Foundations and Trends® in Programming Languages

# Safety and Trust in Artificial Intelligence with Abstract Interpretation

**Suggested Citation:** Gagandeep Singh, Jacob Laurel, Sasa Misailovic, Debangshu Banerjee, Avaljot Singh, Changming Xu, Shubham Ugare and Huan Zhang (2025), "Safety and Trust in Artificial Intelligence with Abstract Interpretation", Foundations and Trends® in Programming Languages: Vol. 8, No. 3-4, pp 250–408. DOI: 10.1561/2500000062.

**Gagandeep Singh**
UIUC
ggnds@illinois.edu

**Jacob Laurel**
Georgia Institute of Technology
jlaurel6@gatech.edu

**Sasa Misailovic**
UIUC
misailo@illinois.edu

**Debangshu Banerjee**
UIUC
db21@illinois.edu

**Avaljot Singh**
UIUC
avaljot2@illinois.edu

**Changming Xu**
UIUC
cx23@illinois.edu

**Shubham Ugare**
UIUC
sugare2@illinois.edu

**Huan Zhang**
UIUC
huanz@illinois.edu

now
the essence of knowledge
Boston — Delft

# Contents

# Safety and Trust in Artificial Intelligence with Abstract Interpretation

Gagandeep Singh[1], Jacob Laurel[2], Sasa Misailovic[1], Debangshu Banerjee[1], Avaljot Singh[1], Changming Xu[1], Shubham Ugare[1] and Huan Zhang[1]

[1] *University of Illinois Urbana-Champaign, USA; {ggnds, misailo, db21, avaljot2, cx23, sugare2, huanz}@illinois.edu*
[2] *Georgia Institute of Technology, USA; jlaurel6@gatech.edu*

ABSTRACT

Deep neural networks (DNNs) now dominate the AI landscape and have shown impressive performance in diverse application domains, including vision, natural language processing (NLP), and healthcare. However, both public and private entities have been increasingly expressing significant concern about the potential of state-of-the-art AI models to cause societal and financial harm. This lack of trust arises from their black-box construction and vulnerability against natural and adversarial noise.

As a result, researchers have spent considerable time developing automated methods for building safe and trustworthy DNNs. Abstract interpretation has emerged as the most popular framework for efficiently analyzing realistic DNNs among the various approaches. However, due to fundamental differences in the computational structure (e.g., high

Gagandeep Singh, Jacob Laurel, Sasa Misailovic, Debangshu Banerjee, Avaljot Singh, Changming Xu, Shubham Ugare and Huan Zhang (2025), “Safety and Trust in Artificial Intelligence with Abstract Interpretation”, Foundations and Trends® in Programming Languages: Vol. 8, No. 3-4, pp 250–408. DOI: 10.1561/2500000062.
©2025 G. Singh *et al.*

nonlinearity) of DNNs compared to traditional programs, developing efficient DNN analyzers has required tackling significantly different research challenges than encountered for programs.

In this monograph, we describe state-of-the-art approaches based on abstract interpretation for analyzing DNNs. These approaches include the design of new abstract domains, synthesis of novel abstract transformers, abstraction refinement, and incremental analysis. We will discuss how the analysis results can be used to: (i) formally check whether a trained DNN satisfies desired output and gradient-based safety properties, (ii) guide the model updates during training towards satisfying safety properties, and (iii) reliably explain and interpret the black-box workings of DNNs.

# 1

# Introduction

Deep neural networks (DNNs) are currently the dominant technology in artificial intelligence (AI) and have shown impressive performance in diverse applications, including autonomous driving, medical diagnosis, text generation, and logical reasoning. However, they lack transparency due to their black-box construction and are vulnerable to environmental and adversarial noise. These issues have caused concerns about their safety and trust when deployed in the real world. Although standard training optimizes the model's accuracy, it does not take into account desirable safety properties such as *robustness* (the DNN should behave similarly for similar inputs), *fairness* (the DNN output should not depend too much on some legally protected attribute, such as gender or race), and *monotonicity* (if the inputs are partially ordered, so should be the outputs). As a result, state-of-the-art models remain untrustworthy. Building trust in DNNs is essential to realizing their vast potential to positively transform society and the economy and is one of the grand challenges in computer science today.

## 1.1 Safety-informed DNN Deployment Cycle

Figure 1.1 presents a general safety-informed pipeline for DNN development, applicable to any application domain. Safety, accuracy, and efficiency can often conflict with each other. DNN accuracy improves with model size but that increases the inference cost. Similarly, models maximizing safety can have reduced accuracy. For example, a DNN classifier that always predicts the same class for all inputs is robust but has very low accuracy. As a result, it may not be possible to obtain DNNs that optimize all three objectives simultaneously. Depending on the target application, a developer may prioritize accuracy over trust or vice-versa. The goal of safety-informed DNN development is to ensure a sufficient balance between accuracy, safety, and efficiency.

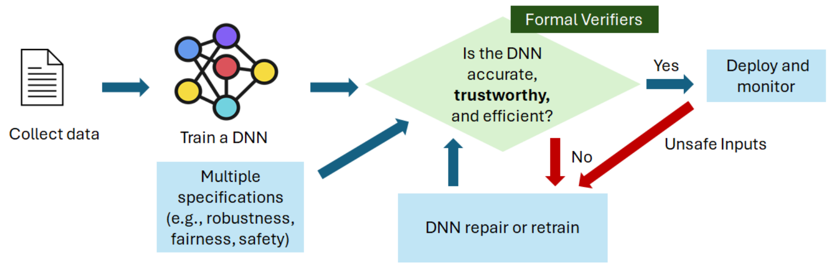

Formal Verifiers
Collect data
Train a DNN
Is the DNN accurate, **trustworthy,** and efficient?
Yes
Deploy and monitor
Unsafe Inputs
No
Multiple specifications (e.g., robustness, fairness, safety)
DNN repair or retrain

**Figure 1.1:** Development pipeline for building accurate, trustworthy, and efficient DNNs. Verification is used for testing model trustworthiness (green diamond).

In this pipeline, first, representative training data for the target application is collected and a DNN is trained to maximize its accuracy on test inputs from the training distribution. Next, a domain expert creates (manually or algorithmically) a set of formal safety specifications (e.g., robustness, fairness) characterizing the expected DNN behavior in different real-world scenarios. The set of inputs covered by these specifications can be infinite.

The expert then checks whether the model meets the safety standards. Since DNNs may not satisfy all the specifications, the standards can require that at least a significant fraction of all specifications be satisfied for trustworthiness. If the model meets the criteria, then the DNN is considered fit for deployment. Otherwise, it is iteratively repaired (e.g., by fine-tuning) until we obtain the desired balance between accuracy, safety, and efficiency.

During deployment, the DNN inputs are monitored for distribution shifts, i.e., the inputs are not from the training distribution. If the runtime system detects a distribution shift, it reports representative samples to the domain experts. They then design new specifications, and the model undergoes another round of repair (or full retraining).

**How formal methods can help.** For checking that the model satisfies safety specifications, the standard practice is to evaluate the DNN behavior on a finite set of inputs satisfying the specifications. However, this cannot guarantee safe and trustworthy DNN behavior on all specification inputs. The unseen set can be huge and contain inputs often seen during real-world deployment. To address these limitations, there is growing work on checking the safety of DNN models and interpreting their behavior, on an infinite set of unseen inputs from safety specifications using formal methods, which provides a more reliable metric for measuring a model's safety than standard empirical methods. For example, a repaired DNN preserving the original test set accuracy and efficiency but satisfying the trustworthy specifications more often is a better model than the unrepaired one as it is less likely to show undesirable behavior during real-world deployment. Formal methods can also be used during training to guide the model to satisfy desirable safety and trustworthiness properties. The models trained this way are more likely to satisfy safety specifications than those without.

This monograph presents a comprehensive treatment of the techniques that guarantee the safety of DNNs by formally modeling the behavior of modern DNNs and efficiently computing with abstractions that represent those behaviors. Our main focus will be on approaches that leverage a general framework for automated analysis of programming languages called abstract interpretation, the most successful formal methods for automatically reasoning about DNNs. We emphasize that the knowledge of the topics covered in this monograph is necessary not only for computer scientists but for practitioners from all areas building DNN-based applications, e.g., natural sciences, aerospace, finance, etc.

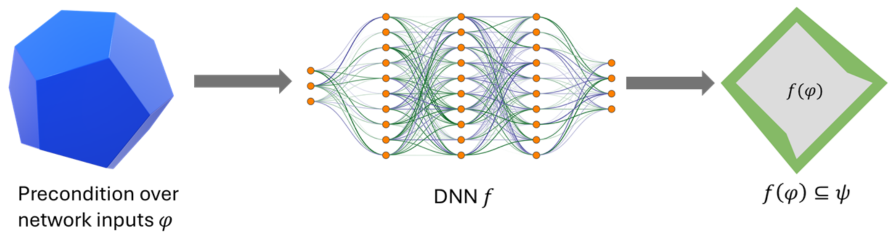
Next, we describe how safety and trustworthy properties can be formally specified for DNNs, then we will discuss the key ideas and design considerations in developing abstract interpretation based methods for the formal verification and training of DNNs. We will also discuss how abstract interpretation enables reliable explanations and interpretations of DNNs as well as analysis of differentiable programs.

## 1.2 Formal Specifications for DNNs

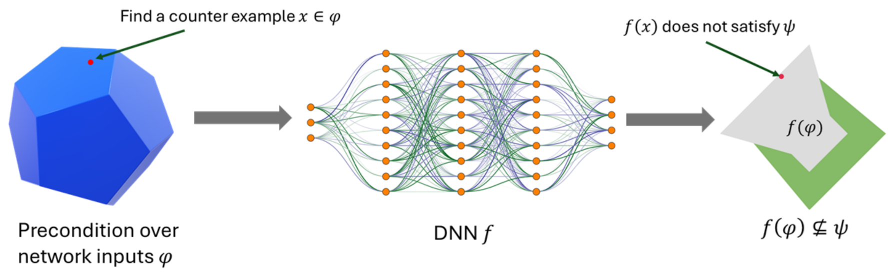
Mathematically, we model a trained DNN as a pure function $f$. Its input $x$ can be images, text, videos, sensor measurements, or other data. We denote the output of the DNN as $f(x)$, which can be a classification of the input into one of the predefined classes, the regression that estimates a continuous value, or the set of tokens generated by a language model. We denote gradients of $f$ as $f'(x)$.

For a trained DNN $f$, a developer specifies the property of interest using two formulas: (1) *the precondition* $\varphi$, which specifies the set of inputs on which the DNN should not misbehave and (2) *the postcondition* $\psi$, which specifies safe and trustworthy behaviors of the DNN for the given inputs. These behaviors are typically constraints on the DNN's outputs or its gradients. The preconditions and postconditions are domain-dependent and usually designed by DNN developers.A tool for DNN verification (*a verifier*) aims to automatically check if the postcondition on the DNN's outputs and/or gradients is satisfied for all inputs specified by the precondition.

A property specification is a tuple $(\varphi, \psi)$, where $\varphi$ is the precondition and $\psi$ is the postcondition. Both formulas $\varphi$ and $\psi$ typically represent *an infinite number of inputs/outputs.* We denote the set of the results of the evaluations of the DNN on all inputs described by the precondition $\varphi$ as $f(\varphi) = \{f(x) \mid x \in \varphi\}$. Similarly, we denote the set of all gradients as $f'(\varphi)$. The verifier then checks for the inclusion of the set of possible executions of the DNN into the set of outputs that satisfy the postcondition, i.e., $f(\phi) \subseteq \psi$ (or $f'(\phi) \subseteq \psi$) holds. *Single execution* specifications, as shown in Figure 1.2, require that each DNN output $f(x)$ where $x \in \varphi$ must independently satisfy $\psi$. *Relational* specifications require reasoning about multiple related executions of the same or different DNNs. As we will show in Section 3, a general way to represent and compute with $\varphi$ and $\psi$ in these settings is as disjunctions


Precondition over network inputs $\varphi$
DNN $f$
$f(\varphi)$
$f(\varphi) \subseteq \psi$

**(a)** Verified


Find a counter example $x \in \varphi$
$f(x)$ does not satisfy $\psi$
Precondition over network inputs $\varphi$
DNN $f$
$f(\varphi)$
$f(\varphi) \nsubseteq \psi$

**(b)** Counterexample

**Figure 1.2:** Single execution specifications require that the DNN output for each input from $\varphi$ must independently satisfy $\psi$.

of convex polyhedra within the framework of abstract interpretation. $\varphi$ and $\psi$ can also define distributions leading to *probabilistic* specifications.
**Local and global properties.** The set of specifications for DNNs can be broadly classified as *local* or *global*. The precondition $\varphi$ for local properties defines a local neighborhood around a sample input from the test set. For example, given a test image correctly classified as a car by a DNN, the commonly used *local robustness property* specifies that if the original image was classified as a car, then all images generated by rotating the original image within $\pm d$ degrees are also classified as a car. We present many local properties in Sections 3.1 and 3.2.

In contrast, global properties are not defined with respect to a specific test input. Verifying global properties yields stronger safety guarantees compared to local properties, however, global properties are difficult to formulate for popular domains, such as vision and NLP, where the individual features processed by the DNN have no clear semantic meaning. While verifying local properties is not ideal, the local verification results enable testing the safety of the model on an infinite set of unseen inputs, not possible with standard methods. We present several concrete global properties in Section 3.1.

## 1.3 Verifying Specifications over DNNs

DNN verification can be seen as an instance of program verification, since one can write a DNN as a program, i.e., there is a direct translation from the mathematical representation of the DNN as a function $f$ into a side-effect free program. However, since it is well-known that program verification is *undecidable* (one cannot prove the correctness of an arbitrary program with respect to an arbitrary property of interest), DNN verification is also undecidable in general. Certain DNN verification problems, such as robustness verification of feedforward networks with ReLU activations, are decidable but still NP-complete in general (Katz *et al.*, 2017).

State-of-the-art verifiers are therefore incomplete in general, i.e., they can fail to prove a specification when it holds. However, when they succeed, the DNN will satisfy the specification. In this monograph, we focus on *white-box* verifiers that require access to the model parameters. Verification of closed-source models requires *black-box* verifiers. We refer the interested readers to the relevant material in this direction in Section 1.7. The white-box verifiers can be formulated using the elegant framework of abstract interpretation. The verifier is parameterized by the choice of an *abstract domain* with two main components: abstract elements and abstract transformers. Abstract elements are mathematical objects symbolically representing an infinite set of numerical points over which the verifier operates. Abstract transformers overapproximate the effect of applying the transformations inside the DNN program (e.g., affine or ReLU assignments) on abstract elements.

There is a tradeoff between the cost and overapproximation error (also known as precision) of an incomplete verifier: expensive verifiers are more precise while cheap verifiers are imprecise. Both are determined by the design of the abstract domain and transformers. The key consideration in designing an efficient verifier applicable to real-world DNNs is managing this tradeoff. The classical domains, such as Polyhedra and Octagons, used for analyzing programs are not well suited for DNN verification. This is because the DNNs have a different structure compared to traditional programs. For example, DNNs have a large number of non-linear assignments but typically do not have loops. For efficient verification, researchers have developed numerous new abstract domains and transformers tailored for DNN verification. These abstract domains can scale to realistic DNNs with millions of neurons, or more than 100 layers, verifying diverse safety properties in different real-world applications. We will present them in Section 3. We will also discuss how verification can be done incrementally to improve efficiency when verifying a large number of similar DNNs and specifications as needed for the developement pipeline in Figure 1.1.

## 1.4 Training Provably Safe DNNs

DNNs trained with standard training often do not satisfy safety specifications as safety satisfaction is not part of their training objective. Adversarial or counter-example guided training augment the training data with violating examples during training, however the trained models still cannot be proven to be safe in most cases. To overcome these limitations, certified training methods have been developed in recent years which directly incorporate the verifier computations within the training loop and generate models with a high degree of provability, i.e., they are more likely to satisfy specifications and are relatively easier to prove than DNNs obtained with competing methods.

In certified training, if the model $f$ does not satisfy the specification, as checked by a verifier, its weights are updated to increase the provability. The gradient updates are derived by formulating a differentiable property loss on the verifier output, which measures how far the model is from satisfying the property. Since gradient updates are derived from the verifier code, its computations must be expressible as a differentiable function of model weights and parallelizable on GPUs for scalability. Overall, certified training can be seen as training $f$ where the model updates are derived by differentiating the surrogate approximation of the DNN within $\varphi$, computed by the verifier.

