B, with details provided in Table 4.6.

Furthermore, the union of benchmarks solved by tools $\alpha\beta$-CROWN and DREFINE_F results in a total of 781 benchmarks, while the union of $\alpha\beta$-CROWN and DREFINE_B covers 784 benchmarks. This demonstrates a clear improvement over the number of benchmarks solved by $\alpha\beta$-CROWN alone. These findings highlight the significant contribution of our techniques in the context of portfolio verifiers, as they complement and enhance the overall performance when integrated with other verification tools.

Analysis of proposed MaxSAT encoding

The constraints in Lemma 2 can also be encoded using implication constraints. Gurobi [40] provides an API for encoding such constraints, namely `addGenConstrIndicator()`. We also proposed an alternative encoding mechanism, as shown in Equation 4.9. Using the proposed encoding, we were able to solve 59 additional benchmarks.

4.6 Conclusion

In this chapter, we introduced two variations of the CEGAR-based refinement technique. In both techniques, the marked neurons are identified as the source of imprecision, and abstraction is removed from these marked neurons during the refinement process. We provide progress and termination guarantees for both variations. Our experiments demonstrate that our approach successfully solves a significant number of benchmarks that are beyond the reach of state-of-the-art verifiers.

Chapter 5

False Positives in Robustness Verification of Neural Networks

Let us recall the definition of local robustness and counterexamples in the context of neural networks. Many of the works for the verification of neural networks [29, 30, 33] focus on the local robustness verification problem, i.e., for a given neural network and an input image, the network is considered locally robust if all images close to the given image are classified the same as the original, where distance is defined in terms of a simple distance metric (e.g., L_1, L_2, or L_∞ norms). In the verification community, such images, which are close to the original but are classified differently, are more often called *counterexamples*, and existing work focuses on either finding counterexamples or providing certification of their absence.

However, in doing so, existing works often lose the focus on evaluating that if end user would consider such counterexamples to be *genuine*. This leads to false positives, i.e., counterexamples that exist from the perspective of the underlying specification, but may not be considered genuine from a domain expert's perspective. In this chapter, our aim is to identify false and true positives and study their prevalence in commonly used image classification benchmarks. Since the definition of true and false positives depends on the perspective of a domain expert, we introduce the notion of an *oracle* that acts as a domain expert and classifies images as genuine or not. To experimentally evaluate the prevalence of true and false positives, we manually examined a set of counterexamples. Since it is infeasible to have a human in the loop for every classification, we approximate the oracle using an ensemble of classifiers for a larger experimental evaluation.

We first motivate about the true and false positives by examples in Section 5.1 also give the motivating examples behind the ensemble-guided property to reduce the false positives in the same section. Next, we introduce the formal definitions of oracle, true positives, and false positives in Section 5.2. In Section 5.3, we provide the experiments followed by a conclusion in Section 5.4.

5.1 Motivating Examples

Consider the two images in Figures 5.1A and 5.1C, which are from the MNIST [65] dataset and have ground truth labels 0 and 2, respectively. The images in Figures 5.1B and 5.1D are counterexamples generated by the state-of-the-art verifier $\alpha\beta$-CROWN [60] for the image labeled 0 (misclassified as 6), and by the same verifier for the image labeled 2 (misclassified as 1). A domain expert may not consider these as actual *bugs* because the counterexample labeled as 6 also visually resembles a 6, and the network predicts it as either 0 or 6, which is reasonable. Similarly, the counterexample misclassified as 1 also resembles a 1, and the network predicts it as either 1 or 2, which is again acceptable. Since the network's predictions are visually justifiable, labeling them as counterexamples is, in fact, a false positive.

On the other hand, a counterexample may be considered a genuine counterexample, which we call "true positive", if the network misclassifies the image and the misclassification is not visually justifiable. In Figure 5.1E, the image is a seed image, and the image in Figure 5.1F is a corresponding counterexample reported by the $\alpha\beta$-CROWN verifier. The image is classified as 3 by the verifier, and a human expert would say that it is a 1. We present a similar example in Figures 5.1G and 5.1H.

In this work, we recall and define the well-known notions of false positive (FP) and true positive (TP) in the context of local robustness verification of neural network image classifiers. Our modified definition requires a domain expert to classify the images and assign the ground truth labels, which we formally call an oracle. An important point to note is that in our setting, even an oracle may not always be able to classify the images since images may resemble many labels. Thus, it may assign a set of possible labels to an image. As a result, we may use the oracle in two possible ways: (1) to determine whether a counterexample produced by the verifier is a true or false positive by checking whether the classification of the counterexample belongs to one of the oracle-provided labels; and (2) to guide the verifier to search only for counterexamples that are not classified as one of

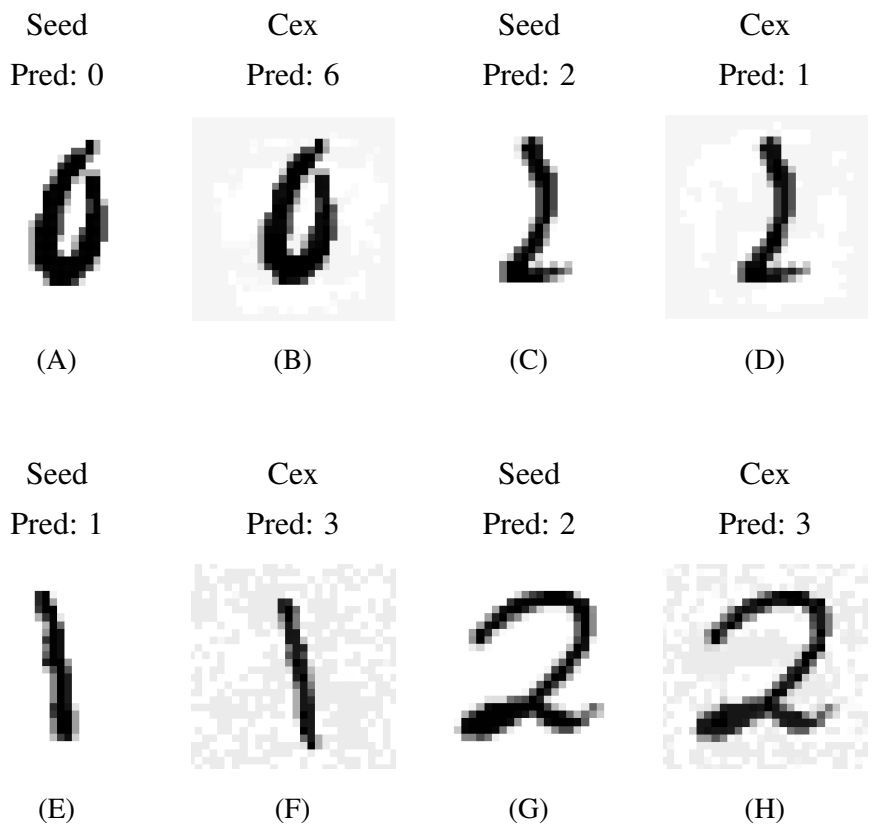


Figure 5.1: 5.1A–5.1D Examples of false positives, 5.1E–5.1H Examples of true positives. For each case, we show two images: the original image ("Seed") and the corresponding counterexample ("Cex") identified by the verifier.

the oracle-provided labels.

For example, in Figure 5.2A, the oracle may state that the original image could be either a 1 or a 7. We first ask the verifier to find a counterexample, and it reports one classified as 1, as shown in Figure 5.2B. We then ask the verifier to look specifically for a counterexample that is not classified as 1 or 7. If such a counterexample is found, as in Figure 5.2C, it qualifies as a true positive. Similarly, in Figures 5.2D-5.2F, the oracle identifies the original image as possibly 4 or 9. We ask the verifier to find a counterexample that is not classified as 4 or 9, and it reports 8—a true positive. Otherwise, it would have reported 4 as a counterexample, which would not be a valid true positive.

It is an open secret that false positives exist in practice. However, we ask a more precise question: what is the rate of false positives in practice (RQ1)? To answer this, we evaluate the standard local robustness property with respect to false positives and true positives.

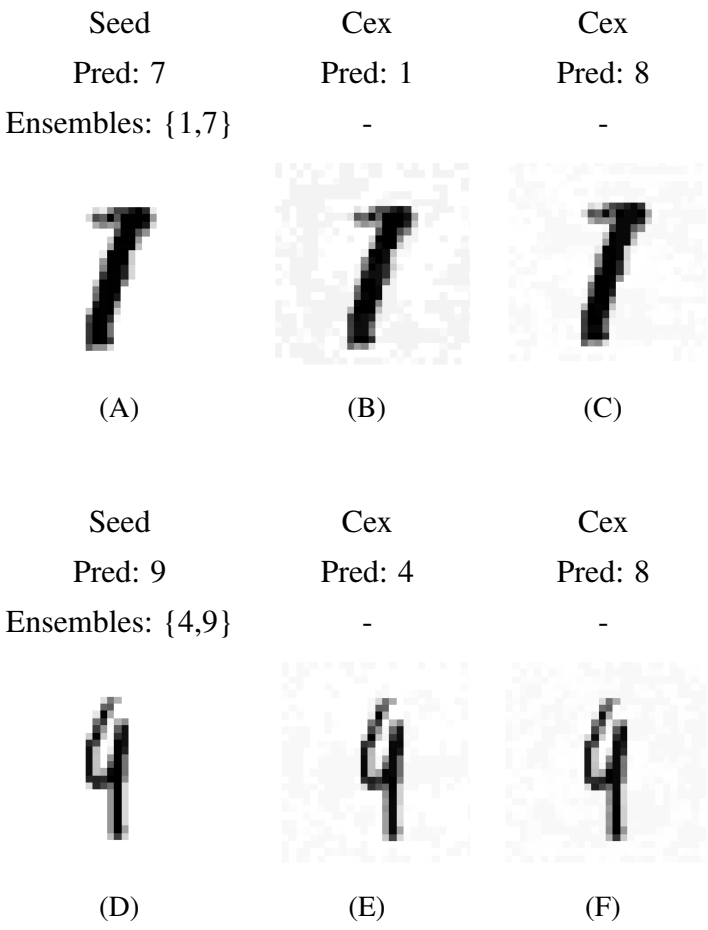


Figure 5.2: Examples of true positives guided by ensemble classifiers. For each case, we show three images: the original image ("Seed"), the counterexample ("Cex") identified without ensemble guidance (middle), and the counterexample ("Cex Ensemble") identified with ensemble guidance (right).

To approximate the domain expert (oracle), we develop an ensemble-based approach that exploits multiple neural network classifiers, as we detail below.

We conduct experiments using state-of-the-art neural network verifier $\alpha\beta$-CROWN [60] on the MNIST [65] and CIFAR-10 [77] datasets. The ensemble consists of 9 classifiers for MNIST and 13 for CIFAR-10, taken from the ERAN benchmark repository [173]. A label is considered valid if at least 30% of the classifiers agree on it. Our experiments reveal that approximately 6.9% and 1.95% of the counterexamples generated by the verifiers are false positives for the MNIST and CIFAR-10 datasets, respectively. We also manually checked whether the label assigned by the ensemble to each counterexample is accurate,

which forms our second experimental research question (RQ2). We found that the false positive rates based on manual analysis were 12.86% for the MNIST benchmarks and 5.65% for the CIFAR-10 benchmarks. We also ask if there are counterexamples that are not classified as one of the ensemble labels (RQ3), which would be considered true positives under the ensemble definition. To answer this, we modified the verifiers to search for such counterexamples and reran them on both the MNIST and CIFAR-10 benchmarks. We found that the false positive rate for MNIST benchmarks reduced significantly from 12.86% to 3.22% and true positives for the same benchmarks increase from 88.63% to 96.78%. In summary, our approach and novel analysis show that the local robustness as is usually considered can be overly conservative. And thus, verifiers are not able to declare a model trustworthy even when the model is actually trustworthy according to the domain experts.

5.2 Definitions

In this section, we define the oracle and explain the classification of counterexamples as true positives and false positives in robustness verification. As we will see in Section 5.3, many counterexamples under standard local robustness are false positives. We also provide an *ensemble-based approach* to reduce false positives.

Recall that a neural network $N : \mathbb{R}^n \to \mathbb{R}^m$ is a function that takes an n-dimensional input and produces an m-dimensional output. Here, $N(x)[i]$ denotes the logit value corresponding to the i^{th} output neuron. The final decision $\hat{N}$ of the neural network is determined using the ARGMAX [101] function, which returns the index of the maximum output value: $\hat{N}(x) = \text{ARGMAX}(N(x))$.

5.2.1 Oracle

An oracle is a function that takes an input x and assigns one or more labels to the input. The oracle models the domain expert's view of the input image x. Let us assume the neural network classifies an input x into one of the classes from the set $[m]$, where $[m] = \{0, 1, \ldots, m-1\}$.

Definition 5. *An oracle $\mathbb{O}$ is a function $\mathbb{O} : \mathbb{R}^n \to 2^{[m]}$, where $2^{[m]}$ is the power set of $[m]$. The oracle assigns one or more labels to the input image x.*

Intuitively, the oracle $\mathbb{O}$ assigns the ground truth labels to the input image x. It may assign more than one label to an image if the image visually resembles more than one class.

For example, in Figure 5.1A, the image is a seed image x, which resembles both 0 and 6, we may have $\mathbb{O}(x) = \{0, 6\}$. Using the oracle, we can define the notions of true and false positives in the context of robustness verification of neural networks.

5.2.2 True Positives and False Positives

Definition 6. *An input counterexample x' is a* true positive *in a neural network N if* $\hat{N}(x') \notin \mathbb{O}(x')$

Definition 7. *An input counterexample x' is a* false positive *in a neural network N if* $\hat{N}(x') \in \mathbb{O}(x')$

In Figure 5.1A, the image is a seed image x with $\hat{N}(x) = 0$, and the image in Figure 5.1B is a corresponding counterexample x' reported under local robustness property with $\hat{N}(x') = 6$. Since $\mathbb{O}(x') = \{0, 6\}$, we have $\hat{N}(x') \in \mathbb{O}(x')$. Therefore, it is not a genuine bug, and we consider it a false positive. A false positive does not violate the oracle's view of local robustness, as the oracle $\mathbb{O}$ considers the counterexample x' to be similar to the seed image x, but violates the standard property. On the other hand, a true positive violates both the oracle's view of local robustness and the standard local robustness property. In Figure 5.1E the image is a seed image x with $\hat{N}(x) = 1$, and the image in Figure 5.1F is a corresponding counterexample x' reported as per the standard property with $\hat{N}(x') = 3$. Since $\mathbb{O}(x') = \{1\}$ and $\hat{N}(x') \notin \mathbb{O}(x')$, this indicates a true counterexample, as the oracle's classification and the neural network's classification are completely different. We can use true positives and false positives to evaluate the results of the standard local robustness property.

5.2.3 Ensemble-Based Approach to Reduce False Positives

As illustrated in Figures 5.1A and 5.1B, the original image has a ground truth label of 0, but it also visually resembles 6. Therefore, it is unreasonable to strictly require the classifier not to predict 6; rather, it should be allowed to classify the image as either 0 or 6. Motivated by this intuition, we define an ensemble and describe its use for reducing false positive cases.

Definition 8. *Let $\mathbb{A}$ be a set of AI models and agreement threshold k be a positive integer. We define an* ensemble oracle *$E(x)$ such that, for each x, $E(x) = \{y \mid |\{N \in \mathbb{A} \mid \hat{N}(x) = y\}| \geq k\}$.*

The ensemble oracle $E(x)$ returns a set of labels for the input image x based on the agreement of the AI models in $\mathbb{A}$. The ensemble is defined such that it returns a label if

Table 5.1: Ensemble of neural network classifiers used for the MNIST dataset

Network name	#layers	adv trained	adv training methods
mnist-relu-5-100.onnx	5-FC	Yes	DiffAI
ffnnRELU-Point-6-500.onnx	6-FC	No	-
ffnnRELU-PGDK-w-0.1-6-500.onnx	6-FC	Yes	PGD
ffnnRELU-PGDK-w-0.3-6-500.onnx	6-FC	Yes	PGD
convSmallRELU-Point.onnx	2-Conv, 2-FC	No	-
convSmallRELU-PGDK.onnx	2-Conv, 2-FC	Yes	PGD
convSmallRELU-DiffAI.onnx	2-Conv, 2-FC	Yes	DiffAI
convMedGRELU-Point.onnx	2-Conv, 2-FC	no	-
convBigRELU-DiffAI.onnx	4-Conv, 3-FC	Yes	DiffAI

Table 5.2: Ensemble of neural network classifiers used for the CIFAR-10 dataset

Network name	#layers	adv trained	arv training method
cifar-relu-9-200.onnx	10-FC	No	-
ffnnRELU-PGDK-w-0.0078-6-500.onnx	7-FC	Yes	PGD
convSmallRELU-Point.onnx	2-Conv, 2-FC	No	-
convSmallRELU-PGDK.onnx	2-Conv, 2-FC	Yes	PGD
convSmallRELU-DiffAI.onnx	2-Conv, 2-FC	Yes	DiffAI
convMedGRELU-Point.onnx	2-Conv, 2-FC	No	-
convMedGRELU-PGDK-w-0.0313.onnx	2-Conv, 2-FC	Yes	PGD
cifar-conv-maxpool.onnx	4-Conv, 2-Maxpool, 3-FC	no	-
convBigRELU-DiffAI.onnx	4-Conv, 3-FC	Yes	DiffAI
ResNetTiny-PGD.onnx	1-Conv, 5-res-blocks (3-Conv), 2-FC	Yes	PGD
ResNet18-PGD.onnx	1-Conv, 5-res-blocks (2-Conv), 3-res-blocks (3-Conv), 2-FC	Yes	PGD
ResNet34-DiffAI.onnx	1-Conv, 13-res-blocks (2-Conv), 3-res-blocks (3-Conv), 3-FC	Yes	DiffAI
SkipNet18-DiffAI.onnx	11-Conv, 3-res-blocks (3-Conv), 3-FC	Yes	DiffAI

at least k classifiers agree on that label. Here, k is a hyperparameter. If no label meets this criterion, we relax the requirement by decrementing the value of k. If k reaches 1, we select the label with the highest prediction confidence. The AI models can be of various types, such as neural networks, decision trees, or random forests. These models may be trained on different datasets, which might not always be publicly accessible. By leveraging such models, we aim to incorporate the collective knowledge of the community into our analysis. In this work, we use neural network classifiers as our AI models. For the MNIST dataset, we use $|A| = 9$ classifiers with a threshold $k = 3$, and for the more complex CIFAR-10 dataset, we use $|A| = 13$ classifiers with $k = 4$. The intuition behind these thresholds is that a label is accepted if approximately 30% of the networks agree on it. If we choose k to be large, we may disallow diversity of opinion, and if we choose k to be small, a single member of the ensemble may pollute the predictions. We use all

the classifiers in ensembles from the ERAN repository [173], ranging from simple fully connected to complex classifiers, from simply trained to adversarially trained classifiers. To adversarially train the networks [173], DiffAI [172] and PGD [174] methods are used. DiffAI uses abstract interpretation to create a range of safe adversarial examples, helping the model become more robust in different input areas. PGD (Projected Gradient Descent) is an attack-based method that trains the model using adversarial examples made by slightly changing inputs through gradient steps within a fixed limit. Tables 5.1 and 5.2 show details about the selected neural network classifiers.

Definition 9. *For a neural network N, an input x, and an input perturbation value ϵ, N is considered* ensemble-guided locally robust *at x if $\hat{N}(x) \in E(x)$ and for every x' such that $dist(x', x) \leq \epsilon$, we have $\hat{N}(x') \in E(x)$.*

We can modify the solver query for the ensemble-aware local robustness property as $\exists x'\ dist(x', x) \leq \epsilon \wedge \bigvee_{i=1, i \notin E(x)}^{m} \bigwedge_{j \in E(x)} N(x')[j] \leq N(x')[i]$. If this query is satisfiable, then we obtain a counterexample x' on which the network N disagrees with the ensemble prediction on the original input x.

The above definition is similar to the property (affinity robustness) presented in [70], where the expert provides a set of sets of possible labels among which the network must classify. In affinity robustness, the top-k classes refer to the k classes with the highest logit values ($N(x)$) in the network output. Affinity robustness requires that for some k, the top-k classes remain the same across both x and x', and moreover, the top-k classes must be a subset of one of the expert-provided sets of labels. In contrast, our proposed local robustness property requires only that the top-1 (predicted) output of the neural network on input x' belongs to the set of labels provided by the ensembles. Our property is incomparable to the notion of affinity robustness.

5.3 Experiments

We evaluate the standard robustness property using the state-of-the-art (SOTA) verifier $\alpha\beta$-CROWN [60] by reporting the number of false positives and true positives on a standard class of benchmarks.

Benchmarks: We considered networks trained on the MNIST [65] and CIFAR-10 [77] datasets. All networks were obtained from VNNCOMP [73, 175, 75, 76]. Table 7.1 provides the details of these networks.

Table 5.3: Networks details

Category	**Network name**	#layers	#activation units	adv trained	ϵ-values
MNIST	mnist-net-256×2.onnx	2-FC	0.51K	No	0.03,0.05
	mnist-net-256×4.onnx	4-FC	1.02K	No	0.03,0.05
	mnist-net-256×6.onnx	6-FC	1.54K	No	0.03,0.05
CIFAR-10	cifar-base-kw.onnx	2-Conv, 2-FC	3.17K	Yes	0.03
	cifar-deep-kw.onnx	4-Conv, 2-FC	6.77K	Yes	0.03
	cifar-wide-kw.onnx	2-Conv, 2-FC	6.24K	Yes	0.03

We randomly selected 1000 images from each dataset. For the neural networks trained on MNIST, we used two different values of input perturbation ϵ, resulting in approximately 2000 benchmark instances per network. For CIFAR-10 networks, we used a single value of ϵ, leading to approximately 1000 benchmark instances per network. The values of ϵ were chosen to match those used in VNNCOMP and are shown in the last column of Table 7.1.

Many neural networks approximate the human decision-making process with the goal of reducing human effort. However, it is impractical for humans to act as oracles for all counterexample images. To address this issue, we propose using an ensemble of AI models as a proxy for the oracle, as defined in Definition 8.

We conduct the following three experiments: we identify the false positives (FP) and true positives (TP) for the standard robustness property using the ensemble, we manually analyze the accuracy of the decisions of the ensemble, and we construct the ensemble-based robustness property verifier and manually validate the resulting false positives and true positives.

False Positives Analysis Using Ensemble (RQ1): Tables 5.4(a) and 5.4(b) present the number of true positives (TP) and false positives (FP) identified using the ensemble-based oracle defined above. We do not analyze false negatives or true negatives, as it is generally infeasible to determine in advance whether a neural network is locally robust around a given input. To empirically evaluate RQ1, we observed false positive rates of 6.9% for the MNIST benchmarks and 1.95% for the CIFAR-10 benchmarks. The FP rate is higher for the MNIST benchmarks compared to the CIFAR-10 benchmarks. A potential reason is that many MNIST images resemble more than one label, whereas CIFAR-10 images tend to be more distinct. Overall, we observe that the number of false positives is significant and merits further consideration.

Manual Analysis of the Ensemble (RQ2): To evaluate the accuracy of the ensemble, we conducted a manual analysis of the false positives (FPs) and true positives (TPs)

Table 5.4: Results obtained using the standard robustness property with $\alpha\beta$-CROWN, along with ensemble-based analysis of False Positives (FP) and True Positives (TP) for the MNIST (Table a) and CIFAR-10 (Table b) networks.

Result	mnist-net-256×2.onnx		mnist-net-256×4.onnx		mnist-net-256×6.onnx	
epsilons	0.03	0.05	0.03	0.05	0.03	0.05
FP	48	49	30	33	16	14
TP	647	832	310	551	297	493
Verified	294	96	518	160	402	102
Timeout	4	16	68	182	125	231

(a) MNIST results

Result	cifar-base-kw.onnx	cifar-deep-kw.onnx	cifar-wide-kw.onnx
epsilons	0.03	0.03	0.03
FP	12	18	45
TP	603	673	880
Verified	14	10	13
Timeout	38	36	62

(b) CIFAR-10 results

Table 5.5: MNIST results using the ensemble-based approach with $\alpha\beta$-CROWN, along with manual analysis of FP and TP

Result	mnist-net-256×2.onnx		mnist-net-256×4.onnx		mnist-net-256×6.onnx	
epsilons	0.03	0.05	0.03	0.05	0.03	0.05
FP	22	28	12	27	13	22
TP	660	846	324	577	321	522
Verified	307	103	551	174	431	116
Timeout	4	16	65	180	121	230

reported in Tables 5.4(a) and 5.4(b). Since it is impractical to manually examine all positive cases, we focused on the first column of each table (`mnist-net-256×2.onnx` and `cifar-base-kw.onnx`, with $\epsilon = 0.03$). For MNIST, we found that 13 out of 48 reported FPs were actually TPs, and 44 out of 647 reported TPs were actually FPs. For CIFAR-10, 4 out of 12 reported FPs were actually TPs, and 27 out of 603 reported TPs were actually FPs. Overall, this results in a false positive rate of 12.86% for MNIST and 5.67% for CIFAR-10 on the selected benchmark networks. This analysis demonstrates that although an approximation of the oracle can reduce human effort, it is not perfect.

Ensemble-based Property Analysis (RQ3): We use the robustness property expressed in Definition 9, and use humans as oracles to analyze counterexamples. Since human evaluation for all counterexamples is impractical, we limit this analysis to the `mnist-net-256×2.onnx` network with $\epsilon = 0.03$. Table 5.5 summarizes the number of FPs and TPs. We observed that the false positive rate dropped from 12.86% to 3.22%, while the true positive rate increased from 88.63% to 96.78%.

5.4 Conclusion

In this chapter, we considered true and false positives in the context of the local robustness using oracles. We used an ensemble of classifiers as an approximation of the oracle and analyzed the robustness of neural networks with respect to this ensemble. We showed that

the standard local robustness property can report many false positives under this oracle and proposed a new ensemble-guided local robustness property that can help to reduce the number of false positives. Robustness property is widely considered to be a useful property to analyze the quality of neural networks. However, we demonstrate that the violation of the property may be false positives. Their prevalence helps guide a user of neural network developers to limit their reliance on the local robustness analysis.

Chapter 6

Confidence-aware Robustness Properties of Neural Networks

Most researchers [100, 25, 34] have analyzed neural networks with respect to the local robustness property. As discussed in Section 2.4 of Chapter 2, the local robustness property captures potential misclassifications within an ϵ-bounded neighborhood of an input. In the previous chapter, we observed a significant number of false positives in the local robustness property. Now, we explore different types of robustness properties. Although neural networks are primarily used for classification, they also output a confidence value associated with each prediction. Several works in the field of adversarial example generation [10] have shown that confidence values play a crucial role in understanding the behavior of neural networks under adversarial perturbations. Building on the prior works, we introduce various robustness properties that explicitly incorporate prediction sensitivity/confidence, including relaxed robustness, strong robustness, and smoothness.

As defined in Equation 2.1, confidence involves highly non-linear operations. However, most of the analysis tools are built on linear solving techniques. We need to model confidence as piecewise linear formulas to leverage these tools for verifying confidence-based robustness properties. We propose linear approximations of confidence within the framework of Linear Real Arithmetic (LRA) [176]. We also show that the property of Top-k robustness [70], which does not involve confidence values, can be modeled using LRA. Furthermore, in Chapter 7, we present an efficient unified approach to handle all LRA-based postconditions.

In this chapter, we first introduce various confidence-based robustness properties in

Section 6.1 and provide LRA-based modeling techniques for these properties in Section 6.2. Next, we present non-confidence-based robustness variants in Section 6.3 and establish a hierarchy among all the robustness properties discussed in this chapter in Section 6.4. Finally, we conclude the chapter in Section 6.5. In the following chapter, we present an encoding framework to verify all the different robustness properties in a unified way.

6.1 Different Robustness Variants

Different applications require different variants of robustness (e.g., see [66]). Further, many existing works on robustness verification regard answers returned by classifiers as binary, ignoring the fact that classifiers provide a confidence in the counterexample in terms of the classification probability, which is computed by the well-known Softmax function [46]. For instance, in Figure 6.1(a-b), an image from the CIFAR-10 [77] dataset is misclassified after an input perturbation, but with a very low confidence. Should such a network be called non-robust if all misclassifications are such low confidence ones? This motivates a more *relaxed* notion of robustness which would allow such examples. At the other extreme, one could argue for a *stronger* notion of robustness, saying that the network in Figure 6.1(c-e), should be called non-robust since under minor input perturbations, the confidence in the classification has vastly changed (even if no misclassification is reported). Finally, we may consider a Lipschitz-continuity motivated smoothness criterion [66, 67, 68, 69] which says that a neural network is not *smooth-robust* if there are counterexamples with varying confidences as depicted in Figure 6.1(f-g).

When confidence is included in the property definition, it becomes necessary to approximate the confidence (Softmax) function to analyze these properties effectively, which we do, with formal guarantees on the error.

6.2 Modeling Confidence Based Robustness Variants

Recall from Section 2.4 of Chapter 2 that the local robustness definition does not capture the network's confidence in its predictions. However, as discussed in the Introduction, it is often meaningful to consider variants that account for prediction confidence. The softmax-based confidence, as defined in Equation 2.1, involves exponential operations, while most existing solver techniques are primarily designed for linear constraints or their Boolean combinations. Consequently, there is no straightforward way to incorporate confidence measures into existing solvers, as also noted in [66, 114]. Therefore, to integrate

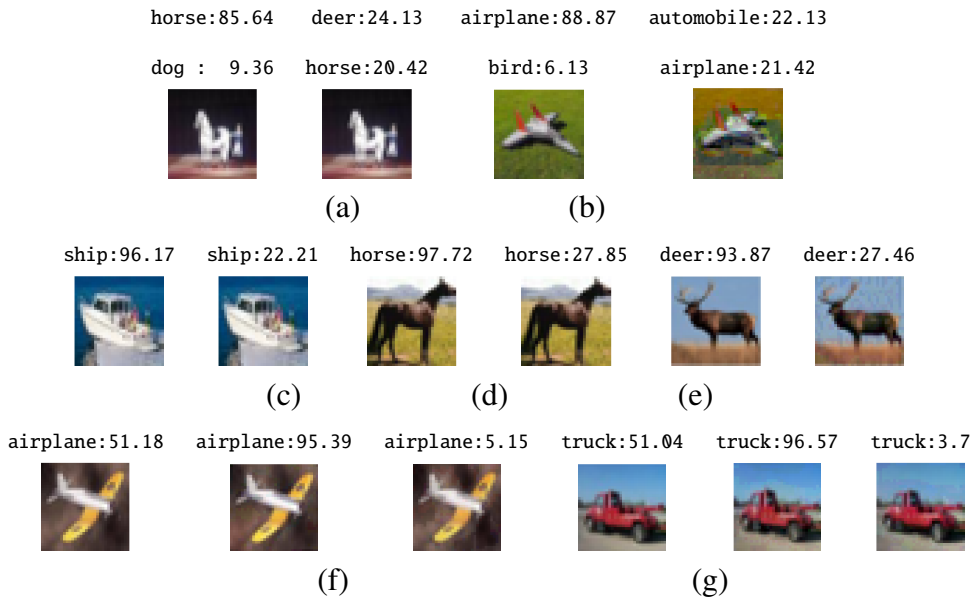


Figure 6.1: **Relaxed Robustness: (a-b)** The network `convBigRELU-PGD.onnx` correctly classified the original image (left) as `horse` (and resp. `airplane`) with high confidence. With an input perturbation of 16/255, we can find misclassified images (right) but with low confidence. In fact, it turns out all counterexamples have low confidence and hence verification succeeds under the relaxed robustness criterion with an 80% confidence threshold, while state-of-the-art verifiers would have declared this network non-robust. **Strong robustness: (c-e)** The network `convBigRELU-PGD.onnx` classified the original image (left) of class `ship/horse/deer` with very high confidence, and we found images (right) within perturbation of 16/255, such that the confidence drops drastically, although the class remains the same. These images are robust with respect to the standard and relaxed robust criteria but not robust with respect to the strong robust criteria if confidence is allowed to fall up to 30%. **Smoothness: (f-g)** The left image, labeled as `Airplane/Truck`, is taken from the Cifar-10 dataset and is classified correctly with a medium confidence of $\sim 50\%$ by the neural network `cifar10-2-255.onnx`. Under an $\epsilon = 16/255$ perturbation, we obtain images with much higher and lower confidences, showing drastic variations. The bottom line is that the above requirements may vary across applications, and users can define many more requirements tailored to specific needs.

confidence with current solver frameworks, one must rely on appropriate approximations.

In this section, we present a list of confidence-aware robustness properties, including both newly proposed ones and those inspired by prior studies. We formalize these properties

as modified verification queries and discuss suitable approximation strategies for enabling their analysis within existing solver frameworks.

6.2.1 Relaxed Robustness

Let us first discuss a weaker notion of robustness that we called *relaxed robustness*, where the intuition is to ignore low-confidence counterexamples in determining robustness of the network, as shown in Figure 6.1(a-b). The main idea is that for all images x close to the given image x^*, their classification should be the same, unless the confidence of the network on x is less than a certain threshold. Incorporating the requirement in Definition 2 gives us Equation 6.1, which permits low-confidence counterexamples to bypass the strict robustness criteria. Let τ be the confidence level threshold, a tunable user specified parameter.

$$\forall x \; \texttt{dist}(x^*, x) \leq \epsilon \implies \Big(\textsc{Conf}(N(x), \hat{N}(x)) < \tau \vee \hat{N}(x^*) = \hat{N}(x)\Big) \tag{6.1}$$

Hence, for the above property, P is as in Definition 2, but we have a new post-condition $Q_{rel} = \textsc{Conf}(N(x), \hat{N}(x)) < \tau \vee \hat{N}(x^*) = \hat{N}(x)$. Thus if the query $\langle N, P, Q_{rel}\rangle$ holds, we say that N is τ-relaxed robust (or just relaxed robust when clear from context), and otherwise, i.e., if there exists x satisfying the negation, $\neg Q_{rel}$, we say that there exists a misclassified counterexample x with confidence at least τ. Since computing $\textsc{Conf}(N(x), \hat{N}(x))$ (a non-linear function) precisely is difficult, we approximate it. Let $y_1, \ldots, y_m = N(x)$, and define $t = \hat{N}(x)$, implying $y_t = \max_{i=1}^{m}(y_i)$. Additionally, let $y_{t'}$ denote the second-highest output: $y_{t'} = \max_{i=1, i\neq t}^{m}(y_i)$. We have the following claim for any $\tau \geq 50$, which leads to our approximation. Let $\delta = -\ln(\frac{100}{\tau} - 1)$

Claim 1. *If $y_t < y_{t'} + \delta$ holds, then $\textsc{Conf}(N(x), \hat{N}(x)) < \tau$*

Proof. Assume LHS of the claim holds, i.e., $y_t < y_{t'} + \delta$. By removing all exponential terms from the denominator except for e^{y_t} and $e^{y_{t'}}$, we obtain $\textsc{Conf}(y_1, \ldots y_m, t) \leq \frac{100e^{y_t}}{e^{y_t} + e^{y_{t'}}}$. After scaling the right-hand side by $\frac{1}{e^{y_t}}$, we get $\textsc{Conf}(y_1, \ldots y_m, t) \leq \frac{100}{1+e^{-(y_t - y_{t'})}}$. Since $\delta > y_t - y_{t'}$, we have $\textsc{Conf}(y_1, \ldots y_m, t) < \frac{100}{1+e^{-\delta}}$. Since $\delta = -\ln(\frac{100}{\tau} - 1)$, $\textsc{Conf}(y_1, \ldots y_m, t) < \tau$. Finally, noting that $(y_1, \ldots, y_m) = N(x)$, and from definition of t, we get $\textsc{Conf}(N(x), \hat{N}(x)) < \tau$. □

Claim 2. *If $y_t \geq y_{t'} + \delta$ holds, then $\textsc{Conf}(N(x), \hat{N}(x)) \geq \frac{100}{1+(m-1)e^{-\delta}}$.*

Proof. Let LHS of the claim be true. Since $y_{t'}$ is the second maximum, we can derive $\textsc{Conf}(y_1, \ldots y_m, t) \geq \frac{100e^{y_t}}{e^{y_t} + (m-1)e^{y_{t'}}}$, replacing e^i by $e^{t'}$ for all $i, i \neq t$. After scaling the fraction

in the RHS by e^{y_t}, we obtain $\textsc{Conf}(y_1, ...y_m, t) \geq \frac{100}{1+(m-1)e^{y_{t'}-y_t}}$. Since $y_t \geq y_{t'} + \delta$, we obtain $\textsc{Conf}(y_1, ...y_m, t) \geq \frac{100}{1+(m-1)e^{-\delta}}$. □

Our approximation is similar to the one presented in [114]. While their approach employs the constraint $y_t < y_{t'} + \delta \wedge t \neq \hat{N}(x)$ to approximate the second-highest output, our formulation compares y_t against every other logit value: $\bigwedge_{j=1, j\neq t}^{m} y_t \geq y_j + \delta$. Similarly, we use a disjunction to identify the maximum y_t. To enforce misclassification, we add the condition $t \neq \hat{N}(x)$. Consequently, we derive $\neg Q'_{rel}$ as follows.

$$\bigvee_{i=1, i\neq\hat{N}(x)}^{m} \quad \bigwedge_{j=1, j\neq i}^{m} \Big(y_i \geq y_j + \delta\Big) \tag{6.2}$$

where, y_i, y_j are the i^{th} and j^{th} outputs in $N(x)$, and $\delta = -\ln(\frac{100}{\tau} - 1)$. The above equation is in LRA form. Note that the conjunct $(\hat{N}(x^*) \neq \hat{N}(x))$ in Eq 6.1 is already subsumed in Eq 6.2 above. Thus, we obtain the following theorem:

Theorem 3. *Given verification query $\langle N, P, Q'_{rel}\rangle$,*

1. *if the query holds true, then $\langle N, P, Q_{rel}\rangle$ holds, i.e., there is no misclassified counter-example in P with confidence greater than $\frac{100}{1+e^{-\delta}} = \tau$.*

2. *else, there is a misclassified input $x \models P$ with confidence greater than or equal to $\frac{100}{1+(m-1)e^{-\delta}}$.*

Proof. 1. Suppose the query holds true, which implies that Eq 6.2 is unsatisfiable. This means that $\forall i, i \neq \hat{N}(x), \exists j, j \neq i, \quad y_i < y_j + \delta$. Consequently, there exists some $j \neq max$ such that $y_{\max} < y_j + \delta$, where $y_{\max} = \max(y_i)$. This further implies $y_{\max} < y_{\text{smax}} + \delta$, since $y_j \leq y_{\text{smax}}$, where y_{smax} is the second highest value in the output layer. By Claim 1, we conclude that no counterexample exists with confidence greater than $\frac{100}{1+e^{-\delta}}$.

2. Suppose the query does not hold, which implies that Eq 6.2 is satisfiable. This means there exists some y_i (where $y_i \neq y_{\hat{N}(x)}$) that is greater than all other logit values by a margin δ, with $\delta \geq 0$. Thus, $y_{\max} \geq y_{\text{smax}} + \delta$. By Claim 2, this counterexample of class *max* has confidence greater than $\frac{100}{1+(m-1)e^{-\delta}}$.

□

Theorem 3 ensures the soundness of our approximation (part 1) and provides a lower bound on the confidence of any counterexample (part 2). Specifically, if the derived query

Q'_{rel} holds, then no counterexample exists with confidence greater than the threshold τ. Otherwise, any counterexample must have a confidence of at least $\frac{100}{1+(m-1)e^{-\delta}}$, which is denoted as τ_{lb}.

In Figure 6.2, we illustrate the estimation of the lower bound by plotting confidence vs the gap between the top two outputs. The thick curve plots the user defined thresholds, and the dashed curve plots the lower bound approximation. For a target confidence threshold τ on the Y-axis, a horizontal line to the user defined threshold curve finds the δ, while the vertical line from the intersection crosses the lower bound curve at τ_{lb}; the lower bound for any counterexample. From Figure 6.2, we observe that τ_{lb} is close to τ when τ is large, indicating a tight approximation. In contrast, for smaller values of τ, the gap between τ_{lb} and τ becomes larger, which indicates a looser abstraction.

Let us discuss the comparison between our softmax approximation and the approximation introduced in [114], which introduced lower and upper bounds for the softmax function as follows. Let c be the index of the maximum value, i.e., $c = \arg\max_{i=1}^{m} y_i$:

$$\text{Sig}(y_c - \max_{i=1,i\neq c}^{m}(y_i) - \log(m-1)) \leq \text{Softmax}(y_1, \ldots, y_m, c) \leq \text{Sig}(y_c - \max_{i=1,i\neq c}^{m}(y_i)) \quad (6.3)$$

where $\text{Sig}(x)$ is the sigmoid function. For analysis, consider the lower bound:

$$\begin{aligned} lb &= \text{Sig}(y_c - \max_{i=1,i\neq c}^{m}(y_i) - \log(m-1)) = \frac{1}{1+e^{-y_c+\max_{i=1,i\neq c}^{m}(y_i)+\log(m-1)}} \\ &= \frac{1}{1+e^{-y_c+\max_{i=1,i\neq c}^{m}(y_i)}e^{\log(m-1)}} = \frac{1}{1+(m-1)e^{-y_c+\max_{i=1,i\neq c}^{m}(y_i)}} \\ &= \frac{1}{1+(m-1)e^{-(y_{\max}-y_{\text{smax}})}}, \end{aligned}$$

where $y_{\max} = \max_{i=1}^{m} y_i$ and $y_{\text{smax}} = \max_{i=1,i\neq\max}^{m} y_i$. Intuitively, $y_{\max}$ and y_{smax} represent the maximum and second maximum values of y_i, respectively. Our approximation also uses the relation between $y_{\max}$ and y_{smax} to define lower and upper bounds, but it introduces a user-defined confidence threshold τ to guide the approximation. Our work identifies that the difference $y_{\max} - y_{\text{smax}}$ plays a crucial role in encoding the property in Linear Real Arithmetic (LRA).

6.2.2 Strong Robustness

The second variant of robustness, that we call *strong robustness*, aims to capture cases where the confidence on the seed image is high, but drops below a certain threshold within an ϵ-bounded perturbation, regardless of whether the classification label changes. In

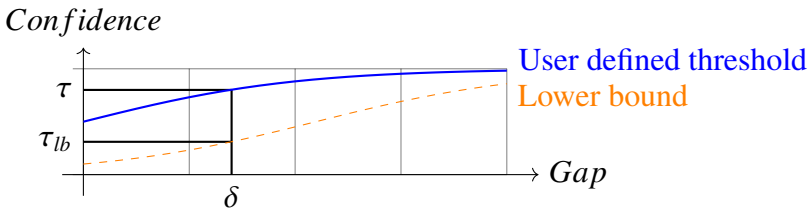


Figure 6.2: Behavior of lower bound τ_{lb} and the user defined threshold τ approximations of $softmaxC$.

other words, a significant drop in confidence itself indicates a weakness in the underlying network. Recall that an example of such robustness was illustrated in Figure 6.1(c-e), where the confidence for a seed image of class `ship` dropped from 96.17% to 22.21%. The definition of strong robustness, which generalizes the notion introduced in [66], can now be formally defined as follows. Let τ_1 and τ_2 be two threshold values such that $\tau_1 > \tau_2$.

$$\forall x \; \texttt{dist}(x^*, x) \leq \epsilon \implies \Big(\textsc{Conf}(N(x), \hat{N}(x^*)) > \tau_2 \wedge \hat{N}(x^*) = \hat{N}(x)\Big) \\ \vee \textsc{Conf}(N(x^*), \hat{N}(x^*)) < \tau_1 \tag{6.4}$$

The above equation asserts that for any x in the ϵ-neighborhood of x^*, if the confidence of the network on x^* is at least τ_1, then x must be classified the same as x^* with a high confidence (above threshold τ_2). Thus, we get the query $\langle N, P, Q_{str}\rangle$ where P remains the same, while $Q_{str} = (\textsc{Conf}(N(x), \hat{N}(x^*)) > \tau_2 \wedge \hat{N}(x^*) = \hat{N}(x)) \vee \textsc{Conf}(N(x^*), \hat{N}(x^*)) < \tau_1$. Note that $\textsc{Conf}(N(x^*), \hat{N}(x^*)) < \tau_1$ can be computed beforehand since x^* is a given seed image. Thus, we will only need to approximate the condition $\textsc{Conf}(N(x'), \hat{N}(x)) > \tau_2$, which we do as follows.

Let $t = \hat{N}(x^*)$ and $y_1, \ldots, y_m = N(x)$, with $y_{t'} = \max_{i=1, i\neq t}^{m}(y_i)$ being the second maximum. For any τ_2, let $\delta = -\ln(\frac{1}{m-1}(\frac{100}{\tau_2} - 1))$, where $\delta \geq 0$. Now we can show:

Claim 3. *If $y_t > y_{t'} + \delta$ holds, then $\textsc{Conf}(N(x'), t) > \tau_2 \wedge \hat{N}(x^*) = \hat{N}(x)$.*

Proof. The following proof is similar to the proof for Claim 2. Let LHS of the claim be true. Since $y_{t'} = \max_{i=1, i\neq t}^{m}(y_i)$, we can derive $\textsc{Conf}(y_1, ...y_m, t) \geq \frac{100e^{y_t}}{e^{y_t} + (m-1)e^{y_{t'}}}$, replacing e^i by $e^{t'}$ for all $i, i \neq t$. After scaling the fraction in the RHS by e^{y_t}, we obtain $\textsc{Conf}(y_1, ...y_m, t) \geq \frac{100}{1+(m-1)e^{y_{t'} - y_t}}$. Since $y_t \geq y_{t'} + \delta$, we obtain $\textsc{Conf}(y_1, ...y_m, t) \geq \frac{100}{1+(m-1)e^{-\delta}}$. Since $\delta = -\ln(\frac{1}{m-1}(\frac{100}{\tau_2} - 1))$, $\textsc{Conf}(y_1, ...y_m, t) > \tau_2$. □

Claim 4. *If $y_t \leq y_{t'} + \delta$ holds, then $\textsc{Conf}(N(x'), t) \leq \frac{100}{1+e^{-\delta}}$.*

Proof. The following proof is similar to the proof for Claim 1. Assume LHS of the claim holds, i.e., $y_t \leq y_{t'} + \delta$. By removing all exponential terms from the denominator except for

e^{y_t} and $e^{y_{t'}}$, we obtain $\textsc{Conf}(y_1, ...y_m, t) \leq \frac{100e^{y_t}}{e^{y_t}+e^{y_{t'}}}$. After scaling the right-hand side by e^{y_t}, we get $\textsc{Conf}(y_1, ...y_m, t) \leq \frac{100}{1+e^{-(y_t-y_{t'})}}$. Since $\delta \geq y_t - y_{t'}$, we have $\textsc{Conf}(y_1, ...y_m, t) \leq \frac{100}{1+e^{-\delta}}$. □

The constraint $y_t > y_{t'} + \delta$ allows us to define the new post-condition Q'_{str}, where instead of explicitly computing $y_{t'}$, we perform comparisons with each y_i using disjunction. Formally, we define $Q'_{str} = \neg\left(\bigvee_{i=1,i\neq t}^{m} y_t \leq y_i + \delta\right)$ and obtain a revised verification query with following guarantees.

Theorem 4. *Given verification query* $\langle N, P, Q'_{str}\rangle$,

1. *if the query holds, then* $\langle N, P, Q_{str}\rangle$ *holds, i.e., all inputs in* P *are correctly classified with confidence greater than* $\frac{100}{1+(m-1)e^{-\delta}} = \tau_2$.

2. *else, there is an example* $x \models P$ *such that* x *is either misclassified or has confidence less than* $\frac{100}{1+e^{-\delta}}$.

Proof. 1. Suppose the query holds, i.e., $\forall i \neq t$, $y_t > y_i + \delta$. This implies that $y_t > y_{t'} + \delta$, where $y_{t'} = \max_{i\neq t}(y_i)$ and $\delta \geq 0$. Hence, y_t is the maximum logit, and no misclassification occurs. Furthermore, by Claim 3, the confidence is greater than $\frac{100}{1+(m-1)e^{-\delta}}$, which corresponds to τ_2.

2. Suppose the query does not hold, i.e., $\exists i \neq t$ such that $y_t \leq y_i + \delta$. This implies that $y_t \leq y_{t'} + \delta$, where $y_{t'} = \max_{i\neq t}(y_i)$ and $\delta \geq 0$. By Claim 4, the confidence is less than or equal to $\frac{100}{1+e^{-\delta}}$. Since $y_t \leq y_{t'} + \delta$ does not imply $y_t \leq y_{t'}$, misclassification may or may not occur.

□

6.2.3 Smoothness

A third variant is smoothness, an instance of Lipschitz continuity in neural networks [67, 68, 69]. The work in [66] introduces the concept of Lipschitz robustness, which bounds each class's logit values using a Lipschitz constant. We propose a simplified instance of Lipschitz robustness, where we ask that within an ϵ-perturbation, the confidence of the network should not exhibit significant variations wrt the seed image. As illustrated in Figure 6.1(f-g) in the Introduction, a seed image of class `Truck` was classified with 51.04% confidence, but under ϵ perturbations, the confidence varied dramatically from 3.7% to 96.57%, violating the smoothness criterion. Such drastic changes in confidence within an

ϵ-bounded perturbation indicate potential issues with the underlying network. Formally, let $t = \hat{N}(x^*)$, and $\tau > 0$ a threshold, then we have:

$$\forall x \; \texttt{dist}(x^*, x) \leq \epsilon \implies |\textsc{Conf}(N(x^*), t) - \textsc{Conf}(N(x), t)| < \tau \tag{6.5}$$

As before, $\textsc{Conf}(N(x^*), t)$ can be pre-computed as x^* is a given seed image; so we can fix $\textsc{Conf}(N(x^*), t) = C$, a constant. Substituting this, we obtain a verification query $\langle N, P, Q_{sm}\rangle$, where $Q_{sm} = C - \textsc{Conf}(N(x), t) > -\tau \wedge C - \textsc{Conf}(N(x), t) < \tau$, with P unchanged. Now again to approximate the non-linear constraint in Q_{sm} let $y_1, \ldots, y_m = N(x)$, $y_{t'} = \max_{i=1, i\neq t}^{m}(y_i)$, $\delta_1 = -\ln\left(\frac{100}{C+\tau} - 1\right)$, and $\delta_2 = -\ln\left(\frac{1}{m-1}\left(\frac{100}{C-\tau} - 1\right)\right)$. We have:

Claim 5. *If $y_t < y_{t'} + \delta_1$ and $y_t > y_{t'} + \delta_2$ holds, then $\textsc{Conf}(N(x'), t) < \tau_1 \wedge \textsc{Conf}(N(x'), t) > \tau_2$.*

Proof. Assume LHS of the claim holds:

1. $y_t < y_{t'} + \delta_1$ holds, proof is similar to the Claim 1, except in Claim 1 y_t is maximum, which is not necessary here. By removing all exponential terms from the denominator except for e^{y_t} and $e^{y_{t'}}$, we obtain $\textsc{Conf}(y_1, \ldots y_m, t) \leq \frac{100e^{y_t}}{e^{y_t} + e^{y_{t'}}}$. After scaling the right-hand side by e^{y_t}, we get $\textsc{Conf}(y_1, \ldots y_m, t) \leq \frac{100}{1+e^{-(y_t - y_{t'})}}$. Since $\delta_1 > y_t - y_{t'}$, we have $\textsc{Conf}(y_1, \ldots y_m, t) < \frac{100}{1+e^{-\delta_1}}$. Since $\delta_1 = -\ln(\frac{100}{\tau_1} - 1)$, $\textsc{Conf}(N(x'), t) < \tau_1$.

2. $y_t > y_{t'} + \delta_2$ holds, by Claim 3 we conclude that $\textsc{Conf}(N(x'), t) > \tau_2$.

By the above two cases we conclude that $\textsc{Conf}(N(x'), t) < \tau_1 \wedge \textsc{Conf}(N(x'), t) > \tau_2$ holds. □

Claim 6. *If $y_t \geq y_{t'} + \delta_1$ or $y_t \leq y_{t'} + \delta_2$ holds, then $\textsc{Conf}(N(x'), t) \geq \frac{100}{1+(m-1)e^{-\delta_1}}$ or $\textsc{Conf}(N(x'), t) \leq \frac{100}{1+e^{-\delta_2}}$*

Proof. Assume LHS of the claim holds:

1. $y_t \geq y_{t'} + \delta_1$ holds, by Claim 2 we conclude that $\textsc{Conf}(N(x'), t) \geq \frac{100}{1+(m-1)e^{-\delta_1}}$ holds.

2. $y_t \leq y_{t'} + \delta_2$ holds, by Claim 4 we conclude that $\textsc{Conf}(N(x'), t) \leq \frac{100}{1+e^{-\delta_2}}$

By the above two cases we conclude that $\textsc{Conf}(N(x'), t) \geq \frac{100}{1+(m-1)e^{-\delta_1}}$ or $\textsc{Conf}(N(x'), t) \leq \frac{100}{1+e^{-\delta_2}}$. □

Again $y_t < y_{t'} + \delta_1 \wedge y_t > y_{t'} + \delta_2$ is our derived post-condition Q'_{sm}. Eq 6.6 shows the encoding of the $\neg Q'_{sm}$. The first part of the equation encodes $y_t \geq y_{t'} + \delta_1$, since $y_{t'} = \max^m_{i=1,i\neq t}(y_i)$, which is equivalent to checking $\forall i, i \neq t, y_t \geq y_i + \delta_1$. The second part of the equation encodes $y_t \leq y_{t'} + \delta_2$, which is equivalent to checking $\exists i, i \neq t, y_t \leq y_i + \delta_2$. Thus,

$$\neg(Q'_{sm}) = \bigwedge\nolimits^m_{i=1,i\neq t} y_t \geq y_i + \delta_1 \vee \bigvee\nolimits^m_{i=1,i\neq t} y_t \leq y_i + \delta_2, \tag{6.6}$$

As we can see, the above equation is also in the form of LRA. It is straightforward to establish a similar guarantee on the approximation for this formulation as well.

Theorem 5. *Given a verification query $\langle N, P, Q'_{sm}\rangle$:*

1. *If the query holds, then $\langle N, P, Q_{sm}\rangle$ holds, i.e., for all inputs $x \models P$, the confidence remains bounded as $|C - \textsc{Conf}(N(x), t)| < \tau$.*

2. *Otherwise, there exists $x \models P$ such that the confidence is either greater than $\frac{100}{1+(m-1)e^{-\delta_1}}$ or less than $\frac{100}{1+e^{-\delta_2}}$.*

Proof. 1. If the query holds, then by Claim 5 and the explanation of Eq 6.6, we can conclude $\textsc{Conf}(N(x'), t) < \tau_1 \wedge \textsc{Conf}(N(x'), t) > \tau_2$. By the above discussion $\tau_1 = C + \tau$ and $\tau_2 = C - \tau$, it follows that $\textsc{Conf}(N(x'), t) < C + \tau \wedge \textsc{Conf}(N(x'), t) > C - \tau$, which implies $|C - \textsc{Conf}(N(x'), t)| < \tau$.

2. If the query does not hold, then by Claim 6 and the explanation of Eq 6.6, we conclude that $\textsc{Conf}(N(x'), t) \geq \frac{100}{1+(m-1)e^{-\delta_1}}$ or $\textsc{Conf}(N(x'), t) \leq \frac{100}{1+e^{-\delta_2}}$.

□

6.3 Non-confidence Based Robustness Variants

So far, we have focused on robustness variants that incorporate confidence. In this section, we examine other forms of robustness that do not rely on confidence measures. These variants, introduced in [70], are included to facilitate comparison with our confidence-based formulations and the local robustness property in the following section. We also discuss the complexity of the postconditions defined by these variants and how effectively they can be encoded within our framework, as described in Chapter 7.

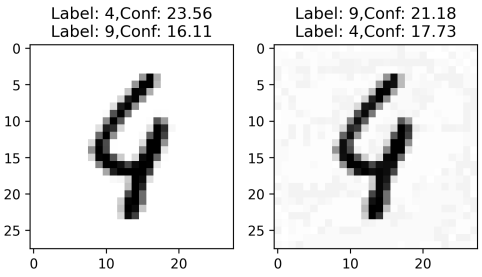


Figure 6.3: **Top-k and Top-k-relaxed**: The image on the left, labeled as 4, is taken from the MNIST dataset and correctly classified with a confidence of 23.56% by the neural network `mnist-net-256x6.onnx`. The tool $\alpha\beta$-CROWN finds a counterexample (right), misclassified as label 9, within an input perturbation of 0.05.The counterexample has a confidence of 21.18%. The top-k property allows misclassification within the top-k predictions of the original image. For $k = 2$, the original image has top-2 predictions: labels 4 and 9 (corresponding to the highest and second-highest confidence scores). This example is top-2 as well as top-2-relaxed robust, but does not satisfy standard robustness or strong robustness criteria.

6.3.1 Top-k Robustness

We consider the notion of top-k robustness introduced in [70]. Suppose a function N computes the logits values at the neural network's output layer, where $N_i(x)$ represents the value at the i^{th} dimension. Let $N(x, k)$ denote the k^{th} highest value of the logits. The function $N^k(x) = \{i \mid N_i(x) \geq N(x, k)\}$ intuitively represents the set of classes with the top k highest values of the logits. Top-k robustness is then defined as:

$$\forall x' \; dist(x, x') \leq \epsilon \implies N^k(x) = N^k(x')$$

See Figure 6.3 for an example. Its negation can be expressed as:

$$\exists x' \; dist(x, x') \leq \epsilon \wedge N^k(x) \neq N^k(x')$$

Letting $y_1, ..., y_m = N(x')$, we can capture this, i.e., $N^k(x) \neq N^k(x')$ constraint using

$$\bigvee_{i=0}^{m} \bigvee_{j \in N^k(x), j \neq i}^{m} y_i > y_j \tag{6.7}$$

Theorem 6. *Eq* (6.7) $\iff N^k(x) \neq N^k(x')$

Proof. ($\Longrightarrow$) If Eq 6.7 holds, for a fixed k there exist indices i and j with $i \notin N^k(x)$ and $j \in N^k(x)$ such that $y_i > y_j$. By definition of $N^k(x)$, $i \in N^k(x')$, but $i \notin N^k(x)$ this means $N^k(x') \neq N^k(x)$.

($\Longleftarrow$) If for a fixed k, $N^k(x) \neq N^k(x')$ holds and $|N^k(x)| = |N^k(x')|$ then there must exist an index i such that $i \in N^k(x')$ but $i \notin N^k(x)$. Consequently, there also exists an index $j \in N^k(x)$ such that $j \notin N^k(x')$. This implies that j is replaced by i in $N^k(x')$, which further implies that $y_i > y_j$. Which is Eq 6.7. □

6.3.2 Top-k Relaxed Robustness

Further, in the same paper [70], a relaxed variant of top-K robustness was also defined as

$$\forall x' \; dist(x, x') \leq \epsilon \implies \exists k \leq K : N^k(x) = N^k(x')$$

The negation of the above post-condition is:

$$\bigwedge_{k=1}^{K} N^k(x) \neq N^k(x')$$

Using the above, the full negation of the post-condition is expressed as:

$$\bigwedge_{k=1}^{K} \left(\bigvee_{i=1, j \in N^k(x), j \neq i}^{m} y_i > y_j \right) \tag{6.8}$$

Theorem 7. *Eq 6.8* $\Longleftrightarrow \bigwedge_{k=1}^{K} N^k(x) \neq N^k(x')$

Proof. ($\Longrightarrow$) If Eq 6.8 holds, then for every $k \in K$ there exist indices i and j with $i \notin N^k(x)$ and $j \in N^k(x)$ such that $y_i > y_j$. By definition of $N^k(x)$, $i \in N^k(x')$, but $i \notin N^k(x)$ this means $N^k(x') \neq N^k(x)$ for all $k \in K$.