While certified training improves the provability, safety specifications can be in conflict with accuracy. Using an imprecise verifier during training can result in overregularization and a significant reduction in the standard accuracy. However, precise verifiers often have complicated code which makes the optimization problem too complicated to solve during training, yielding suboptimal results. Also, employing a verifier during training is more expensive than when used for checking specifications on an already trained DNN, as now the verifier is called during every training iteration. Balancing the provability, accuracy, and cost is therefore the main challenge when developing state-of-the-art methods. Researchers have developed a variety of abstractions, refinements, and loss formulations to enable efficient training. We will cover these methods in detail in Section 4.

## 1.5 Explaining and Interpreting DNNs

Popular methods for explaining DNN predictions identify relevant input features that influence the DNN output the most. However, they do not give guarantees about the robustness of the generated explanations. Relying on non-robust explanations can lead to a false sense of confidence in an untrustworthy model. We will discuss how abstract interpretation can be leveraged to generate explanations with robustness guarantees in Section 5, reliably improving DNN transparency.

Abstract interpretation-based DNN verifiers generate high-dimensional abstract elements at different layers capturing complex relationships between neurons and DNN inputs to prove DNN safety. However, the individual neurons and inputs in the DNN do not have any semantic meaning, unlike the variables in programs, therefore it is not clear whether the safety proofs are based on any meaningful features learned by the DNN. If the DNN is proven to be safe but the proof is based on meaningless features not aligned with human intuition, then the DNN behavior cannot be considered trustworthy. While there has been a lot of work on interpreting black-box DNNs, standard methods can only explain the DNN behavior on individual inputs and cannot interpret the complex invariants encoded by the abstract elements capturing DNN behavior on an infinite set of inputs.

The main challenge in interpreting DNN proofs is mapping the complex abstract elements to human-understandable interpretations.

Section 5 presents ProFIt, the first method for interpreting robustness proofs computed by DNN verifiers. The technique can interpret proofs computed by different verifiers. It builds upon the novel concept of *proof features* computed by projecting the high-dimensional abstract elements onto individual neurons. The proof features can be analyzed independently by generating the corresponding interpretations. Since certain proof features can be more important for the proof than others, a priority function over the proof features that signifies the importance of each proof feature in the complete proof is defined. The method extracts a set of proof features by retaining only the more important parts of the proof that preserve the property.

A comparison of proof interpretations for DNNs trained with standard and robust training methods shows that the proof features corresponding to the standard networks rely on spurious input features that are not aligned with human intuition. The proofs of adversarially trained DNNs filter out some of these spurious features. In contrast, the networks trained with certifiable training produce proofs that do not rely on any spurious features but they also miss out on some meaningful features. Proofs for training methods that combine both empirical and certified robustness not only preserve meaningful features but also selectively filter out spurious ones. These insights suggest that DNNs can satisfy safety properties but their behavior can still be untrustworthy.

## 1.6 Analyzing and Verifying Differentiable Programs

Differentiable programming, which includes automatic differentiation (AD), is the backbone of machine learning. AD computes the gradients alongside the values of the program's output variables. AD computations generalize many machine learning and signal processing applications. Thus, generalized abstractions for AD analysis can be deployed across applications: a neural network, an image filter, and a differential equation solver can be expressed and analyzed in the same language, even when combined in complex programs. Despite AD's ubiquity, automated formal reasoning of derivatives that AD computes has lagged.

Analyzing gradient properties is important for today's trustworthy AI: for instance, the sensitivity of DNN's output to input noise can be expressed as finding bound for the absolute gradients values. The same bounds can help with selecting low precision data types in machine learning algorithms to prevent overflows. Fairness can be formalized as a monotonicity property on a specific attribute, which is satisfied when all derivatives are strictly positive.

To answer these questions, it is not sufficient to reason about the output of a function (e.g., DNN) $f$ for all inputs in $\varphi$. Instead one has to reason about $f'$, the derivative of $f$. For instance, to prove the monotonicity of $f$, one should ensure that its derivative $f'$ is strictly positive or negative for all inputs in $\varphi$. Figure 1.3 presents an intuition of this workflow.

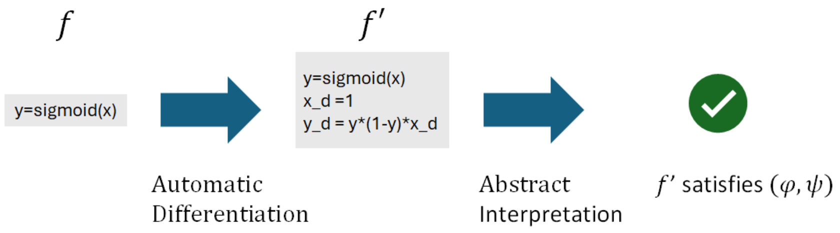


**Figure 1.3:** Verifying derivative properties requires first computing the derivative of a function $f$ (given as a piece of code) using automatic differentiation. The derivative program is then analyzed with abstract interpretation to prove the desired property $(\varphi, \psi)$ holds for the derivative $f'$ instead of $f$ itself.

Section 6 will present a general framework for precise analysis of AD computations. This approach leverages ideas from abstract interpretation of DNNs and generalizes them to find precise abstract transformers of gradient computation. It overcomes the limitation of standard program analysis, which treats the gradient computation as any other code, and leads to significant imprecision, and is in some cases ill-defined. We will present the advantage of the AD-specific abstract transformers on the case study for monotonicity analysis for a decision-making DNN.

## 1.7 Sources and Further Reading

Many recent studies demonstrate the power of modern DNNs, e.g., Bojarski *et al.* (2016) for autonomous driving, Amato *et al.* (2013) for medical diagnosis, Brown *et al.* (2020) for text generation, and Pan *et al.* (2023) for logical reasoning. Many domains have standard datasets for training and inference, e.g., in vision MNIST (LeCun *et al.*, 1989), CIFAR10 (Krizhevsky, 2009), and ImageNet (Deng *et al.*, 2009).

At the same time, recent research also points out key concerns: Ribeiro *et al.* (2016) discusses the issues of black-box model construction and non-interpretability; Szegedy *et al.* (2014) and Kurakin *et al.* (2017) discuss vulnerability against environmental and adversarial noise. Works that pointed out problems with standard training include Shafique *et al.* (2020) for robustness, Dwork *et al.* (2012) for fairness, and Sill (1997) for monotonicity. Tsipras *et al.* (2019) and Wong *et al.* (2021) point out problems with using a finite set of test inputs to ensure DNN safety during deployment. Many recent works, identify classes of slight adversarial perturbations that impact the DNN decisions (Madry *et al.*, 2017; Goodfellow *et al.*, 2014; Heo *et al.*, 2019).

For examples of local robustness to image rotations and its classification see, e.g., Balunovic *et al.* (2019). For examples of global properties in air traffic collision avoidance systems see, e.g., Katz *et al.* (2017), and in security vulnerability classification see, e.g., Chen *et al.* (2021). Beyond manual design, there is a growing line of work on automatically generating formal specifications for DNNs. These include Geng *et al.* (2022), Chaudhary *et al.* (2024b), Geng *et al.* (2024), and Jin *et al.* (2024).

Checking the safety of DNNs has been a very active area of research with many publications, primarily during inference and relying on white-box access to the model, such as Balunovic *et al.* (2019), Singh *et al.* (2019b), Zhang *et al.* (2018a), Singh *et al.* (2018), Singh *et al.* (2019d), Paulsen *et al.* (2020), Xu *et al.* (2021), Tran *et al.* (2019b), Wu *et al.* (2022b), Anderson *et al.* (2019), Katz *et al.* (2019), Singh *et al.* (2019a), Wong and Kolter (2018), Lan *et al.* (2022), Wang *et al.* (2018), Bunel *et al.* (2020), Wang *et al.* (2021), Ugare *et al.* (2022), Kabaha and Drachsler-Cohen (2022), Palma *et al.* (2021a), Dathathri *et al.* (2020), Munakata *et al.* (2023), Ranzato *et al.* (2021), Banerjee *et al.* (2024b), Banerjee *et al.* (2024a), and Zhou *et al.* (2024). Black-box DNN verifiers are based on collecting DNN output for inputs from $\varphi$ and providing probabilistic guarantees. These include Baluta *et al.* (2021), Webb *et al.* (2019), Chaudhary *et al.* (2024a), and Chaudhary *et al.* (2025).

Certified training leverages DNN verifiers during training obtaining models that have higher provability than those with standard training. Examples include Gowal *et al.* (2019), Mirman *et al.* (2018), Xu *et al.* (2020), Zhang *et al.* (2020), Shi *et al.* (2021), Yang *et al.* (2023), Müller *et al.* (2023a), Balunovic and Vechev (2020), and Hu *et al.* (2023b).

Numerous methods aim to provide transparency of DNNs. Standard methods include Ribeiro *et al.* (2016) and Wu *et al.* (2023) and Wong *et al.* (2021). Marques-Silva and Ignatiev (2022), Malfa *et al.* (2021), Ignatiev *et al.* (2019), Darwiche and Hirth (2020), and Wu *et al.* (2023) generate explanations with formal guarantees. The work of Banerjee *et al.* (2024a) presents ProFIt, the first method for interpreting robustness proofs computed by DNN verifiers.

Various uses of automatic differentiation are presented (Hückelheim *et al.*, 2018). Static analysis of AD computations is introduced by Laurel *et al.* (2022a), Laurel *et al.* (2022b), and Laurel *et al.* (2023). Verification of properties involving gradients and Jacobians are discussed by Zhang *et al.* (2019), Fazlyab *et al.* (2019b), and Shi *et al.* (2022)

Abstract interpretation was introduced in the seminal work by Cousot and Cousot (1977). Over the past almost 50 years, this approach to program analysis has flourished and demonstrated many uses. There are numerous books, monographs, and tutorials describing the foundations of abstract interpretations, for instance Cousot (2021), Nielson *et al.* (2005), Miné (2017), and Rival and Yi (2020).

Examples of abstract domains for neural networks include DeepPoly/CROWN (Singh *et al.*, 2019b; Zhang *et al.*, 2018a), DeepZ/Fast-Lin (Singh *et al.*, 2018; Weng *et al.*, 2018), Star sets (Tran *et al.*, 2019b), and DeepJ (Laurel *et al.*, 2022a). These custom solutions can scale to realistic DNNs with up to a million neurons (Müller *et al.*, 2021a), or more than 100 layers (Wu *et al.*, 2022b), verifying diverse safety properties in different real-world applications. Examples include autonomous driving (Yang *et al.*, 2023), job-scheduling (Wu *et al.*, 2022b), data center management (Chakravarthy *et al.*, 2022), biology (Mohr *et al.*, 2021), aerospace (Cohen *et al.*, 2024), and financial modeling (Laurel *et al.*, 2023). For examples of refinements of abstract domains used in machine learning see e.g., Wang *et al.* (2018), Singh *et al.* (2019d), Müller *et al.* (2021b), Ryou *et al.* (2021), Wang *et al.* (2021), Wu *et al.* (2022b), and Yang *et al.* (2021).

# 2

# Background

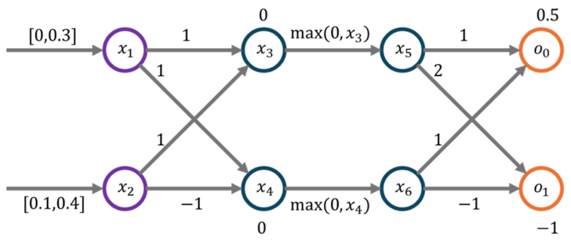
In this section, we introduce the necessary background for understanding our monograph.

## 2.1 Deep Neural Networks

Deterministic DNNs $f\colon \mathbb{R}^m \to \mathbb{R}^n$ are vector-valued functions that can be implemented using straight-line programs (i.e., without loops) of a certain form. A DNN composes a set of *layers* according to an *architecture* (e.g., residual, transformer), given by a directed acyclic graph (DAG). Each layer $f^i$ is one of the following: (i) an affine transformation $f^i(x) = Wx + b$ based on learned weights $W \in \mathbb{R}^{n_i \times n_{i-1}}$ and biases $b \in \mathbb{R}^{n_i}$ and (ii) the non-linear functions $f^i(x) = \sigma(x)$ applied component-wise. Common examples of $\sigma$ include the ReLU $(max(0, x))$, the sigmoid $(\frac{e^x}{e^x+1})$, and the tanh $(\frac{e^x-e^{-x}}{e^x+e^{-x}})$ activation functions. Stochastic DNNs $f\colon \mathbb{R}^m \to \mathcal{P}(\mathbb{R}^n)$ apply similar transformations as deterministic DNNs but map inputs to distributions $p \in \mathcal{P}(\mathbb{R}^n)$ over outputs. The stochasticity can be introduced through noise injection (Srivastava *et al.*, 2014), parameter sampling (Goan and Fookes, 2020), or latent variable sampling (Kingma and Welling, 2019).

**Neurons and activations.** During execution on a given input vector, each layer in the DNN receives a vector from its predecessor layer (as determined by its architecture), applies the layer-specific transformation (e.g., affine, ReLU) to obtain a new output vector, and passes the output to all its successors. Each component of one of the vectors passed along through the layers is called a *neuron* $x$, and its value $v \in \mathbb{R}$ is called an *activation*. There are three types of neurons: $m$ input neurons whose activations form the input to the network, $n$ output neurons whose activations form the output of the network, and all other neurons, called *hidden*, as they are not directly observed.

**Illustrative example.** Figure 2.1 (a) shows a simple, deterministic neural network $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ with sequential architecture. The network has three layers: two affine layers and one ReLU layer between the affine layers. Each layer has two neurons. Each affine layer performs two affine assignments based on learned weights (shown on the edges) and biases (shown above or below neurons). The output of one layer serves as the input to the next. The computations in the neural network can be written as straight-line code, as shown in Figure 2.1 (b). Note that there are multiple semantically equivalent program representations for the same program. For example, the assignments to $x_3, x_4$ in Figure 2.1 (b) can be interchanged without changing the semantics. In Figure 2.1 (a), for input $(x_1, x_2) = (0.2, 0.3)$, we obtain the hidden activation vectors $(x_3, x_4) = (0.5, -0.1)$, $(x_6, x_7) = (0.5, 0)$, and the output vector $(o_0, o_1) = (1, 0)$.



```
def f(x1,x2):
   x3:= x1 + x2
   x4:= x1 − x2
   x5:= max(0,x3)
   x6:= max(0,x4)
   o0:= x5 + x6 + 0.5
   o1:= 2 · x5 − x6 − 1
   return o0, o1
```

**Figure 2.1:** A deterministic DNN and its corresponding program representation.

**Discriminative DNNs.** Discriminative DNNs can be used for classification or regression. For a DNN $f$ that classifies its inputs to multiple possible labels, $n$ is the number of distinct classes, and the neural network classifies a given input $x$ to a given class $k$ if $f(x)_k > f(x)_j$ for all $j$ with $1 \leq j \leq n$ and $j \neq k$. The output of the regression model is simply $f(x)$.

**Generative DNNs.** A generator DNN $f$ takes a latent variable $z$ sampled from a prior distribution $p_z(z)$ and produces a sample $x \sim p_f(x \mid z)$, based on the learned parameters, in the data space $\mathbb{R}^m$, which could represent an image, a piece of text, or other data. The generator $f$ can be deterministic or stochastic. A generative model defines a probability distribution $p_f(x)$ over the data space by considering the generator's outputs and marginalizing over the latent space $\mathcal{Z}$, integrating over all possible latent variables.

$$p_f(x) = \int_{\mathcal{Z}} p_f(x \mid z) p_z(z) dz$$

## 2.2 Abstract Interpretation

Abstract interpretation computes an over approximation of the system (e.g., a program or a neural network) behaviors for a set of (potentially infinite) executions (Cousot and Cousot, 1977). It works via two domains: the concrete domain $(\mathcal{C}, \subseteq)$ and the abstract domain $(\mathcal{D}, \sqsubseteq)$. A DAG-based DNN can be translated into a semantically equivalent straight-line program where neurons are the program variables and each neuron is assigned at most once in any given DNN execution. We assume the neurons in a DNN are obtained from a sequence $\mathcal{X}$ where the input neurons appear first, then hidden, and finally the output neurons.

**Concrete domain.** A concrete domain $(\mathcal{C}, \subseteq)$ consists of a set $\mathcal{C}$ of *concrete elements* partially ordered by the *precision* relation $\subseteq$ and a set of *concrete transformers* that model the effect of applying different DNN statements on concrete elements. We next define concrete elements and concrete transformers in the context of DNN analysis:

**Definition 2.1.** (Store) A store $s \colon \mathcal{X} \to \mathbb{R}$ is a partial mapping from neurons $x \in \mathcal{X}$ to activations $v \in \mathbb{R}$.

We use $dom(s)$ to denote the domain of a store $s$.

**Definition 2.2.** (Store comparison) $s_1 \subseteq s_2$ iff $dom(s_1) \subseteq dom(s_2)$ and for all $x \in dom(s_1)$, $s_1(x) = s_2(x)$.

**Definition 2.3.** (Concrete element) A concrete element $C \in \mathcal{C}$ is a set of stores.

A concrete element $C$ for the DNN in Figure 2.1 (a) is $\{\{x_3 \mapsto 0, x_4 \mapsto 1\}, \{x_3 \mapsto 3, x_4 \mapsto -9\}\}$. Note that the domain for both stores in $C$ is $\{x_3, x_4\}$.

**Definition 2.4.** (Concrete comparison) $C \subseteq C'$ iff each store $s \in C$ is part of $C'$.

When $C \subseteq C'$ holds, we say that $C$ is more precise than $C'$. The set of all possible concrete elements in $\mathcal{C}$ typically forms a lattice $(\mathcal{C}, \subseteq, \cup, \cap, \top, \bot)$. Here, the least upper bound of two concrete elements is computed as set union $\cup$, and the greatest lower bound is computed via set intersection $\cap$. $\top$ is the top element in the lattice containing all possible stores for all neurons in $\mathcal{X}$. $\bot$ is the bottom element representing an empty set of stores.

**Definition 2.5.** (Concrete transformer) The output $T^{\#}(C)$ of a concrete transformer $T^{\#} : \mathcal{C} \to \mathcal{C}$ corresponding to a DNN statement (e.g., affine or ReLU assignment) applied on a concrete element $C \in \mathcal{C}$ contains all possible stores after applying the statement to all stores in $C$.

The concrete transformer for any ReLU assignment $y := max(0, x)$ (here $y, x$ are arbitrary neurons from $\mathcal{X}$) sets $y = 0$ for each store $s$ in $C$ where $s(x) \leq 0$ and sets $y = s(x)$ for stores where $s(x) > 0$. When considering $C = \{\{x_3 \mapsto 0, x_4 \mapsto 1\}, \{x_3 \mapsto 3, x_4 \mapsto -9\}\}$ and the ReLU assignment $x_6 := max(0, x_4)$, we get the output $\{\{x_3 \mapsto 0, x_4 \mapsto 1, x_6 \mapsto 1\}, \{x_3 \mapsto 3, x_4 \mapsto -9, x_6 \mapsto 0\}\}$.

**Abstract domain.** An abstract domain $(\mathcal{D}, \sqsubseteq)$ consists of a set $\mathcal{D}$ of *abstract elements* $D \in \mathcal{D}$ partially ordered by the relation $\sqsubseteq$ and a set of *abstract transformers* $T$ that model the effect of DNN statements on abstract elements.

The concrete and the abstract domains are connected by two functions, as shown in Figure 2.2. The monotonic concretization function $\gamma\colon \mathcal{D} \to \mathcal{C}$ for a given abstract element $D \in \mathcal{D}$ computes the concrete element $\gamma(D) \in \mathcal{C}$ containing all stores represented by $D$. The abstraction function $\alpha\colon \mathcal{C} \to \mathcal{D}$ for a concrete element $C \in \mathcal{C}$ computes an abstract element $\alpha(C) \in \mathcal{D}$ for the stores in $C$.

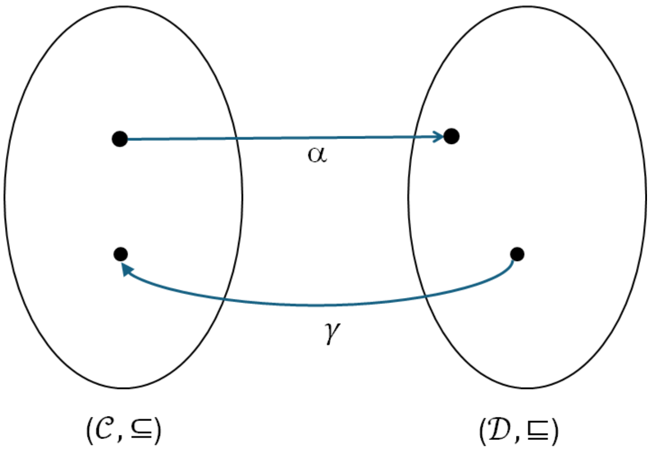


**Figure 2.2:** The concrete domain $(\mathcal{C}, \subseteq)$ and the abstract domain $(\mathcal{D}, \sqsubseteq)$ are connected by the abstraction $\alpha$ and monotonic concretization functions $\gamma$.

**Definition 2.6.** (Sound abstraction function) An abstraction function $\alpha\colon \mathcal{C} \to \mathcal{D}$ is sound iff for all concrete elements $C \in \mathcal{C}$, we have that $C \subseteq \gamma(\alpha(C))$ holds.

Soundness implies that all concrete stores in $C$ are captured by $\alpha(C)$. However, some stores captured by $\alpha(C)$ may not occur in $C$. As an example, consider the Box domain (Gehr *et al.*, 2018; Gowal *et al.*, 2018), which associates a lower bound $l$ and upper bound $u$ with each neuron $x$ such that $l \leq x \leq u$, written as $x \in [l, u]$. For $C = \{\{x_3 \mapsto 0, x_4 \mapsto 1\}, \{x_3 \mapsto 3, x_4 \mapsto -9\}\}$, $\alpha(C)$ in the Box domain yields $x_3 \in [0, 3]$ and $x_4 \in [-9, 1]$. Note that $\gamma(\alpha(C))$ contains all stores satisfying $0 \leq x_3 \leq 3$ and $-9 \leq x_4 \leq 1$ and therefore it contains extra stores like $\{x_3 \mapsto 1, x_4 \mapsto 0\}$ not present in $C$. Traditionally, $\alpha$ computes the smallest abstract element capturing the stores in $C$.

However, for abstract domains used for analyzing DNNs, the smallest element typically does not exist. Therefore, in this monograph, we only require that $\alpha$ computes a sound abstraction.

An abstract lattice $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ can be defined using the join ($\sqcup$) operator for computing the least upper bound of two abstract elements and the meet ($\sqcap$) operator for computing the greatest lower bound. Unlike programs, popular abstract domains for analyzing DNNs typically do not form a lattice.

**Definition 2.7.** (Precision) An abstract element $D \in \mathcal{D}$ is more precise compared to $D' \in \mathcal{D}$ iff $\gamma(D) \subseteq \gamma(D')$.

We next formally define a sound abstract transformer.

**Definition 2.8.** (Sound abstract transformer) A given abstract transformer $T$ is sound w.r.t to its concrete transformer $T^{\#}$ iff for all elements $D \in \mathcal{D}$, $T^{\#}(\gamma(D)) \subseteq \gamma(T(D))$.

Figure 2.3 visualizes Definition 2.8. A sound abstract transformer for the ReLU assignment $y := max(0, x)$ in the Box domain sets the lower bound of $y$ to $max(0, l)$ and upper bound to $max(0, u)$, keeping the intervals for other neurons the same. The soundness follows from the fact that the smallest and the largest values of $y$ from $T^{\#}(\gamma(D))$ cannot exceed these values. When considering $\alpha(C)$ and $x_6 := max(0, x_4)$, we obtain the interval $[0, 1]$ for $x_6$ while keeping the intervals for $x_3, x_4$ the same.

For abstract transformers $T_1, T_2$, since $T^{\#}(\gamma(D)) \subseteq \gamma(T_1(D))$ and $T^{\#}(\gamma(D)) \subseteq \gamma(T_2(D))$, we have that the intersection $\gamma(T_1(D)) \cap \gamma(T_2(D)) \supseteq T^{\#}(\gamma(D))$ is also sound. The composition of two sound abstract transformers is sound with respect to the composition of the corresponding concrete transformers.

**Definition 2.9.** (Completeness) We say an abstract domain $\mathcal{D}$ is *complete* for a concrete transformer $T^{\#}$ (e.g., affine, ReLU) iff it can be captured exactly in the domain, i.e., if there exists an abstract transformer $T$ corresponding to that concrete transformer such that for all abstract elements $D \in \mathcal{D}$, $T^{\#}(\gamma(D)) = \gamma(T(D))$.

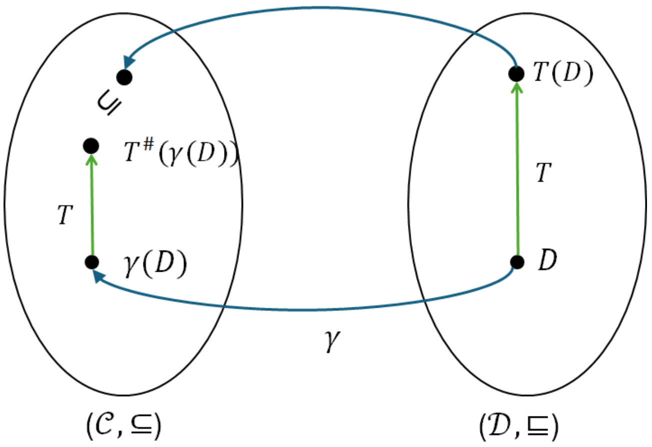


**Figure 2.3:** The concretization $\gamma(T(D))$ of the output of a sound abstract transformer $T$ is $\supseteq$ than the output of $T^{\#}$ operating on $\gamma(D)$.

The Box domain is complete for the ReLU assignment $y := max(0, x)$ when $|\mathcal{X}| = 2$, as the abstract transformer mentioned above satisfies $T^{\#}(\gamma(D)) = \gamma(T(D))$ for all $D \in \mathcal{D}$. For $|\mathcal{X}| > 2$, it is not complete.

A useful concept is that of a best abstract transformer.

**Definition 2.10.** (Best abstract transformer) An abstract transformer $T$ in $\mathcal{D}$ is best iff for any other sound abstract transformer $T'$ (corresponding to the same concrete transformer $T^{\#}$) it holds that for all elements $D \in \mathcal{D}$, $T$ always produces a more precise result (in the concrete), that is, $\gamma(T(D)) \subseteq \gamma(T'(D))$.

The composition of two best abstract transformers may not be the best transformer for the composition of the corresponding concrete transformers. For example, the composition of the best Box transformers for $x_5 := max(0, x_3)$ and $x_6 := max(0, x_4)$ does not yield the most precise output in the Box domain.

**Definition 2.11.** (Expressiveness of an abstract domain) An abstract domain $(\mathcal{D}, \sqsubseteq)$ is more expressive than another domain $(\mathcal{D}', \sqsubseteq')$ iff $\mathcal{D} \supseteq \mathcal{D}'$.

# 3

# Formal Verification of DNNs

This section describes how the classical abstract interpretation framework can be leveraged to design diverse state-of-the-art DNN verifiers. We focus on three broad categories of specifications, defined over an already trained DNN $f$, that capture a range of important properties.

- **Single Execution**: Requires that the output of each execution of a single deterministic DNN should satisfy a given postcondition. The set of executions is specified by a precondition. Examples include robustness, safety, and stability.
- **Relational**: Requires that the outputs of multiple executions (given by a precondition) of the same or different deterministic DNNs satisfy a given postcondition. In contrast to single execution properties, verifying relational properties requires capturing relationships between multiple related executions. Examples include fairness, monotonicity, DNN equivalence, and the safety of a DNN ensemble.
- **Probabilistic**: Typically require that the outputs of executions (given by a precondition) of a given deterministic or stochastic

DNN satisfy a given postcondition with high probability. Probabilistic specifications are useful for reasoning about stochastic models such as variational autoencoders (VAEs). For deterministic models, these specifications quantify undesirable behavior.

The design of preconditions and postconditions depends highly on the application within which the DNN is used. We will also discuss incremental verification to improve scalability when verifying multiple models and properties. While we focus on the verification of DNNs, the DNN verifiers discussed in this monograph can be combined with approaches for program/system verification to provide end-to-end safety guarantees of AI-enabled systems (Tran *et al.*, 2020b; Yang and Chaudhuri, 2022; Habeeb *et al.*, 2024; Mitra *et al.*, 2024). Next, we discuss the verification of single execution properties.

## 3.1 Single Execution Properties

We formally define the verification problem for single execution properties, which is parameterized by two concrete elements $C_\varphi, C_\psi \in \mathcal{C}$. $C_\varphi$ is used to specify the set of inputs where the DNN should not misbehave. The desired behavior is captured by $C_\psi$. Let $x \in \mathbb{R}^m$ be the vector containing the activations of the input neurons for a store $s \in C_\varphi$. The problem involves checking for each $x$ if the store corresponding to the activation vector $f(x) \in \mathbb{R}^n$ for the output neurons is $\subseteq s'$ where $s'$ is a store in $C_\psi$.

$C_\varphi$ and $C_\psi$ can contain an infinite number of stores, and therefore, it is not possible to solve the problem by simply running the DNN for each $x$. Instead of set representation, the verification algorithms work with the symbolic representations $\varphi, \psi$ of $C_\varphi, C_\psi$. $\varphi, \psi$ are typically represented as a disjunction of conjunctions of linear constraints over the input and output neurons, respectively (we will discuss handling of an atypical case). Geometrically, the constraints describe the union of convex polyhedra. We say an input activation vector $x \in \mathbb{R}^m$ satisfies $\varphi$, written as $x \in \varphi$, iff it satisfies all constraints in $\varphi$. The satisfaction of $\psi$ by $f(x) \in \mathbb{R}^n$ written as $f(x) \in \psi$ can be defined similarly. We use $f(\varphi)$ to describe the symbolic representation of the set of DNN outputs corresponding to the inputs $x$ in $\varphi$.

A DNN verifier checks if $f(\varphi) \subseteq \psi$ holds. If it can prove the property, then it produces an independently checkable certificate. If it can disprove the property, then it generates a counter-example $x \in \varphi$ such that $f(x) \notin \psi$. It returns unknown if it cannot prove or disprove a property.

### 3.1.1 Representative Properties

Next, we describe some representative single execution properties that have been verified in the literature. For robustness properties, $y$ denotes the correct class.

- **Robustness.** Given a correctly classified input $x$, the local robustness property requires that all inputs obtained after applying a set of transformations are classified correctly. Formally, $\varphi := \{x' \mid x' \in T(x), T \in \mathcal{T}\}$ and $\psi := f(x')_y - f(x')_{i \neq y} > 0$ where $\mathcal{T}$ is the set of transformations. Examples of $\mathcal{T}$ include changes in pixel intensity (Gehr *et al.*, 2018; Singh *et al.*, 2019b), geometric transformations (e.g., rotations, translations) (Balunovic *et al.*, 2019; Yang *et al.*, 2023; Mohapatra *et al.*, 2020) applied to images, and string transformations (e.g., deletions, insertions) (Jia *et al.*, 2019; Zhang *et al.*, 2021b) applied to an input text. The local robustness can be extended to the global robustness property by universal quantification over the input $x$ (Kabaha and Drachsler-Cohen, 2024; Yang and Rinard, 2019).

- **Safety.** This is a global property requiring that the DNN outputs $f(x)$ corresponding to each input $x$ in the precondition $\phi$ satisfy the postcondition $\psi$ capturing safe behaviors (Katz *et al.*, 2017; Wang *et al.*, 2018). Here $\phi$ and $\psi$ respectively describe disjunctions of polyhedra over the input and output spaces of $f$.

- **Stability.** This property can be defined both locally and globally. In the local version, given an input $x$, the output of the DNN corresponding to each input $x'$ within a local neighborhood parameterized by $\epsilon \in \mathbb{R}$ should not deviate from $f(x)$ by more than a constant $c \in \mathbb{R}$ (Zhang *et al.*, 2019; Jordan and Dimakis, 2020). The distances in the input and output spaces are typically measured using norm functions $d_\phi$ and $d_\psi$ respectively. $\phi := \{x' \mid d_\phi(x, x') \leq \epsilon\}$ and $\psi := d_\psi(f(x), f(x')) \leq c$. The global property is defined by universal quantification over $x$ (Chen *et al.*, 2021).

Many of these properties can be written as a combination of a pre-defined computation graph (represented as an ONNX file) and a specification file in VNNLIB format (Demarchi *et al.*, 2023). The VNNLIB specification typically defines a precondition $\phi$ represented by element-wise constraints on $x$, and a postcondition $\psi$ represented by a logical formula over the outputs of the computation graph. The standardized benchmarks from the Verification of Neural Network Competitions (VNN-COMPs) include many properties in this general form, arising from different applications such as computer vision, computer systems (Lin *et al.*, 2024), control (Yang *et al.*, 2024b), aerospace, and power systems (Chevalier *et al.*, 2023). The details of these benchmarks are available in VNN-COMP reports (Brix *et al.*, 2024a; Brix *et al.*, 2023; Müller *et al.*, 2022; Bak *et al.*, 2021), which can serve as a starting point for readers who want to apply neural network verification to their domain-specific problems. Readers can also find the latest performance reports of practical verification toolboxes, such as $\alpha$,$\beta$-CROWN, ERAN, Marabou, and PyRAT, in the VNN-COMP reports.