($\Longleftarrow$) If $\bigwedge_{k=1}^{K} N^k(x) \neq N^k(x')$ holds and $|N^k(x)| = |N^k(x')|$ then for each k there must exist an index i such that $i \in N^k(x')$ but $i \notin N^k(x)$. Consequently, there also exists an index $j \in N^k(x)$ such that $j \notin N^k(x')$. This implies that j is replaced by i in $N^k(x')$, which further implies that $y_i > y_j$. Which is Eq 6.8. □

Theorem 6.8 establishes the equivalence between the encoding constraints in Eq 6.8 and the negation of the post-condition $\bigwedge_{k=1}^{K} N^k(x) \neq N^k(x')$ for relaxed Top-k robustness. This equivalence confirms the correctness of our encoding.

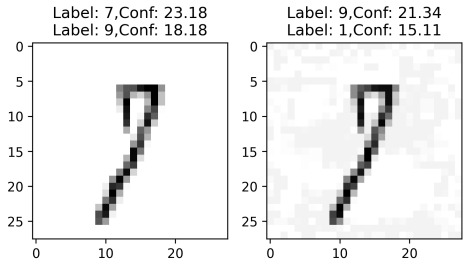


Figure 6.4: **Top-k-affinity Robustness**: The image on the left, labeled as 7, is taken from the MNIST dataset and correctly classified with a confidence of 23.18% by the neural network `mnist-net-256x6.onnx`. A state-of-the-art verification tool, $\alpha\beta$-CROWN, finds a counterexample (right), misclassified as label 9, within an input perturbation of 0.05 (consistent with the perturbation used in VNN-COMP). The counterexample has a confidence of 21.34%. In affinity robustness, the user provides prior knowledge about acceptable misclassifications. Suppose the user specifies the following affinity groups: $\{\{1,7,9\},\{2,7\},\{0,8\},\{4,8\},\{3,5\},\{6\}\}$. This means, a class-7 image may be misclassified as 1, 2, or 9. A class-0 image may only be misclassified as 8. And a class-6 image must not be misclassified at all. According to this specified misclassification knowledge, the above example is affinity robust as well as top-k relaxed robust. However, the images in Figure 6.3 are not affinity robust under the same misclassification constraints because 4 is not allowed to be misclassified in class 9. Additionally, the above image does not satisfy standard or strong robustness criteria.

6.3.3 Top-k-affinity Robustness

The paper [70] also introduced the concept of affinity robustness. The intuition behind this robustness is to incorporate expert knowledge into robustness evaluation. Experts provide sets of similar categories, and the network is considered robust as long as misclassifications remain within these predefined sets. The key idea is to allow misclassification within similar categories but not across completely different ones.

For instance, if an input image of a pine tree is misclassified as a palm tree, this might be acceptable. However, it should not be misclassified as a mammal or another unrelated category. Similarly, a tiger could be misclassified as a leopard but not as an elephant. Suppose experts define these sets as $\mathbb{S}$, where each set $S \in \mathbb{S}$ represents a collection of classes within which misclassification is allowed.

The definition of affinity robustness is given as:

$$\forall x' \; dist(x, x') \leq \epsilon \implies \exists k \leq K \; S \in \mathbb{S} : N^k(x) = N^k(x') \wedge N^k(x) \subseteq S \tag{6.9}$$

The negation of the post-condition can be written as follows:

$$\bigwedge_{k=1}^{K} \bigwedge_{S \in \mathbb{S}} \left(N^k(x) \neq N^k(x') \vee N^k(x) \nsubseteq S \right). \tag{6.10}$$

For a given image x, the condition $N^k(x) \nsubseteq S$ can be determined beforehand. For all pairs $\langle k, S \rangle$, if $N^k(x) \nsubseteq S$ is satisfied, the entire constraint is satisfied due to the disjunction. Therefore, we only need to construct constraints for those pairs $\langle k, S \rangle$ where $N^k(x) \subseteq S$ holds. Let K' and S' denote the sets of pairs $\langle k, S \rangle$ for which $N^k(x) \subseteq S$ is satisfied. The new constraints for the post-condition can be written as follows:

$$\bigwedge_{\langle k,S \rangle \in (K',S')} N^k(x) \neq N^k(x').$$

Using the equation above, this can be rewritten as:

$$\bigwedge_{\langle k,S \rangle \in (K',S')} \left(\bigvee_{i=1, j \in N^k(x), j \neq i}^{m} y_i \geq y_j \right) \tag{6.11}$$

Theorem 8. *Eq 6.11* $\iff \bigwedge_{\langle k,S \rangle \in (K',S')} N^k(x) \neq N^k(x')$

Proof. ($\implies$) If Eq 6.11 holds, then for every $k, S \in (K', S')$ there exist indices i and j with $i \notin N^k(x)$ and $j \in N^k(x)$ such that $y_i > y_j$. By definition of $N^k(x)$, $i \in N^k(x')$, but $i \notin N^k(x)$ this means $N^k(x') \neq N^k(x)$ for all $k, S \in (K', S')$.

($\impliedby$) If $\bigwedge_{\langle k,S \rangle \in (K',S')} N^k(x) \neq N^k(x')$ holds and $|N^k(x)| = |N^k(x')|$ then for each $k, S \in (K', S')$ there must exist an index i such that $i \in N^k(x')$ but $i \notin N^k(x)$. Consequently, there also exists an index $j \in N^k(x)$ such that $j \notin N^k(x')$. This implies that j is replaced by i in $N^k(x')$, which further implies that $y_i > y_j$ for each $k, S \in (K', S')$. Which is Eq 6.11. □

The postconditions defined by Equations 6.8 and 6.11 are relatively complex and expressed in Conjunctive Normal Form (CNF), whereas those associated with confidence-based robustness variants are primarily formulated in Disjunctive Normal Form (DNF).

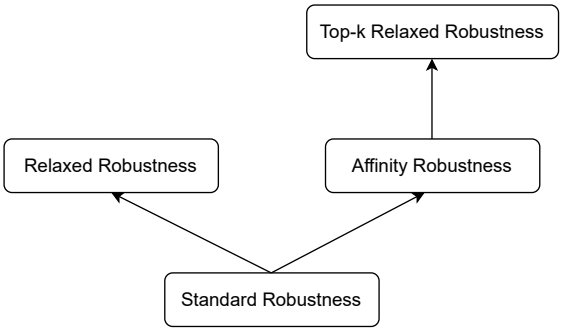


Figure 6.5: Hierarchy of various robustness

This diversity in postcondition structures implies that each must be handled manually and separately during verification, which is both effort-intensive and error-prone. In the next chapter, we demonstrate how our framework enables verification of all these properties in a unifying manner.

6.4 Hierarchy of Various Robustness

In this section, we discuss the connections among various robustness properties. Figure 6.5 illustrates the hierarchy of robustness definitions. The arrows in the figure denote logical implication relationships among these robustness properties. If a network satisfies the local robustness property, it also satisfies relaxed robustness, affinity robustness, and top-k relaxed robustness. However, the converse does not necessarily hold; a property that is relaxed robust, affinity robust, or top-k relaxed robust may not satisfy standard local robustness. Furthermore, there is no direct implication between relaxed robustness and either affinity or top-k robustness. Finally, if strong robustness holds non-trivially, meaning that the confidence of the seed image is greater than the specified threshold and satisfies the strong robustness criterion, then it implies all other forms of robustness considered in Figure 6.5.

Theorem 9. *The implications in Figure 6.5 hold.*

Proof. **Standard Robustness $\implies$ Relaxed Robustness:** By Definition 2 of the local robustness property, $\hat{N}(x) = \hat{N}(x')$ holds. This directly falsifies the premise of Equation 6.1 of relaxed robustness, implying that standard robustness ensures relaxed robustness.

Standard Robustness $\implies$ Affinity Robustness: We prove this by contradiction. Suppose affinity robustness does not hold, which means that Equation 6.10 holds. Consider an instance of Equation 6.10 with $k = 1$. The equation can be rewritten as follows:

$$\forall S \in \mathbb{S}, \left(N^1(x) \neq N^1(x') \vee N^1(x) \nsubseteq S\right).$$
$$\implies \forall S \in \mathbb{S}, \left(\hat{N}(x) \neq \hat{N}(x') \vee \hat{N}(x) \nsubseteq S\right).$$

By the definition of affinity robustness, we have:

$$\forall c \in C, \exists S, \text{ such that } c \in S,$$
$$\text{where } C \text{ is the set of all class labels.}$$

Now, consider the set S for which $\hat{N}(x) \subseteq S$. This implies:

$$\hat{N}(x) \neq \hat{N}(x') \Rightarrow \text{Standard robustness does not hold,}$$

which contradicts our assumption that standard robustness holds. Thus, standard robustness implies affinity robustness.

Affinity Robustness $\implies$ Top-k Relaxed Robustness: By Equation 6.9 of affinity robustness, there exist k and S such that:

$$N^k(x) = N^k(x') \wedge N^k(x) \subseteq S.$$

Picking the same k, we obtain:

$$N^k(x) = N^k(x'),$$

which satisfies the condition for top-k relaxed robustness. Hence, affinity robustness implies top-k relaxed robustness.

Strong Robustness $\implies$ Standard Robustness: By Equation 6.4, strong robustness holds non-trivially when $\textsc{Conf}(N(x^*), \hat{N}(x^*)) \geq \tau_1$. This implies that $\textsc{Conf}(N(x), \hat{N}(x^*)) > \tau_2 \wedge \hat{N}(x^*) = \hat{N}(x)$ for all x in the ϵ-neighborhood of x^*. From this, we can directly infer that $\hat{N}(x^*) = \hat{N}(x)$ holds, which corresponds to the definition of standard robustness. Therefore, strong robustness implies standard robustness, and standard robustness, in turn, implies all other forms of robustness discussed earlier.

□

6.5 Conclusion

In this chapter, we introduced several confidence-based robustness properties, each applicable to different use cases. We also proposed an approximation of the confidence measure derived from the SOFTMAX function, resulting in constraints expressed in Linear Real Arithmetic (LRA) form. In addition, we examined non-confidence-based robustness variants and showed that while each imposes distinct postconditions, all can be represented in LRA form. Handling each postcondition separately is both manual and error-prone. Finally, we discussed the hierarchy among different robustness variants and set the stage for the next chapter, where we present a unified framework to verify all these postconditions efficiently.

Chapter 7

Verifying Rich Robustness Properties for Neural Networks

The confidence-based robustness variants discussed in the previous chapter produce separate post-conditions on the neural network outputs. Handling each post-condition individually requires substantial manual effort and remains error-prone. We present a grammar-based framework that captures families of post-conditions and provides an algorithmic translation from these post-conditions to corresponding layers. These additional layers transform an arbitrary post-condition into a simplified canonical form. The resulting layers are then appended to the original neural network, yielding an augmented network whose simplified post-condition can be verified using any state-of-the-art neural network verifier.

To be able to handle variants such as those given in Chapter 6, the first step is to have a common way to specify all such variants. In this work, we use a simple grammar via arbitrary Boolean combinations of linear inequalities and show that it captures all the above-introduced variants and many more.

Difficulties in encoding. The main difficulty that remains is whether such rich properties modeled as Boolean combinations of linear inequalities can directly be encoded in state-of-the-art tools. In fact, as standardized in VNN-COMP, robustness properties are specified in a special VNNLIB format as pre- and post-conditions that a neural network must satisfy. Although VNNLIB supports arbitrary Boolean combinations of linear constraints, most state-of-the-art neural network verifiers are optimized for simpler post-conditions which typically involve only disjunctions or conjunctions of linear atoms over the outputs.

Thus, verifying properties with multi-layered conjunctions, disjunctions, or combinations of both poses significant challenges: (i) Modifying the verifier's code to handle complex post-conditions requires a deep understanding of its implementation, which can be challenging for users unfamiliar with the codebase, i.e. $\alpha\beta$-CROWN [60] (ii) The source code for some commercial verifiers may not be publicly available, restricting the ability to make modifications. (iii) Even when access is available, adapting the code for each new property format is time-consuming and prone to errors. (iv) Constraint-based tools, i.e., MARABOU [62], allow such properties to be encoded directly as constraints, but users are then limited to a particular solver only and lack to achieve the scalability. (v) Advanced techniques like Projected Gradient Descent (PGD) attacks, which are highly effective at finding counterexamples in neural networks, cannot always be applied directly due to the complexity of certain post-conditions.

Simplifying post-conditions by adding layers. To overcome these challenges, we propose a technique that simplifies arbitrary post-conditions by appending a few additional layers to the neural network. This transformation converts the post-condition into a straightforward form, such as $y \geq 0$ or $y > 0$, where y is the output of an added node in the neural network, while maintaining low error bounds. This intricate transformation generates new layers that encode parts of the formulas and compose the outputs of these layers according to the Boolean operations in the formulas. Since most verifiers support the RELU activation function, we employ RELU to model the Boolean operations in the post-conditions. The output of a sum of ReLUs can model conjunction or disjunction: if all inputs are negative, the output is zero; if any input is positive, the output is positive. Since we use ReLU operators to model all Boolean operations, conjunctions and disjunctions interpret input signals in opposite ways, i.e., for a conjunction, a negative input yields 1 and a positive input yields 0, while for a disjunction, a positive input yields 0 and a negative input yields 1. To enable the composition of conjunction outputs into disjunctions and vice versa, we introduce a novel technique that reverses the outputs using a flip operation, while maintaining low error bounds. These simplified post-conditions can be verified using any state-of-the-art verifier as a black box, eliminating the need for code modifications and ensuring seamless integration with existing tools.

Our encoding mechanism is similar to the one used in proving the NP-hardness of local robustness [20], but differs in a key aspect—our approach enables the estimation of the error introduced during translation. In their encoding, the inputs to the added layers can take

values either close to 0 or close to 1 with the help of an auxiliary gadget, and the output is a single variable indicating whether the property is satisfied or not. However, our translation operates on real-valued inputs and outputs, allowing us to quantify the approximation error introduced during counterexample generation. Moreover, their method assumes that the property formula must be expressed in conjunctive normal form (CNF), where converting arbitrary formulas into CNF can cause exponential blowup. In contrast, our approach generalizes to arbitrary properties and supports formulas of any depth, without requiring them to be in CNF or DNF form.

Summarizing, *our main contributions* are:

- We define a grammar for post-conditions that captures a rich set of robustness properties for neural networks. Our grammar allows us to capture existing notions from the literature, such as strong [66] and top-k robustness [70].
- We provide a new technique to analyze all variants discussed in previous chapter. Instead of changing the encoding for each variant, our idea is to provide a generalized encoding (for all properties that can be expressed using this grammar) by adding layers to the neural network. Our encoding enables the use of state-of-the-art neural network verifiers like $\alpha\beta$-CROWN, PyRAT.
- We perform a wide set of experiments over benchmarks ranging from $0.51K$ to $13.16M$ non-activation units. Our evaluation shows that we can use our technique to verify different robustness, and that we outperform the direct encodings that have been tried in the literature [66, 114].

In this chapter, we begin with the formal grammar in Section 7.1. In Section 7.2, we present the translation from properties to additional layers. The experimental evaluation is provided in Section 7.4, and we conclude the chapter in Section 7.5.

7.1 A grammar for different robustness variants

We have seen three variants of robustness. One way to check these variants is to have separate encodings for each, i.e., change the neural network encoding manually and obtain different robustness algorithms (as done, e.g., in [66, 114]). But can we have a common technique that can handle all three? This leads us to two simple yet important observations: First, all three are defined by only changing the post-condition of the verification query. Second, after the approximation described, all three can be written as a

Boolean combination of linear constraints. This motivates us to define a simple grammar on the post-conditions that unifies all the variants (similar to the standard *Satisfiability Modulo Linear Real Arithmetic* [176], but we use this grammar in the context of post-conditions). The following grammar describes the rules for the post-conditions considered in our analysis. *LE* denotes the linear expressions, and *LC* represents the constraints on *LE* defined using conditional operators. The final property is represented by *PC* (for PostCond), which denotes the Boolean combinations of *LC* and is essentially a quantifier-free linear arithmetic formula.

$$\text{LE} ::= c_1 x_1 + c_2 x_2 + \cdots + c_m x_m + b, \forall i \in [m], c_i, b \in \mathbb{R}$$

$$\text{LC} ::= \text{LE} > 0 \mid \text{LE} \geq 0 \mid \text{LE} < 0 \mid \text{LE} \leq 0$$

$$\text{PC} ::= \text{PC} \wedge \text{PC} \mid \text{PC} \vee \text{PC} \mid \text{LC} \mid \neg \text{LC}$$

We let $\mathcal{PC}$ denote the set of post-conditions generated using the above grammar. It is then easy to see that $Q, Q_{rel}, Q_{str}, Q_{sm}$ from Chapter 6, are all in $\mathcal{PC}$, i.e., derivable in this grammar. As we show next, the main reason to do so is that the post-condition can algorithmically be encoded into additional layers of the neural network, which allows us to use any state-of-the-art neural network verifier.

7.2 Encoding Mechanism via Additional Layers

In this section, we provide an encoding for all robustness variants definable by the grammar, such that they can be integrated in state-of-the-art robustness verification engines. We begin by noting that the International Verification of Neural Networks Competition (VNN-COMP) [76] standardizes both the neural network format (ONNX) and the property format (VNNLIB). All neural network verifiers participating in VNN-COMP take as input a neural network in ONNX format and a property file in VNNLIB format. Importantly, the VNNLIB format encodes the pre-condition P and the negation of the post-condition Q of the neural network N, expressed as an arbitrary Boolean combination of linear constraints, i.e., exactly in the grammar that we described in the previous section. However, most state-of-the-art neural network verifiers are optimized for *simplified post-conditions*, typically involving only disjunctions or conjunctions of linear atoms over the outputs of the network. In what follows, we call a post-condition *simplified* if it is either atomic $y > 0$, $y \geq 0$, $y < 0$, or $y \leq 0$, where y is one of the outputs of N or conjunctions/disjunctions of the atomics.

In this section, we show how we can convert post-conditions into simplified ones *by just appending a few additional layers* to the neural network, without having to change

the encoding within the verifiers. In doing so, we only add a linear number of neurons in the size of the post-condition, and the number of layers added is at most the depth of the formula (represented as a Directed Acyclic Graph) in the post-condition. As we will see, while it is easy to do this for conjunctions, it is non-trivial when both conjunctions and disjunctions are present, and we achieve a solution with an approximation factor.

7.2.1 Translating Conjunctions

Translating conjunctions is relatively simple (as also done in [158]). Suppose we have a conjunction $\bigwedge_{i=0}^{n} LE_i \leq 0$, where nodes $y_1, \ldots, y_m$ represent the original network's output nodes and $LE_i = \sum_{j=1}^{m} w_{ij} y_j + b_j$, say. Then, we observe that defining a new output as $y = \sum_{i=0}^{n} \text{ReLU}(LE_i)$ can be encoded easily as an additional layer, as depicted in Figure 7.1(a). The nodes $LE_1, LE_2, \ldots, LE_n$ represent the actions of corresponding linear expressions, which are followed by ReLU nodes. We immediately obtain:

Lemma 8. $y \leq 0 \iff \bigwedge_{i=0}^{n} LE_i \leq 0$

Proof. ($\implies$) Since $y = \sum_{i=0}^{n} \text{ReLU}(LE_i)$, if $y \leq 0$ then $\forall i$, $\text{ReLU}(LE_i) = 0$, because ReLU's output is always ≥ 0. Therefore, $\forall i, LE_i \leq 0$ by the definition of ReLU. Therefore, $\bigwedge_{i=0}^{n} LE_i \leq 0$ holds. ($\Longleftarrow$)Since $\bigwedge_{i=0}^{n} LE_i \leq 0$, we conclude $\bigwedge_{i=0}^{n} \text{ReLU}(LE_i) = 0$. Therefore, $\sum_{i=0}^{n} \text{ReLU}(LE_i) = 0$. Since $y = \sum_{i=0}^{n} \text{ReLU}(LE_i)$, $y \leq 0$. □ □

The number of ReLU nodes added is just the number of clauses in the property, e.g., in Figure 7.1(a), there are a total of n clauses; therefore, n ReLU nodes are added. We can also deduce that $y > 0 \Leftrightarrow \bigvee_{i=0}^{n} LE_i > 0$, which represents the encoding of the disjunction.

7.2.2 The general translation

When we have both disjunctions and conjunctions in the post-condition, we cannot directly feed the output of conjunctions to the input of a disjunction. The reason is that in the case of conjunction, the condition $y = 0$ is interpreted as true, whereas $y > 0$ is interpreted as false, while for disjunction, the roles are reversed. A straightforward approach to handle this asymmetry is to convert the post-condition into Disjunctive Normal Form (DNF) and execute each clause in parallel. However, this method suffers from several drawbacks: (i) the conversion to DNF may cause exponential blowup, (ii) information derived from one clause cannot be exploited by another, and (iii) the underlying verifier codebase may still need modifications to accommodate each clause. Another way to encode

disjunctions is by using products, however, this introduces highly non-linear constraints, which are very inefficient to handle. Although all three post-conditions above are expressed in DNF form, this is not always the case. For instance, for the top-k properties introduced in Section 6.3 of Chapter 6, the post-conditions are not in DNF.

Therefore, we take an alternate approach via a transformation $flip(b, v) = b - v$ of a signal v for some $b > 0$ (defined later), which allows us to feed the output of disjunction to conjunction and vice-versa. Our translation will also introduce non-linear constraints, but only as *Relu*s, which can be easily modeled in verifiers. In the following, let $\dagger \in \{\vee, \wedge\}$ and if $\dagger = \vee$, then $\bar{\dagger} = \wedge$, if $\dagger = \wedge$, then $\bar{\dagger} = \vee$. Wlog., we also assume that the post-condition is given in Negation Normal Form, with negations pushed inside the linear inequalities, with $\wedge$ and $\vee$ flattened (i.e., $\wedge$ and $\vee$ alternate when traversing the formula top-down). Then, for $\dagger \in \{\vee, \wedge\}$, for any post-condition $Q \in \mathcal{PC}$ and a parameter $\eta \in \mathbb{R}$, we define the gadget $V(\dagger, Q, \eta)$ as an expression made up of $flip$s, *Relu*, and summation functions. The parameter η is an approximation parameter that is used as a constant in the simplified post-condition (Theorem 10) and to derive the value of b in the flip operation (Eq. 7.3). Hence, for every such expression we can build a circuit $\mathfrak{C}_{V(\dagger,Q,\eta)}$ that can be integrated with the Neural network using additional layers (for $flip$ we use -1 weight on edges, while *Relu* and summation are standard operators in neural networks). Thus, given a set of input values to Q, $V(\dagger, Q, \eta)$ evaluates to a real output value y, satisfying some properties, as we shall describe below. Both V and its circuit are defined inductively based on the structure of the post-condition.

For atomic formulas, we construct the gadgets V as given below, one for each value of $\dagger \in \{\wedge, \vee\}$:

$$V(\wedge, LE \leq 0, \eta) = LE + \eta \qquad V(\vee, LE \leq 0, \eta) = -LE + \eta \tag{7.1}$$

The following gives the gadget for $Q = Q_1 \dagger .. \dagger Q_k$ (where $\dagger \in \{\vee, \wedge\}$), whose circuit is pictorially illustrated in Figure 7.1(b).

$$V(\dagger, Q, \eta) = \sum_{i=1}^{k} Relu(flip(b(\dagger, k, \eta), V(\bar{\dagger}, Q_i, \eta))) \tag{7.2}$$

$$\text{where,} b(\dagger, k, \eta) = \begin{cases} \eta(1 + 1/k) & \text{if } \dagger = \wedge \\ 2 * \eta & \text{if } \dagger = \vee \end{cases} \tag{7.3}$$

The main intuition of the above construction is that after crossing every disjunct/conjunct, the direction of the bounds flips, i.e., the upper bound on an output becomes a lower bound

and vice versa. This allows us to propagate bounds when the circuit has disjuncts and conjuncts. Before formalizing this intuition and proving the properties of the translation, we present in Figure 7.1(c) an illustrative example. In this example, the negation of the post-condition $\neg Q$ is $((y_1 + y_2 \leq 0 \wedge y_2 \leq 0) \vee (y_1 - y_3 \leq 0 \wedge y_3 \leq 2))$, with atomic formulae $y_1 + y_2 \leq 0$, $y_2 \leq 0$, and so on. The parameter η in the input constraint is set to 0.2 (user-defined). The part of the circuit before the first dashed vertical line are the gadgets for atomic formulae. For instance, $y_1 + y_2 \leq 0$ is converted into $-y_1 - y_2 + 0.2$, which is represented by the red part of the circuit. Thus, the output at the red line satisfies $0.2 \leq -y_1 - y_2 + 0.2$, which is equivalent to $y_1 + y_2 \leq 0$. The reason to have this flipped representation is that $y_1 + y_2 \leq 0$ and $y_2 \leq 0$ are connected with conjunction, but the formulation of Eq 7.2 expects the atomic formulae to be connected with disjunction, thus we apply the disjunctive rule in right hand side of (7.1). The resulting outputs are connected with conjunctions, and the gadget between the two vertical lines corresponds to these conjunctions. Since this gadget represents conjunction, the value of b is determined by the conjunction case in Eq 7.3, i.e., $b = \eta\left(1 + \frac{1}{k}\right) = 0.2\left(1 + \frac{1}{2}\right) = 0.3$. In other words, at the blue line, the output will be less than or equal to 0.2 if $(y_1 + y_2 \leq 0 \wedge y_2 \leq 0)$ (as shown more generally in Lemma 9 below). Finally, the conjunctions are connected with disjunctions, represented by the gadget after the second vertical line, which is also constructed following Eq 7.3, where $b = 2\eta = 0.2 \times 2 = 0.4$.

Here, the value of η needs to be nonzero. Let us consider the case $\eta = 0$. In this setting, the outputs of both conjunctions become 0, as shown just before the second vertical line in Figure 7.1(c). Consequently, the output of the disjunction, denoted by y, is always 0 because the effect of the ReLU activation is disabled. Hence, the condition $y \geq \eta$ is always satisfied.

Our choice of $b(\dagger, k, \eta)$ is crucial for ensuring the desired properties of the gadget. The following technical lemma establishes how we can propagate the bounds on the signal $V(\dagger, Q, \eta)$ to the sub-formulas Q_i and vice-versa, which will help us relating bounds on the inputs of the gadget and the bounds on the output of the gadget.

Lemma 9. *For $\eta > 0$ and $\beta > 0$, the following holds for Q.*

1. $\forall i.\ \eta \leq V(\vee, Q_i, \eta) \Rightarrow V(\wedge, Q, \eta) \leq \eta$.
2. $\exists i.\ V(\wedge, Q_i, \eta) \leq \eta \Rightarrow \eta \leq V(\vee, Q, \eta)$.
3. $V(\wedge, Q, \eta) \leq \beta \Rightarrow \forall i.\ (1 + 1/k)\eta - \beta \leq V(\vee, Q_i, \eta)$.

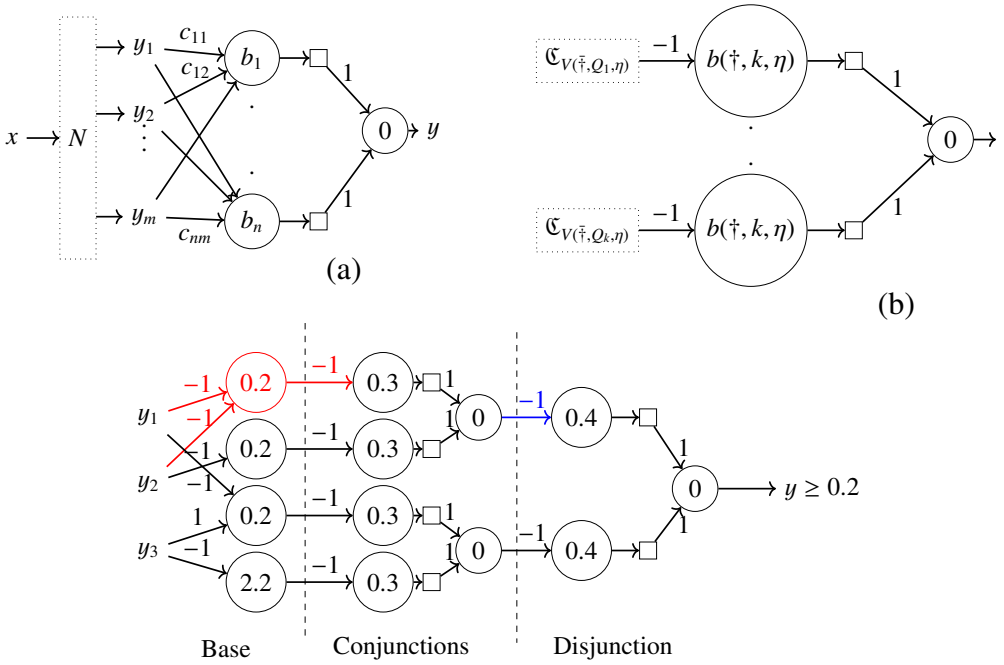


Figure 7.1: (a) Neural network N appended with neural network layer that encodes either $\bigwedge_{i=0}^{n} LE_i \leq 0$ and its negation $\bigvee_{i=0}^{n} LE_i > 0$, where $LE_i = \sum_{j=1}^{m} c_{ij} y_j + b_i$. The square nodes are ReLU and the circular nodes are linear combinations. (b) The circuit for $\mathfrak{C}_{V(\dagger,Q,\eta)}$ (c) Translation of post-condition $\neg((y_1 + y_2 \leq 0 \wedge y_2 \leq 0) \vee (y_1 - y_3 \leq 0 \wedge y_3 \leq 2))$ using our scheme and $\eta = 0.2$.

4. $\beta \leq V(\vee, Q, \eta) \Rightarrow \exists i.\ V(\wedge, Q_i, \eta) \leq 2\eta - \beta/k$.

Proof. The following is the proof of the four parts.

1. We assume for each i, $\eta \leq V(\vee, Q_i, \eta)$. After a rewrite, $(1+1/k)\eta - V(\vee, Q_i, \eta) \leq \eta/k$. After applying definition of b and $flip$, $flip(b(\wedge, k, \eta), V(\vee, Q_i, \eta)) \leq \eta/k$ for each i. Due to the definition of V, we conclude $V(\wedge, Q, \eta) \leq \eta$.

2. We assume for some i, $V(\wedge, Q_i, \eta) \leq \eta$. After a rewrite, $\eta \leq 2\eta - V(\vee, Q_i, \eta)$. After applying definition of b and $flip$, $\eta \leq flip(b(\vee, k, \eta), V(\wedge, Q_i, \eta))$ for some i. Due to the definition of V, we conclude $\eta \leq V(\wedge, Q, \eta)$.

3. If $V(\wedge, Q, \eta) \leq \beta$, we conclude $flip(b(\wedge, k, \eta), V(\vee, Q_i, \eta)) \leq \beta$. After applying definition of b, $flip((1+1/k)\eta, V(\vee, Q_i, \eta)) \leq \beta$ for each i. After expanding definition

of $flip$, $(1+1/k)\eta - V(\vee, Q_i, \eta) \leq \beta$. After simplification, $(1+1/k)\eta - \beta \leq V(\vee, Q_i, \eta)$.

4. If $\beta \leq V(\vee, Q, \eta)$, we conclude $\beta/k \leq flip(b(\wedge, k, \eta), V(\wedge, Q_i, \eta))$ for some i. After applying definition of b, $\beta/k \leq flip(2\eta, V(\wedge, Q_i, \eta))$. After expanding definition of $flip$, $\beta/k \leq 2\eta - V(\wedge, Q_i, \eta)$. After simplification, $V(\wedge, Q_i, \eta) \leq 2\eta - \beta/k$.

□

Using the first two parts of the above lemma (the other two parts will be used for providing error bounds), we are now ready to state the lemma that establishes the correctness of our translation scheme. The first two cases demonstrate correctness for atomic formulae, while Cases 3 and 4 establish correctness for general formulae.

Lemma 10. *For a given post condition Q and an $\eta > 0$, the following holds*

1. If $Q = LE \leq 0$, $V(\wedge, Q, \eta) \leq \eta \Leftrightarrow Q$
2. If $Q = LE \leq 0$, $V(\vee, Q, \eta) \geq \eta \Leftrightarrow Q$
3. If Q is conjunctive, i.e., $Q = Q_1 \wedge ... \wedge Q_k$, then $Q \Rightarrow V(\wedge, Q, \eta) \leq \eta$
4. If Q is disjunctive, i.e., $Q = Q_1 \vee ... \wedge Q_k$, then $Q \Rightarrow V(\vee, Q, \eta) \geq \eta$

Proof. We prove this lemma by induction.

- **Base case:**
  1. If $Q = LE \leq 0$, then by definition, $V(\wedge, Q, \eta) = LE + \eta$. Given that $V(\wedge, Q, \eta) \leq \eta$, we derive: $LE + \eta \leq \eta \Leftrightarrow LE \leq 0$.
  2. If $Q = LE \leq 0$, then by definition, $V(\vee, Q, \eta) = -LE + \eta$. Given that $V(\vee, Q, \eta) \geq \eta$, we derive: $-LE + \eta \geq \eta \Leftrightarrow LE \leq 0$.
- **Inductive step:** Assume Lemma 10 holds for all formulae of depth d, denoted as Q^d. Consider a formula of depth $d+1$, given by: $Q^{d+1} = Q_1^d \dagger \cdots \dagger Q_k^d$.
  3. If $\dagger = \wedge$, Q_i^ds are true. Due to the induction hypothesis, $\forall i,\ \eta \leq V(\vee, Q_i^d, \eta)$. By the part 1 of Lemma 9, we have: $\forall i,\ \eta \leq V(\vee, Q_i^d, \eta) \implies V(\wedge, Q^{d+1}, \eta) \leq \eta$. Therefore, $V(\wedge, Q^{d+1}, \eta) \leq \eta$ holds.

4. If $\dagger = \vee$, Q_i^d is true for some i. Due to the induction hypothesis, $\exists i,\ V(\wedge, Q_i^d, \eta) \leq \eta$. By the part 2 of Lemma 9, we have: $\exists i,\ V(\wedge, Q_i^d, \eta) \leq \eta \implies \eta \leq V(\vee, Q^{d+1}, \eta)$. Therefore, $\eta \leq V(\vee, Q^{d+1}, \eta)$ holds.

□

Using the above, we can finally define $N' = Append(N, V(\dagger, Q, \eta))$ to be the neural network obtained by attaching the circuit $\mathfrak{C}_{V(\dagger,Q,\eta)}$ after the neural network N with y being the final output.More precisely, the inputs of $V(\dagger, Q, \eta)$ are connected to the outputs of N, and the output y of $V(\dagger, Q, \eta)$ is the only output of N'. Then, from the above lemma, we can prove our first theorem regarding using it as a verification query.

Theorem 10. *Consider neural network N, pre-condition P, a post-condition $\neg Q$, and $\eta > 0$. Let $N' = Append(N, V(\dagger, Q, \eta))$ be the neural network obtained by attaching $\mathfrak{C}_{V(\dagger,Q,\eta)}$ after neural network N and y be the final output.*

1. *If Q is disjunctive, $\langle N', P, y < \eta\rangle \Rightarrow \langle N, P, \neg Q\rangle$*

2. *If Q is conjunctive, $\langle N', P, y > \eta\rangle \Rightarrow \langle N, P, \neg Q\rangle$*

Proof. The following is the proof of the first part. Let us consider the successful query $\langle N', P, y < \eta\rangle$ to the solver. We know $\neg(y < \eta)$ is infeasible. Therefore, $\eta \leq V(\vee, Q, \eta)$ is infeasible. Therefore, due to the contrapositive of the third part of Lemma 10, we conclude that post-condition Q cannot be satisfied. Therefore, query $\langle N, P, \neg Q\rangle$ holds. Similarly, we can prove the second part using the last part of Lemma 10. □

7.2.3 Integrating All Components: Soundness.

Now, we establish the soundness of our framework by integrating the two main components: confidence approximation, as defined in Section 6.2 of Chapter 6, and the encoding mechanism, as defined in this section. While our framework remains sound as long as the soundness conditions of the first component hold, we can show how soundness is preserved across all examples presented in Section 6.2 of Chapter 6.

Theorem 11. *Let N be a neural network, P a pre-condition, Q any of the three post-conditions specified in Section 6.2 of Chapter 6. Let N' be the appended neural network and Q' be the simplified post-condition obtained through confidence approximation. If the query $\langle N', P, Q'\rangle$ holds, then the original query $\langle N, P, Q\rangle$ also holds.*

Proof. Let Q'' denote an intermediate post-condition obtained after applying confidence approximations, but prior to full simplification and encoding. From Theorems 3, 4, and 5 of Chapter 6, we have: $\langle N, P, Q'' \rangle \implies \langle N, P, Q \rangle$. Furthermore, from Theorem 10 of this section, we have: $\langle N', P, Q' \rangle \implies \langle N, P, Q'' \rangle$. and hence, together, we conclude $\langle N', P, Q' \rangle \implies \langle N, P, Q \rangle$. □

7.2.4 Bound on the error in counterexamples

If the query $\langle N', P, y < \eta \rangle$ fails, the verifier generates a counterexample input x to N' that violates $y < \eta$. Since the neural network translation is approximate, x may not violate $\neg Q$ in N. Moreover, the following Theorem uses parts 3 and 4 of Lemma 9 to bound the error. Let $Q[\eta]$ be a formula obtained by replacing each $LE \leq 0$ by $LE \leq \eta$ in Q.

Theorem 12. *If $\eta \leq V(\vee, Q, \eta)$ is satisfiable and Q is DNF (disjunctive normal form), we can conclude that $Q[2\eta]$ is satisfiable.*

Proof. Since Q is DNF, $Q = Q_1 \vee ... \vee Q_k$. Due to part four of Lemma 9, there is $V(\wedge, Q_i, \eta) \leq 2\eta - \eta/k = (2 - 1/k)\eta$. Let $Q_i = LE_1 \leq 0 \wedge ... \wedge LE_l \leq 0$. Due to part three of Lemma 9 for each LE_i, $(1 + 1/l)\eta - (2 - 1/k)\eta \leq V(\vee, LE_i \leq 0, \eta)$. After simplification, $(-1 + 1/k + 1/l)\eta \leq V(\vee, LE_i \leq 0, \eta)$. Therefore, $(-1 + 1/k + 1/l)\eta \leq -LE_i + \eta$. Therefore, $LE_i \leq (2 - 1/k - 1/l)\eta \leq 2\eta$. □

The above theorem indicates that the satisfying assignment may violate Q by the margin of 2η. We must choose η as small as possible, but not smaller than the minimum precision of the underlying solver, which will lead to the numerical instability of the solver. The above theorem requires Q to be in DNF. However, we can prove a similar theorem if Q is in CNF (conjunctive normal form). Since all our properties are in CNF or DNF, we can use the above theorem to bound the error of the counterexample for the properties that we are interested in.

7.3 Support of strict inequalities

For the simplicity of the presentation, we only considered non-strict inequalities above. In this section, we provide the versions of the theorems that handle strict inequalities. The proof of theorems work in a similar line as we have provided for the earlier theorems. Let $<^* \in \{<, \leq\}$. Let $\overline{<} = \leq$ and $\overline{\leq} = <$. Let us define V for $<^*$:

$$V(\wedge, LE <^* 0, \eta) = LE + \eta \qquad V(\vee, LE <^* 0, \eta) = -LE + \eta$$

We can rewrite Lemma 9 as follows to support strict inequalities.

Lemma 11. *For $\eta > 0$ and $\beta > 0$, the following holds for Q.*

1. $\forall i.\ \eta <_i^* V(\vee, Q_i, \eta) \Rightarrow V(\wedge, Q, \eta) <^* \eta$, *if for some* i, $<_i^* = <$, *then* $<^* = <$. *Otherwise,* $<^* = \leq$.

2. $\exists i.\ V(\wedge, Q_i, \eta) <^* \eta \Rightarrow \eta <^* V(\vee, Q, \eta)$.

3. $V(\wedge, Q, \eta) <^* \beta \Rightarrow \forall i.\ (1 + 1/k)\eta - \beta <^* V(\vee, Q_i, \eta)$.

4. $\beta <^* V(\vee, Q, \eta) \Rightarrow \exists i.\ V(\wedge, Q_i, \eta) <^* 2\eta - \beta/k$.

Proof. 1. Suppose I_1 and I_2 are indexes for which $<^*=<$ and $<^*=\leq$ respectively and $|I_1| + |I_2| = k$ and $I_1 \cap I_2 = \Phi$, where k are the number of subformulae. We have the following two cases:

- **Case:1** $|I_1| \geq 1$ For all $i \in I_1$ $<_i^*=<$, for all i we have $\eta < V(\vee, Q_i, \eta)$, which is same as: $\eta(1 + 1/k) - V(\vee, Q_i, \eta) < \eta/k$, by using the definitions of $flip$ and b we have: $flip(\eta(1 + 1/k), V(\vee, Q_i, \eta)) < \eta/k$. By the definition of V, we have the following:

$$\sum_{i \in I_1} flip(\eta(1 + 1/k), V(\vee, Q_i, \eta)) < \eta/k. \tag{7.4}$$

  By similar arguments, we hold the following inequality for all $j \in I_2$:

$$\sum_{j \in I_2} flip(\eta(1 + 1/k), V(\vee, Q_j, \eta)) \leq \eta/k. \tag{7.5}$$

  By Equations 7.4 and 7.5, we conclude the following:

$$\sum_{i \in I_1 \cup I_2} flip(\eta(1 + 1/k), V(\vee, Q_i, \eta)) < \eta/k,$$

  which is $V(\wedge, Q, \eta) <^* \eta$.

- **Case:2** $|I_1| = 0$, for each $i \in I_2$ $<_i^*=\leq$. Proof follows from the proof of part 1 of Lemma 9.

2. For some i, $V(\wedge, Q_i, \eta) <^* \eta$, for the same i this is equivalent to: $\eta <^* 2\eta - V(\wedge, Q_i, \eta)$, By using the definitions of b and $flip$: $\eta <^* flip(2\eta, V(\wedge, Q_i, \eta))$. By definition of V we conclude: $\eta <^* V(\vee, Q, \eta)$.

3. If $V(\wedge, Q, \eta) <^* \beta$, we conclude $flip(b(\wedge, k, \eta), V(\vee, Q_i, \eta)) <^* \beta$. After applying definition of b, $flip((1 + 1/k)\eta, V(\vee, Q_i, \eta)) <^* \beta$ for each i. After expanding definition of $flip$, $(1+1/k)\eta - V(\vee, Q_i, \eta) <^* \beta$. After simplification, $(1+1/k)\eta - \beta <^* V(\vee, Q_i, \eta)$.

4. If $\beta <^* V(\vee, Q, \eta)$, we conclude $\beta/k <^* flip(b(\wedge, k, \eta), V(\wedge, Q_i, \eta))$ for some i. After applying definition of b, $\beta/k <^* flip(2\eta, V(\wedge, Q_i, \eta))$. After expanding definition of $flip$, $\beta/k <^* 2\eta - V(\wedge, Q_i, \eta)$. After simplification, $V(\wedge, Q_i, \eta) <^* 2\eta - \beta/k$.

□

Let us recursively define $<^*_Q$ such that we can rewrite Lemma 10 as follows to support strict inequalities.

1. For $Q = LE <^* 0$, $<^*_Q = <^*$.

2. For conjunctive Q,
$$<^*_Q = \begin{cases} \leq & \forall i.\ <_{Q_i} = \leq \\ < & Otherwise. \end{cases}$$

3. For disjunctive Q,
$$<^*_Q = \begin{cases} < & \forall i.\ <_{Q_i} = < \\ \leq & Otherwise. \end{cases}$$

Lemma 12. *For a given post condition Q and an $\eta > 0$, the following holds*

1. If $Q = LE \leq 0$, $V(\wedge, Q, \eta) \leq \eta \Leftrightarrow Q$

2. If $Q = LE \leq 0$, $V(\vee, Q, \eta) \geq \eta \Leftrightarrow Q$

3. If $Q = LE > 0$, $V(\wedge, Q, \eta) < \eta \Leftrightarrow Q$

4. If $Q = LE > 0$, $V(\vee, Q, \eta) > \eta \Leftrightarrow Q$

5. If Q is conjunctive, $Q \Rightarrow V(\wedge, Q, \eta) <^*_Q \eta$

6. If Q is disjunctive, $Q \Rightarrow \eta <^*_Q V(\vee, Q, \eta)$

Proof. We prove this lemma by induction.

- **Base case:**

1. If $Q = LE \leq 0$, then by definition, $V(\wedge, Q, \eta) = LE + \eta$. Given that $V(\wedge, Q, \eta) \leq \eta$, we derive: $LE + \eta \leq \eta \Leftrightarrow LE \leq 0$.

2. If $Q = LE \leq 0$, then by definition, $V(\vee, Q, \eta) = -LE + \eta$. Given that $V(\vee, Q, \eta) \geq \eta$, we derive: $-LE + \eta \geq \eta \Leftrightarrow LE \leq 0$.

3. If $Q = LE > 0$, then by definition, $V(\wedge, Q, \eta) = -LE + \eta$. Given that $V(\wedge, Q, \eta) < \eta$, we derive: $-LE + \eta < \eta \Leftrightarrow LE > 0$.

4. If $Q = LE > 0$, then by definition, $V(\vee, Q, \eta) = LE + \eta$. Given that $V(\vee, Q, \eta) > \eta$, we derive: $LE + \eta > \eta \Leftrightarrow LE > 0$.

- **Inductive step:** Assume Lemma 12 holds for all formulae of depth d, denoted as Q^d. Consider a formula of depth $d+1$, given by: $Q^{d+1} = Q_1^d \dagger \cdots \dagger Q_k^d$.

3. If $\dagger = \wedge$, Q_i^ds are true. Due to the induction hypothesis, $\forall i,\ \eta <^*_{Q_i} V(\vee, Q_i^d, \eta)$ hold. By the part 1 of Lemma 11 and definition of $<^*_Q$, we have: $\forall i,\ \eta <^*_{Q_i} V(\vee, Q_i^d, \eta) \implies V(\wedge, Q^{d+1}, \eta) <^*_Q \eta$. Therefore, $V(\wedge, Q^{d+1}, \eta) <^*_Q \eta$ holds.

4. If $\dagger = \vee$, Q_i^d is true for some i. Due to the induction hypothesis, $\exists i,\ V(\wedge, Q_i^d, \eta) <^*_{Q_i} \eta$. By the part 2 of Lemma 11 and definition of $<^*_Q$, we have: $\exists i,\ V(\wedge, Q_i^d, \eta) <^*_{Q_i} \eta \implies \eta <^*_Q V(\vee, Q^{d+1}, \eta)$. Therefore, $\eta <^*_Q V(\vee, Q^{d+1}, \eta)$ holds.

□

Now we can rewrite Theorem 10 as follows to support strict inequalities, which illustrates that our encoding mechanism works for strict inequalities as well.

Theorem 13. *Consider neural network N, pre-condition P, a post-condition $\neg Q$, and $\eta > 0$. Let $N' = Append(N, V(\dagger, Q, \eta))$ be the neural network obtained by attaching $\mathfrak{C}_{V(\dagger,Q,\eta)}$ after neural network N and y be the final output.*

1. *If Q is disjunctive, $\langle N', P, y\overline{<^*_Q}\eta\rangle \Rightarrow \langle N, P, \neg Q\rangle$*

2. *If Q is conjunctive, $\langle N', P, \eta\overline{<^*_Q}y\rangle \Rightarrow \langle N, P, \neg Q\rangle$*

Proof. The proof follows similarly to that of Theorem 10. □

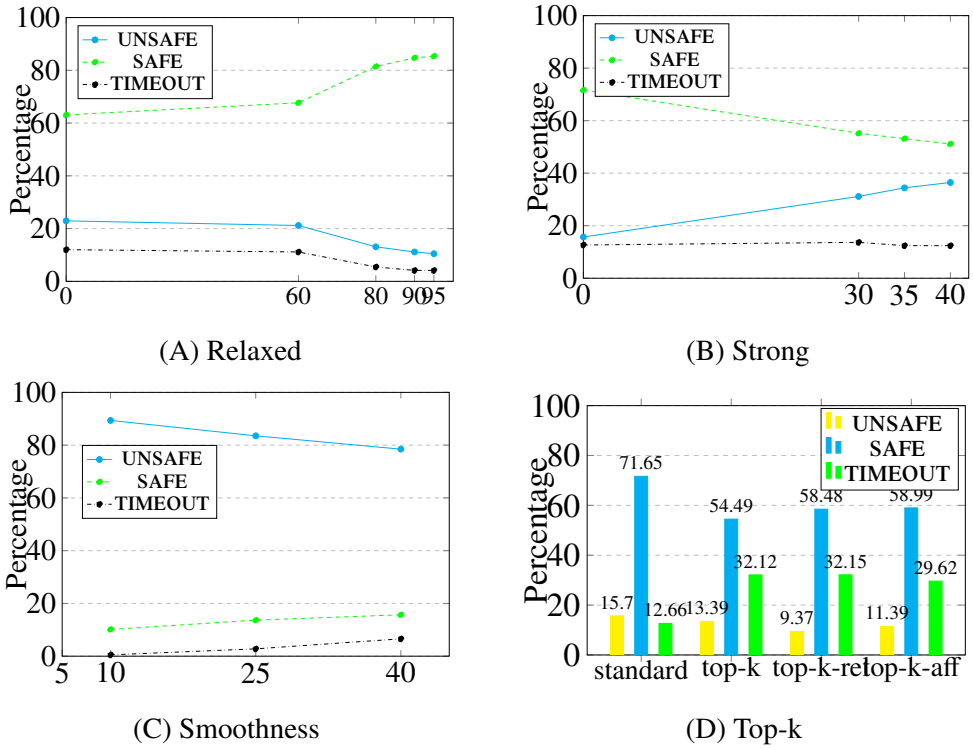
Table 7.1: Networks details

Category	Network name	#layers	#activation units	adv trained
MNIST	mnist-net-256×2.onnx	2-FC	0.51K	No
	mnist-net-256×4.onnx	4-FC	1.02K	No
	mnist-net-256×6.onnx	6-FC	1.54K	No
CIFAR-10	cifar-base-kw.onnx	2-Conv, 2-FC	3.17K	Yes
	cifar-deep-kw.onnx	4-Conv, 2-FC	6.77K	Yes
	cifar-wide-kw.onnx	2-Conv, 2-FC	6.24K	Yes
	cifar10-2-255.onnx	3-Conv, 2-FC	49.15K	Yes
	cifar10-8-255.onnx	2-Conv, 2-FC	16.39K	Yes
	convBigRELU-PGD.onnx	4-Conv, 3-FC	62.46K	Yes
	resnet-2b.onnx	2-res-blocks (5-Conv, 2-FC)	6.24K	Yes
	resnet-4b.onnx	4-res-blocks (9-Conv, 2-FC)	14.45K	Yes
GTSRB	net-1	2-QConv, 1-FC	-	No
	net-2	3-QConv, 3-BN, 2-Maxpool, 2-FC	-	No
	net-3	3-QConv, , 3-BN, 3-Maxpool, 2-FC	-	No
IMAGENET	vggnet-16	13-Conv, 5-Maxpool, 3-FC	13.16M	No

7.4 Experiments

Our implementation encodes our post-conditions into additional neural network layers as described in the previous section and then invokes a Neural network verification engine. For experiments, we selected the state-of-the-art tool $\alpha\beta$-CROWN [33, 71, 72, 42], a portfolio verifier, which has consistently ranked 1^{st} in VNN-COMP 2021-2024. Our experiments below are designed to address the following research questions: **RQ1:** Does our layer-based approach work/scale for different variants of robustness for *large* VNN-COMP benchmarks? **RQ2:** How does our approach compare across *varying thresholds* for the different robustness variants? **RQ3:** How does our approach compare wrt a direct encoding of properties in *a constraint-based state-of-the-art solver*, e.g.,MARABOU?

Intuitively, RQ1 aims to demonstrate *scalability* for each of the (richer) properties we consider, in comparison to other tools. In RQ2, we study the effect of changing thresholds on performance. While it is expected that higher thresholds yield more "safe" cases (e.g., in relaxed robustness), this experiment serves as a sanity check to validate that the framework behaves consistently across variants and demonstrates flexibility in supporting different thresholds. RQ3 compares our encoding framework with direct encoding. We performed experiments on a GPU machine: Tesla V100-SXM2-32GB, 9 vCPUs, and 32GB RAM, and set the value of η to 1×10^{-4}.



(A) Relaxed (B) Strong

(C) Smoothness (D) Top-k

Figure 7.2: Figures 7.2A, 7.2B, and 7.2C show the confidence thresholds on the x-axis and the percentage of SAFE, UNSAFE, and TIMEOUT instances on the y-axis. Figure 7.2D presents a comparison between standard robustness and top-k robustness, including top-k relaxed robustness and top-k affinity robustness. For each robustness metric, the left/middle/right bars represent the percentage of UNSAFE, SAFE, and TIMEOUT cases, respectively.

7.4.1 Benchmarks

We conducted experiments on four different datasets: MNIST [65], CIFAR-10 [77], Traffic Sign Recognition (TSR) [78], and IMAGENET [8]. All networks used in the experiments, along with their properties (VNNLIB files), were taken from VNN-COMP 2021 to VNN-COMP 2024. The provided VNNLIB files correspond to standard robustness properties. We have listed our benchmarks in Table 7.1. Our experiments covered a diverse set of benchmarks, ranging from fully connected networks to complex architectures, including convolutional layers, max-pooling layers, residual blocks, and ReLU activations, including standard and adversarially trained models. *Additionally, the network sizes ranged from small architectures with* 512 *ReLUs to large networks with* 11.16*M ReLUs. Each neural network, along with its corresponding property (VNNLIB) file, represents a single benchmark. In total, we evaluated 8,870 benchmarks in our experiments.* We provide the details of each dataset below:

MNIST: We utilized benchmarks from VNN-COMP 2022, which included a total of three networks and 30 VNNLIB files, resulting in 90 benchmarks. These benchmarks were constructed with l_∞ input perturbations of $\epsilon \in \{0.03, 0.05\}$ in 15 randomly chosen images from mnist dataset. All networks are fully connected with RELU activation functions, featuring between $0.51K$ and $1.54K$ ReLU activations.

CIFAR-10: We selected the first six networks, ranging from `cifar-base-kw.onnx` to `convBigRELU-PGD.onnx`, from VNN-COMP 2022. These networks include convolutional layers, fully connected layers, and RELU activation functions. Additionally, the networks `resnet-2b` and `resnet-4b`, both residual networks [177] (ResNet), were sourced from VNN-COMP 2021. These residual networks contain two and four residual blocks, respectively. All networks for this dataset were adversarially trained either using COLT [174] or using the method described in [178]. The dataset includes a total of 305 benchmarks, with l_∞ input perturbations ϵ ranging from $\frac{1}{255}$ to $\frac{16}{255}$.

GTSRB: These benchmarks are sourced from VNN-COMP 2024 and are based on binary neural networks (BNNs) trained on the German Traffic Sign Recognition Benchmark (GTSRB) dataset [179]. This multi-class dataset comprises images of German road signs spanning 43 classes, presenting challenges for both humans and models due to factors such as perspective changes, shading, color degradation, and varying lighting conditions. The networks include QConv layers (which binarize the corresponding convolutional layers), Batch Normalization (BN), Max Pooling (MP), and Fully Connected (FC) layers. A total of 45 benchmarks are available for this dataset, created using l_∞ input perturbations with parameter epsilon values of $\{1, 3, 5, 10, 15\}$. These networks do not use explicit activation functions but instead employ a `sign` operator, which converts the input to 1, -1, or 0 if the input is positive, negative, or zero, respectively.

IMAGENET-1K: To analyze the effect on scalability when using appended networks, we used the VGGNET-16 architecture [106], the only one from VNN-COMP that runs on Imagenet. The network consists of convolutional layers, RELU activation functions, and max pooling layers, with a total of $138M$ parameters and $13.16M$ RELU activations. The properties were generated by applying perturbations on single input pixels to all 150528 input pixels using l_∞ perturbations. The post condition was generated wrt target robustness, where we check for misclassification wrt a fixed target class.

7.4.2 Evaluating robustness variants and comparing across thresholds (RQ1, RQ2)

We conducted experiments on various robustness properties discussed in Chapter 6 across multiple datasets. The combined results for all datasets, evaluated with respect to each property, are presented in Figure 7.2. Detailed analyses for individual datasets are provided in the subsequent figures within this subsection. For each property, a corresponding figure illustrates the results and analysis across the different datasets.

Relaxed robustness:

As defined in Equation 6.1 of Chapter 6, we need a user defined confidence level τ to analyse this property. We took five different confidence thresholds $\tau \in \{0, 60, 80, 90, 95\}$, where $\tau = 0$ corresponds to standard robustness. For a given threshold value τ, a result of SAFE indicates either no misclassification or a misclassification with confidence below $\tau\%$, whereas a result of UNSAFE indicates a counterexample with confidence above the lower bound derived in Chapter 6.

The results in Figure 7.2A present the combined outcomes across all datasets for the relaxed robustness property. As the confidence level increases, the number of safe (verified) cases rises while the number of unsafe (counterexample) cases decreases. This trend occurs because higher confidence levels impose stricter bounds on counterexamples, reducing their occurrence. Additionally, the number of TIMEOUT cases decreases with increasing confidence, as the search space becomes more constrained. Consequently, most cases are efficiently verified by abstraction-based modules such as CROWN [42] and MILP-based bound tightening within the $\alpha\beta$-CROWN framework.

Figure 7.3 presents the analysis across different datasets. For the MNIST benchmarks, we observe zero TIMEOUT cases beyond a confidence threshold of 80%. Since these are small networks, $\alpha\beta$-CROWN employs the Gurobi [40] solver for verification. As the confidence threshold increases, the search space tightens, allowing all benchmarks to be verified either by CROWN [42] or the MILP bound-tightening technique. A similar trend is observed for the CIFAR-10 benchmarks—higher confidence thresholds make the definitions more relaxed, resulting in more SAFE cases, which can be efficiently verified by incomplete methods like CROWN.