### 3.1.2 Verification Algorithms

A diverse set of verifiers exist for single execution properties. We broadly classify them based on their completeness guarantees:
**Complete vs. incomplete verifiers.** Complete verifiers can provide exact answers: they either prove the property or provide a counterexample. In contrast, an incomplete verifier may be indecisive (returns "unknown") on some properties. Incomplete verifiers, such as those based on abstract interpretation, have been developed to scale to large neural networks. On the other hand, complete verification is an undecidable problem for general neural network architectures and arbitrary property specifications (Ivanov *et al.*, 2019). In cases when completeness is possible, complete verifiers can be built upon incomplete verifiers with techniques such as branch-and-bound (Bunel *et al.*, 2020; Wang *et al.*,

2021; Ferrari *et al.*, 2022), and they may be practically used with a timeout to verify as many properties as possible. Incomplete verifiers excel in situations where efficiency and scalability are the key concerns, while complete verifiers enable us to improve the precision of incomplete verifiers with different techniques, such as iterative branch-and-bound (Bunel *et al.*, 2020) and refinement (Singh *et al.*, 2019c). In this monograph, we focus on incomplete verifiers as they are more scalable and applicable to verifying general architectures and properties.

**Abstract interpreters for DNNs.** Figure 3.1 shows the high-level idea behind abstract interpretation-based DNN verifiers. The verifier first runs a *verification analysis*, with a chosen abstract domain $\mathcal{D}$, to compute an overapproximation of $f(\varphi)$. The analysis starts by computing an abstract element $\alpha(\varphi)$ using a sound abstraction function $\alpha$. The soundness of $\alpha$ ensures that $\gamma(\alpha(\varphi)) \supseteq \varphi$ (Definition 2.6). Next, the analyzer symbolically propagates $\alpha(\varphi)$ through the different layers of the network. At each layer, the analyzer computes an abstract element (in blue) overapproximating the exact layer output (in white) corresponding to $\varphi$ computed by a concrete transformer $T^{\#}$. The formula is computed by applying a sound abstract transformer $T$, overapproximating the effect of the operations in the layer, on the abstract element input to the layer. $T$ can be obtained by combining the abstract transformers for individual statements within the layer or designed to handle all individual statements jointly. The propagation yields a sound approximation $\gamma(g(\alpha(\varphi))) \supseteq f(\varphi)$ at the output layer. Next, the verifier checks if $\gamma(g(\alpha(\varphi))) \subseteq \psi$ holds for the bigger region $\gamma(g(\alpha(\varphi)))$ by either calling an off-the-shelf solver (Salman *et al.*, 2019; Wang *et al.*, 2018) or using custom approximations (Zhang *et al.*, 2018a; Weng *et al.*, 2018; Boopathy *et al.*, 2019; Singh *et al.*, 2019b; Singh *et al.*, 2018). If the answer is yes, then $f(\varphi) \subseteq \psi$ also holds for the smaller region $f(\varphi)$. Because of the overapproximation, it can be the case that $\gamma(g(\alpha(\varphi))) \subseteq \psi$ does not hold while $f(\varphi) \subseteq \psi$ holds. To reduce the amount of overapproximation, refinements (Wang *et al.*, 2018; Singh *et al.*, 2019c; Müller *et al.*, 2021b; Ryou *et al.*, 2021; Lyu *et al.*, 2020; Wang *et al.*, 2021) can be applied.

To obtain an effective verifier, it is essential to design an analysis such that the concretization of $g(\alpha(\varphi))$ is as close as possible to the true output $f(\varphi)$ while $g$ can also be computed in a reasonable amount

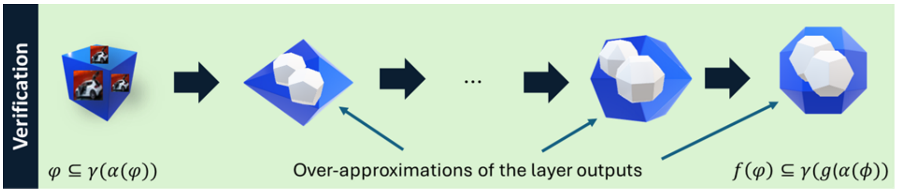

Verification
...
$\varphi \subseteq \gamma(\alpha(\varphi))$
Over-approximations of the layer outputs
$f(\varphi) \subseteq \gamma(g(\alpha(\phi)))$

**Figure 3.1:** An illustration of the workings of neural network verifiers based on abstract interpretation.

of time for practical networks. The cost and precision of $g$ depend upon the expressivity of the abstract domains and the cost of abstract transformers for different layers and preconditions. For scalability, state-of-the-art analyzers restrict the abstract domains to describe a convex shape, e.g., Zonotope (Anderson *et al.*, 2019; Singh *et al.*, 2018; Gehr *et al.*, 2018; Goubault and Putot, 2022; Goubault and Putot, 2024), Star sets (Tran *et al.*, 2020b), Polyhedral (Gehr *et al.*, 2018; Singh *et al.*, 2019b; Zhang *et al.*, 2018a), etc. Note that the verifier does not require $\varphi, \psi$ to have the same shape as supported by the analyzer ($\varphi$ is approximated using $\alpha(\varphi)$ and $\psi$ is handled exactly via a solver or approximated).

### 3.1.3 Abstract Domains for Analyzing DNNs

Various analyzers exist based on the abstract domain they support, which we describe next. Traditional relational abstract domains useful for analyzing programs such as Octagons (Miné, 2006), Zones (Miné, 2002), TVPI (Simon and King, 2010), etc., have not proven useful for DNNs as they are as imprecise for affine assignments as simpler domains like Box while being more expensive. As a result, specialized abstractions have been designed for DNNs that can efficiently handle affine transformations and non-linearities. For a given domain, the precision and cost of the analysis are not only determined by the size of the DNN but also by the complexity of the specification and the way the DNN is trained. It is possible that a given domain proves a specification on a large DNN but fails to prove it on a smaller DNN. Next, we describe four popular abstract domains for DNN verification and compare their expressivity and precision on an illustrative example.

**Example network.** We focus on the simple fully connected neural network with ReLU activation shown in Figure 3.2. This network has already been trained, and we have the learned weights shown in the figure. The network has two affine layers and one ReLU layer between the affine layers. The weights on the edges represent the learned coefficients of the weight matrix that the affine layers use. The learned bias for each neuron is shown above or below it. All of the biases in one layer constitute the translation vector of the affine transformation.

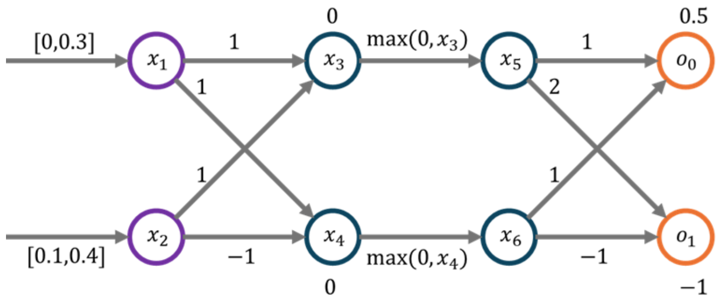
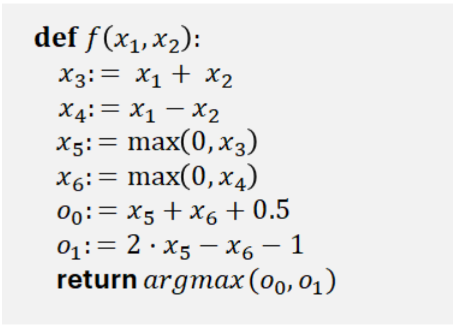


**Figure 3.2:** Example fully connected neural network with ReLU activations.

To compute its output, each neuron $\in \{x_3, x_4\}$ in the first affine layer applies an affine transformation based on the weight matrix and bias to its inputs $x_1, x_2$, producing a value $v$. Then, an activation function is applied to $v$, in our example, ReLU, which outputs $v$, if $v > 0$, and 0 otherwise. In the last layer, a final affine transform is applied to yield the output of the entire network, typically a class label that describes how the input is classified. The predicted class is usually the one with the highest value. The computations in the neural network can be written as straight-line code, as shown in Figure 3.3.

**Specification.** We consider local robustness specification defined around the point $(0.3, 0.4)$. This point is originally classified as the class $o_0$, as for this input, the value at $o_0$ is higher. Next, we define a set of transformations that can perturb the original point by adding or subtracting at most 0.3 to the original input in both dimensions. Mathematically, this defines an $L_\infty$ ball of radius 0.3 around the original point. This yields the precondition $\varphi$ where both $x_1, x_2$ can take any



```
def f(x_1, x_2):
  x_3 := x_1 + x_2
  x_4 := x_1 − x_2
  x_5 := max(0, x_3)
  x_6 := max(0, x_4)
  o_0 := x_5 + x_6 + 0.5
  o_1 := 2 · x_5 − x_6 − 1
  return argmax(o_0, o_1)
```

**Figure 3.3:** Straight-line code representing the computations in the neural network of Figure 3.2 for a concrete input.

values in the intervals $[0, 0.6]$ and $[0.1, 0.7]$ respectively. Our goal will be to check whether the network's output is the same as for the original point, for any possible inputs in $\varphi$. If the proof is successful, it implies that the network is robust and produces the same classification label for all of these inputs.

**Box.** This is the simplest abstract domain that associates an interval $[l, u]$, $l, u \in \mathbb{R}$ with each neuron $x$ in the DNN (Pulina and Tacchella, 2010; Gehr *et al.*, 2018). Geometrically, an n-dimensional abstract element in this domain represents a hyperbox shape in $\mathbb{R}^n$. Figure 3.4 (a) visualizes an abstract shape for two variables. This domain can exactly represent the precondition $\varphi$ from our specification. Therefore $\gamma(\alpha(\varphi)) = \varphi$ for our example. The analysis starts by associating the intervals $[0, 0.6]$ and $[0.1, 0.7]$ with $x_1$ and $x_2$ respectively. Next, it computes the intervals for $x_3$ and $x_4$ by applying the Box abstract transformer for the affine assignments $x_3 := x_1 + x_2$ and $x_4 := x_1 - x_2$, respectively. There is an infinite number of Box transformers for these assignments. We consider the best transformer for each assignment that computes the tightest interval bounds. This transformer can be constructed by composing two types of transformers: interval addition and multiplication by a scalar $\lambda \in \mathbb{R}$ shown in Figure 3.5. First, each coefficient in the affine assignment is multiplied to the interval for the corresponding variable using the multiplication transformer. Then, all intervals are added together using interval addition. For $x_3$, the

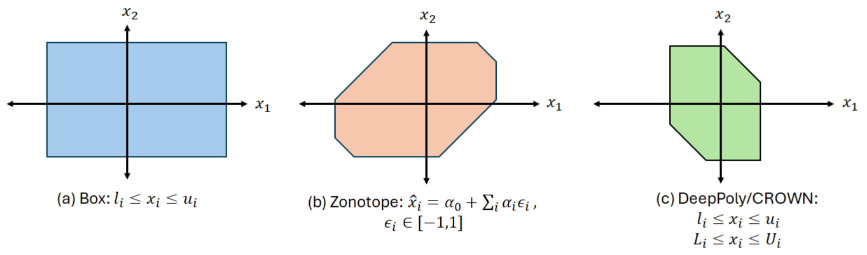
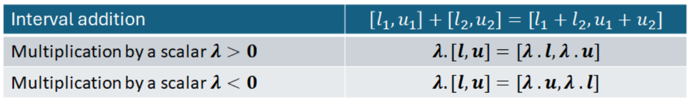

$x_2$
$x_1$
(a) Box: $l_i \leq x_i \leq u_i$
(b) Zonotope: $\hat{x}_i = \alpha_0 + \sum_i \alpha_i \epsilon_i$, $\epsilon_i \in [-1,1]$
(c) DeepPoly/CROWN: $l_i \leq x_i \leq u_i$ $L_i \leq x_i \leq U_i$

**Figure 3.4:** Popular abstract shapes used for verifying DNNs



| Interval addition | $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$ |
|---|---|
| Multiplication by a scalar $\boldsymbol{\lambda} > \mathbf{0}$ | $\boldsymbol{\lambda}.[\boldsymbol{l}, \boldsymbol{u}] = [\boldsymbol{\lambda}.\boldsymbol{l}, \boldsymbol{\lambda}.\boldsymbol{u}]$ |
| Multiplication by a scalar $\boldsymbol{\lambda} < \mathbf{0}$ | $\boldsymbol{\lambda}.[\boldsymbol{l}, \boldsymbol{u}] = [\boldsymbol{\lambda}.\boldsymbol{u}, \boldsymbol{\lambda}.\boldsymbol{l}]$ |

**Figure 3.5:** Abstract transformers for interval addition and scalar multiplication.

output is computed as $[0.1, 1.3] = [0, 0.6] + [0.1, 0.7]$. For $x_4$, the output is $[-0.7, 0.5] = [0, 0.6] + (-1 * [0.1, 0.7])$. For each affine assignment, the best abstract transformer has linear complexity in the number of neurons on the right-hand side (RHS) of the assignment expression. The composition of the best Box transformers for the individual assignments yields the best abstract transformer for the full affine layer. Both affine assignments can be handled independently; therefore, their composition can be easily parallelized.

Even though the best affine transformer computes the tightest box, it is not exact and, therefore, loses precision. For example, the point $(x_3, x_4) = (1.3, 0.5)$ is part of the output box but not feasible in any concrete execution. This is because the Box domain does not track dependencies between neurons caused by the affine assignments where the same variables are used on the RHS of the assignment expressions for $x_3$ and $x_4$.

Next, the analysis computes the intervals for $x_5$ and $x_6$ by applying the ReLU abstract transformer. The best output interval for the ReLU assignment $y := ReLU(x)$ can be computed as $[ReLU(l), ReLU(u)]$. The transformer is visualized in Figure 3.6 for the case when $l < 0 < u$. This computation can be done in constant time and efficiently parallelized across neurons in a layer. The best ReLU transformer for

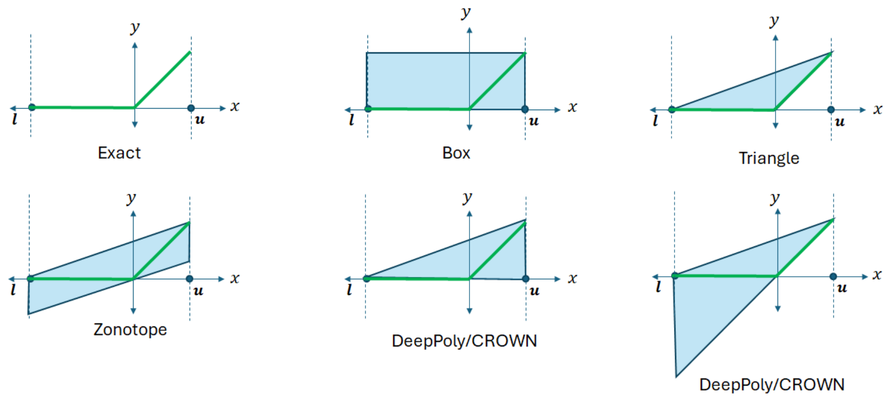


**Figure 3.6:** Approximations of the popular ReLU activations $y := ReLU(x)$ with different domains. We consider the most challenging case $l < 0 < u$, where the exact output is piecewise-linear (shown in green in all figures).

the layer can be obtained by composing the best transformers for the individual ReLU using the *Single Neuron* construction defined in Section 3.1.4. Similar to affine layers, the best abstract transformer for the ReLU layers is not exact. Other popular non-linear operations used in DNNs, such as Tanh, Sigmoid, etc., can be handled similarly in constant time by the Box domain. For our example, we get the intervals $[0.1, 1.3]$ and $[0, 0.5]$ for $x_5$ and $x_6$ respectively. Finally, the best affine transformer is applied again to compute the intervals $[0.6, 2.3]$ and $[-1.3, 1.6]$ for $o_0$ and $o_1$, respectively. The output $g(\alpha(\varphi))$ of the Box analysis for each neuron is shown in Figure 3.7.