We observed surprising behavior with the GTSRB dataset, as shown in Figure 7.3. The number of SAFE, UNSAFE, and TIMEOUT cases remained constant, despite increasing the confidence threshold. There were a total of 45 benchmarks available for this category,

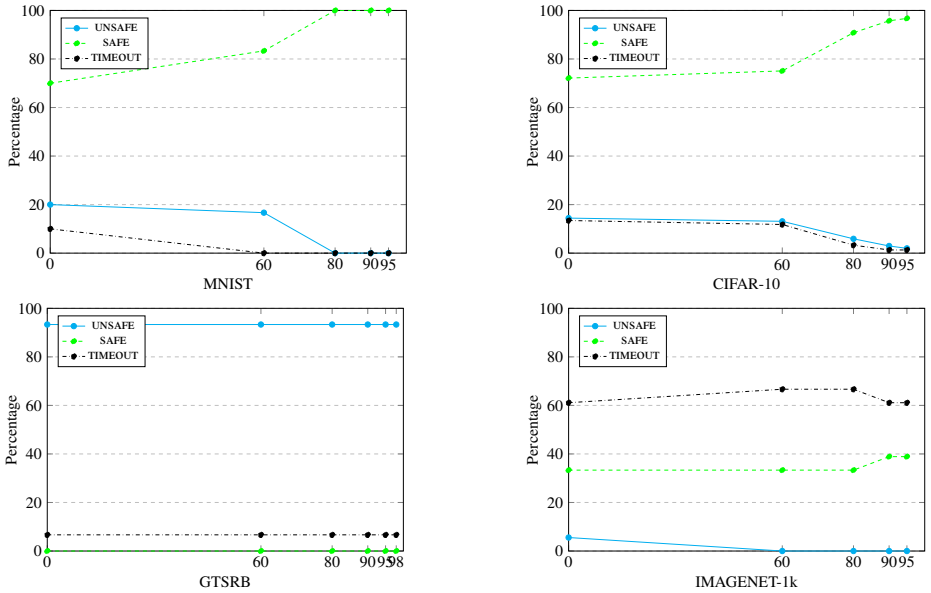


Figure 7.3: Analysis of relaxed robustness with respect to each dataset separately.

resulting in 45 benchmarks for each confidence threshold. For each threshold, we found 42 UNSAFE, 3 TIMEOUT, and 0 SAFE cases. This behavior raised curiosity, prompting us to experiment with a confidence threshold of 98%, but the same pattern persisted. Additionally, we noticed that the confidence for all seed images was 100%. This behavior raises serious concerns about the vulnerability of these networks. If both the seed image and counterexample have near-100% confidence, it is a red flag for these networks. Confidently making incorrect decisions is more dangerous than doing so with lower confidence, as a user might intervene to validate decisions made with lower confidence.

In the IMAGENET-1K benchmarks, there are a total of 18 benchmarks, implying 18 benchmarks for each confidence threshold. We observed on a GPU machine that many of the benchmarks ran out of memory with 32GB of GPU RAM. As a result, we had to run them on a CPU machine: Intel(R) Xeon(R) Gold 6314U CPU @ 2.30GHz with 64GB of RAM. For the standard robustness property with a confidence of 0%, we observed 1 UNSAFE, 6 SAFE, and 11 TIMEOUT cases. At confidence levels of 60% and 80%, the 1 UNSAFE case was converted to a TIMEOUT. At 90% and 95% confidence, it was converted to a SAFE case. The 11 TIMEOUT cases remained as TIMEOUT for all confidence levels.

Figures 7.4, 7.5, and 7.6, and 7.7 provide more insightful examples of the relaxed robustness definition.

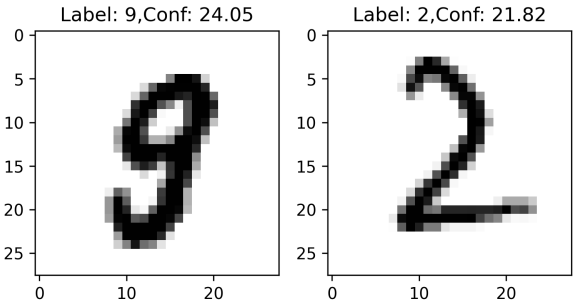


Figure 7.4: The above images were verified using relaxed robustness with a confidence threshold of 80%.

Figure 7.5: The image on the left was correctly classified as label AIRPLAN by the network `convBigRELU-PGD.onnx` with a confidence of 91.37%, but it was misclassified as `automobile` with a confidence of 22.13% within an ϵ perturbation of 0.007 under the standard robustness property. The same image was then verified by relaxed robustness with a confidence threshold of 90%.

Strong robustness:

We took the threshold value τ_1 as 57.5 for the seed image in Eq 6.4 for CIFAR-10 and GTSRB datasets, because it is the average value of all the confidences on the seed images of the properties. We took three different values of the confidence thresholds $\tau_2 = 30, 35, 40$. Intuitively, we want our model to be robust against the misclassification, at the same time, confidence should not go below τ_2. As shown in Figure 7.2B, the counterexamples increase as we increase the confidence τ_2, because the higher the threshold, means higher the chances of confidence falling below the threshold, with almost no increase in time taken. Figure 7.8 shows the strong robustness behavior with varying threshold levels. In addition to SAFE, UNSAFE, and TIMEOUT, we also introduced a new attribute to the graphs, namely TRIVIAL SAFE. The TRIVIAL SAFE cases occur when the confidence of the seed image is below the threshold (τ_1).

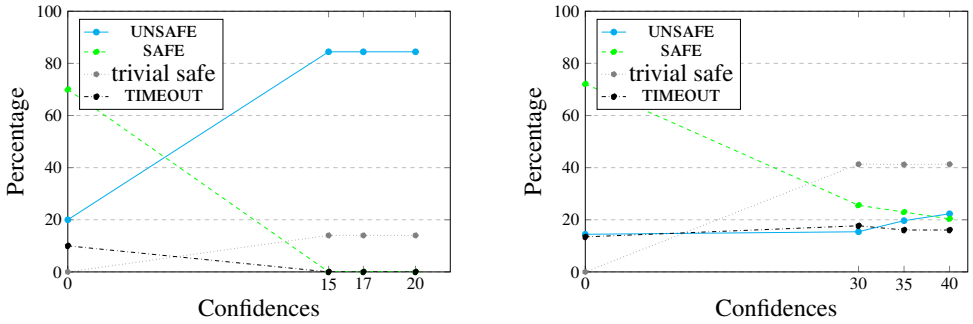
We observed that all the MNIST benchmarks became trivially safe, as the confidence

Figure 7.6: The left image is taken from the German Traffic Sign Recognition Benchmark (GTSRB) and belongs to the class `Speed limit (80 km/h)`. It is correctly classified by `net-1` (Table 7.1) with 100% confidence. However, under an ϵ-perturbation of $5/255$, it is misclassified as `Speed limit (120 km/h)` with a high confidence of 99.99%, highlighting a potential vulnerability in the network. The right image belongs to the class `Right-of-way at the next intersection` and is classified with 100% confidence. With the same ϵ-perturbation, it is misclassified as `Beware of ice/snow` with a confidence of 98.19%.

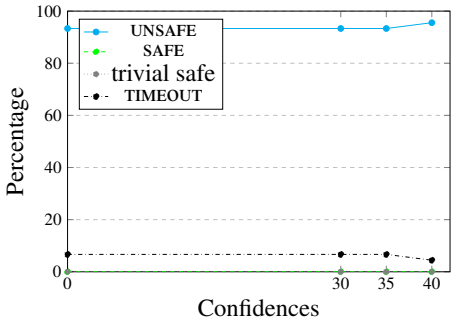
Figure 7.7: The image on the left is correctly classified as CARPENTERS KIT with 94.0% confidence by the VGGNET-16 network. However, within an epsilon perturbation of $1e-5$, it is misclassified as ABACUS with 38.23% confidence. This image was verified under relaxed robustness with a 95% confidence threshold.

of all seed images was below 57.5%. Therefore, we conducted a separate analysis for the MNIST dataset with varying threshold values. We took 22% as the threshold on seed images for MNIST benchmarks, and threshold values 15, 17, and 20 on the right side thresholds of the strong robustness equation. We found that after a confidence threshold 15%, the SAFE and TIMEOUT cases became 0, only UNSAFE and TRIVIAL SAFE cases remained. This implies these networks are not strongly robust with respect to the given threshold values.

For the CIFAR-10 dataset, as we increased the confidence threshold (τ_2) in the strong robustness equation, the number of UNSAFE cases increased, while the number of SAFE



(A) MNIST (B) CIFAR-10



(C) GTSRB

Figure 7.8: Analysis of strong robustness with respect to each dataset separately.

cases decreased. This trend is intuitive, as a higher confidence requirement increases the likelihood of falsifying the condition. The number of TIMEOUT cases remained almost unchanged. For the GTSRB dataset. The number of TRIVIAL SAFE cases is zero since all seed images have a confidence of 100%. We observed almost the same behavior as in relaxed robustness, except for one case that converted from TIMEOUT to UNSAFE at the confidence threshold of 40%.

Figure 7.9: This image from the GTSRB benchmark was classified correctly `Turn right ahead (33)` with 100% confidence by the network *net* – 3 in Table 7.1. However, this case resulted in a timeout in all three definitions: relaxed robustness, strong robustness, and smoothness.

Smoothness:

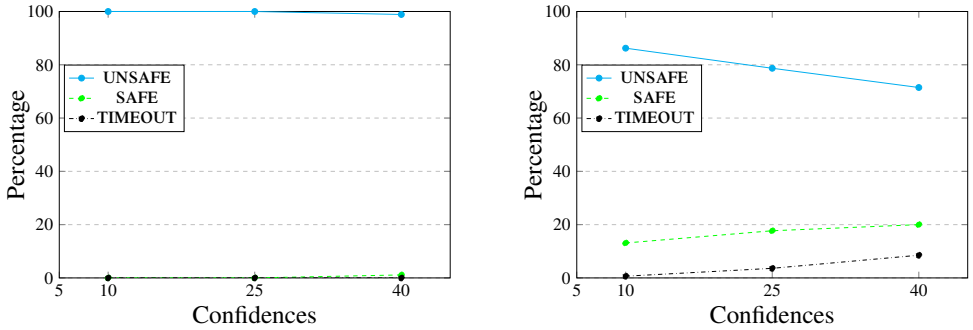
We considered varying threshold values of 10, 25, and 40 for smoothness. Intuitively, a threshold of 10 means that if, within an ϵ perturbation, the confidence of the seed image's class fluctuates within ±10 of its original confidence, the smoothness property holds (SAFE); otherwise, it does not hold (UNSAFE). Figure 7.2C illustrates how smoothness changes as the threshold varies. Increasing the smoothness threshold allows for greater variations in the output, resulting in more SAFE cases and fewer UNSAFE cases. However, the number of timeout cases slightly increases. A possible reason is that a lower threshold implies even slight changes in output confidence can lead to UNSAFE cases.

Figure 7.10 illustrates the behavior of the smoothness property across all three datasets. We observed that for the MNIST and GTSRB datasets, almost 99% of the benchmarks did not satisfy the smoothness property, indicating highly unsmooth behavior in these networks. The Figure 7.9 shows the timeout case. The CIFAR-10 benchmarks performed reasonably well, with the number of SAFE cases increasing as the smoothness threshold increased. This trend is intuitive since a higher smoothness threshold allows for greater variations in the output. Notably, we observed a slight increase in TIMEOUT cases, which may be due to the higher complexity of this property compared to relaxed and strong robustness. Unlike the other properties, smoothness requires verifying both upper and lower bounds of the output confidence.

Top-k, top-k-relaxed, and top-k-affinity robustness:

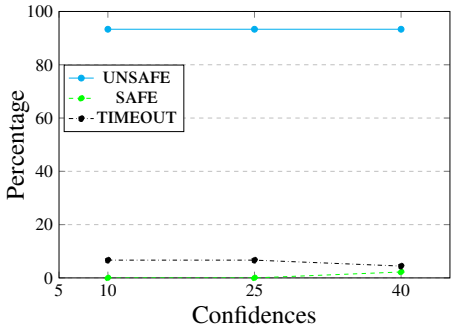
In both variations of top-k robustness, no threshold is involved, so the number of benchmarks remains the same as in standard robustness. We observed in previous experiments that for the GTSRB dataset properties, the confidence values for all seed images are 100%, meaning the confidence for all other classes is 0%. As a result, top-k properties cannot be applied to these benchmarks. Figure 7.2D compares standard robustness with top-k, top-k relaxed, and top-k affinity robustness. The number of UNSAFE cases decreases from standard robustness to the other three robustness definitions.

Affinity robustness requires a predefined set of classes in which misclassification is allowed (prior knowledge). For MNIST, this affinity set is defined as: {{0,8}, {4,9}, {1,9,7},{2}, {3}, {5}, {6}}. Intuitively, this means that 0 is allowed to be misclassified as 8 but not as any other class, while 9 is allowed to be misclassified as 4, 1, or 7 only. Similarly, 2 is not allowed to be misclassified into any other class. For



(A) MNIST

(B) CIFAR-10



(C) GTSRB

Figure 7.10: Analysis of smoothness with respect to each dataset separately.

CIFAR-10, we define the affinity set as: $\{\{bird\}, \{airplane, automobile, ship, truck\}, \{cat\}$ $\{deer, dog, horse\}, \{dog\}, \{frog\}\}$. Here, machines (airplane, automobile, ship, truck) are allowed to be classified among themselves, and the same applies to animals (deer, dog, horse). The affinity set is user-defined and may vary depending on the user or application. We took $k = 2$ for both mnist and cifar-10 benchmarks.

Figure 7.11 compares the results on the MNIST and CIFAR-10 datasets separately. As mentioned in the experiment section of the main paper, all the seed images in the GTSRB dataset have 100% confidence, making top-k analysis inapplicable to those benchmarks.

For the MNIST benchmarks, the results align with intuition: SAFE cases increase while UNSAFE cases decrease as we move from standard robustness to affinity and then to top-k relaxed robustness. This trend is likely due to the relatively small network sizes, making it easier to find solutions. On the other hand, for the CIFAR-10 benchmarks, while SAFE cases are theoretically expected to increase, we observed a rise in TIMEOUT cases instead. A potential reason for this could be the increased complexity of the problem, specifically the presence of conjunctions of disjunctions, making verification more computationally challenging.

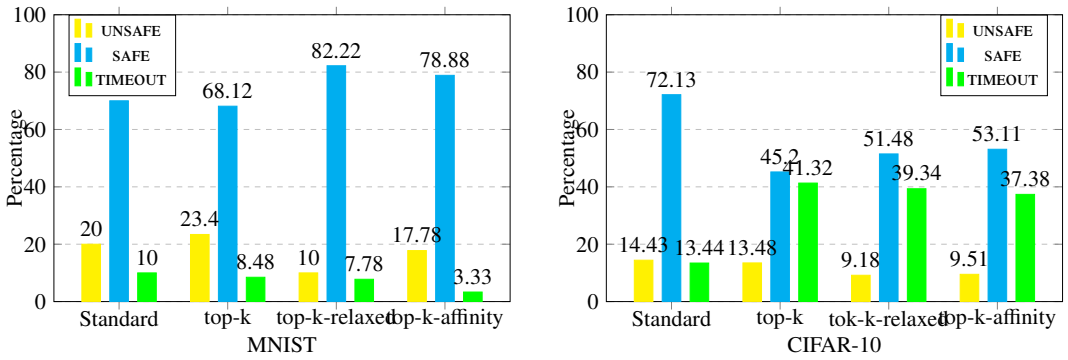


Figure 7.11: Top-K: The left figure shows the comparison on MNIST benchmarks, right figure shows comparison on CIFAR-10 benchmarks

We wish to recall that Figure 7.2D compares standard robustness with top-k relaxed and top-k affinity robustness. The number of UNSAFE cases decreases from standard robustness to the other two robustness definitions. Although both versions of top-k robustness are more relaxed than standard robustness, the number of SAFE cases also decreases due to an increase in the number of TIMEOUT cases. One possible reason for the increase in timeouts is the added complexity of the constraints in both top-k robustness definitions. In standard robustness, the property consists of a disjunction of atomic properties, whereas in top-k robustness, it involves a conjunction of disjunctive clauses. This requires additional layers to be appended to the existing neural network, increasing computational complexity.

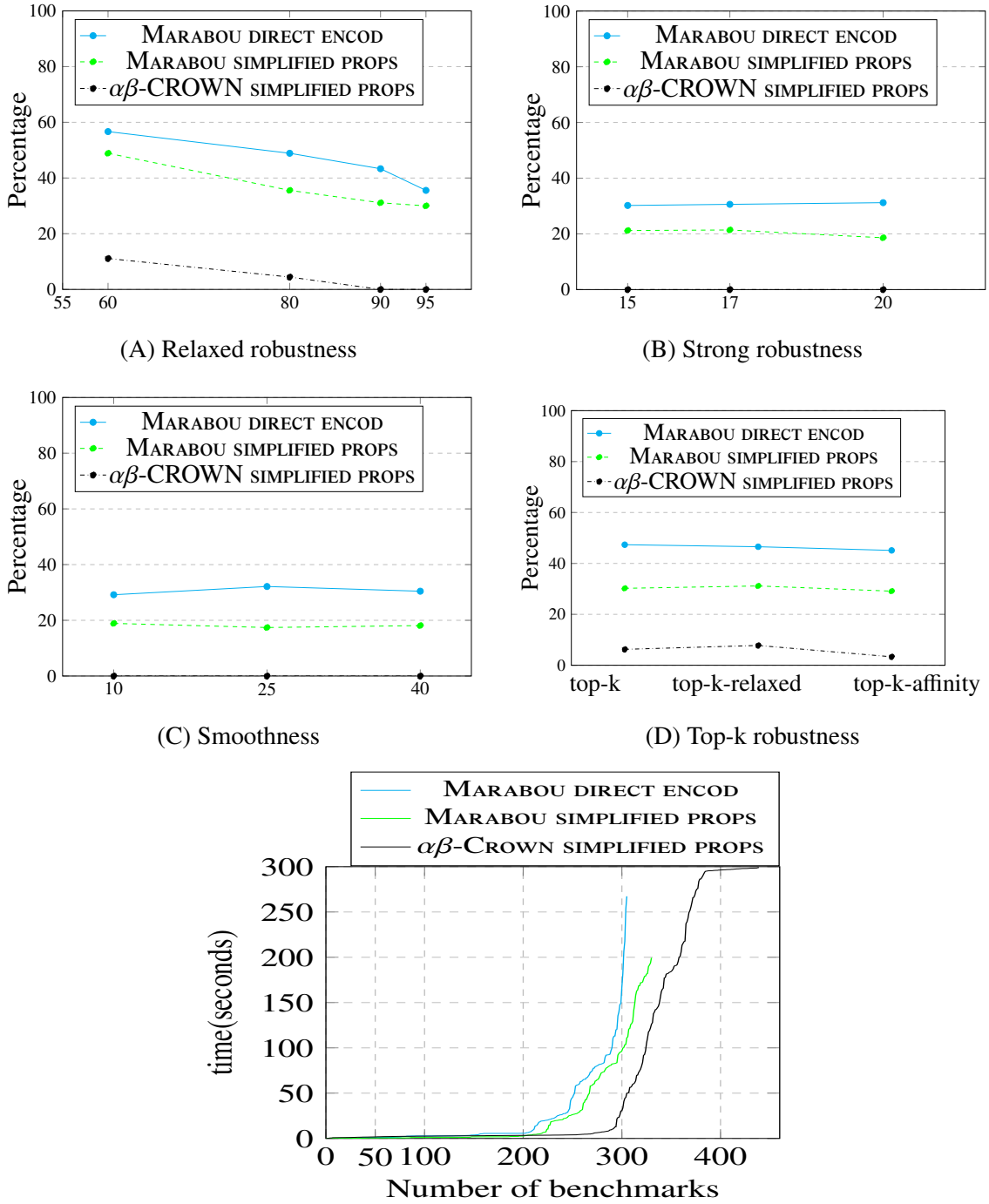
7.4.3 Comparing with a direct encoding (RQ3)

To compare our layer-based encoding with a direct encoding, for each of the specific richer properties considered, we encoded them directly with the constraint-based solver MARABOU using its Python interface. We compared this with our layer-based encoding on MARABOU (i.e., an appended network with a simplified property) as well as $\alpha\beta$-CROWN. Since MARABOU is a CPU-based solver [76], we conducted the experiments on an Intel(R) Xeon(R) Gold 6314U CPU @ 2.30GHz with a 64GB RAM machine. We observed that MARABOU ran out of memory for many CIFAR-10 benchmarks, so we restricted this comparison to MNIST benchmarks. Figure 7.12A - 7.12D presents the comparison with the relaxed robustness, strong robustness, smoothness, and top-k properties, using the same confidence threshold as in previous experiments. Figure 7.12E shows the cactus plot of the comparison. Our results indicate that $\alpha\beta$-CROWN using our layer-based encoding outperforms MARABOU, both with the direct encoding and with the layer-based one. This is

perhaps not surprising, as $\alpha\beta$-CROWN is a portfolio tool incorporating efficient techniques such as PGD-attack [11] and CROWN [33]. However, as an ablation study, we can see that MARABOU with our layer-encoding does perform better than MARABOU without it (i.e., with the direct encoding). *This comparison demonstrates that our framework not only enables the encoding of richer properties but also is efficient compared to directly encoding the properties.*

7.5 Conclusion

In this chapter, we introduced a grammar of post-conditions that captures all these variants and provided a unifying framework to transform them into a gadget that can be appended to an existing neural network, thereby simplifying arbitrary properties. This enables the use of verifiers such as $\alpha\beta$-CROWN. Our experiments demonstrate that our approach is more efficient than directly encoding the properties as constraints into state-of-the-art constraint-based solvers.



(A) Relaxed robustness

(B) Strong robustness

(C) Smoothness

(D) Top-k robustness

(E) Cactus plots for only solved benchmarks

Figure 7.12: (A-D) Comparison of the constraint-based solver MARABOU with and without the simplified property, along with $\alpha\beta$-CROWN using the simplified property. The y-axis represents the percentage of timeout cases, while the x-axis denotes the confidence thresholds. (E) The x-axis shows the number of benchmarks solved, ordered by increasing solving time, and the y-axis indicates the time taken to solve them.

Chapter 8

Explainable Reward Learning in Inverse Reinforcement Learning

If a model is verified, one may still ask why a model behaves the way it does, since these models are not explicitly designed by humans but are instead learned from input-output pairs. Therefore, we also consider the problem of explainability in machine learning models, where the goal is to understand the reasoning behind the decisions of a model. In this context, we focus on the explainability of policies learned by reinforcement learning models. Synthesizing explanations from demonstrations has emerged as an effective approach in scenarios where domain experts or performant agents can provide examples of both desirable and undesirable behaviors. One important embodiment of this form of learning is known as *inverse reinforcement learning* [79] (IRL), whereby an apprentice agent learns the reward function being optimized by a given expert policy or behavior. IRL is indispensable in settings where it is difficult or error-prone to explicate a reward signal that captures the underlying learning objective. In explainability, we go one step further and summarize the learned reward function into an interpretable formula. Our approach expresses the behavior of such policies in the form of *Linear Temporal Logic* (LTL) formulas, providing an interpretable description of the learned behavior.

Formally, the key computational problem for the LTL-based IRL is the following: given a pair $\mathcal{S} = (P, N)$ of samples consisting of positive traces P and negative traces N (both are sets of finite words), produce the highest-ranking LTL specification consistent with the sample where rank is informed by some user-tunable notion of simplicity over the LTL specifications. And, here lies our central challenge of induction: we need to infer an

LTL specification over unbounded length traces by observing a finite set of finite examples and counterexamples!

To address the challenge, we define quantitative semantics of satisfaction of an LTL specification over a finite word guided by a notion of parsimony of explanation. The complexity of an LTL formula can result from two aspects: the complexity of the *temporal structure* ($\mathsf{G}p$ is simpler than $p \wedge \mathsf{X}p \wedge \mathsf{XX}p$ in explaining the sample $P = \{\{p\}\{p\}\{p\}\}$ and $N = \emptyset$) and the complexity of the *nesting structure* (the formula p is simpler than $p \wedge \neg q$ for explaining the sample $P = \{p\}$ and $N = \emptyset$). We expose hyperparameters (temporal discounting α and nesting discounting δ) to take user preference in weighing these sources of complexity. To avoid overfitting, we focus on the GF fragment [94, 95] of LTL (temporal operators are restricted to G and F).

We implemented our algorithms as an open-source tool. We demonstrate the effectiveness of our method on randomized gridworld environments by computing the inverse learning error (ILE) for policies computed using the reward learned by our tool and compare it with the reward learned by a competing approach.

In this chapter, we begin with the formal definitions and preliminaries in Section 8.1. We provide an illustrative example in Section 8.2. Next, we present the quantitative semantics for LTL over finite words in Section 8.3. In Section 8.4, we present three algorithms to solve the LTL formula learning problem. Finally, we provide the experimental evaluation in Section 8.6 and conclude the chapter in Section 8.8.

8.1 Preliminaries

Propositional Logic. Let *Var* be a set of *propositional* variables, which take values from $\mathbb{B} = \{0, 1\}$ (0 interpreted as $false$ and 1 as *true*). The set of formulae $\mathcal{W}$ in propositional logic — with formulae denoted herein as Greek letters — is defined inductively as follows:

$$\varphi ::= p \in \mathcal{P} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi$$

We use the usual syntactic sugar $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, and $\varphi \Leftrightarrow \psi$.

A *propositional valuation* is defined as a mapping $v : Var \to \mathbb{B}$, which maps propositional variables to Boolean values. The semantics of this logic, given by the satisfaction relation $\models$, are defined inductively as: (1) $v \models x$ iff $v(x) = 1$; (2) $v \models \neg\varphi$ iff $v \not\models \varphi$; (3) $v \models \varphi \wedge \psi$ iff $v \models \varphi$ and $v \models \psi$. If $v \models \varphi$ we say v models φ. A formula is said to be *satisfiable* if there exists a model for it. There are practical tools, called SAT solvers, that can check the satisfiability of the formulae.

An *alphabet* Σ is a non-empty, finite set of *symbols*. A *finite word* w over Σ is a finite sequence $a_1 a_2 ... a_n$ of symbols from Σ. The empty sequence is called the empty word, denoted ϵ. The domain of w, denoted $\mathsf{dom}(w)$, is the set of positions in w. Thus, $\mathsf{dom}(a_1 \dots a_n) = \{1, 2, \dots, n\}$ and $\mathsf{dom}(\epsilon) = \emptyset$. The length of a finite word w is denoted $|w|$, with $|\epsilon| = 0$. The set of all finite words over Σ is denoted Σ^*. An *infinite word* over Σ is an infinite sequence $w = a_0 a_1 a_2 ...$ of symbols $a_i \in \Sigma$ with $i \in \mathbb{N}$. The set of all infinite words over Σ is denoted Σ^ω.

Given a finite word w, we write $w(i)$ for the symbol at position i. The subsequence of w from index i to index j, both inclusive, is denoted as $w[i : j]$, while $w[i :]$ denotes the suffix of w starting from index i. When clear from context, we write w_i for $w[i :]$.

Linear Temporal Logic (LTL). Let $\mathcal{P}$ be a set of propositional variables. LTL [180] is an extension of propositional logic with temporal modalities, which allows the expression of temporal properties. Formulae in LTL are defined inductively. An LTL formula φ over $\mathcal{P}$ is defined by the following grammar:

$$\varphi ::= \mathit{true} \mid a \in \mathcal{P} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

Using the above, the formulae $\mathsf{F}\varphi = \mathit{true}\,\mathsf{U}\,\varphi$ and $\mathsf{G}\varphi = \neg\mathsf{F}\,\neg\varphi$ can be derived. The size of an LTL formula φ denoted by $|\varphi|$ is the number of subformulae in it. For example, if $\varphi = p\,\mathsf{U}\,\psi$, then $|\varphi| = |\psi| + 2$. We say that an infinite word α satisfies an LTL formula ϕ, and we write $\alpha \models \phi$, if:

- $\alpha \models a$ iff $a \in \alpha[0 : 0]$
- $\alpha \models \neg\Phi$ iff $\alpha \not\models \Phi$
- $\alpha \models \Phi \wedge \Psi$ iff $\alpha \models \Phi$ and $\alpha \models \Psi$
- $\alpha \models \mathsf{X}\Phi$ iff $\alpha[1 :] \models \Phi$
- $\alpha \models \Phi\,\mathsf{U}\,\Psi$ iff $\exists\, i$ s.t. $\alpha[i :] \models \Psi$, and $\forall\, j < i, \alpha[j :] \models \Phi$.

The language $L(\varphi)$ of an LTL formula φ is defined as

$$L(\varphi) = \{\alpha \in \Sigma^* | \alpha \models \varphi\}\ .$$

Two LTL formulae having the same language are called *equivalent*. In this paper, we learn formulae that are in the GF-fragment of LTL, where only the G and F modalities are

allowed apart from Boolean connectives. Since the LTL formulae can be converted into negation normal form (NNF), we learn formulae only in NNF. Given an LTL formula φ, the syntax tree of φ is a tree labeled with variables, Boolean connectives, and temporal modalities. The variables always appear at the leaf nodes, while temporal modalities and Boolean connectives are internal nodes. For example, we present the syntax tree of the formula $\varphi = [\mathsf{G}q \wedge \mathsf{F}r] \vee (\mathsf{FG}q)$ in Figure 8.1(b). The depth of an LTL formula is the maximum of distances of the root to leaves in the corresponding syntax tree. The depth in Figure 8.1(b) is 3. Although LTL is defined over infinite words, we observe only finite executions of the systems. Therefore, we defined samples with finite words as follows.

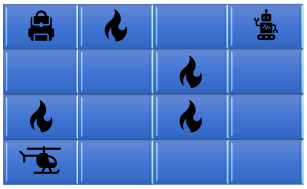
Samples. A *sample* is a pair $\mathcal{S} = (P, N)$ of two finite, disjoint sets $P, N \subseteq (2^{\mathcal{P}})^*$. The words in P are *positive traces* while words in N are *negative traces*. For an LTL formula φ, we say $\mathcal{S} \models \varphi$ iff $\forall \tau \in P,\ \tau \models \varphi$ and $\forall \tau' \in N,\ \tau' \not\models \varphi$. We learn an LTL formula from a given sample.

We assume the setting of non-Markovian reward decision processes (NMRDPs) as the model of the agent-environment interactions. Since we are interested in learning qualitative behavior, we assume a binary reward signal that captures the acceptance semantics of the underlying language.

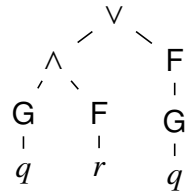
Definition 10 (Non-Markovian Reward Decision Process)**.** *An NMRDP is a tuple* $M = (S, s_0, A, T, R, L, \Sigma)$*, where* S *is a finite set of states,* $s_0 \in S$ *is a distinguished initial state,* A *is a finite set of actions,* $T : S \times A \times S \to [0, 1]$ *is a probabilistic transition function,* Σ *is an alphabet (i.e. the power set* $2^{\mathcal{P}}$ *of a set of atomic propositions* $\mathcal{P}$*),* $L : S \to \Sigma$ *is a labeling function, and* $R : S^* \to \{0, 1\}$ *is a non-Markovian reward function.*

The labeling function $L : S \to 2^{\mathcal{P}}$ maps states of the NMRDP to the alphabet $\Sigma = 2^{\mathcal{P}}$ that defines the language of the underlying objective. This alphabet denotes semantically meaningful events observed in given states. Given a sequence of states $s_i, s_j, \ldots, s_k$, the corresponding trace is given by $L(s_i), L(s_j), \ldots, L(s_k)$. Given positive and negative examples of such traces, the proposed non-Markovian IRL solution can learn the underlying LTL formula that captures the objective of the agent.

Given an NMRDP, the objective of reinforcement learning (RL) is to learn an optimal policy $\pi : S^* \to \mathcal{D}(A)$, where $\mathcal{D}(A)$ denotes the space of probability distributions over the action space A. An optimal policy π is one that maximizes the expected reward observed by



(a)



(b)

Figure 8.1: (a) Disaster response example (b) The syntax tree for $[\mathsf{G}q \wedge \mathsf{F}r] \vee (\mathsf{FG}q)$

the agent, as given by Equation (8.4). We define the random variable $(\tilde{s}_0, ..., \tilde{s}_n$ as follows.

$$\mathbb{P}(\tilde{s}_0, ..., \tilde{s}_n = s_0 ... s_n) = \Pi_{i=1}^{n} \sum_{a} \pi(s_0, ..., s_i)(a).T(s_{i-1}, a, s_i) \tag{8.1}$$

$$score(s_0, ..., s_n) = \sum_{t=1}^{n} \gamma^t R(s_0, ..., s_t) \tag{8.2}$$

$$\mathbb{E}\left[score(\tilde{s}_0, ..., \tilde{s}_n) | \tilde{s}_0 = s\right] = \sum_{s_0,...,s_n \in S^{n+1} \text{ such that } s_0 = s} score(s_0, ..., s_n)\mathbb{P}(\tilde{s}_0, ..., \tilde{s}_n = s_0, ..., s_n) \tag{8.3}$$

Due to the policy π, the expected reward signal $V_\pi(s)$ in non-Markovian setting also depends on the history, which is defined as follows.

$$V_\pi(s) = \lim_{n \to \infty} \mathbb{E}\left[score(\tilde{s}_0, ..., \tilde{s}_n) | \tilde{s}_0 = s\right] \tag{8.4}$$

The objective is to learn a reward signal that best characterizes the given observations of behavior from an expert. For a given process, the performance of the learned reward can be measured in terms of the inverse learning error (ILE) given by $\|V_{\pi^*_{\text{true}}} - V_{\pi^*_{\text{learned}}}\|_2$ [98], where $V_{\pi^*_{\text{true}}}$ denotes the value function computed using the optimal policy π^*_{true} from the true reward signal and $V_{\pi^*_{\text{learned}}}$ denotes the same for the policy π^*_{learned} derived from the reward signal learned using IRL. The value functions are computed over the NMRDP M by leveraging the learned policy π^*_{learned} using the true reward function. A learned reward signal is said to generalize if the ILE is low when compared to the ground truth.

8.2 A Motivating Example

Consider the RL problem illustrated in Figure 8.1(a). The NMRDP that defines this problem is given by $M = (S, s_{14}, A, P, L, R)$, where S consists of sixteen states as denoted by the grid position of the agent, with starting state s_{14} (top right, at position $(1, 4)$). The action space

A is given by the cardinal directions with the obvious deterministic transition dynamics P. The labeling function is given by $L(s_{12}) = L(s_{23}) = L(s_{31}) = L(s_{33}) = \{\texttt{danger}\}$, $L(s_{11}) = \{\texttt{cargo}\}$, and $L(s_{41}) = \{\texttt{rendezvous}\}$. Suppose that the objective of the agent is to eventually obtain the cargo and, once the cargo is obtained, the agent must avoid the dangerous areas and eventually reach the rendezvous point. We also required agent to avoid dangerous areas even after reaching the rendezvous point. This is given by the LTL formula $\phi = (\mathsf{F}\texttt{cargo}) \wedge [\mathsf{G}(\texttt{cargo} \implies (\mathsf{G}\neg\texttt{danger}) \wedge (\mathsf{F}\texttt{rendezvous}))]$. In the IRL context, this objective is not known to the agent directly, and it must instead learn from expert demonstrations.

Our proposed approach enables the use of both positive and negative traces of behavior to guide the learning process. In this example, a positive trace can be $\{\texttt{danger}\}\ \{\texttt{cargo}\}\ \{\texttt{rendezvous}\}$ since it clearly satisfies the underlying objective given by ϕ. A bad trace can be $\{\texttt{danger}\}\ \{\texttt{cargo}\}\ \{\texttt{danger}\}\ \{\texttt{rendezvous}\}$ as generated by the policy of moving left three times and then down three times. Since trajectories of states given by the expert (or produced by the expert policy) have a corresponding trace induced by the labeling function L, we assume that the traces are given over the alphabet $\Sigma = 2^{\mathcal{P}}$ as opposed to the state space of the NMRDP.

8.3 Occam's Razor for LTL

Given an LTL formula φ and a finite word w, we design a valuation function $V(\varphi, w)$ that quantifies the parsimony of φ in explaining w. Intuitively, a pair scores high if all of the subformulae of the formula φ contribute in accepting w in $L(\varphi)$. However, we do so in a nuanced fashion by geometrically attenuating the effect of parsimony with the length of the word. For example, $\mathsf{G}(p \vee q)$ should score well along with word $(\{p\}\{q\})^3$ but should not do well with word $(\{p\})^6$, since the subformula q did not contribute to the acceptance. Similarly, $\mathsf{G}(p \vee q)$ should score better along with word $(\{p\}\{q\})^3$ than $(\{p\})^5\{q\}$.

8.3.1 Quantifying Expressive Parsimony

Let us present a valuation function first. Let $\mathcal{F}$ represent the set of NNF GF-fragment formulae over $\mathcal{P}$. We interpret LTL formulae over finite words and define the quantitative semantics in terms of a *valuation mapping* $V : \mathcal{F} \times \Sigma^* \to \mathbb{R}^+ \cup \{0\}$, where $\Sigma = 2^{\mathcal{P}}$. The

valuation mapping is defined over a word $w \in \Sigma^*$ inductively:

$$V(p, w) = \begin{cases} 1 & \text{if } p \in w(1) \\ 0 & \text{otherwise} \end{cases}$$

$$V(\neg p, w) = \begin{cases} 1 & \text{if } p \notin w(1) \\ 0 & \text{otherwise} \end{cases}$$

$$V(\varphi \wedge \psi, w) = \delta \cdot V(\varphi, w) \cdot V(\psi, w)$$

$$V(\varphi \vee \psi, w) = \delta \cdot \frac{V(\varphi, w) + V(\psi, w)}{2}$$

$$V(\mathsf{G}\varphi, w) = \begin{cases} \delta \sum_{i=0}^{|w|} \alpha^i V(\varphi, w_i) & \text{if } V(\neg\varphi, w_t)=0,\ \forall t \\ 0 & \text{otherwise} \end{cases}$$

$$V(\mathsf{F}\varphi, w) = \begin{cases} \delta\alpha^t V(\varphi, w_t) & t = \min\{j \mid V(\varphi, w_j) > 0\} \\ 0 & \text{if } V(\varphi, w_t) = 0,\ \forall t \end{cases}$$

Here w_j is a shorthand for $w[j:]$. If $w \models \varphi$, then $V(\varphi, w)$ is non-zero. This scheme is parameterized by two discount factors: the *temporal discount factor* α and the *nesting discount factor* δ.

For the literals, we assign a valuation of zero or one if the word satisfies the literals or not. We interpret conjunction as multiplication, which implies we need both subformulae to do well on the word. We interpret disjunction as an addition, which implies that we give a high score to the formula if any of the two subformulae does well on the word. Our interpretation of $\mathsf{G}\varphi$ computes the discounted sum of the value of φ at each position of the word. To reduce the weight of a letter appearing further in the word, we apply the temporal discount of $0 < \alpha < 1$. Our interpretation of $\mathsf{F}\varphi$ computes the discounted score of φ at the earliest position where φ has a non-zero score. We apply the nesting $0 < \delta < 1$ each time we construct a more complex formula, i.e. we go deeper in the nested structure of the LTL formula. The discount values δ and α are parameters defined by the user to control nesting complexity and temporal complexity.

Example 1. *Consider the formula $\varphi = \mathsf{F}q$ for $p, q \in \mathcal{P}$ and the word $w = \{p\}\{p, r\}\{p\}\{p, s\}\{p\}\{p\}\{p, q\}(\{r\}\{q\})^*$. Then q holds for the first time at the position $t = 7$, and for all $t' < t$, q is not present in w. Thus, $V(q, w[7:]) = 1$, making $V(\varphi, w) = \delta\alpha^7$. Note that our valuation assigns non-zero scores only for satisfiable formulas.*

We extend the notion of a valuation from a word to a sample in a natural fashion. For a sample $\mathcal{S}$, the valuation of φ is taken as the sum of valuations over all positive traces in the sample:

$$V(\varphi, \mathcal{S} = (P, N)) = \sum_{w \in P} V(\varphi, w).$$

This scheme attempts to match intuition about the operators. We posit that while the definition of the valuation functions is subjective, natural variations do not provide any significant theoretical advantages to our paradigm. For example, One may assign $V(\varphi \wedge \psi, w)$ as the minimum of $V(\varphi, w)$ and $V(\psi, w)$ [181]. This valuation ensures that both φ and ψ must score high for $\varphi \wedge \psi$ to score high. However, the function is not sensitive to the formula that has a higher value. Therefore, the learning algorithm becomes unguided for one part of the formula. This suggests a modification to the valuation function that takes both the subformulae into account symmetrically, without flattening one of the subformulae. Our valuation function for the conjunction of two formulae, defined as their product $V(\varphi, w) \cdot V(\psi, w)$ is based on this idea.

8.4 Learning Algorithms

As our main contribution, we propose learning algorithms to solve the following problem. *Given a sample $\mathcal{S}$=(P,N) over finite words, compute an LTL formula φ in the GF-fragment that best describes $\mathcal{S}$ and is consistent with $\mathcal{S}$. That is, φ has the highest score, based on the valuation described above, among all formulae such that for all $w \in P$, $w \models \varphi$ and for all $w' \in N$, $w' \not\models \varphi$.*

To achieve the goal of ranking formulae based on a quantitative notion of satisfiability, we propose the techniques of *Constraint System Optimization* (Section 8.4.1) *Optimized Pattern Matching* (Section 8.4.2) and *Hybrid Pattern Matching* (Section 8.4.3). In the first one, we get a sample and a depth d as input. We encode the syntax tree of this unknown formula of depth d along with constraints to compute the score of each node in the tree. The second one makes use of a formula template pattern provided by the user, but has unknown propositional variables. We encode constraints that allow mapping these variables to unique variables occurring in the sample. The third one is a "hybrid" approach, a middle ground incorporating both of the above techniques. We use an optimizing SMT solver to solve the constraints to find the *best* formula for the sample with the highest score according to the valuation function.

Algorithm 6 Computing the optimal formula given a sample

```
1: procedure ConstraintOpt(S = (P, N), d)
2:     construct Φ_d^S                                  ▷ Constraints in eq(8.5)
3:     max. min({y^τ_{1,0} | τ ∈ P}) with Φ_d^S as a constraint
4:     if optimization succeeds with model m then       ▷ SAT
5:         construct formula tree from m
6:         return optimized formula tree
7:     else
8:         return UNSAT
```

8.4.1 Constraint System Optimization

In Algorithm 6, we present the method to compute the optimal formula with the highest score for a sample $\mathcal{S} = (P, N)$ along with the desired depth d of the formula. This is obtained by computing and optimizing the scores of a class of formulae of depth d, constrained to be well-formed and to be satisfied by $\mathcal{S}$. We reduce the construction of an LTL formula for a sample $\mathcal{S}$ to a constraint system $\Phi_d^{\mathcal{S}}$, which is constructed in three parts as:

$$\Phi_d^{\mathcal{S}} = \varphi_d^{\mathrm{ST}} \wedge \bigwedge_{\tau \in P} \varphi_d^{\tau} \wedge \bigwedge_{\tau' \in N} \neg \varphi_d^{\tau'} \tag{8.5}$$

The first part encodes the structure of a *syntax tree* (ST) representing an unknown formula, while the second and third encode the functional constraints on the *score* of each node enforced by the operators that make up the formula.

To encode the formula structure, we use a syntax tree with identifiers $\mathcal{N} = \{1, 2, \ldots, n\}$ for the set of nodes, where $n = |\mathcal{N}|$. We assume herein that the root node is identified as 1. We have the child relations L and R, such that $(i, j) \in L$ (or R) iff the node j is the left (right) child of node i. The only child of unary operators is considered as a left child by assumption. For each node $i \in \mathcal{N}$ and possible label $\lambda \in \mathcal{O} \cup \mathcal{P}$, we introduce a Boolean variable $x_{i,\lambda}$ indicating whether the node is labelled with an operator ($\mathcal{O}$) or variable ($\mathcal{P}$). The formula φ_n^{ST} is constructed as the conjunction of formulae (8.6) through (8.14).

$$\left[\bigwedge_{1 \le i \le n} \bigvee_{\lambda \in \mathcal{P} \cup \mathcal{O}} x_{i,\lambda}\right] \wedge \left[\bigwedge_{1 \le i \le n} \bigwedge_{\substack{\lambda \neq \lambda' \\ \in \mathcal{P} \cup \mathcal{O}}} (\neg x_{i,\lambda} \vee \neg x_{i,\lambda'})\right] \tag{8.6}$$

$$\bigwedge_{\nexists j (i,j) \in L} \bigvee_{p \in \mathcal{P}} x_{i,p} \tag{8.7}$$

The constraint given by expression (8.6) ensures the two properties that each node must accept at least one label, and that it must accept at most one label, while expression (8.7) ensures that the leaf nodes are labelled with propositional variables, and not operators (since they have no children).

Next, we encode the functional constraints imposed by the operators. For this, to each node $i \in \mathcal{N}$, we attach a set of real variables $Y_i^\tau = \{y_{i,t}^\tau \mid 0 \leq t \leq |\tau|, i \in \mathcal{N}\}$ representing its *score* at each point $0 \leq t \leq |\tau|$ in a trace $\tau \in P \cup N$, where $y_{i,t}^\tau$ is defined below. For a given trace τ we construct φ_d^τ as the conjunction of:

$$y_{1,0}^\tau > 0 \tag{8.8}$$

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{p \in \mathcal{P}} x_{i,p} \Rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \begin{cases} 1 & \text{if } p \in \tau(t) \\ 0 & \text{if } p \notin \tau(t) \end{cases} \right] \tag{8.9}$$

$$\bigwedge_{\substack{1 \leq i \leq n, \\ (i,j) \in L}} x_{i,\neg} \Rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \delta \cdot \max(0, 1 - y_{j,t}^\tau) \right] \tag{8.10}$$

$$\bigwedge_{\substack{1 \leq i \leq n, \\ (i,j) \in L, \\ (i,j') \in R}} x_{i,\wedge} \Rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \delta \cdot y_{j,t}^\tau \cdot y_{j',t}^\tau \right] \tag{8.11}$$

$$\bigwedge_{\substack{1 \leq i \leq n, \\ (i,j) \in L, \\ (i,j') \in R}} x_{i,\vee} \Rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} y_{i,t}^\tau = \delta \cdot \frac{y_{j,t}^\tau + y_{j',t}^\tau}{2} \right] \tag{8.12}$$

$$\bigwedge_{\substack{1 \leq i \leq n \\ (i,j) \in L}} x_{i,\mathsf{G}} \Rightarrow \bigwedge_{1 \leq t \leq |\tau|} \left[\left[y_{i,t}^\tau = \delta \sum_{t \leq t' < |\tau|} \alpha^{t'-t} \cdot y_{j,t'}^\tau \right] \wedge \left[\bigwedge_{t \leq t' < |\tau|} (y_{j,t'}^\tau > 0) \right] \right] \vee y_{i,t}^\tau = 0 \tag{8.13}$$

$$\bigwedge_{1 \leq i \leq n, (i,j) \in L} x_{i,\mathsf{F}} \Rightarrow \left[\bigwedge_{1 \leq t \leq |\tau|} \exists\, t'.\ \left[y_{i,t}^\tau = \delta \cdot \alpha^{t'-t} \cdot y_{j,t'}^\tau \right] \wedge (t \leq t') \wedge \left[\bigwedge_{t < t'' < t'} \neg (y_{j,t''}^\tau > 0) \right] \wedge (y_{j,t'}^\tau > 0) \right] \tag{8.14}$$

Corresponding to Boolean variables and to each operator, the constraints encode the calculation of the valuation of a given node as a function of the valuation of its children, as defined in Section 8.3. The score for a node labelled G at a position t in the trace is described by the constraint (8.13). The first conjunct encodes the actual score as a function of its child, adding the child's score over all positions in the input word, scaled by an exponential. The second conjunct simply ensures that the G-property holds in a classical sense, i.e., that its child has positive valuation at all positions. Similarly, F is encoded in constraint (8.14), where we look for the first position t' where its child has a positive valuation. The score of its child is exponentially scaled so as to diminish the contribution from an occurrence far away from the start.

Finally, we optimize the score of the syntax tree using the score of the root node $y^{\tau}_{1,0}$ w.r.t. the constraint system $\Phi^{\mathcal{S}}_{d}$ (Algorithm 6). Then we use the resulting model to label the tree, obtaining the optimal formula of chosen depth. By iterating over d, we may obtain the minimal such formula. Next, we establish the correctness and soundness of this algorithm.

Lemma 13 (Well-Formedness)**.** *Any satisfying assignment of the constraint system (8.5-8.14) encodes a well-formed LTL formula, and every well-formed formula of depth $\leq d$ can be encoded within $\Phi^{\mathcal{S}}_{d}$.*

The proof is a straight forward argument. In one direction, the constraints ensure that the $\{x_{i,\lambda} \mid \lambda \in \mathcal{P} \cup \mathcal{O}\}$ forms a well formed binary tree. The encoding requires matching this binary tree with the parse tree of the given formula. In the converse direction, one can recursively construct the parse tree of a formula that satisfies the constraint system.

Theorem 14 (Completeness)**.** *Given a sample $\mathcal{S} = (P, N)$, if there exists an LTL formula φ of depth d such that $\mathcal{S} \models \varphi$, then there exists a model m satisfying the constraint system (8.5 - 8.14), i.e. $m \models \Phi^{\mathcal{S}}_{d}$ such that the parse tree (x_i) encodes the formula φ and for each node i and position t in each trace τ, $y_{i,t} = V(\varphi_i, \tau[t:])$, where φ_i is the formula encoded by the subtree with root at node i.*

Proof. We induct on the depth d of the formula.

Base case: $d = 0$

$\varphi = p$ for some propositional variable p, and $\mathcal{S} \models \varphi$.

Setting $x_{1,p} = 1$ in m and resolving the constraints, for each positive trace τ we are left with

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \tag{8.15}$$

$$\bigwedge_{\tau \in P} \bigwedge_{t} y^{\tau}_{1,t} = \begin{cases} 1 \text{ if } p \in \tau(t) \\ 0 \text{ otherwise} \end{cases} \tag{8.16}$$

Constraint 8.16 encodes the valuation function $V(p, \tau[t:])$. We assign the same value to it in m.

Since for each positive trace $\tau, \tau \models \varphi$, we have $p \in \tau(0)$, and hence, $y^{\tau}_{1,0} = 1 > 0$, and the constraint system is satisfied.

Similarly for negative traces, constraint 8.15 changes to $y^{\tau}_{1,0} = 0$, and since $\tau \not\models \varphi$, we have $p \notin \tau(0)$, and the constraint system is satisfied.

Induction: Given the system is complete for all depths $\leq (d-1)$, we show it is complete for depth d.

We prove this for each top-level operator:

Assuming $\varphi = \psi \vee \chi$ — since $\mathcal{S} \models \varphi$, for each positive trace τ, we must have atleast one of $\tau \models \psi$ and $\tau \models \chi$, and for each negative trace τ', we have $\tau' \not\models \psi, \chi$.

Construct new samples $\mathcal{S}_1 = (P_1, N)$ and $\mathcal{S}_2 = (P_2, N)$ with $P_1 \subseteq P$ containing the positive traces which ψ satisfies, and P_2 for χ. By the induction hypothesis, we can find satisfying models for the constraint systems $\Phi^{\mathcal{S}_1}_{d-1}$ and $\Phi^{\mathcal{S}_2}_{d-1}$, say m_1 and m_2.

Using these, we construct a satisfying model for $\Phi^{\mathcal{S}}_{d}$, m. Assign to the $\{x_i\}$ the syntax tree encoding for φ. Through m_1, the values $\{y_i\}$ of ψ are known over $\mathcal{S}_1$. Without loss of generality, set the unknown values (over $P \setminus P_1$) to 0. Similarly for χ and P_2.

The constraint system for positive traces reduces again to

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \tag{8.17}$$

$$\bigwedge_{\tau \in P} \bigwedge_{t} y^{\tau}_{1,t} = \beta \cdot \frac{y^{\tau}_{m,t} + y^{\tau}_{m',t}}{2} \tag{8.18}$$

where nodes $1, m, m'$ are the root node and its left and right children respectively. The models m_1 and m_2 ensure that both the terms on the RHS for $y^{\tau}_{1,0}$ are non-negative, and for each trace, at least one is positive. Assigning the respective values to $y^{\tau}_{n,t}$ in m, since the sum of a non-negative, and a positive value is greater than 0, the system is satisfied.

For negative traces, we require $y^{\tau}_{1,0} = 0$. Clearly, the models of the subformulae ensure that both terms on the RHS for $y^{\tau}_{1,0}$ are zero. Hence, their sum is zero as well. Hence, the negative trace constraints are also satisfied under this m.

m is the required model.

Assuming $\varphi = \psi \wedge \chi$ — we must have $\tau \models \psi, \chi$, and for each negative trace τ', we have atleast one of $\tau' \not\models \psi$ and $\tau' \not\models \chi$.

The proof proceeds identically to the previous case, except with a partitioning of the negative traces. Similar to the sum-based constraint system for $\vee$, we have the product-based system for positive traces with the same notation as before

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \tag{8.19}$$

$$\bigwedge_{\tau \in P} \bigwedge_{t} y^{\tau}_{1,t} = \beta \cdot y^{\tau}_{m,t} \cdot y^{\tau}_{m',t} \, . \tag{8.20}$$

Again, the models of the subformulae ensure that both terms on the RHS for $y^{\tau}_{1,0}$ are positive, hence so is their product.

For negative traces, without loss of generality, we assign $\mathbf{1}$ to the unknown values over the complements $(N \setminus N_1, N \setminus N_2)$. We have

$$\bigwedge_{\tau \in N} y^{\tau}_{1,0} = 0 \tag{8.21}$$

$$\bigwedge_{\tau \in N} \bigwedge_{t} y^{\tau}_{1,t} = \beta \cdot y^{\tau}_{m,t} \cdot y^{\tau}_{m',t} \, , \tag{8.22}$$

and the models ensure at least one of the terms in the RHS for $y^{\tau'}_{1,0}$ is zero, and hence so is the product.

The constraint system is satisfied, and m is the required model.

Assuming $\varphi = \mathsf{G}\psi$ — we must have ψ satisfied at every point on every positive trace and not to be satisfied on some point on each negative trace, i.e.

$$\mathcal{S} \models \varphi = \mathsf{G}\psi \tag{8.23}$$

$$\forall \tau \in P \; \forall t < |\tau|, \;\; \tau[t:] \models \psi \, , \text{ and} \tag{8.24}$$

$$\forall \tau \in N \; \exists t < |\tau|, \;\; \tau[t:] \not\models \psi \, . \tag{8.25}$$

By taking all possible suffixes of traces in P, construct P^*, and by resolving the existential quantifier, collect relevant (non-satisfying) suffixes of N as N^*. Generate the new sample $\mathcal{S}^* = (P^*, N^*)$.

By construction, $\mathcal{S}^* \models \psi$. Using the induction hypothesis, there exists a model m^* satisfying the constraint system $\Phi_{d-1}^{\mathcal{S}^*}$. We use m^* to construct a model m for $\Phi_d^{\mathcal{S}}$.

Setting the values for $\{x_i\}$ as encoding φ, we are left with the reduced constraint system

$$\bigwedge_{\tau \in P} y_{1,0}^{\tau} > 0 \,, \bigwedge_{\tau' \in N} y_{1,0}^{\tau'} = 0 \tag{8.26}$$

(8.27)

$$\bigwedge_{\tau \in P \cup N} \bigwedge_{1 \le t \le |\tau|}$$

$$\left[\left[y_{1,t}^{\tau} = \beta \cdot \sum_{t \le t' < |\tau|} \alpha^{t'-t} \cdot y_{m,t'}^{\tau} \right] \wedge \left[\bigwedge_{t \le t' < |\tau|} (y_{m,t'}^{\tau} > 0) \right]\right] \vee y_{1,t}^{\tau} = 0 \,. \tag{8.28}$$

For positive traces, from m^*, we have an assignment for each of the terms in the sum for $y_{1,0}^{\tau}$ being positive (we posed them as positive traces in P^*). Hence, their sum is positive too, and adding these inferred assignments for $y_{1,t}$, the constraints are satisfied.

For the negative traces, adding the assignments from m^*, clearly the constraints $y_{m,t} > 0$ cannot be satisfied for each parameter in the sum for $y_{1,0}$, by construction. Thus, $y_{1,0} = 0$.

m is the required model.

Assuming $\varphi = \mathsf{F}\psi$ — the proof proceeds identically to the previous case, with existential quantification in the positive case, and suffix construction in the negative, since these are dual operators.

The induction is complete.

□

Theorem 15 (Soundness)**.** *Given a sample $\mathcal{S}$, if there exists a model m satisfying the constraint system (8.5 - 8.14), i.e. $m \models \Phi_d^{\mathcal{S}}$, then there exists an LTL formula φ of depth d such that the parse tree (x_i) encodes the formula φ and for each node i and position t in each trace τ, $y_{i,t} = V(\varphi_i, \tau[t:])$, where φ_i is the formula encoded by the subtree with root at node i.*

Proof. We induct on the depth parameter d of the constraint system.

Base case: d = 0.

Since there is only a single leaf node, the model must have one of the propositional variable mapped to it, say p. With this, the constraint system reduces to

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \,, \bigwedge_{\tau \in N} y^{\tau}_{1,0} = 0 \tag{8.29}$$

$$\bigwedge_{\tau \in P \cup N} \bigwedge_{t} y^{\tau}_{1,t} = \begin{cases} 1 \text{ if } p \in \tau(t) \\ 0 \text{ otherwise} \end{cases} \tag{8.30}$$

Clearly, due to the constraints, for each trace $y^{\tau}_{1,t} = V(p, \tau[t:])$. In particular, $y^{\tau}_{1,0} = 1$ for $\tau \in P$, and with the constraints, we must have $p \in \tau(0)$, which, by the LTL semantics implies $\tau \models p$.

Similarly for negative traces $\tau' \in N$ we find $\tau' \not\models p$.

Combining these, we have $\mathcal{S} \models p$, and hence $\varphi = p$ is the required LTL formula of depth 0.

Induction: Given the system is sound for all depths $\leq (d-1)$, we show it is sound for depth d.

We prove this for each top-level operator in the constraint encoding:

The top-level operator is obtained by checking which of $\{x_{1,O}\}$ holds under m. The constraint system ensures there is exactly one. We assume the $\{x_{i,O}\}$ have been processed to obtain the encoded formula, say φ.

Our task is now to show that $\mathcal{S} \models \varphi$.

Assuming $\varphi = \psi \vee \chi$ — We have the reduced constraints

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \,, \bigwedge_{\tau' \in N} y^{\tau'}_{1,0} = 0 \tag{8.31}$$

$$\bigwedge_{\tau \in P \cup N} \bigwedge_{t} y^{\tau}_{1,t} = \beta \cdot \frac{y^{\tau}_{m,t} + y^{\tau}_{m',t}}{2} \tag{8.32}$$

For positive traces, in particular, the constraints on the root node $y^{\tau}_{1,0}$ imply at least one of $y^{\tau}_{m,0} > 0$ and $y^{\tau}_{m',0} > 0$ must hold, where m, m' are the left and right children of the root-node 1, respectively. And for negative traces, we have $y^{\tau'}_{m,0} = y^{\tau'}_{m',0} = 0$.

In conjunction with the rest of the constraints, this is precisely two copies of the constraint system $\Phi^{\mathcal{S}}_{d-1}$.

In either case, i.e., for the left or right child being non-zero, the induction hypothesis implies that the subformula holds over the sample $\mathcal{S}$, i.e. $\mathcal{S} \models \varphi_m$ (or φ'_m). And since this is a subformula of φ, by introduction of $\vee$, $\mathcal{S} \models \varphi$.

Assuming $\varphi = \psi \wedge \chi$ — Proof proceeds similar to the previous case, with a case-bifurcation in the negative traces instead of the positive ones.