Now, we need to check whether the postcondition $o_0 > o_1$ holds on the output $g(\alpha(\varphi))$. This can be done symbolically without concretizing $g(\alpha(\varphi))$. Since there is no dependency between the intervals for different neurons, we only need to look at the intervals for $o_0$ and $o_1$. To determine whether the constraint holds, the analysis checks whether the lower bound of $o_0$ is $>$ than the upper bound for $o_1$. This is not the case for our example, so the analysis fails to prove our specification. Note that while the Box domain may appear quite imprecise, it is not useless. If the size of the precondition was smaller (e.g., if the $L_\infty$ ball had a radius of 0.15), then Box analysis could prove the specification. As we will

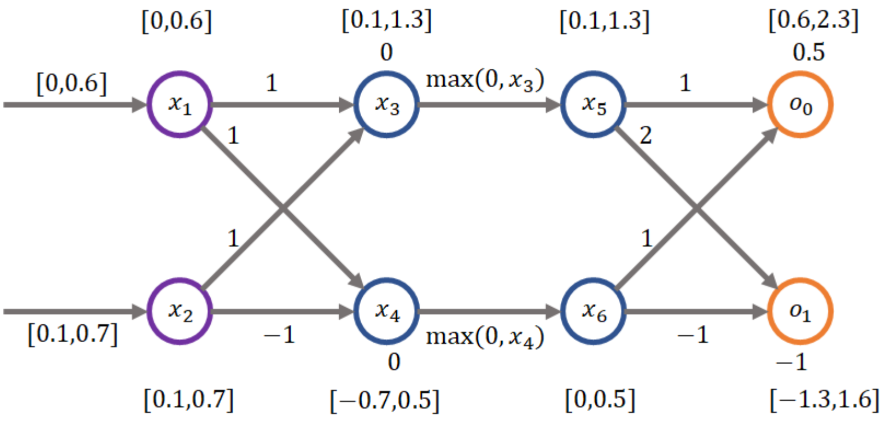
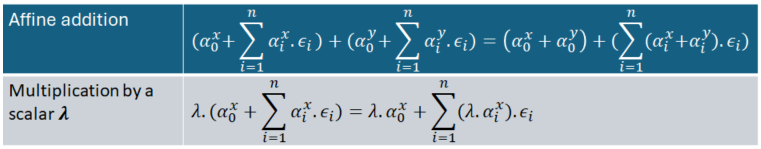


**Figure 3.7:** Box analysis results on the example DNN.

see in the next section, Box analysis is quite useful for training DNNs to satisfy specifications and can achieve high precision when verifying complex specifications on larger DNNs trained in this way. Since Box analysis is cheap, many verifiers often first run the Box analysis to avoid running more expensive analyzers for easier specifications.

**Zonotope.** The Zonotope domain associates an affine expression $\hat{x} = \alpha_0 + \sum_{i=1}^{r} \alpha_i \cdot \epsilon_i$ with each neuron $x$ (Ghorbal *et al.*, 2009). Here the expression consists of a center coefficient $\alpha_0 \in \mathbb{R}$, a set of noise symbols $\epsilon_i \in [-1, 1]$ , and coefficients $\alpha_i \in \mathbb{R}$ representing partial deviations around the center. Geometrically, an n-dimensional zonotope describes a center-symmetric polytope in $\mathbb{R}^n$. Figure 3.4 (b) visualizes a Zonotope shape in 2-D. The Zonotope domain is more expressive than Box and can capture linear dependencies between the neurons. The affine form for an interval $[l, u]$ can be computed as $\frac{l+u}{2} + \frac{u-l}{2} \cdot \epsilon_i$, where $\epsilon_i$ is a new noise symbol. Therefore we get the affine forms $\hat{x_1} = 0.3 + 0.3 \cdot \epsilon_1$ and $\hat{x_2} = 0.4 + 0.3 \cdot \epsilon_2$ from $\varphi$. As for the Box domain, we have $\gamma(\alpha(\varphi)) = \varphi$. Next, we apply the best affine transformer for the Zonotope domain, which can be constructed by composing the transformers for affine addition and multiplication of affine expressions by a scalar shown in Figure 3.8. This yields



| | |
|---|---|
| Affine addition | $(\alpha_0^x + \sum_{i=1}^{n} \alpha_i^x.\epsilon_i) + (\alpha_0^y + \sum_{i=1}^{n} \alpha_i^y.\epsilon_i) = (\alpha_0^x + \alpha_0^y) + (\sum_{i=1}^{n} (\alpha_i^x + \alpha_i^y).\epsilon_i)$ |
| Multiplication by a scalar $\boldsymbol{\lambda}$ | $\lambda.(\alpha_0^x + \sum_{i=1}^{n} \alpha_i^x.\epsilon_i) = \lambda.\alpha_0^x + \sum_{i=1}^{n} (\lambda.\alpha_i^x).\epsilon_i$ |

**Figure 3.8:** Abstract transformers for addition and scalar multiplication in the Zonotope domain.

$$\begin{aligned}
\hat{x_3} &= \hat{x_1} + \hat{x_2} = (0.3 + 0.3 \cdot \epsilon_1) + (0.4 + 0.3 \cdot \epsilon_2) \\
&= 0.7 + 0.3 \cdot \epsilon_1 + 0.4 \cdot \epsilon_2 \\
\hat{x_4} &= \hat{x_1} + (-1 \cdot \hat{x_2}) = (0.3 + 0.3 \cdot \epsilon_1) + (-0.4 - 0.3 \cdot \epsilon_2) \\
&= -0.1 + 0.3 \cdot \epsilon_1 - 0.3 \cdot \epsilon_2
\end{aligned}$$

$\hat{x_3}$ and $\hat{x_4}$ share the same noise variables, capturing dependency due to the affine assignments. Unlike the corresponding Box transformer, the best Zonotope transformer is exact for affine assignments. Its complexity is linear in the number of variables in the RHS of the assignment and the maximum number of noise terms in an affine form for the RHS variables. The composition of the exact transformers for the individual assignments preserves the exactness when computing the output for the full layer. For different neurons in a layer, the transformers can be applied, and therefore, their composition can be efficiently parallelized.

Next, we need to handle the ReLU layer. We will describe ReLU transformers for individual assignments. The layerwise Zonotope ReLU transformer can be obtained by combining the transformers for individual neurons using the Single Neuron construction formalized in Section 3.1.4. For $y := ReLU(x)$, we consider three cases depending on the interval $[l, u]$ for $x$. This interval can be easily extracted from $\hat{x}$ as $[l, u] = [\alpha_0 - \sum_{i=1}^{r} |\alpha_i|, \alpha_0 + \sum_{i=1}^{r} |\alpha_i|]$. If the lower bound $l > 0$, then $y = x$, and therefor, we we can set $\hat{y} = \hat{x}$. The complexity of this operation is linear in the number of noise symbols $r$. If $u \leq 0$, then $y = 0$ and we set $\hat{y} = 0$ in constant time. The Zonotope output is exact in both of these cases. For our example, the interval bound for $x_3$ is $[0.1, 1.3]$ and therefore we get $\hat{x_5} = \hat{x_3} = 0.7 + 0.3 \cdot \epsilon_1 + 0.3 \cdot \epsilon_2$. The interval bounds $[-0.7, 0.5]$ for $x_4$ represent the challenging case of $l < 0 < u$. Here the exact output shown in Figure 3.6 cannot be exactly represented as a Zonotope. Unlike popular abstract domains used in program analysis like Octagons, TVPI, etc., Zonotopes are not closed under intersection. As a result, there is no best Zonotope ReLU transformer for this challenging case (otherwise, the best transformer can be constructed by intersecting the output of all other transformers). Therefore, various heuristic-guided abstract transformers have been designed in the literature. Figure 3.6 shows a popular abstract transformer that heuristically minimizes the area of the Zonotope approximation in the $xy$-plane (Singh *et al.*, 2018). The affine form for $y$ depends only on the affine form of $x$ and therefore the computation can be easily parallelized across neurons in a layer. The complexity of this transformer is linear in $r$. The output is computed as $\hat{y} = \omega \cdot \hat{x} + \mu/2 + \mu/2 \cdot \epsilon_n$ where $\epsilon_n$ is a new noise symbol and $\omega = \frac{u}{u-l}$ and $\mu = \frac{-l \cdot u}{u-l}$. For our example, $\omega = \frac{5}{12}$ and $\mu = \frac{3.5}{12}$. This yields

$$\begin{aligned}\hat{x_6} &= \omega \cdot \hat{x_4} + \mu/2 + \mu/2 \cdot \epsilon_3 \\ &= \frac{5}{12} \cdot (-0.1 + 0.3 \cdot \epsilon_1 - 0.3 \cdot \epsilon_2) + \frac{1.75}{12} + \frac{1.75}{12} \cdot \epsilon_3 \\ &= \frac{1.25}{12} + \frac{1.5}{12} \cdot \epsilon_1 + \frac{-1.5}{12} \cdot \epsilon_2 + \frac{1.75}{12} \cdot \epsilon_3\end{aligned}$$

Theoretically, this abstract transformer is not more precise than the one from Box but yields more precise verification results in practice due to capturing the dependence between $y$ and $x$. Similar transformers computing the output of the form $\hat{y} = \omega \cdot \hat{x} + \mu/2 + \mu/2 \cdot \epsilon_n$ can be constructed for handling other common activations like Tanh, Sigmoid, etc., where $\omega, \mu$ depend on the activation. We refer the reader to Singh *et al.* (2018) and Bonaert *et al.* (2021) for more details. Notice that the precision of the transformer depends on the quality of the interval bounds. As we shall see in Section 3.1.5, the interval bounds can be refined by using a more precise analysis to improve the precision of the Zonotope analysis.

Finally, we can apply the affine transformer for the next layer to compute the output affine form for $o_0$. The Single Neuron construction ensures that we can soundly use the affine forms for $x_5, x_6$ computed above to obtain:

$$\begin{aligned}\hat{o_0} =& \hat{x_5} + \hat{x_6} + 0.5\\ =&(0.7 + 0.3 \cdot \epsilon_1 + 0.3 \cdot \epsilon_2)+\\ &(\frac{1.25}{12} + \frac{1.5}{12} \cdot \epsilon_1 + \frac{-1.5}{12} \cdot \epsilon_2 + \frac{1.75}{12} \cdot \epsilon_3) + 0.5\\ =&\frac{15.65}{12} + \frac{5.1}{12} \cdot \epsilon_1 + \frac{2.1}{12} \cdot \epsilon_2 + \frac{1.75}{12} \cdot \epsilon_3\end{aligned}$$

Notice that the coefficients for $\epsilon_2$ in the affine forms $\hat{x_5}$ and $\hat{x_6}$ have opposite signs. When we add these affine forms to compute $\hat{o_0}$, there is cancellation between the coefficients which improves the upper bound compared to the Box analysis yielding $[0.56, 2.05]$ and contributes to better analysis precision. Similar cancellation occurs when computing $o_1$ yielding more precise bounds $[-1.05, 1.64]$. The output $g(\alpha(\varphi))$ from the Zonotope analysis is shown in Figure 3.9.

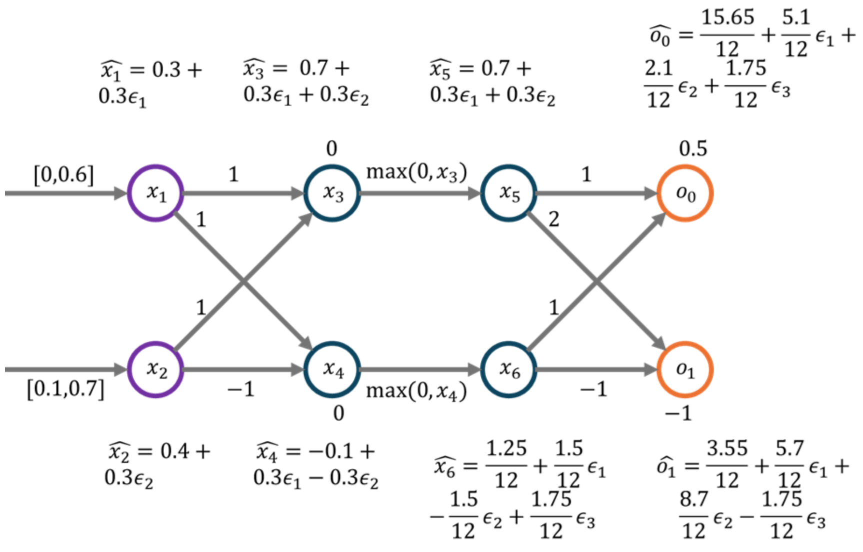

$\widehat{x_1} = 0.3 + 0.3\epsilon_1$
$\widehat{x_3} = 0.7 + 0.3\epsilon_1 + 0.3\epsilon_2$
$\widehat{x_5} = 0.7 + 0.3\epsilon_1 + 0.3\epsilon_2$
$\widehat{o_0} = \frac{15.65}{12} + \frac{5.1}{12}\epsilon_1 + \frac{2.1}{12}\epsilon_2 + \frac{1.75}{12}\epsilon_3$
[0,0.6]
[0.1,0.7]
$x_1$
$x_2$
$x_3$
$x_4$
$x_5$
$x_6$
$o_0$
$o_1$
1
1
1
−1
0
0
max(0,$x_3$)
max(0,$x_4$)
1
2
1
−1
0.5
−1
$\widehat{x_2} = 0.4 + 0.3\epsilon_2$
$\widehat{x_4} = -0.1 + 0.3\epsilon_1 - 0.3\epsilon_2$
$\widehat{x_6} = \frac{1.25}{12} + \frac{1.5}{12}\epsilon_1 - \frac{1.5}{12}\epsilon_2 + \frac{1.75}{12}\epsilon_3$
$\widehat{o_1} = \frac{3.55}{12} + \frac{5.7}{12}\epsilon_1 + \frac{8.7}{12}\epsilon_2 - \frac{1.75}{12}\epsilon_3$

**Figure 3.9:** Zonotope analysis results on the example DNN.

A straightforward way to check whether the specification holds is to compare the interval bounds for $o_0, o_1$ as for the Box analysis. However, this is not sufficient to prove the property. Instead, we can compute the affine form for $o_0 - o_1$ and then check whether its lower bound is $> 0$. Computing this, we get the affine form $\frac{12.1}{12} + \frac{-0.6}{12} \cdot \epsilon_1 + \frac{-6.6}{12} \cdot \epsilon_2 + \frac{3.5}{12} \cdot \epsilon_3$ with the lower bound 0.11 which is sufficient to prove our specification.

**DeepPoly/CROWN.** The DeepPoly/CROWN abstract domain associates four constraints with each neuron (Singh *et al.*, 2019b; Zhang *et al.*, 2018a). Out of these, two are box constraints $l \leq x \leq u$, and two are polyhedral $L \leq x \leq U$ where both $L, U$ are linear expressions of the form $\sum_{i=1}^{m} a_i \cdot x_i + b$ with $a_i, b \in \mathbb{R}$. Figure 3.4 (c) shows a DeepPoly/CROWN shape in 2-D. The polyhedral bounds enable DeepPoly to capture dependencies between the neurons while interval bounds are used for efficiently computing precise approximations of non-linear activations. Since this domain tracks interval bounds, it can represent $\varphi$ exactly setting $0 \leq x_1 \leq 0.6$ and $0.1 \leq x_2 \leq 0.7$ (the polyhedral bounds are the same as interval bounds in this case). Next, the affine assignments are handled exactly and in a parallelizable manner by adding the polyhedral bounds $x_1 + x_2 \leq x_3 \leq x_1 + x_2$ and $x_1 - x_2 \leq x_4 \leq x_1 - x_2$. This has linear complexity in the number of variables in the RHS of the affine assignment. To compute interval bounds, we can use interval affine transformers. This yields $0.1 \leq x_3 \leq 1.3$ and $-0.7 \leq x_4 \leq 0.5$.

As for the Zonotope domain, DeepPoly can exactly handle the ReLU assignment $y := ReLU(x)$ when $l > 0$ or $u \leq 0$. For the former, it adds the polyhedral bounds $x \leq y \leq x$ and interval bounds $l \leq y \leq u$. For the latter, it sets all bounds to 0. Since the lower bounds for $x_3$ is $> 0$ we can compute $x_3 \leq x_5 \leq x_3$ and $0 \leq x_5 \leq 0.6$. Like the Zonotope domain, the DeepPoly domain cannot exactly capture the ReLU output when $l < 0 < u$, as is the case for computing the bounds for $x_6$. DeepPoly is also not closed under intersection and, therefore, does not have a best transformer for ReLU. As a result, specialized transformers have been designed based on various heuristics (Weng *et al.*, 2018; Wong and Kolter, 2018; Singh *et al.*, 2019b; Zhang *et al.*, 2018a; Xu *et al.*, 2021). Figure 3.6 shows two parallelizable approximations that can minimize the area in the $xy$-plane. Both are incomparable with the Zonotope approximation we used previously. Note that the Zonotope approximation can be captured by the DeepPoly domain. However, in general, the two domains are incomparable, i.e., there are DeepPoly shapes that are not Zonotopes and vice-versa. However, in practice, the DeepPoly domain often provides more precise results.