Assuming $\varphi = \mathsf{G}\psi$ — Again, we have the reduced constraint system

$$\bigwedge_{\tau \in P} y^{\tau}_{1,0} > 0 \, , \bigwedge_{\tau' \in N} y^{\tau'}_{1,0} = 0 \tag{8.33}$$

$$\bigwedge_{\tau \in P \cup N} \bigwedge_{1 \leq t \leq |\tau|}$$

$$\left[\left[y^{\tau}_{1,t} = \beta \cdot \sum_{t \leq t' < |\tau|} \alpha^{t'-t} \cdot y^{\tau}_{m,t'} \right] \wedge \left[\bigwedge_{t \leq t' < |\tau|} (y^{\tau}_{m,t'} > 0) \right]\right] \vee y^{\tau}_{1,t} = 0 \, , \tag{8.34}$$

First, in the case of the positive traces, since $y^{\tau}_{1,0} > 0$, we must have the first clause of the score constraint holding (since the second clause $y^{\tau}_{1,0} = 0$ evaluates to false). Thus, for each point t in the trace, we must have $y_{m,t} > 0$ with notation as before.

For negative traces, we can reduce the constraints on $y^{\tau}_{1,0}$ further to obtain

$$\bigwedge_{\tau \in N} \bigvee_{t} y_{m,t} = 0 \, , \tag{8.35}$$

implying there exists a position for which $y_{m,t} = 0$.

As before, taking all possible suffixes of traces in P, construct P^*, and by resolving the existential quantifier above, collect relevant (non-satisfying) suffixes of N as N^*. Generate the new sample $\mathcal{S}^* = (P^*, N^*)$. Combined with the constraints above, we have a satisfying assignment to $\Phi^{\mathcal{S}^*}_{d-1}$ by restricting m as needed to the smaller system.

Thus, by the induction hypothesis, for each trace $\tau \in P$, ψ holds on every suffix of τ. By the semantics of the operator G, we have $\tau \models \mathsf{G}\psi = \varphi$.

Further, for each negative trace $\tau' \in N$, ψ does not hold on some suffix of τ. Again, by the semantics of G, $\tau \not\models \mathsf{G}\psi = \varphi$.

Thus, $\mathcal{S} \models \varphi$ as required.

Assuming $\varphi = \mathsf{F}\psi$ — Proof proceeds similar to the previous case, with existential quantification on the positive traces instead of the negative.

The induction is complete.

□

Corollary 16 (Valuation Equivalence)**.** *The valuation semantics are equivalent to the LTL semantics. For any LTL formula φ and trace τ, $\tau \models \varphi$ iff $V(\varphi, \tau) > 0$.*

Proof. Forward Direction $\Rightarrow$:

Given an LTL formula φ and trace τ such that $\tau \models \varphi$, we can construct a satisfying assignment to the constraint system $\Phi_d^{\mathcal{S}}$ by Theorem 14 where $\mathcal{S} = (\{\tau\}, \emptyset)$ and d is the depth of φ. Further, the theorem guarantees $y_{1,0}^{\tau} = V(\varphi, \tau) > 0$. This is the required condition.

Backward Direction $\Leftarrow$:

Given an LTL formula φ and trace τ such that $V(\varphi, \tau) > 0$, construct the constraint system $\Phi_d^{\mathcal{S}}$ as before. Iteratively assign to each $y_{i,t}^{\tau}$ the value $V(\varphi_i, \tau[t:])$, where φ_i is the subformula at node i in the parse tree. After the assignment with constraints, $y_{1,0}^{\tau} = V(\varphi, \tau) > 0$ as given, satisfying the constraint. By Theorem 15, $\tau \models \varphi$ as required.

□

8.4.2 Optimized Pattern Matching

We now present a variation of Algorithm 6 where the input consists of a sample along with a user-provided formula pattern, where the propositional variables are unknown. This approach is in the spirit of [182], where a formula template is used instead of just a depth as in Algorithm 6. We generate a static syntax tree by parsing the given pattern, with pattern propositional variables becoming the leaves. In addition to the constraints discussed in Section 6, constraint (8.36) ensures the mapping of the pattern variables to exactly one variable in the given sample, where Var represents the set of variables in the given pattern and $m_{x,p}$ stands for the mapping of pattern variable x to the sample variable p.

$$\varphi_{\mathcal{P}}^{Var} = \left[\bigwedge_{x \in Var} \bigvee_{p \in \mathcal{P}} m_{x,p} \right] \wedge \left[\bigwedge_{\substack{x \in Var \\ p \neq p' \in \mathcal{P}}} \neg m_{x,p} \vee \neg m_{x,p'} \right] \tag{8.36}$$

The constraint (8.36) specifies that each pattern variable x is mapped to a sample variable, and no pattern variable is mapped to two sample variables. For an example of how this

Algorithm 7 Compositional Ranking

procedure CompRank($\mathcal{S} = (P, N), depth$)
▷ Returns list of satisfying formulae $\mathcal{F}$ sorted by score
    curr_depth $\leftarrow 0$, $\mathcal{F} \leftarrow$ {literals in $\mathcal{S}$}
    used $\leftarrow \{\}$, unary $\leftarrow \{\mathsf{G}, \mathsf{F}\}$, binary $\leftarrow \{\wedge, \vee\}$
    while *curr_depth* $\neq$ *depth* **do**
        for all f $\in \mathcal{F}$ **do**
            if $\mathsf{F}(f)$ does not hold on any trace **then**
▷ F-Check
                $\mathcal{F} \leftarrow \mathcal{F} \setminus \{\mathrm{f}\}$
        used $\leftarrow$ used $\cup\, \mathcal{F}$
        if curr_depth $\neq$ depth - 1 **then**
            $\mathcal{F} \leftarrow \bigcup_{T \in \text{unary}} T(\text{used}) \cup \bigcup_{T \in \text{binary}} T(\text{used}, \text{used})$
        curr_depth $\leftarrow$ curr_depth + 1
    scores $\leftarrow \{(f, V(f, P, N)) | f \in \mathcal{F}\}$
    return sort(scores)
▷ sort list w.r.t. scores

approach and the one defined in the previous subsection are combined into a hybrid approach, which is presented as follows.

8.4.3 Hybrid Pattern Matching

The algorithms defined in sections 8.4.1 and 8.4.2 suffer from a common drawback, though at different ends of the spectrum. In the first, we work with increasing depth to find the optimal formula, and constraint sizes may grow quickly as a result. In the second, we start with a formula template, and many formulae are not considered since we are guided by the template pattern. This makes this approach *insufficiently* expressive in comparison with constrained system optimization.

To remedy this, we introduce a middle ground, where, instead of attempting to learn formulae from scratch or from explicit patterns, we learn *subformulae* within some pattern. A subformulae argument $\varphi(d)$ with d being a prescribed maximum depth for the subtree is provided as part of the pattern, parsed into the tree as an abstracted empty formula with constraints constructed for the specified nodes explicitly, and for the subformulae recursively in the manner as described in Algorithm 8. In Figure 8.2, we show an example

Algorithm 8 Computing the optimal formula given a partial pattern

procedure HYBRIDPATTERN($\mathcal{S} = (P, N)$, *pattern*)
▷ Returns optimal formula fitting a pattern
    parse pattern
    construct $\varphi_{\mathcal{P}}^{Var}$ for propositional patterns in tree
    construct $\varphi_{\mathrm{n}}^{ST}$ for every subformula pattern $\varphi(n)$ in the tree
    constraint $\Phi \leftarrow \varphi_{\mathcal{P}}^{Var} \wedge \bigwedge \varphi_{\mathrm{n}}^{ST}$
    maximize $\min(\{y_{1,0}^{\tau} \mid \tau \in P\})$ with Φ as constraint
    if optimization succeeds with model m **then**
▷ satisfiable
        construct formula tree from m
        return optimized formula tree
    else
        return UNSAT

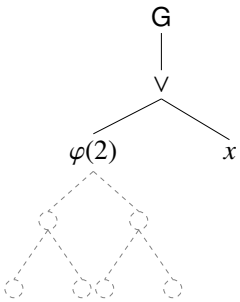


Figure 8.2: Subtree for hybrid pattern matching.

of the hybrid pattern $\mathsf{G}(\varphi(2) \vee x)$, where $\varphi(2)$ is an unknown formula of depth ≤ 2 and x is an unknown proposition.

8.4.4 Compositional Ranking

We describe an alternative greedy search for an optimal formula, which bypasses constraint solving and optimizations, by pruning the search space of formulae. We begin by enumerating all formulae of depth zero, i.e., all literals in our system as obtained after parsing input traces. We consider all compositions of these literals with the operators present. After enumerating the literals, we perform an "F-check" : for any φ, the F-check tests whether, in any input sample, $\mathsf{F}\varphi$ holds. If a formula passes an F-check, it is retained to produce formulae of higher depth, else it is removed (Algorithm 7). We have implemented

more custom options to priorities certain parts of the search space (Section 8.4.5).

8.4.5 Prioritize Variables.

Finally, we have added one more heuristic in our implementation. In case there is a large set of events and we want to bias the focus of our search towards certain letters in the traces that do not occur very often, we may adjust the value $V(p, w)$ assigned to each propositional variable p. In our default scheme, we assign $V(p, w) = 1$ if $w(1)$ contains p. A user may assign a value greater than 1 to variables p that are desirable and assign less than 1 for the variables p that are not. This allows for mining specifications pertaining to the prioritized variables in cases where several competing well ranked specifications are present. Let π be the map from the propositional variables $\mathcal{P}$ to their priority. We replace equation (8.9) by the following formula where we return score $\pi(p)$ instead of 1.

$$\bigwedge_{1\leq i\leq N}\bigwedge_{p\in\mathcal{P}} \quad x_{i,p} \rightarrow \left[\bigwedge_{1\leq t\leq|\tau|} y^{\tau}_{i,t} = \begin{cases}\pi(p) & \text{if } p\in\tau(t)\\ 0 & \text{if } p\notin\tau(t)\end{cases}\right] \tag{8.37}$$

In the Dining Philosophers problem (see Section 8.6.3), we may wish to verify individually whether the properties are being satisfied for a single philosopher (thread). By giving a higher weight to the properties of this philosopher, we can guide the tool to learn the relevant properties and verify them. This can be expanded to studying specific applications or threads in varied noisy data where the target of interest is either known apriori or is inferred from preliminary unguided results.

The suggested variations and their results indicate that our method is viable to be adapted to an application at hand, where we want to bias our ranking to give preference to a desired class of formulae.

8.5 Computing ILE via optimal policies

We will evaluate the generated formulae via ILE, which needs the optimal policy computed for a given NMRDP and the formula. In this section, we will present our method for computing the optimal policy, and its use for computing ILE.

We first convert an LTL formula into a Rabin automaton. We then compute the intersection of the NMRDP and the Rabin automaton by constructing their product. Next, we apply the value iteration algorithm [183] to the resulting product automaton in order to

compute the optimal policy and the reward signal for each state. Using the reward signal, we will compute the ILE as described in Section 8.1. The details are described below.

Since we are learning LTL formulae, the corresponding structures are deterministic Rabin automata, though other choices of automata are possible.

Definition 11 (Deterministic Rabin Automaton (DRA))**.** *A DRA $\mathcal{A}$ is a tuple $(\Sigma, Q, q_0, \delta, F)$, where Σ is a finite* alphabet*, Q is a finite set of* states*, $q_0 \in Q$ is the* initial state*, $\delta : Q \times \Sigma \to Q$ is the transition function, and $F = \{(B_i, G_i) \in 2^Q \times 2^Q\}$ is the Rabin condition.*

By taking the DRA representation of a regular reward signal and the underlying NMRDP, a product Markov Decision Process (MDP) can be computed. In particular, the reward signal of the MDP is defined only over the current state and action, thereby enabling the adoption of conventional Markovian solutions, such as value iteration [183]. In order to define a reward signal for the product MDP, we must ensure that, when the acceptance condition of the underlying DRA is satisfied, the agent is rewarded. For this, we must compute maximal sub-MDPs known as maximal end components (MECs). Formally, for a DRA with acceptance condition $F = (G, B)$, a MEC $E = (S^E, A^E)$ of the product MDP $\mathcal{M} \times \mathcal{A}$ is accepting if $S^E \cap (S \times B) = \emptyset$ and $S^E \cap (S \times G) \neq \emptyset$ for some $(B, G) \in F$. The reward signal of the product MDP is defined whenever states with labels in G are entered.

Definition 12 (Product LMDP)**.** *Given NMRDP $\mathcal{M} = (S, s_0, A, T, R, \Sigma, L)$, DRA $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$, the* product *LMDP $M \times A$ is the tuple $(S^\times, (s_0, q_0), A, T^\times, R^\times, L, \Sigma)$, where $S^\times = S \times Q$; $T^\times(s', q'|s, q, a)$ equals $T(s'|s, a)$ if $q' = \delta(q, L(s'))$ and is 0 otherwise; and $R^\times(s, q, a, s', q') = 1$ if $q' \in G_i$ for some i and is 0 otherwise.*

For a product LMDP $(S^\times, (s_0, q_0), A, T^\times, R^\times, L^\times, \Sigma)$, Equation (8.4) reduces to the equation below [183], where $r = R(s, q, \pi(s, q), s', q')$.

$$V_\pi(s, q) = \sum_{(s', q') \in S^\times} T(s', q'|s, q, \pi(s, q)) \left[r + \gamma V_\pi(s', q') \right]$$

Since we are now reasoning about a Markovian reward over the product, the optimal value for a state $V^*(s, q) = \max_\pi V_\pi(s, q)$ can be computed using value iteration [183]. In particular, we can initialize $V(s, q) = 0$ for all $q \notin G_i$ for some $G_i \in F$ and $V(s, q) = 1$ for all $q \in F$. We can then iteratively refine these value functions by applying the following equation until convergence, where $r = R(s, q, a, s', q')$.

$$V(s, q) = \max_a \sum_{(s', q') \in S^\times} T(s', q'|s, q, a) \left[r + \gamma V(s', q') \right]$$

These equations converge to $V^*(s,q)$ for all $(s,q) \in S^\times$. The optimal policy can then be extracted as

$$\pi^*(s,q) = \operatorname*{argmax}_a \sum_{(s',q') \in S^\times} T(s',q'|s,q,a)\left[r + \gamma V^*(s',q')\right] \tag{8.38}$$

8.6 Experiments

We have implemented the preceding algorithms in a tool called QuantLearn, which is implemented in C++ and is publicly available at https://github.com/sankalpgambhir/quantlearn. Our implementation uses the SMT solver Z3 [96] to perform the required optimizations, and all queries to Z3 are quantifier-free. We compare QuantLearn against the existing tools Traces2LTL [97] and Texada [184]. We answer the following three research questions empirically – **RQ1:** Does our tool generalize well, as measured by the mean inverse learning error (MeanILE)?, **RQ2:** How does our approach compare with Traces2LTL and Texada in terms of correctness and efficiency?, and **RQ3:** What is the impact of our approach on the well-known dining philosophers problem? All experiments were conducted on a 64-bit Linux system equipped with an AMD Renoir Ryzen 5 (4500U) laptop CPU.

8.6.1 Benchmarks

For experiments, we chose three different benchmarks. One is a set of traces sampled from the grid-world environment running under OpenAI Gym. After generating a randomised 10×10 grid environment labelled with propositional variables, we uniformly sample the grid, taking actions that are compatible with an input automaton. This ensures that the generated traces satisfy a given formula. We use the same LTL properties as the previous case. We allow the MDP to randomly simulate for at least 100 steps, after which we wait for it to reach an accepting state. Through this method, we generated traces of length varying between 100 and 150, with 1000 positive and 1000 negative traces for each formula, amounting to a total trace length of at least 10^5 across all positive and negative inputs. However, for constraint system optimisation and Traces2LTL, due to timeouts, a smaller subset of the traces was randomly selected.

Second, we systematically generate traces of different sizes to compare the performance and expressiveness of the different methods. After generating uniformly random traces, we add noise to each point in the trace in the form of two extraneous propositional variables with probability p_{noise} each. For these experiments, we maintained $p_{\text{noise}} = 0.25$. We

consider LTL formulae encoding some popular requirements. Third, we also considered the Dining Philosophers Problem [185] from Texada tests [184], having 5 philosophers.

8.6.2 Experimental Setup

Due to the fundamentally different mechanisms employed by the competing tools and algorithms, an apples-to-apples comparison is not possible. In QuantLearn, the optimization procedure takes as input a set of positive traces, a (possibly empty) set of negative traces, and a formula template. The template may simply be $\varphi(d)$, corresponding to a search depth d with no specification, or a combination of templates. The procedure returns a formula that maximizes the score according to our scoring scheme. The compositional ranking component takes the trace samples along with a maximum search depth as input and returns a list of all satisfying formulae in the search space, sorted by score. In our experiments, we use a discount factor $\alpha = e^{-1}$ and a decay factor $\beta = 0.8$, which is applied each time deeper formulae are constructed, in order to bias the ranking toward simpler formulae. In contrast, Texada requires a complete template specification as input and does not utilize negative trace samples. Traces2LTL accepts both positive and negative samples but restricts itself to generating any satisfying LTL formulae.

8.6.3 Results

We conduct three experiments to answer the three research questions described previously. First, on grid-world traces, we compute the MeanILE with respect to the learned formula obtained using different tools. A lower MeanILE indicates better generalization of the learned formula. Second, on synthetically generated traces, we learn LTL formulae and compare them against the ground-truth formulae to evaluate accuracy and also compare the efficiency. Third, from an application perspective, we consider the well-known dining philosophers problem [185], to check well studied properties – liveness, mutual exclusion, and deadlock-freedom.

RQ1: Non-Markovian IRL (Comparison Using MeanILE)

We apply learning techniques to generate a reward function over an MDP defined as a grid world to obtain an NMRDP, similar to the example in Figure 8.1(a). After generating a randomized 10×10 grid environment labelled with propositional variables, we uniformly sample the grid, taking actions compatible with an input automaton. This ensures that the generated traces satisfy a given formula. We use the same LTL properties as the

	Constraint Optimization	Comp. Ranking	Traces2LTL
Mean ILE	0.031	0.037	0.112
Input size	10^3	2×10^5	10^3

Table 8.1: Comparison of MeanILE with different techniques.

previous case. We allow the MDP to randomly simulate for at least 100 steps, after which we wait for it to reach an accepting state. Through this method, we generated traces of length varying between 100 and 150, with 1000 positive and 1000 negative traces for each formula, amounting to a total trace length of at least 10^5 across all positive and negative inputs. However, for constraint system optimization and Traces2LTL [97], due to timeouts, a smaller subset was randomly selected from the traces. For our experiments, given an NMRDP M in the form of a gridworld, we can compute the optimal policy using Equation 8.38 for three different DRA. The first DRA objective is what we are trying to learn. We will denote the optimal policy here as π^*_{true} computed on $M \times A_{\text{true}}$, where A_{true} is the DRA representation of the LTL objective we are trying to learn. Then, we have the policy π^*_{QL} computed on $M \times A_{\text{QL}}$, where A_{QL} is the DRA learnt using QuantLearn. Finally, we have the policy π^*_{T2L} computed on $M \times A_{\text{T2L}}$, where A_{T2L} is the DRA learnt using Traces2LTL [97]. We will take these three policies to generate our results in the form of the inverse learning error (ILE). We compute these value functions using uniformly random sampling of trajectories from every state in the NMRDP. We can then take a simple ratio (MeanILE) of the number of trajectories satisfied by A_{true}, and divide it by the total number of trajectories, and report an average over multiple runs and inputs. In particular, we will compute two ILE values, comparing $\|V_{\pi^*_{\text{true}}} - V_{\pi^*_{\text{QL}}}\|_2$ and $\|V_{\pi^*_{\text{true}}} - V_{\pi^*_{\text{T2L}}}\|_2$. Our experiments demonstrate that the former is smaller than the latter, as shown in Table 8.1, thereby providing evidence that our approach generalizes better for non-Markovian IRL than a competing one adapted to the IRL.

RQ2: Performance Analysis of the Tools

We report the average runtimes for our approach with and without compositional ranking and compare them to those of Texada. In Figure 8.3, we show the runtimes with different patterns of user specification over the different lengths of traces for constraint system optimization and optimized pattern matching. In Figure 8.4, we show runtimes for synthetic

Property	Formulae
Absence	$\mathsf{G}\neg p$, $\mathsf{G}(q \rightarrow \mathsf{G}(\neg p))$
Response	$\mathsf{G}(p \rightarrow \mathsf{F}s)$, $\mathsf{G}(q \rightarrow \mathsf{G}(p \rightarrow \mathsf{F}s))$
Existence	$\mathsf{F}p$, $\mathsf{G}(\neg p \vee \mathsf{F}(p \wedge \mathsf{F}q))$
Universality	$\mathsf{G}p$, $\mathsf{G}(p \rightarrow \mathsf{G}q)$

Table 8.2: LTL Formulae to generate synthetic traces

traces generated from the common formulae in Table 8.2, with compositional ranking [97, 186].

The length of the individual traces varies from 5 to 10,000. In Figure 8.5 we show runtimes for the generated traces with Texada. The comparison with Texada is difficult because it requires the a complete formula template as input while we do not. The tests are thus performed with complete formula template as input, with which Texada outputs a list of possible formulae with propositions substituted in. In our experiments, we found that Texada and compositional ranking perform comparably despite the fact that our approach requires no pattern input.

RQ3: Mining Formulae from the traces of Dining Philosophers

The dining philosophers problem [185] is a widely used example of a control problem in distributed systems and has become an important benchmark for testing the expressiveness of concurrent languages and resource allocation strategies. We consider the problem with five philosophers `p1`, `p2`, `p3`, `p4`, `p5` sitting at a round table. They are being served food, with a fork placed between each pair. Each philosopher proceeds to think till they are hungry, after which they attempt to pick up the forks on both of their sides, eating till they are full, but only when forks on both sides are available. After they are done eating, they put the forks down back on to either of their sides, and continue thinking. The goal is to establish *lockout-freedom*, i.e., each hungry philosopher is eventually able to eat. We use QuantLearn to mine LTL formulae from a trace of size 250 from Texada tests [184]. We searched several mined properties using different templates presented in Table 8.3.

- When we gave the pattern $\mathsf{G}(\varphi(1))$ (invariant, depth 1) to QuantLearn and required it to learn a property, we obtained the property GF`p1 is thinking`. The mined property acts as a verification of the liveness of the system. QuantLearn took 12

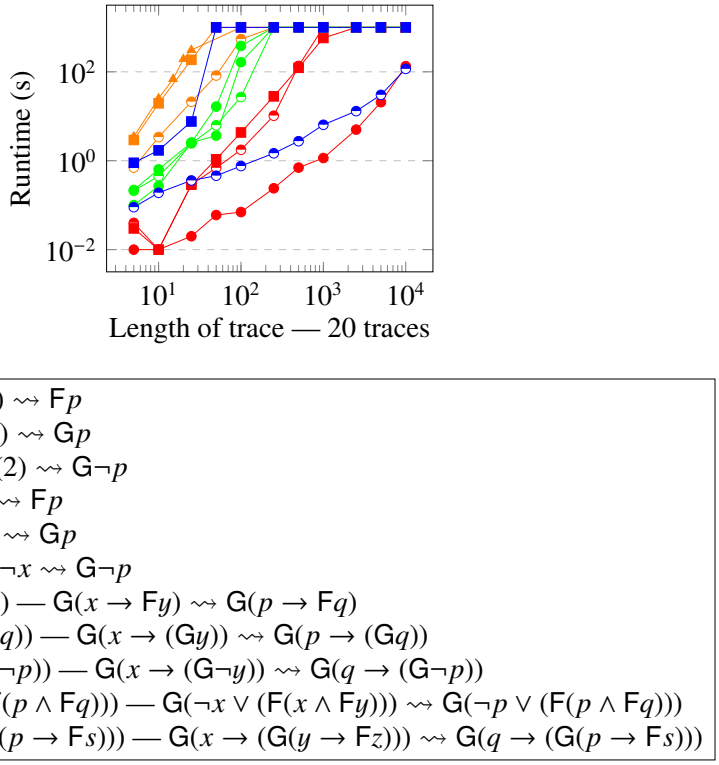


Figure 8.3: Runtime for traces generated for formulae, full and partial pattern specification. In the legend, φ — *pattern* $\rightsquigarrow \psi$ represents the following : φ is the formula used to generate the traces; *pattern* is the input to QuantLearn and ψ is the learned output formula.

seconds to find the property. While this provides a general fact about the system, building upon this result, we can guide the tool to find more relevant properties for higher depths.

- When we gave the pattern $\mathsf{G}\ (x \rightarrow \varphi(1))$ to QuantLearn and required it to learn a property, we obtained the property $\mathsf{G}\ ((\texttt{p4 is eating}) \rightarrow \neg(\texttt{p3 is eating}))$. The mined property illustrates that adjacent philosophers cannot acquire forks at the same time, ensuring that our *lock*, the availability of forks does indeed prevent philosophers from eating. QuantLearn took 72 seconds to find the property.

- When we gave the pattern $\mathsf{G}\ (x \rightarrow \mathsf{F}\ y \wedge \mathsf{F}\ z)$ to QuantLearn and required it to learn a property, we obtained $\mathsf{G}\ ((\texttt{p1 is hungry}) \rightarrow (\mathsf{F}\ ((\texttt{p1 is eating}) \wedge \mathsf{F}\ (\texttt{p1 is thinking}))))$. This property is a richer demonstration of deadlock freedom for philosopher 1, ensuring that they both enter and exit their *critical section*, i.e., the `eating` state. QuantLearn took 165 seconds to find the property.

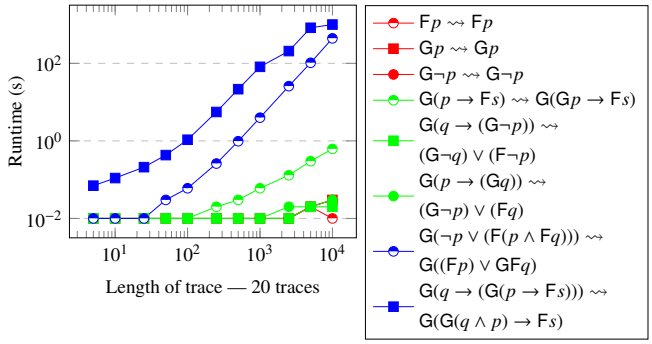


Figure 8.4: Runtime for traces generated for formulae with compositional ranking, and only a depth as input. In the legend, $\varphi \rightsquigarrow \psi$ indicates the traces in the sample were generated using φ, and ψ is the learnt formula.

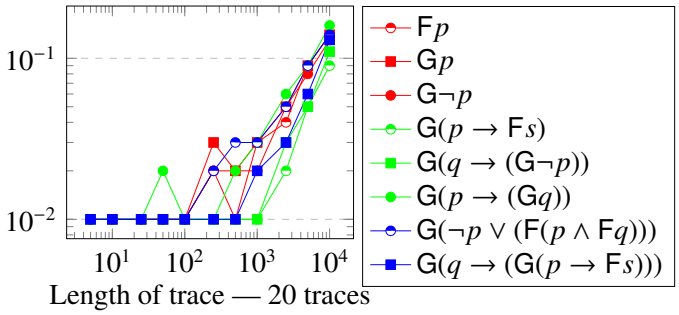


Figure 8.5: Runtime for traces generated for formulae with Texada [184], with complete pattern specification.

by observing their runs. Using patterns, QuantLearn can be guided to learn properties of specific interest.

We observe that the MeanILE achieved by QuantLearn is lower than that of Traces2LTL, which indicates better generalization. Furthermore, even with 25% noise in the synthetic data, QuantLearn is able to generate LTL formulae that match the ground-truth formulae, demonstrating its robustness to noise. In terms of efficiency, Traces2LTL and Texada sometimes outperform our optimization-based approach by a margin. This is expected, as our method optimizes for the best-scoring formula under a given template (optional), whereas Traces2LTL generates any satisfiable formula and Texada requires a fully specified template. However, our compositional ranking algorithm performs comparably to both tools while achieving better generalization of the learned LTL formulae.

Input	Output	Interpretation
$\mathsf{G}(\varphi(1))$	GF`p1 is thinking`	Liveness property
$\mathsf{G}\,(x \rightarrow \varphi(1))$	$\mathsf{G}\,($(`p4 is eating`) $\rightarrow \neg$(`p3 is eating`))	Mutual exclusion
$\mathsf{G}\,(x \rightarrow \mathsf{F}\,y \wedge \mathsf{F}\,z)$	$\mathsf{G}\,($(`p1 is hungry`) $\rightarrow$ ($\mathsf{F}\,($(`p1 is eating`) $\wedge\, \mathsf{F}\,$(`p1 is thinking`))))	Deadlock freedom

Table 8.3: Results of running QuantLearn on Dining Philosophers traces

8.7 Related Work

The paper [187] presents a detailed landscape of the algorithms, challenges, and the state-of-the-art in IRL. The work [80] formalized the first computational solution to IRL based on linear programming to demonstrate the effectiveness of IRL. Among more recent efforts, IRL is solved using techniques from entropy optimization [188, 189, 190], maximum likelihood estimation [191, 192], and reformulating the problem as a classification task [193]. These techniques have shown much promise. They have been applied successfully to problems such as maneuvering remote-controlled helicopters [194] and Atari games [195].

Grammatical inference is a related area [196] concerned with learning grammars and their automata representations [197, 198]. Active techniques rely on querying the system under learning to guide the inference process, whereas passive grammatical inference leverages a static set of trace behavior without making further queries for additional data. The former is exemplified by $L^\star$ algorithm [199], whereas the latter generally relies on state-merging procedures and can be used to learn probabilistic automata [200], MDPs in the context of model checking [201], timed automata [202], and regular decision processes [203]. There is a growing literature on the application of grammatical inference to RL. This typically entails the learning of weighted DFAs, known as reward machines [204, 205, 206].

However, the application of grammatical inference for IRL has not been explored. Indeed, the foregoing RL methods learn automata from traces of the underlying decision process with a given reward signal, not from traces of an expert policy over an unknown and unobservable environment. The problem of learning LTL formulae from traces is a form of grammatical inference that has been well-studied. Two of the methods most related to our own are presented in [182] and [97]. The focus of [97] is to produce the minimal formula which is consistent with a rational sample represented as a lasso. The problem of matching formulae with traces is encoded as a constraint system, and a satisfying assignment yields the learned property. However, requiring the inputs to be lassos significantly restricts the

application to real scenarios. The method in [182] requires a user-defined input template of the LTL formula which they would like to satisfy and outputs all possible propositional substitutions consistent with the sample.

We seek to combine some of these ideas and try to eliminate their restrictions with a method that may be used on finite traces obtained from real systems, and output results relevant to the user. To allow the algorithms to quantitatively distinguish formulae, we supplement them with the idea of a ranking scheme as a parameter to the methods. Our ranking scheme quantitatively scores each formula against a finite word. It expands on intuitive ideas used to formulate distances in regular language spaces [207, 208]. The suggested scheme assigns a formula a high score if it expresses most features of the word. A formula can score well on a word if it provides longer evidence of validity with respect to larger parts of the word. For example, $\mathsf{G}a$ will score more on the word $aaaa$ than on the word aa, since it contains more evidence for the word to have been a prefix of $a^\omega \in L(\mathsf{G}a)$. Furthermore, our suggested scoring scheme encourages simpler formulae over complex ones in equivalence classes; that is, from an equivalence class of LTL formulae, the ones with smaller parse trees will be preferred.

The papers closest to our work are [182] and [97]. The focus of [97] is to produce the minimal formula which is consistent with a rational sample irrespective of the expressiveness, while [182] requires a user-defined input template of the LTL formula which they would like to satisfy. Both work with infinite traces: [97] takes rational traces in the form uv^ω where u, v are typically words of length ~ 10, while [182] mines specifications from finite traces, and appends them with an infinite sequence of terminal events. [209] considers finite traces, and develops an LTL checker which takes an event log and an LTL property and verifies if the observed behaviour matches some bad behaviour. The papers [210, 211] look at the application of monitoring the execution of Java programs, and check LTL formulae on finite traces of these programs. In [212], the authors focus on mining quantified temporal rules which help in establishing data flow analysis between variables in a program, while in [213], software bugs are exposed using a mining algorithm, especially for control flow paths. [214] looks at process mining in the context of workflow management. The tool PISA [215] is developed to extract the performance matrix from workflow logs, while a declarative language is developed to formulate workflow-log properties in [216]. Tool Synoptic [217], on the other hand, follows a different approach and takes event logs and regular expressions as input and produces a model that satisfies a temporal

invariant which has been mined from the trace.

8.8 Conclusion

In this chapter, we presented a novel approach to learning and explaining the reward function of a learned policy. The policy is provided in the form of demonstrations, which include both desirable and undesirable traces. From these traces, we learn an LTL formula that captures the behavior of desirable traces while excluding the undesirable ones. The explainability of the learned policy naturally emerges from the interpretability of the resulting LTL formulae. Our novel scheme quantitatively evaluates LTL formulae. The schema is designed such that the score assigned to a word is proportional to how well it represents the formula. Consequently, words that are "good representatives" receive higher scores than those that merely satisfy the formula and are considered "poor representatives."

Chapter 9

Conclusion

In this thesis, we have explored several aspects of neural network verification and explainability. At the start of our research, we observed that most existing techniques for robustness verification of neural networks are either incomplete or achieve completeness by splitting non-linear activation units. However, the splitting in these methods is typically guided by heuristics that do not guarantee the elimination of spurious counterexamples. To address this limitation, we proposed a counterexample-guided refinement approach that systematically selects activation units for splitting based on the analysis of counterexamples, thereby ensuring the removal of spurious counterexamples. Furthermore, we explored the limitations of the local robustness property in identifying false positives, which led us to propose new robustness properties based on classification confidence. Each property defines distinct postconditions that are difficult to handle individually. To address this challenge, we introduced a unifying framework that transforms arbitrary properties into a set of layers appended to the underlying neural network. This framework enables the verification of diverse properties in a unified manner, independent of their specific formulations or the verifier employed. We also explored the explainability aspect of machine learning systems. We investigated this problem in the context of reinforcement learning. Specifically, we proposed a method for learning reward functions in inverse reinforcement learning, with a focus on enhancing both the explainability and expressiveness of the learned reward functions.

We introduced two variants of a counterexample-guided refinement technique to improve the robustness verification of neural networks. Our empirical results show that these techniques are orthogonal to state-of-the-art approaches. Currently, we use an MILP-based

optimizer for MAXSAT queries, which may not scale well on large neural networks. To address this, we plan to explore gradient-based optimizers and integrate our techniques with solvers such as $\alpha\beta$-CROWN, which scale efficiently on GPUs.

Next, we discussed true and false positives in the context of local robustness using oracles. We used an ensemble of classifiers as an approximation of the oracle and analyzed the robustness of neural networks with respect to this ensemble. Our results show that the standard local robustness property can report many false positives under this oracle. To mitigate this, we proposed a new ensemble-guided local robustness property that helps reduce false positives. It would be interesting to use false positives as part of a counterexample abstraction-refinement loop to further improve verification engines.

We also introduced variants of properties that incorporate the sensitivity of network decisions, making them more semantic and useful than standard local robustness. In addition, we defined a grammar to express a rich set of post-conditions (properties) and developed a unifying framework to translate these properties into gadgets that can be appended to existing neural networks, thus simplifying arbitrary properties. This enables the use of verifiers such as $\alpha\beta$-CROWN. Our experiments show that this approach is more efficient than directly encoding the properties as constraints into state-of-the-art solvers.

Finally, we presented a novel scheme for quantitatively evaluating LTL formulae. Our evaluation framework ensures that the score assigned to a word is proportional to how well it represents the formula, so that "good representatives" score higher than words that merely satisfy the formula. One contribution of this work is the use of this framework to mine LTL formulae from traces of reactive systems. Our approach provides a viable solution to synthesizing explanations for policies of a reinforcement learned system.

9.1 Future Direction

One of the future works is to integrate our refinement techniques into other abstraction-based verifiers such as $\alpha\beta$-CROWN and PYRAT, because both the tools use heuristics-based branching, and we believe our CEGAR-based approach may benefit. We also plan to explore the use of different oracles, such as alternative neural AI models, such as LLMs, to further analyze false positives. We will also plan to systematically study the trade-offs between different confidence-based robustness properties in various application contexts and false positive/true positive analyses. We also would like to provide generalized reasoning on the network's output using the confidence, by allowing the linear combination over

the conditions on the confidence. We want to bring our layer-based encoding framework to more verifiers and study its scalability on larger neural networks. Naturally, we also want to apply our methods to other AI models such as transformers and large language models (LLMs). A promising direction for for explainability work is to enhance our optimizer so that it can seamlessly handle a broader spectrum of explanations, such as formulae in CTL and CTL*.

Bibliography

[1] Mariusz Bojarski, Davide Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Larry Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *https://arxiv.org/abs/1604.07316*, 04 2016.

[2] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis. *Journal of applied biomedicine*, 11(2):47–58, 2013.

[3] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[4] Joel Janai, Fatma Güney, Aseem Behl, Andreas Geiger, et al. Computer vision for autonomous vehicles: Problems, datasets and state of the art. *Foundations and trends® in computer graphics and vision*, 12(1–3):1–308, 2020.

[5] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[6] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[7] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19(1):221–248, 2017.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[10] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[11] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. Boosting adversarial attacks with momentum. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9185–9193, 2018.

[12] Wieland Brendel, Jonas Rauber, and Matthias Bethge. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *International Conference on Learning Representations (ICLR)*, 2018.

[13] Jianbo Chen, Michael I Jordan, and Martin J Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In *2020 ieee symposium on security and privacy (sp)*, pages 1277–1294. IEEE, 2020.

[14] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[15] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, pages 1–30, 2021.

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

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

[18] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[19] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017.

[20] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International conference on computer aided verification*, pages 97–117. Springer, 2017.

[21] 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 *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[22] Ruediger 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.

[23] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.

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

[25] 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. *arXiv preprint arXiv:2011.13824*, 2020.

[26] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 25150–25164, 2022.

[27] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. A dual approach to scalable verification of deep networks. In *UAI*, volume 1, page 3, 2018.

[28] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2018.

[29] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 31, 2018.

[30] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. Boosting robustness certification of neural networks. In *International Conference on Learning Representations*, 2018.

[31] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, pages 5276–5285. PMLR, 2018.

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

[33] 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, 2018.

[34] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[35] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1599–1614, 2018.

[36] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, pages 6367–6377, 2018.

[37] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In *International Conference on Computer Aided Verification*, pages 43–65. Springer, 2020.

[38] Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. Improving neural network verification through spurious region guided refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–408. Springer, 2021.

[39] Xuankang Lin, He Zhu, Roopsha Samanta, and Suresh Jagannathan. Art: abstraction refinement-guided training for provably correct neural networks. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, volume 1, pages 148–157. TU Wien Academic Press, 2020.

[40] A. Gurobi optimizer. 2020.

[41] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32, 2019.

[42] 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.

[43] Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0210–0215. IEEE, 2018.

[44] Nadia Burkart and Marco F Huber. A survey on the explainability of supervised machine learning. *Journal of Artificial Intelligence Research*, 70:245–317, 2021.

[45] Mohammad Afzal, Ashutosh Gupta, and S Akshay. Using counterexamples to improve robustness verification in neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 422–443. Springer, 2023.

[46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[47] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[48] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[49] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, 2002.

[50] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K Rajamani. Refining approximations in software predicate abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 388–403. Springer, 2004.

[51] Thomas Ball, Andreas Podelski, and Sriram K Rajamani. Boolean and cartesian abstraction for model checking c programs. In *International Conference on Tools*

and Algorithms for the Construction and Analysis of Systems, pages 268–283. Springer, 2001.

[52] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7):68–76, 2011.

[53] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[54] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, 2001.

[55] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *International Static Analysis Symposium*, pages 238–254. Springer, 2008.

[56] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.

[57] Rudy Bunel, P Mudigonda, Ilker Turkaslan, P Torr, Jingyue Lu, and Pushmeet Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.

[58] Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *CoRR*, abs/2104.06718, 2021.

[59] Jingyue Lu and M. Pawan Kumar. Neural network branching for neural network verification. In *International Conference on Learning Representations*, 2020.

[60] α, β-crown. In *VNNCOMP21*. 2021.

[61] Oval. In *VNNCOMP21*. 2021.

[62] Haoze Wu, Omri Isac, Aleksandar Zeljić, Teruhiro Tagomori, Matthew Daggitt, Wen Kokke, Idan Refaeli, Guy Amir, Kyle Julian, Shahaf Bassan, et al. Marabou 2.0: a versatile formal analyzer of neural networks. In *International Conference on Computer Aided Verification*, pages 249–264. Springer, 2024.

[63] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T. Johnson. Improved geometric path enumeration for verifying relu neural networks. In *Proceedings of the 32nd International Conference on Computer Aided Verification*. Springer, 2020.

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

[65] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[66] Marco Casadio, Ekaterina Komendantskaya, Matthew L Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property: a principled case study. In *International conference on computer aided verification*, pages 219–231. Springer, 2022.

[67] Zi Wang, Gautam Prakriya, and Somesh Jha. A quantitative geometric approach to neural-network smoothness. *Advances in Neural Information Processing Systems*, 35:34201–34215, 2022.

[68] Mihaela Rosca, Theophane Weber, Arthur Gretton, and Shakir Mohamed. A case for new neural network smoothness constraints. In Jessica Zosa Forde, Francisco Ruiz, Melanie F. Pradier, and Aaron Schein, editors, *Proceedings on "I Can't Believe It's Not Better!" at NeurIPS Workshops*, volume 137 of *Proceedings of Machine Learning Research*, pages 21–32. PMLR, 12 Dec 2020.

[69] Blerta Lindqvist. A novel method for function smoothness in neural networks. *IEEE Access*, 10:75354–75364, 2022.

[70] Klas Leino and Matt Fredrikson. Relaxing local robustness. *Advances in Neural Information Processing Systems*, 34:17072–17083, 2021.

[71] 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.

[72] 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.

[73] Stanley Bak, Changliu Liu, and Taylor Johnson. The second international verification of neural networks competition (vnn-comp 2021): Summary and results. *arXiv preprint arXiv:2109.00498*, 2021.

[74] 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. *arXiv preprint arXiv:2212.10376*, 2022.

[75] Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. The fourth international verification of neural networks competition (vnn-comp 2023): Summary and results, 2023.

[76] Christopher Brix, Stanley Bak, Taylor T Johnson, and Haoze Wu. The fifth international verification of neural networks competition (vnn-comp 2024): Summary and results. *arXiv preprint arXiv:2412.19985*, 2024.

[77] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, ON, Canada, 2009.

[78] Andreea Postovan and Mădălina Eraşcu. Architecturing binarized neural networks for traffic sign recognition. *arXiv preprint arXiv:2303.15005*, 2023.

[79] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[80] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*,

ICML '00, page 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[81] David Abel, Will Dabney, Anna Harutyunyan, Mark K Ho, Michael Littman, Doina Precup, and Satinder Singh. On the expressivity of markov reward. *Advances in Neural Information Processing Systems*, 34, 2021.

[82] Rodrigo Toro Icarte, Toryn Klassen, Richard Valenzano, and Sheila McIlraith. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, pages 2107–2116. PMLR, 2018.

[83] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[84] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[85] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *International Conference on Computer Aided Verification*, pages 53–65. Springer, 2001.

[86] Ronen I Brafman and Giuseppe De Giacomo. Planning for ltlf/ldlf goals in non-markovian fully observable nondeterministic domains. In *IJCAI*, pages 1602–1608, 2019.

[87] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. Association for Computing Machinery, 2013.

[88] E. M. Hahn, M. Perez, S. Schewe, F. Somenzi, A. Trivedi, and D. Wojtczak. Omega-regular objectives in model-free reinforcement learning. In *TACAS 2019, Proceedings, Part I*, volume 11427 of *LNCS*, pages 395–412. Springer, 2019.

[89] A. K. Bozkurt, Y. Wang, M. M. Zavlanos, and M. Pajic. Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In *ICRA*, pages 10349–10355. IEEE, 2020.

[90] D. Sadigh, E. S. Kim, S. Coogan, S. S. Sastry, and S. A. Seshia. A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In *CDC*, pages 1091–1096, 2014.

[91] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q Klassen, Richard Anthony Valenzano, and Sheila A McIlraith. Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, volume 19, pages 6065–6073, 2019.

[92] Alper Kamil Bozkurt, Yu Wang, Michael M Zavlanos, and Miroslav Pajic. Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10349–10355. IEEE, 2020.

[93] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Mungojerrie: Reinforcement learning of linear-time objectives. *arXiv preprint arXiv:2106.09161*, 2021.

[94] Jan Kretínský and Javier Esparza. Deterministic automata for the (f, g)-fragment of LTL. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 7–22, 2012.

[95] Javier Esparza, Jan Kretínský, and Salomon Sickert. One theorem to rule them all: A unified translation of LTL into ω-automata. In *LICS 2018*, 2018.

[96] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer Berlin Heidelberg, 2008.

[97] D. Neider and I. Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

[98] JD Choi and Kee-Eung Kim. Inverse reinforcement learning in partially observable environments. *Journal of Machine Learning Research*, 12:691–730, 2011.

[99] Mohammad Afzal, S Akshay, and Ashutosh Gupta. Verifying rich robustness properties for neural networks. *arXiv preprint arXiv:2511.07293*, 2025.

[100] Mohammad Afzal, Sankalp Gambhir, Ashutosh Gupta, Ashutosh Trivedi, and Alvaro Velasquez. Ltl-based non-markovian inverse reinforcement learning. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 2857–2859, 2023.

[101] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[102] Constituency Parsing. Speech and language processing. *Power Point Slides*, page 20, 2009.

[103] Andreas Mueller. Modern robotics: Mechanics, planning, and control [bookshelf]. *IEEE Control Systems Magazine*, 39(6):100–102, 2019.

[104] Hao Zheng, Zhanlei Yang, Wenju Liu, Jizhong Liang, and Yanpeng Li. Improving deep neural networks using softplus units. In *2015 International joint conference on neural networks (IJCNN)*, pages 1–4. IEEE, 2015.

[105] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[106] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[107] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[108] Mingxing Tan, Q Efficientnet Le, et al. Rethinking model scaling for convolutional neural networks. In *Proceedings of the International conference on machine learning, Long Beach, CA, USA*, volume 15, 2019.

[109] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent. *Advances in neural information processing systems*, 12, 1999.

[110] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[111] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International workshop on artificial neural networks*, pages 195–201. Springer, 1995.

[112] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[113] Dennis Wei, Haoze Wu, Min Wu, Pin-Yu Chen, Clark Barrett, and Eitan Farchi. Convex bounds on the softmax function with applications to robustness verification. In *International Conference on Artificial Intelligence and Statistics*, pages 6853–6878. PMLR, 2023.

[114] Anagha Athavale, Ezio Bartocci, Maria Christakis, Matteo Maffei, Dejan Nickovic, and Georg Weissenbacher. Verifying global two-safety properties in neural networks with confidence. In *International Conference on Computer Aided Verification*, pages 329–351. Springer, 2024.

[115] Steven Rakitin. *Software verification and validation for practitioners and managers*. Artech, 2001.

[116] Carl-Johan Seger. *An introduction to formal hardware verification*. University of British Columbia, Department of Computer Science, 1992.

[117] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 2000.

[118] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[119] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[120] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[121] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[122] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[123] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[124] Duo Zhou, Christopher Brix, Grani A Hanasusanto, and Huan Zhang. Scalable neural network verification with branch-and-bound inferred cutting planes. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[125] CPLEX User's Manual. Ibm ilog cplex optimization studio. *Version*, 12(1987-2018):1, 1987.

[126] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper Van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, et al. Enabling research through the scip optimization suite 8.0. *ACM Transactions on Mathematical Software*, 49(2):1–21, 2023.

[127] Laurence A Wolsey and George L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 1999.

[128] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2009.

[129] Ralph E Gomory. Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem. In *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 77–103. Springer, 2009.

[130] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[131] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[132] Robert Nieuwenhuis, Albert Oliveras, and Cesare 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.

[133] Clark W Barrett, David L Dill, and Jeremy R Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, pages 522–527, 1998.

[134] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[135] Gagandeep Singh, Jacob Laurel, Sasa Misailovic, Debangshu Banerjee, Avaljot Singh, Changming Xu, Shubham Ugare, Huan Zhang, et al. Safety and trust in artificial intelligence with abstract interpretation. *Foundations and Trends® in Programming Languages*, 8(3-4):250–408, 2025.

[136] Sudeep Kanav, Jan Křetínskỳ, and Sabine Rieder. A literature review on verification and abstraction of neural networks within the formal methods community. *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, pages 39–65, 2024.

[137] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[138] Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. Scaling polyhedral neural network verification on gpus. *Proceedings of Machine Learning and Systems*, 3:733–746, 2021.

[139] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin 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.

[140] Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, Krunal Kishor Patel, and Juan Pablo Vielma. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. *Advances in Neural Information Processing Systems*, 33:21675–21686, 2020.

[141] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *International conference on computer aided verification*, pages 627–633. Springer, 2009.

[142] Marshall L Fisher. The lagrangian relaxation method for solving integer programming problems. *Management science*, 50(12_supplement):1861–1871, 2004.

[143] Pranav Ashok, Vahid Hashemi, Jan Křetínský, and Stefanie Mohr. Deepabstract: Neural network abstraction for accelerating verification. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 92–107, Cham, 2020. Springer International Publishing.

[144] Fateh Boudardara, Abderraouf Boussif, Pierre-Jean Meyer, and Mohamed Ghazel. Interval weight-based abstraction for neural network verification. In *International Conference on Computer Safety, Reliability, and Security*, pages 330–342. Springer, 2022.

[145] Fateh Boudardara, Abderraouf Boussif, Pierre-Jean Meyer, and Mohamed Ghazel. Innabstract: an inn-based abstraction method for large-scale neural network verification. *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[146] Sanaa Siddiqui, Diganta Mukhopadhyay, Mohammad Afzal, Hrishikesh Karmarkar, and Kumar Madhukar. Unifying syntactic and semantic abstractions for deep neural networks. In Anne E. Haxthausen and Wendelin Serwe, editors, *Formal Methods for Industrial Critical Systems*, pages 201–219, Cham, 2024. Springer Nature Switzerland.

[147] Florian Jaeckle, Jingyue Lu, and M Pawan Kumar. Neural network branch-and-bound for neural network verification. *arXiv preprint arXiv:2107.12855*, 2021.

[148] Patrick Henriksen and Alessio Lomuscio. Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In *IJCAI*, pages 2549–2555, 2021.

[149] Yuke Liao, Blaise Genest, Kuldeep Meel, and Shaan Aryaman. Solution-aware vs global relu selection: partial milp strikes back for dnn verification. In *International Symposium on Automated Technology for Verification and Analysis*, pages 299–320. Springer, 2025.

[150] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 731–744, 2019.

[151] Tobias Ladner and Matthias Althoff. Automatic abstraction refinement in neural network verification using sensitivity analysis. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–13, 2023.

[152] Matan Ostrovsky, Clark Barrett, and Guy Katz. An abstraction-refinement approach to verifying convolutional neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 391–396. Springer, 2022.

[153] Jiaxiang Liu, Yunhan Xing, Xiaomu Shi, Fu Song, Zhiwu Xu, and Zhong Ming. Abstraction and refinement: towards scalable and exact verification of neural networks. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–35, 2024.

[154] Zhe Zhao, Yedi Zhang, Guangke Chen, Fu Song, Taolue Chen, and Jiaxiang Liu. Cleverest: accelerating cegar-based neural network verification via adversarial attacks. In *International Static Analysis Symposium*, pages 449–473. Springer, 2022.

[155] Samuel Teuber, Philipp Kern, Marvin Janzen, and Bernhard Beckert. Revisiting differential verification: Equivalence verification with confidence. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 257–278. Springer, 2025.

[156] Peter Blohm, Patrick Indri, Thomas Gärtner, and SAGAR MALHOTRA. Probably approximately global robustness certification. In *Forty-second International Conference on Machine Learning*, 2025.

[157] Eleonora Giunchiglia, Alex Tatomir, Mihaela Cătălina Stoian, and Thomas Lukasiewicz. Ccn+: A neuro-symbolic framework for deep learning with requirements. *International Journal of Approximate Reasoning*, 171:109124, 2024. Synergies between Machine Learning and Reasoning.

[158] David Shriver, Sebastian Elbaum, and Matthew B Dwyer. Dnnv: A framework for deep neural network verification. In *International Conference on Computer Aided Verification*, pages 137–150. Springer, 2021.

[159] Eric Wong, Frank R. Schmidt, Jan Hendrik Metzen, and J. Zico Kolter. Scaling provable adversarial defenses. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 8410–8419, Red Hook, NY, USA, 2018. Curran Associates Inc.

[160] Jiameng Fan and Wenchao Li. Adversarial training and provable robustness: A tale of two objectives. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7367–7376, 2021.

[161] Sven Gowal, Krishnamurthy Dj Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. Scalable verified training for provably robust image classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4842–4851, 2019.

[162] Sungyoon Lee, Woojin Lee, Jinseong Park, and Jaewook Lee. Towards better understanding of training certifiably robust models against adversarial examples. *Advances in Neural Information Processing Systems*, 34:953–964, 2021.

[163] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

[164] Zhaodi Zhang, Zhiyi Xue, Yang Chen, Si Liu, Yueling Zhang, Jing Liu, and Min Zhang. Boosting verified training for robust image classifications via abstraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16251–16260, 2023.

[165] Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. *arXiv preprint arXiv:1906.06316*, 2019.

[166] Aditi Raghunathan, Jacob Steinhardt, and Percy S Liang. Semidefinite relaxations for certifying robustness to adversarial examples. *Advances in neural information processing systems*, 31, 2018.

[167] 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 *International Conference on Machine Learning*, pages 26591–26604. PMLR, 2022.

[168] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[169] Rudy Bunel, Alessandro De Palma, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip Torr, and M Pawan Kumar. Lagrangian decomposition for neural network verification. In *Conference on Uncertainty in Artificial Intelligence*, pages 370–379. PMLR, 2020.

[170] Alessandro De Palma, Harkirat S Behl, Rudy Bunel, Philip Torr, and M Pawan Kumar. Scaling the convex barrier with active sets. In *Proceedings of the ICLR 2021 Conference*. Open Review, 2021.

[171] Alessandro De Palma, Harkirat Singh Behl, Rudy Bunel, Philip H. S. Torr, and M. Pawan Kumar. Scaling the convex barrier with sparse dual algorithms. *CoRR*, abs/2101.05844, 2021.

[172] Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.

[173] Mark Niklas Müller, Gagandeep Singh, Mislav Balunovic, Gleb Makarchuk, Anian Ruoss, François Serre, Maximilian Baader, Dana Drachsler Cohen, Timon Gehr, Adrian Hoffmann, Jonathan Maurer, Matthew Mirman, Christoph Müller, Markus Püschel, Petar Tsankov, and Martin Vechev. Eran: Eth robustness analyzer for neural networks.
urlhttps://github.com/eth-sri/eran. Secure, Reliable, and Intelligent Systems Lab (SRI), ETH Zurich. Accessed: 2025-07-17.

[174] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[175] Seyyed Ali Emami, Paolo Castaldi, and Afshin Banazadeh. Neural network-based flight control systems: Present and future. *Annual Reviews in Control*, 53:97–137, 2022.

[176] Daniel Kroening and Ofer Strichman. *Decision procedures*, volume 5. Springer, 2008.

[177] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[178] Mislav Balunovic and Martin Vechev. Adversarial training and provable defenses: Bridging the gap. In *International Conference on Learning Representations*, 2020.

[179] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: A multi-class classification competition. In *The 2011 International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[180] Christel B and Joost-Pieter K. *Principles of model checking*. MIT Press, 2008.

[181] Paulo Tabuada and Daniel Neider. Robust linear temporal logic. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, Marseille, France, August 29 - September 1, 2016*, volume 62 of *LIPIcs*, pages 10:1–10:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[182] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92. IEEE, 2015.

[183] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[184] Texada. https://github.com/ModelInference/texada, 2015.

[185] Edsger W Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. Springer, 1971.

[186] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, page 7–15, New York, NY, USA, 1998. Association for Computing Machinery.

[187] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

[188] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[189] Abdeslam Boularias, Jens Kober, and Jan Peters. Relative entropy inverse reinforcement learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 182–189. JMLR Workshop and Conference Proceedings, 2011.

[190] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[191] Monica C Vroman. *Maximum likelihood inverse reinforcement learning*. Rutgers The State University of New Jersey-New Brunswick, 2014.

[192] Dexter R.R. Scobee and S. Shankar Sastry. Maximum likelihood constraint inference for inverse reinforcement learning. In *International Conference on Learning Representations*, 2020.

[193] Edouard Klein, Matthieu Geist, Bilal Piot, and Olivier Pietquin. Inverse reinforcement learning through structured classification. *Advances in neural information processing systems*, 25, 2012.

[194] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, 19:1, 2007.

[195] Aaron Tucker, Adam Gleave, and Stuart Russell. Inverse reinforcement learning for video games. *arXiv preprint arXiv:1810.10593*, 2018.

[196] Manuel Lopes, Francisco Melo, and Luis Montesano. Active learning for reward estimation in inverse reinforcement learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 31–46. Springer, 2009.

[197] James Jay Horning. A study of grammatical inference. Technical report, STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1969.

[198] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[199] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[200] Rafael C Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference*, pages 139–152. Springer, 1994.

[201] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas Dyhre Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning markov decision processes for model checking. *Electronic Proceedings in Theoretical Computer Science*, 103:49–63, 2012.

[202] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D Nielsen, Kim G Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016.

[203] Eden Abadi and Ronen I. Brafman. Learning and solving regular decision processes. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, pages 1948–1954. ijcai.org, 2020.

[204] Rodrigo Toro Icarte, Ethan Waldie, Toryn Klassen, Rick Valenzano, Margarita Castro, and Sheila McIlraith. Learning reward machines for partially observable reinforcement learning. *Advances in Neural Information Processing Systems*, 32:15523–15534, 2019.

[205] Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. Joint inference of reward machines and policies for reinforcement learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 590–598, 2020.

[206] Zhe Xu, Bo Wu, Aditya Ojha, Daniel Neider, and Ufuk Topcu. Active finite reward automaton inference and reinforcement learning using queries and counterexamples. In *Machine Learning and Knowledge Extraction - 5th IFIP TC 5, TC 12, WG 8.4, WG 8.9, WG 12.9 International Cross-Domain Conference, CD-MAKE 2021, Virtual Event, August 17-20, 2021, Proceedings*, volume 12844 of *Lecture Notes in Computer Science*, pages 115–135. Springer, 2021.

[207] Sean A Fulop and David Kephart. Topology of language classes. In *Proceedings of the 14th Meeting on the Mathematics of Language (MoL 2015)*, pages 26–38, 2015.

[208] Austin J. Parker, Kelly B. Yancey, and Matthew P. Yancey. Regular language distance and entropy. In *International Symposium on Mathematical Foundations of Computer Science*, 2016.

[209] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. Process mining and verification of properties: an approach based on temporal logic. In *Proceedings of the 2005 Confederated International Conference on On the Move to Meaningful Internet Systems - Volume >Part I*, OTM'05, page 130–147, Berlin, Heidelberg, 2005. Springer-Verlag.

[210] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, page 135, USA, 2001. IEEE Computer Society.

[211] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[212] David Lo, Ganesan Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, page 62–71, USA, 2009. IEEE Computer Society.

[213] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, page 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[214] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. In *Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '98, page 469–483, Berlin, Heidelberg, 1998. Springer-Verlag.

[215] Michael zur Muehlen and Michael Rosemann. Workflow-based process monitoring and controlling ¾ technical and organizational issues. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6 - Volume 6*, HICSS '00, page 6032, USA, 2000. IEEE Computer Society.

[216] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using

declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Softw. Eng.*, 2015.

[217] Ivan B, Yuriy B, Sigurd S, Michael S, and Michael D E. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE*, 2011.