In Figure 3.6, the first DeepPoly/CROWN transformer adds the constraints $0 \leq y \leq u, 0 \leq y \leq \lambda \cdot x + \mu$ while the second one adds

$l \leq y \leq u, x \leq y \leq \lambda \cdot x + \mu$. Since any sound transformers can work here, using different transformers may lead to different verification results, and we hope to choose the tightest one. For a given value of $l, u$, a common heuristic is to choose the transformer with the smaller area in the $x, y$ plane, and we will discuss the opportunity to further refine these transformers in Section 3.1.5. The transformer can be extended to handle other activations. We refer the reader to Singh *et al.* (2019b), Zhang *et al.* (2020), and Paulsen and Wang (2022) for details. In terms of cost, all three cases can be handled in constant time. As one can notice, similar to the Zonotope domain, the constraints for $y$ are computed based only on the constraints in $x$, ignoring the constraints for the other neurons in the same layer as $x$. The precision of the transformer therefore depends on the quality of the interval bounds $[l, u]$. Coming back to our example, the bounds we computed using the interval arithmetic are the tightest for $x_4$. However, this is only the case for the output of the first affine layers. For other layers, as we will see, this method loses precision.

In our example, we choose the first approximation for bounding $x_6$ since it has a smaller area. This yields the constraints $0 \leq x_6 \leq \frac{5}{12} \cdot x_4 + \frac{3.5}{12}$ and $0 \leq x_6 \leq 0.5$. The Single Neuron construction is used to combine the individual ReLU transformer to compute the layerwise ReLU transformer. This enables using the bounds for $x_5, x_6$ computed above for bounding $o_0$.

The polyhedral bounds for $o_0$ can be easily computed as $x_5 + x_6 + 0.5 \leq o_0 \leq x_5 + x_6 + 0.5$. However, using the interval transformers for computing the interval bounds yields imprecise output $[0.6, 2.3]$, which can reduce the precision of subsequent analysis. The best interval bounds can be obtained via linear programming where the objective is to minimize/maximize $o_0$ subject to the DeepPoly constraints for the other variables. While LP computations can be parallelized across neurons, running LP two times for every neuron can be expensive for larger DNNs. Therefore, standard DeepPoly /CROWN uses a custom algorithm called *backsubstitution* that, in practice, obtains the bounds of a similar quality as LP but faster (Singh *et al.*, 2019b; Zhang *et al.*, 2018a; Müller *et al.*, 2021a; Zelazny *et al.*, 2022). We next show how backsubstitution yields a more precise upper bound for $o_0$.

Instead of using the interval bounds, backsubstitution uses polyhedral bounds. It substitutes the upper polyhedral bounds for $x_5$ and $x_6$ into the upper polyhedral bound for $o_0$. This yields

$$\begin{aligned} o_0 &\leq x_5 + x_6 + 0.5 \leq x_3 + \frac{5}{12} \cdot x_4 + \frac{3.5}{12} + 0.5 \\ &\leq x_3 + \frac{5}{12} \cdot x_4 + \frac{9.5}{12} \end{aligned}$$

We can compute the upper bound using the interval upper bounds for $x_3$ and $x_4$. However, this does not improve the upper bound. Next, we can substitute the upper bounds for $x_3$ and $x_4$ to obtain a new upper bound

$$\begin{aligned} o_0 &\leq x_3 + \frac{5}{12} \cdot x_4 + \frac{9.5}{12} \\ &\leq (x_1 + x_2) + \frac{5}{12} \cdot (x_1 - x_2) + \frac{9.5}{12} \\ &\leq \frac{17}{12} \cdot x_1 + \frac{7}{12} \cdot x_2 + \frac{9.5}{12} \end{aligned}$$

Computing the upper bound using the expression yields 2.05 which is more precise and is the output of the algorithm. The precision improves because of the cancellation of the coefficients for $x_2$. In general, the backsubstitution algorithm, starting from a layer $l > 2$, recursively substitutes the polyhedral bounds for the variables in the polyhedral expression, till the first layer. At each layer, it computes interval bounds and chooses the best among them. Backsubstitution is the most expensive operation in the DeepPoly/CROWN analysis and its cost for computing the interval bounds of a single neuron is quadratic in terms of the maximum number of neurons in a layer in the DNN and linear in the number of layers. The cost can be controlled by using different heuristics such as stopping after a few layers or using interval bounds instead of polyhedral ones for substitution for some variables. Similar computations can be performed for the bounds of $o_1$. $g(\alpha(\varphi))$ from the DeepPoly analysis is shown in Figure 3.10.

As for the Zonotope analysis, interval bounds from DeepPoly's $g(\alpha(\varphi))$ cannot prove the postcondition. Instead, we can compute the

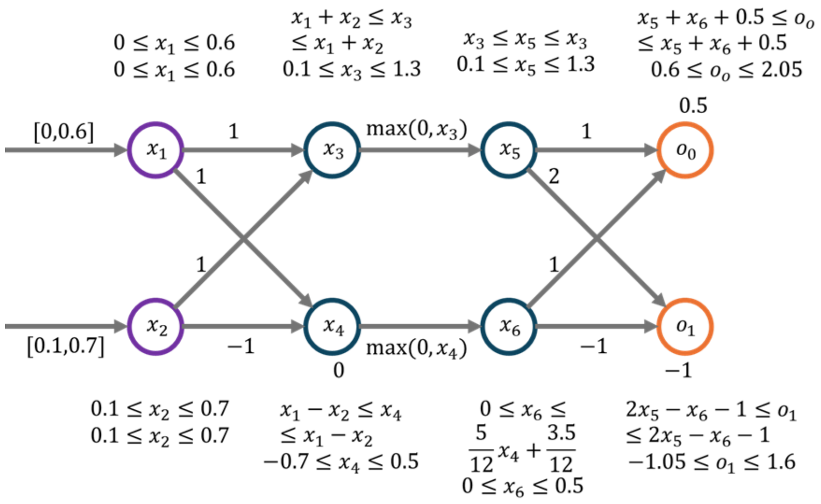


**Figure 3.10:** DeepPoly/CROWN analysis results on the example DNN.

lower bound for $o_0 - o_1$ using backsubstitution and check whether that is $> 0$. The computed lower bound is 0.2, which is more precise than the Zonotope analysis and proves the specification.

**Star Set.** Star Sets are a generalization of Zonotopes where the noise terms are constrained by a predicate $P(\epsilon_1, \epsilon_2, ...\epsilon_p)$ instead of the interval $[-1, 1]$. In the context of DNN verification, the most common predicate is defined using a conjunction of linear constraints $\sum_{j=1}^{p} c_j \cdot \epsilon_j \le d$, where $c_j, d \in \mathbb{R}$, over the noise symbols (Tran *et al.*, 2021; Tran *et al.*, 2019a). For our example, the analysis starts by creating the affine forms $\hat{x_1} = \epsilon_1, \hat{x_2} = \epsilon_2$ with the constraints $0 \le \epsilon_1 \le 0.6$ and $0.1 \le \epsilon_2 \le 0.7$. The affine assignments are handled exactly by creating the affine forms $\hat{x_3} = \epsilon_1 + \epsilon_2$ and $\hat{x_4} = \epsilon_1 - \epsilon_2$. This transformer can be easily parallelized across neurons in a layer and does not modify the constraints over the $\epsilon$'s. The complexity of the exact transformer for one affine assignment is the same as Zonotopes. Next, the ReLU assignment is handled based on the interval bounds which can be computed using linear programming (LP). We find that the lower bound for $x_3 > 0$, so we can set $\hat{x_5} = \hat{x_3} = \epsilon_1 + \epsilon_2$. We have the challenging case $l < 0 < u$ on the bounds for $x_4$. The Star

Sets domain uses the triangle approximation shown in Figure 3.6, which cannot be represented in the Zonotope domain. This adds the affine form $\hat{x_4} = \epsilon_3$ with the new noise symbol $\epsilon_3$ constrained by $\epsilon_3 \geq 0, \epsilon_3 \geq \epsilon_1 - \epsilon_2, \epsilon_3 \leq \frac{5}{12} \cdot (\epsilon_1 - \epsilon_2) + \frac{3.5}{12}$. All three cases for the ReLU require two LP calls to compute the interval bounds in the worst case. While these calls can be easily parallelized and combined using the Single Neuron construction to compute a layerwise approximation, the use of LP makes Star Sets more expensive than Zonotopes. Note that the triangle approximation can be represented by the DeepPoly/CROWN domain. However, the standard implementations use less precise relaxations as backsubstitution cannot exploit the extra precision from the triangle approximation. Overall, the Star Sets domain is more expressive than Zonotopes and DeepPoly/CROWN but also the most expensive.

The final affine assignments add the affine forms $\hat{o_0} = \epsilon_1 + \epsilon_2 + \epsilon_3 + 0.5$ and $\hat{o_1} = 2 \cdot \epsilon_1 + 2 \cdot \epsilon_2 - \epsilon_3 - 1$. Figure 3.11 shows the final output shape $g(\alpha(\varphi))$ computed by the Star Sets domain. To prove the specification, an LP call is made to minimize $o_0 - o_1$ under the constraints defining $g(\alpha(\varphi))$. The positive result is sufficient to prove the specification.

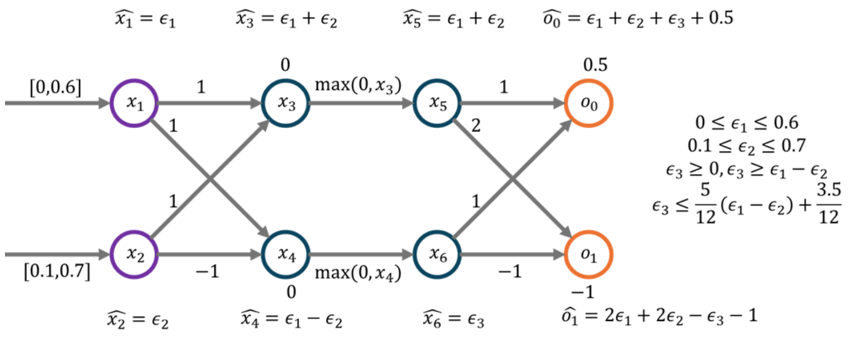

$\widehat{x_1} = \epsilon_1$
$\widehat{x_3} = \epsilon_1 + \epsilon_2$
$\widehat{x_5} = \epsilon_1 + \epsilon_2$
$\widehat{o_0} = \epsilon_1 + \epsilon_2 + \epsilon_3 + 0.5$
[0,0.6]
[0.1,0.7]
$x_1$
$x_2$
$x_3$
$x_4$
$x_5$
$x_6$
$o_0$
$o_1$
0
0.5
$\max(0, x_3)$
$\max(0, x_4)$
1
1
1
−1
1
2
1
−1
0
−1
$0 \leq \epsilon_1 \leq 0.6$
$0.1 \leq \epsilon_2 \leq 0.7$
$\epsilon_3 \geq 0, \epsilon_3 \geq \epsilon_1 - \epsilon_2$
$\epsilon_3 \leq \frac{5}{12}(\epsilon_1 - \epsilon_2) + \frac{3.5}{12}$
$\widehat{x_2} = \epsilon_2$
$\widehat{x_4} = \epsilon_1 - \epsilon_2$
$\widehat{x_6} = \epsilon_3$
$\widehat{o_1} = 2\epsilon_1 + 2\epsilon_2 - \epsilon_3 - 1$

**Figure 3.11:** Star Sets analysis results on the example DNN.

We had $\gamma(\alpha(\varphi))) = \varphi$ with all four domains for our example specification. However, equality does not always hold, and a sound approximation needs to be computed. Next, we discuss geometric perturbations on 2D-images, which apply non-linear transformations resulting in an atypical $\varphi$ that cannot be described as the union of convex polyhedra.

**Geometric perturbations.** A geometric image perturbation is a function $P$, which takes an input image $x$ and a multi-dimensional parameter vector $\theta \in \mathbb{R}^k$ encoding transformations on $x$, e.g., angle of rotation and amount of scaling, and produces the geometrically perturbed image $x' = P(x, \theta)$. Geometric perturbations involve an affine transformation on each pixel's row and column indices, followed by an interpolation operation. As these operations are performed in the 2D plane, we first interpret the row and column indices $(i, j)$ as points $(u, v) \in \mathbb{R}^2$, where the $u$-axis is the horizontal axis and the $v$-axis is the vertical axis. Here, we define functions $\phi_u(j) = j - (W-1)/2$ and $\phi_v(i) = (H-1)/2 - i$, which convert zero-indexed $i, j$ indices to $u, v$ coordinates with respect to the center of an $H \times W$ image. Let $G_\theta \colon \mathbb{R}^2 \to \mathbb{R}^2$ be an invertible affine transformation (e.g., rotation, translation) parameterized by $\theta$ (e.g., rotation angle, amount of horizontal shift). Having converted row-column indices to $\mathbb{R}^2$, we compute for each location $(\phi_u(j), \phi_v(i))$ the (real-valued) coordinate that maps to this location under $G_\theta$; we can obtain this coordinate as $(u', v') = G_\theta^{-1}(\phi_u(j), \phi_v(i))$, where $G_\theta^{-1}$ is the inverse transformation. Since these transformed coordinates may not align exactly with integer-valued pixel indices, we must interpolate. The most popular interpolation method is bilinear interpolation (Jaderberg *et al.*, 2015), given as:

$$I_x(u, v) = \sum_{p=0}^{H-1} \sum_{q=0}^{W-1} x_{p,q} \cdot \max(0, 1 - |v - \phi_v(q)|) \cdot \max(0, 1 - |u - \phi_u(p)|) \tag{3.1}$$

where $x$ is the original image. The value of each pixel in the interpolated image $x'$ is then:

$$x'_{i,j} = P(x, \theta) = I_x(G_\theta^{-1}(\phi_u(j), \phi_v(i))) \tag{3.2}$$

$\varphi$ for geometric perturbations contains the set of all images generated by (3.2) for a range $\theta \in [\alpha, \beta]$, where $\alpha, \beta \in \mathbb{R}^k$, of parameter values. Verification against geometric transformations requires two key steps: (1) obtaining bounds on the set of perturbed images $P(x, \theta)$ obtainable after applying geometric perturbation $P$ corresponding to $\theta \in [\alpha, \beta]$ and (2) propagating these bounds through the neural network. This is shown in Figure 3.12. As a concrete example, we consider computing

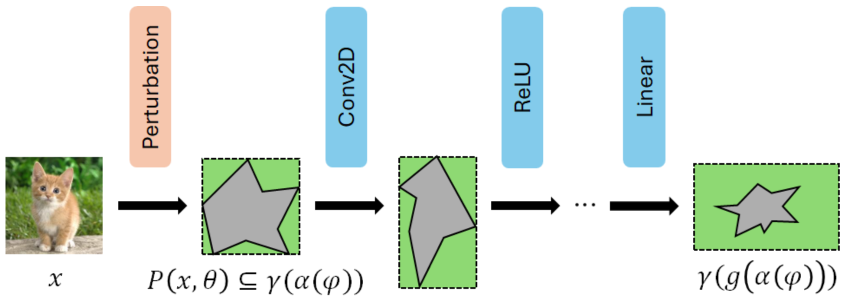

Perturbation
Conv2D
ReLU
Linear
$x$
$P(x,\theta) \subseteq \gamma(\alpha(\varphi))$
…
$\gamma(g(\alpha(\varphi)))$

**Figure 3.12:** DNN verifier against geometric perturbations based on the Box analysis. It first computes a box approximation $\alpha(\varphi)$ of the set of images $P(x,\theta)$ produced by applying geometric perturbations from $\theta \in [\alpha,\beta]$. The result is propagated through the DNN with standard Box analysis using the same transformers for DNN transformations as described before.

$\alpha(\varphi)$ for the scaling transformation with $\theta \in [-0.02, 0.02]$. The inverse transformation for scaling is shown below:

$$G_\theta^{-1}(u,v) = \begin{bmatrix} \frac{1}{1+\theta} & 0 \\ 0 & \frac{1}{1+\theta} \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u/(1+\theta) \\ v/(1+\theta) \end{bmatrix} \tag{3.3}$$

We consider the Box domain because it is the most common abstraction for computing $\alpha(\varphi)$ for geometric perturbations (Singh *et al.*, 2019b; Yang *et al.*, 2023; Mohapatra *et al.*, 2020). One benefit of the Box approximation of $\varphi$ is that $\alpha(\varphi)$ can be easily propagated through the DNN with other abstract domains (e.g., Zonotope, DeepPoly). We refer the reader to the work of Balunovic *et al.* (2019) that computes $\alpha(\varphi)$ in the DeepPoly domain. We start by associating the interval $[-0.02, 0.02]$ with $\theta$. We next apply the Box transformer for the inverse transformation, which involves interval division for every $u, v$ coordinate

$$\left(\frac{u}{1+[-0.02,0.02]}, \frac{v}{1+[-0.02,0.02]}\right) = \left(\frac{u}{[0.98,1.02]}, \frac{v}{[0.98,1.02]}\right) \tag{3.4}$$

For example with $(u,v) = (1,1)$ we get the set of pixels that map to $(u,v)$ in the scaled image as inside the square $[0.98, 1.02] \times [0.98, 1.02]$ in the original image. The Box output from (3.4) containing the set of points in the original image mapping to different $(u,v)$ in the scaled image is then fed to the to the abstract transformer corresponding to

(3.1), obtaining $\alpha(\varphi)$. This abstract transformer can be constructed by combining the interval transformer for the absolute, max, addition, subtraction, and multiplication transformations.

Next, we list some other abstract domains that have been employed for neural network verification.

- **CH-Zonotope.** CH-Zonotopes are more expressive than Zonotopes and can be seen as the Minkowski sum of a Hyperbox and Zonotope (Müller *et al.*, 2023b; Mirman *et al.*, 2018). Formally, the affine form for the variable $x$ in this abstraction can be written as $\hat{x} = \alpha_0 + \sum_{i=1}^{r} \alpha_i \cdot \epsilon_i + \beta_x \cdot \kappa$ where $\beta \in \mathbb{R}$. Here $\kappa \in [-1, 1]$ is a special noise term that is specific to each variable and is not shared across different affine forms like $\epsilon_i$.

- **Multi-Norm Zonotope.** Standard zonotopes can exactly represent preconditions defined by $L_\infty$ distances or intervals. However, they lose precision when handling $L_1$ or $L_2$ balls. To overcome this limitation, multi-norm Zonotopes were developed (Bonaert *et al.*, 2021), parameterized by a given $L_p$ norm. Here the affine expression $\hat{x} = \alpha_0 + \sum_{i=1}^{r} \alpha_i \cdot \epsilon_i + \sum_{j=1}^{q} \beta_j \cdot \kappa_j$, in addition to the standard Zonotope noise symbols $\alpha_i$, also contains the set of noise variables $\kappa := (\kappa_1, \kappa_2, \ldots, \kappa_q)$ constrained by the $L_p$-based inequality $||\kappa||_p \leq 1$. Overall, this domain is more expressive than the Zonotope domain. For handling the non-linearities such as ReLU, Sigmoid, and Tanh, the domain adapts the transformers from the Zonotope domain. The domain provides specialized efficient transformers for the Softmax and dot product operations that are common in the transformer architecture making the domain suitable for verifying this architecture.

- **Polynomial Zonotope.** Polynomial Zonotope (Kochdumper *et al.*, 2023; Ladner and Althoff, 2024; Kochdumper and Althoff, 2021; Ladner and Althoff, 2023; Ladner *et al.*, 2024) is a non-convex abstraction parameterized by a degree $m$. Here the affine form $\hat{x} = \alpha_0 + \sum_{i=1}^{r} \alpha_i \cdot \epsilon_i + \sum_{j1=1}^{q} \beta_1 \cdot \kappa_{j1} + \sum_{j1=1}^{q} \sum_{j2=j1}^{q} \beta_{12} \cdot \kappa_{j1} \cdot \kappa_{j2} + \ldots + \sum_{j1=1}^{q} \sum_{j2=j1}^{q}, \ldots, \sum_{jm=j(m-1)}^{q} \beta_{12\ldots m} \cdot \kappa_{j1} \cdot \kappa_{j2} \cdot \ldots . \kappa_{jm}$ where all the noise terms $\kappa_j, \epsilon_i \in [-1, 1]$ as for standard Zonotopes. The noise terms $\kappa_j$ are called dependent as a change in their value affects multiple terms in the affine form.

- **ImageStars.** An extension of Star Sets, called ImageStars, was developed in Tran *et al.* (2020a) to handle convolutional networks in the vision domain efficiently. The coefficients $\alpha_i$ in the Star abstraction denote an image instead of scalars in this abstraction. The abstract transformers for ImageStars are an adaptation of the transformers for Star Sets.

- **Symbolic Linear Relaxation (SLR).** Like DeepPoly/CROWN domain, the SLR domain keeps the polyhedral bounds $L_i \leq x_i \leq U_i$ for each neuron $x_i$. Instead of keeping interval bounds on the value of $x_i$, it tracks interval bounds on the expressions $L_i$ and $U_i$. Thus SLR associates six constraints per neuron (Wang *et al.*, 2018). The abstraction is exact for affine assignments and heuristically approximates the effect of non-linearities based on the interval bounds computed by affine transformer. The interval bounds for $L_i, U_i$ can be computed in a similar manner as for DeepPoly/CROWN.

- **Octatope.** The Octatope domain is a restricted form of Star Sets but is more expressive than the Zonotopes. Here, the noise symbols are constrained by Octagon constraints (Miné, 2006). Geometrically, Octatopes are affine transformations of n-dimensional Octagons (Bak *et al.*, 2023). As a result, the Octatope domain can exactly capture affine transformation. Recall that the use of LP for computing the interval bounds make Star Sets expensive. The restriction to Octagon constraints makes LP more efficient for the Octatope domain. It can be solved in strongly polynomial time via a reduction to the minimum cost flow (MCF) problem (Goldberg and Tarjan, 1989).

- **HexaTope.** This domain is less expressive than Octatope (Bak *et al.*, 2024). Here, the noise symbols in the affine form are constrained by difference constraints, as is the case for the Zones domain (Miné, 2002). This domain can also capture affine assignments exactly and enables efficient LP solving.

- **Polyhedra.** The Polyhedra domain is the most expressive linear relational domain (Cousot and Halbwachs, 1978; Singh *et al.*, 2017). The elements in this domain can be represented as a conjunction of a finite number of linear constraints between neurons of the form $\sum_i a_i \cdot x_i \leq b$ where $a_i, b \in \mathbb{R}$ and $x_i \in \mathcal{X}$. The domain can exactly capture affine assignments. The best transformer for piecewise-linear activations like ReLU, Maxpool has exponential cost, making the domain prohibitively expensive for handling larger DNNs.

- **Tropical Polyhedra.** The tropical polyhedra domain was used for verifying ReLU-based networks in Goubault *et al.* (2021). A tropical polyhedron (Allamigeon *et al.*, 2008) is a conjunction of a finite number of tropical constraints of the form $max(max_i(a_i + x_i), b) \leq max(max_i(c_i + x_i), d)$ where $a_i, c_i, b, d \in \mathbb{R}$ and $x_i \in \mathcal{X}$. Geometrically, a tropical polyhedron encodes a union of zones (Miné, 2002). Compared to other domains, this one is exact for ReLU but loses precision for affine transformation.

### 3.1.4 Generic Recipes for Non-Linear Layers

Affine layers can be efficiently handled exactly in most existing domains by composing the exact transformers for individual affine assignments. This computation can often be parallelized. The most challenging computation inside DNNs is the non-linear layers. Next, we describe four generic recipes for constructing abstract transformers, with varying precision and cost, for approximating the effect of a given layer $Y := \sigma(X)$ containing $n$ non-linear assignments $y_i := \sigma(x_i)$. The construction can be instantiated for any domain $\mathcal{D}$ (e.g., DeepPoly, Octatope). Let $D \in \mathcal{D}$ denote the input abstract element and consider the common case where $D$ does not contain any constraints involving any of the variables $y_i$. The output $O \in \mathcal{D}$ can be calculated as:

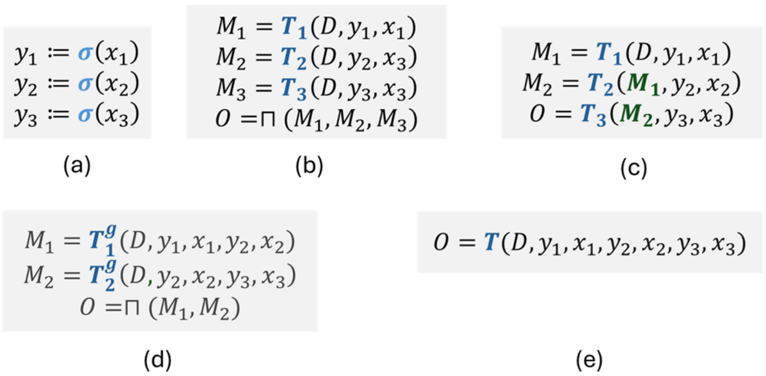
- **Single Neuron.** This method requires individual transformers $T_i$ for each non-linear assignment $y_i := \sigma(x_i)$. Each $T_i$ can be based on different algorithms. However, $T_i$ for different assignments can be applied independently with $D$ as input for each $T_i$, allowing efficient parallelization on GPUs. $O$ is computed by applying the meet operator ($\sqcap$) (or its approximation) on the outputs $O_i$ computed by each $T_i$. The best $O$ obtained from this method can be less precise than that from the sequential method. This is the most common construction for handling non-linear layers in the literature. We followed this recipe for handling the ReLU layers in our illustrative example in Section 3.1.3.

- **Sequential.** This method also requires abstract transformers $T_i$ corresponding to each assignment. The individual assignments $y_i := \sigma(x_i)$ are processed sequentially where the output of the abstract transformer $T_i$ serves as the input to $T_{i+1}$. The order in which the assignments are processed can affect the verifier's precision. $O$ is obtained after processing the last assignment in the chosen sequence. As the assignments are processed one after the other, the transformer computations cannot be done in parallel. Even if each $T_i$ is the best transformer for each assignment, this method's output $O$ need not be the best approximation of the layerwise output in the domain $\mathcal{D}$.

- **Multi Neuron.** This method processes a group of $1 < k < n$ assignments jointly using the same or different abstract transformers $T^g$ that operate on a group of assignments. The set of groups $G$ must form a covering (the groups can have overlapping assignments) of the set of assignments in the layer. The individual groups can be processed independently, enabling parallelization opportunities. $O$ is computed by applying the meet operator ($\sqcap$) (or its approximation) on the outputs corresponding to different groups. The precision of $O$ increases with the size $k$ of each group and the number of groups in the covering. The best $O$ obtained from this method is at least as precise as the single-neuron method but need not be the best layerwise approximation in the domain $\mathcal{D}$. For a given group, the best output from this method is more precise than the best from the sequential method.

- **Layerwise.** All $n$ assignments are processed at once using a single abstract transformer $T$. The best layerwise approximation in $\mathcal{D}$ can be computed using this method.

Figure 3.13 shows the application of the four constructions on a toy example. We refer the reader to Gehr *et al.* (2018) for a detailed description of sequential transformers with any domain $\mathcal{D}$ for the ReLU layer based on using the Join ($\sqcup$) operator. The work of Singh *et al.* (2019a) describes the construction of single neuron, multi neuron, and layerwise transformers for ReLU using joins. This was generalized to more diverse non-linearities in Müller *et al.* (2021b). Anderson *et al.* (2020) and Tjandraatmadja *et al.* (2020) discussed generating layerwise approximations directly for the composition of affine and ReLU transformations, which typically involve an exponential number of constraints.



(a)
$$y_1 := \sigma(x_1) \quad y_2 := \sigma(x_2) \quad y_3 := \sigma(x_3)$$

(b)
$$M_1 = T_1(D, y_1, x_1) \quad M_2 = T_2(D, y_2, x_3) \quad M_3 = T_3(D, y_3, x_3) \quad O = \sqcap(M_1, M_2, M_3)$$

(c)
$$M_1 = T_1(D, y_1, x_1) \quad M_2 = T_2(M_1, y_2, x_2) \quad O = T_3(M_2, y_3, x_3)$$

(d)
$$M_1 = T_1^g(D, y_1, x_1, y_2, x_2) \quad M_2 = T_2^g(D, y_2, x_2, y_3, x_3) \quad O = \sqcap(M_1, M_2)$$

(e)
$$O = T(D, y_1, x_1, y_2, x_2, y_3, x_3)$$

**Figure 3.13:** Illustration of the generic constructions for handling non-linear layers. (a) shows an example layer with 3 non-linear assignments. (b) The single neuron construction applies the transformer for each assignment independently taking $D$ as input. The results are combined using the meet ($\sqcap$) operator. (c) Starting from $D$, the sequential construction applies the transformer for each assignment on the output of the previous assignment. (d) The multi neuron construction first constructs a covering $\{\{(y_1, x_1), (y_2, x_2)\}, \{(y_2, x_2), (y_3, x_3)\}\}$ of the set of assignments and applies group transformers on each group independently with $D$ as the input. The results are combined using $\sqcap$. (e) The layerwise construction handles all assignments with a single transformer $T$.

Since abstract interpretation is compositional, once we have transformers for non-linear and affine layers, any DNN architecture, e.g., residual (Xu *et al.*, 2020), recurrent (Ko *et al.*, 2019), graph neural network (Wu *et al.*, 2022b), can be handled by simply composing the corresponding transformers according to the DNN architecture defined by a computation graph (Xu *et al.*, 2020).

### 3.1.5 Refinement

When the verifier cannot prove a specification due to imprecise $g(\alpha(\varphi))$, refinement can be applied to strategically improve the approximation error. In this section, we will discuss general refinement strategies that can be adapted for different DNN architectures. State-of-the-art verifiers combine several refinement strategies with abstract interpretation-based analysis.

**Input/output splitting.** The precondition $\varphi$ is partitioned into $m$ smaller regions $\varphi_i$. An abstract interpreter is then run to compute $g(\alpha(\varphi_i))$ for each $\varphi_i$. This computation can be parallelized across different splits. If the postcondition $\psi$ can be proven on each $\varphi_i$, then the original specification is also proved. If a counterexample is found on one of the split (e.g., using a bug finding algorithm), then the specification is disproved. Otherwise, we can further partition the preconditions for which $g(\alpha(\varphi_i))$ is imprecise. This process can continue till the full specification is proved, a concrete counter example is found, or a limit on the number of splits is reached. The cost of this strategy depends upon the number of splits and the size of each split. Several heuristics exist in the literature for partitioning. We refer the reader to Wang *et al.* (2018), Singh *et al.* (2019d), Balunovic *et al.* (2019), Yin *et al.* (2022), Wei *et al.* (2023), Yang and Rinard (2019), and Brückner and Lomuscio (2024) for examples. Output splitting works similarly by partitioning the postcondition (Henriksen and Lomuscio, 2021). Input splitting and activation splitting (next paragraph) methods are also known in the literature as branch-and-bound (BaB) based methods.

**Activation splitting.** In this strategy, the set of non-linear assignments in the DNN are ranked according to a heuristic that measures the importance of different assignments for proving the specification. Next, the first assignment $y_i := \sigma(x_i)$ from the ranking is chosen and the interval range $[l_i, u_i]$ for $x_i$ is partitioned into $m$ regions $[l_{ij}, u_{ij}]$. For each split $[l_{ij}, u_{ij}]$, we need to compute the DNN output corresponding to $\varphi \wedge (l_{ij} \leq x_i \leq u_{ij})$. The abstract interpreters from the previous section can be adapted to handle the interval constraint $l_{ij} \leq x_i \leq u_{ij}$

on an intermediate neuron $x_i$ by designing (or leveraging) an (existing) abstract transformer for computing the intersection of an abstract element $D \in \mathcal{D}$ with a linear constraint. For our case, $D$ is the element computed for $\varphi$ before any split. To handle the linear constraint, an LP solver (Bunel *et al.*, 2020) or Lagrangian multipliers (Wang *et al.*, 2021; Ferrari *et al.*, 2022; Shi *et al.*, 2024) can be applied. After applying the intersection, the assignment $y_i := \sigma(x_i)$ and any other operations in the DNN that come after it are handled by applying the corresponding transformers. If the specification is proved on each split, then the original specification is also proved. If a counterexample is found on one of the splits $[l_{ij}, u_{ij}]$, then the specification is disproved. If the result is inconclusive on a set of splits $[l_{ij}, u_{ij}]$, then for each such split, the process is repeated to select the next non-linear assignment $y_k := \sigma(x_k)$ (this can be in a layer before $x_i$) and the abstract interpreter is run for each region $\varphi \wedge (l_{ij} \leq x_i \leq u_{ij}) \wedge (l_{kj} \leq x_k \leq u_{kj})$. In the case of ReLU networks, activation splitting can achieve complete verification (Wang *et al.*, 2021) since ReLU is piece-wise linear, and the number of linear regions in a ReLU network is finite (exponential to the number of ReLU neurons). For non-ReLU networks, although splitting is also possible for improving precision (Sidrane *et al.*, 2022; Shi *et al.*, 2024), the verification problem is, in general, undecidable.

There exists a variety of heuristics for ranking the non-linear assignments and for partitioning the interval ranges. We refer the reader to Pulina and Tacchella (2010), Lu and Kumar (2019), Yang *et al.* (2024c), Shi *et al.* (2024), Duong *et al.* (2023), Henriksen and Lomuscio (2020), Palma *et al.* (2021b), Xue and Sun (2024), and Lemesle *et al.* (2024) for state-of-the-art heuristics. A generalization of activation splitting discovers *cut constraints* between neurons in different layers in the DNN and runs the abstract interpreter on the conjunction of $\varphi$ with the discovered constraints (Zhang *et al.*, 2022; Zhou *et al.*, 2024). Abstract transformers for intersecting abstract elements with cutting constraints are needed for this approach.

**Optimizing parametric analysis.** This approach leverages parametric abstract transformers $T_\lambda$ for non-linearities: their output has parameters $\lambda$ that can be optimized for the given specification. An example is designing the lower polyhedral bound in the DeepPoly/CROWN domain for the ReLU assignment $y := ReLU(x)$ to be $y \geq \lambda \cdot x$ where $\lambda \in [0, 1]$ can be optimized (Figure 3.14). The setting of $\lambda$ for each neuron can be independent. Notice that setting $\lambda = 0$ and $\lambda = 1$ gives us the transformers shown in Figure 3.6 based on minimizing the area in the $xy$-plane, and $\lambda = \frac{u}{u-l}$ gives the Zonetope transformer. While they may decrease the number of spurious points in the output, they do not yield optimal precision. All lower bounds with $\lambda \in [0, 1]$ are sound, and better results can be obtained by tuning the $\lambda$ for each assignment guided by the specification. The verification of the specification is then expressed as an optimization problem either directly or via a surrogate differentiable loss over the parametric analysis output $g_\lambda(\alpha(\varphi))$. The optimization can be solved with LP, SMT, or gradient-based methods. $\alpha$-CROWN is a prominent example of this approach (Xu *et al.*, 2021) using gradient descent on GPUs to optimize all $\lambda$s in parallel. Please see Ryou *et al.* (2021), Chevalier *et al.* (2024), Lyu *et al.* (2020), König *et al.* (2024b), and Dvijotham *et al.* (2018b) for more examples.

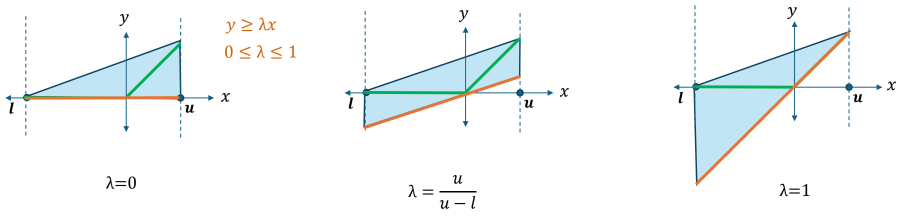


**Figure 3.14:** Parametric DeepPoly/CROWN transformers $T_\lambda$, where $\lambda$ denotes the slope of the lower polyhedral bound of ReLU. Any $0 \leq \lambda \leq 1$ yields a sound transformer, and $\lambda$ can be optimized for every ReLU to achieve the tightest verification result.

**Reduction with precise approximations.** This approach runs two analyzers in parallel where one is fast but imprecise and the second one is precise but slow. The analysis results from the imprecise analysis are used to speed up the computations in the precise analysis while the precise results are used to reduce the approximation error of the imprecise analysis. To manage the cost, the precise analysis maybe employed on a strategically selected subset of neurons in the DNN. The final output is the intersection of the results from the two analyzers. For example, the RefineZono analysis runs (i) a cheap but imprecise analysis based on a reduced product of the Box and Zonotope domains on all neurons in the DNN (Singh *et al.*, 2019d) and (ii) LP and MILP-based analysis to compute precise interval bounds on the inputs to strategically selected ReLU assignments. The Box bounds from the imprecise analysis speedup the precise analysis by reducing the search area for the solver. The tighter interval bounds on the ReLU inputs are passed to the imprecise analysis. Since the Zonotope approximation of ReLU assignments (as shown in Figure 3.6) depends on the tightness of the input interval bounds, the output has a smaller area, and this contributes to improved precision. Another popular example of this approach is DeepPoly combined with multi neuron transformers (Müller *et al.*, 2021b; Singh *et al.*, 2019a; Ma *et al.*, 2024; Ferrari *et al.*, 2022; Tang *et al.*, 2023; Ma, 2023). We refer the reader to Bak *et al.* (2020) and Bak (2021) for more examples of this strategy.

**Backward analysis.** The analyzers described so far ignore the postcondition. This makes the output $g(\alpha(\varphi))$ useful for proving a variety of postcondition. However, incorporating the constraints from the postcondition can be used to refine the verifier output (Yang *et al.*, 2021; Wu *et al.*, 2022b; Rober *et al.*, 2023; Kotha *et al.*, 2023). This type of refinement uses two abstract domains $\mathcal{D}_f, \mathcal{D}_b$ for the forward and backward analysis, respectively. The backward pass first updates the backward element at the output layer by (i) applying an abstract transformer in $\mathcal{D}_f$ to intersect $g(\alpha(\varphi))$ with $\neg\psi$ and (ii) abstracting the result in $\mathcal{D}_b$. The backward pass at a non-output layer $k$ performs two steps. First, the network behavior from layer $k$ till the output and $\neg\psi$ is encoded using the elements from both the forward and backward analysis. Next, interval bounds on each neuron in the layer are computed using LP/SMT solvers (Yang *et al.*, 2021; Wu *et al.*, 2022b), or gradient-based optimization (Kotha *et al.*, 2023). The backward abstract element is then refined by (i) applying an abstract transformer in $\mathcal{D}_f$ to intersect the forward element at the layer with the interval constraints from the solver, (ii) abstracting the output in $\mathcal{D}_b$, and (iii) applying meet operator ($\sqcap$) on the original backward element and the one from (ii). If the output of (iii) is empty, then the specification is proved. To control the cost of the backward pass, the solver can be run to compute interval bounds for only a subset of heuristically selected neurons. The forward and backward passes can be applied iteratively till the specification is proved/disproved or a stopping criteria is met. The final output is the intersection of the abstract elements from the forward and backward analysis.

Independent from refinement, specialized backward analysis have been designed to compute an under or overapproximation of the DNN preimage with respect to the postcondition $\psi$. We refer the reader to Dimitrov *et al.* (2022), Urban *et al.* (2020a), Kotha *et al.* (2023), Zhang *et al.* (2018b), Gopinath *et al.* (2020), and Zhang *et al.* (2024) for details. We conclude our discussion of the verification of single execution properties by referring the interested readers to excellent evaluations from Li *et al.* (2020), König *et al.* (2024a), and Brix *et al.* (2024b) comparing the performance of state-of-the-art verifiers on standard benchmarks.

## 3.2 Relational Properties

The verification methods covered in Section 3.1 lack the expressiveness needed to specify properties such as monotonicity, which requires comparing the DNN's output on two distinct inputs. To capture such properties, we need input and output specifications that can characterize the DNN's behavior across multiple distinct but related inputs, rather than just perturbations of a single input as in non-relational verifiers (Singh *et al.*, 2019b). In this section, we discuss the verification of relational properties that encode desirable DNN behavior in two settings: a) multiple executions of the same DNN on related inputs (input-relational) (Banerjee *et al.*, 2024b), and b) multiple executions of different DNNs on the same input (network-relational) (Paulsen *et al.*, 2020). We begin by formally defining the input and output specifications for these relational properties and illustrate practical examples where verifying these properties is useful.

### 3.2.1 Relational Specifications

Suppose we want to ensure that a DNN applied in the financial domain, such as housing price prediction (Banerjee *et al.*, 2024b), makes sensible decisions—e.g., predicting a higher price for a larger house with more rooms, assuming all other features remain constant. Non-relational verifiers cannot capture relationships between pairs of inputs, such as comparing the number of rooms in a larger house and a smaller one. Similarly, if we want to verify that a smaller, compressed model behaves equivalently to the larger network from which it was derived within a specific input region (Paulsen *et al.*, 2020), existing non-relational verifiers are inadequate, as they are unable to reason about two network executions simultaneously. We now go into the details of both input-relational and network-relational properties.

**Input-relational properties.** Input-relational properties check whether a DNN behaves as expected across $k$ distinct but related inputs. In this case, we define $k$ input regions, $\varphi_1, \dots, \varphi_k$, along with a **cross-executional constraint** $\varphi_\delta$ that encodes the relationships between the $k$ distinct inputs used in different executions of the DNN $f$. For any set of $k$ inputs $x_1, \dots, x_k$ satisfying $\wedge_{i=1}^{k}(x_i \in \varphi_i) \wedge (x_1, \dots, x_k) \in \varphi_\delta$, we aim to prove that the corresponding outputs $f(x_1), \dots, f(x_k)$ meet the condition $(f(x_1), \dots, f(x_k)) \in \psi$. The key distinction here is that the cross-executional constraint captures relationships across inputs, while the output specification ensures that the outputs of all $k$ executions are related (see Figure 3.15). For example, to demonstrate that the given DNN $f$ is monotonically increasing, we can use $\varphi_\delta = (x - y \geq 0)$ for a pair of inputs $x$ and $y$, and $\psi = (f(x) - f(y) \geq 0)$ as the output specification. Another interesting property expressible as a relational specification is the DNN's robustness against Universal Adversarial Perturbations (UAP). The UAP (Universal Adversarial Perturbation) robustness verification problem examines whether a single perturbation $\delta \in \mathbb{R}^m$ can be applied to $k$ DNN inputs, causing the model to misclassify all of them. This problem differs fundamentally from the more commonly studied local robustness verification (Singh *et al.*, 2019b), where the adversary is allowed to perturb each input independently. However, recent research has shown that generating input-specific ad-

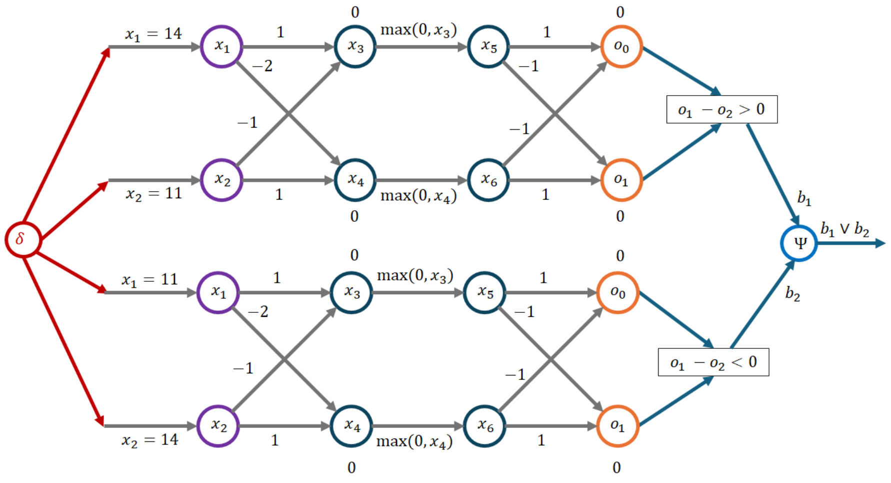


**Figure 3.15:** Relational specification for defining UAP robustness on two distinct inputs. Here the same perturbation $\delta \in \mathbb{R}^2$ is applied to both inputs and $\psi$ ensures that at least one of the perturbed inputs remains correctly classified.

versarial perturbations is often unrealistic, as practical attacks typically involve finding a perturbation that works across a set of inputs rather than targeting a single input. These studies suggest that focusing solely on robustness against input-specific adversarial attacks is overly conservative and offers a pessimistic view of real-world DNN robustness. Because the same adversarial perturbation is applied to all $k$ inputs, the perturbed inputs are related. We can express this relationship using the following cross-executional constraint: $x' - y' = x - y$, where $x'$ and $y'$ represent the perturbed versions of the original inputs $x$ and $y$. Beyond monotonicity and UAP robustness, previous works have also considered other important input-relational properties, such as DNN fairness (Urban *et al.*, 2020b) and global robustness (Wang *et al.*, 2022a).

**Network-relational properties.** Network-relational properties are used to compare the outputs of two different networks on the same input. For instance, if we want to verify whether two networks behave identically over a specific input region, we can express this as a network-relational property. DNNs are often compressed using techniques like quantization (Gholami *et al.*, 2022) and pruning (Liang *et al.*, 2021) to meet practical requirements, such as reducing energy consumption and computational costs. However, it's essential to quantify any potential performance loss post-compression. Paulsen *et al.* (2020) employs relational verification to formally prove that, within specific input regions, the compressed model does not experience any performance degradation.

### 3.2.2 Relational Verifier

In this section, we primarily discuss three input-relational verification techniques: a) **RaVeN** (Banerjee *et al.*, 2024b), b) **RACoon** (Banerjee and Singh, 2024), and c) **RABBit** (Suresh *et al.*, 2024), due to their scalability in terms of both DNN size and the number of executions $k$. Note that the input relational verifiers like RACoon can handle network-relational properties, such as the local network equivalence problem (Paulsen *et al.*, 2020). Hence, we will limit our discussion to input-relational verifiers.

The main difference between non-relational and relational verification problems lies in the cross-executional input constraints $\varphi_\delta$, which capture dependencies between inputs used in different DNN executions. Leveraging these cross-executional dependencies is crucial for improving the precision of relational verifiers. Before going into the details of specific verification algorithms, we first discuss the significance of utilizing cross-executional dependencies through an illustrative example.

#### Illustrative Example

**Network.** For this example, we consider the network, $f_{ex}$, with three layers: two affine layers and one ReLU layer with two neurons each (Figure 3.16). The weights on the edges represent the coefficients of the weight matrix used by the affine transformations applied at each layer, and the learned bias for each neuron is shown above or below it.
**Relational property.** We verify the UAP verification problem described in Section 3.2.1 on $f_{ex}$ where the relational property is defined over 2 separate executions of $f_{ex}$. $\varphi_1$, $\varphi_2$ and $\varphi_\delta$ are defined as follows where $i_1^* = [14, 11]^T$, $i_2^* = [11, 14]^T$, and $\epsilon = 6$.

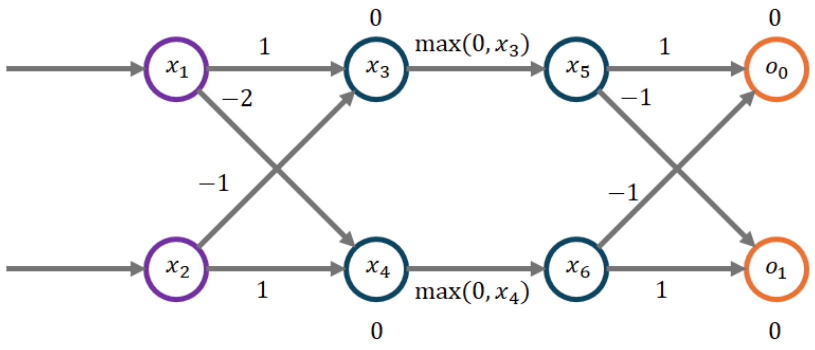


**Figure 3.16:** Representation of $f_{ex}$ used in the illustrative example

$$\varphi_1 \coloneqq (\|i_1 - i_1^*\|_\infty \le \epsilon) \;\; \varphi_2 \coloneqq (\|i_2 - i_2^*\|_\infty \le \epsilon)$$
$$\varphi_\delta \coloneqq ((i_1 - i_2) = (i_1^* - i_2^*))$$

In UAP verification, an adversary can select to attack the DNN with any perturbation $\delta$ such that $\|\delta\|_\infty \le \epsilon$ but the same perturbation $\delta$ must be applied to both inputs - $i_1^*, i_2^* \in \mathbb{R}^2$. Therefore the two executions are related and tracking this relationship improves precision. In contrast, in the common local robustness problem, an adversary can choose different perturbations for the two inputs and therefore the two executions are unrelated and can be verified independently. Any input $i_1 \in \mathbb{R}^2$ inside the $L_\infty$ ball defined by $\|i_1 - i_1^*\|_\infty \le \epsilon$ is not misclassified if $(f_{ex}(i_1) = [o_1, o_2]^T) \wedge (o_1 - o_2 > 0)$ holds. Conversely, any input $i_2 \in \mathbb{R}^2$ lying inside the $L_\infty$ ball - $\|i_2 - i_2^*\|_\infty \le \epsilon$ is not misclassified if $(f_{ex}(i_2) = [o_1, o_2]^T) \wedge (o_2 - o_1 > 0)$ holds. We want to formally verify that there does not exist an adversarial perturbation $\delta \in \mathbb{R}^2$ with $\|\delta\|_\infty \le \epsilon$ such that both the inferences on inputs $i_1 = i_1^* + \delta$ and $i_2 = i_2^* + \delta$ produces incorrect classification results (encoding in Figure 3.15).

The key challenge lies in utilizing the cross-executional input constraint $\varphi_\delta$ (e.g., ensuring that the perturbation $\|\delta\|_\infty$ remains the same across inputs). For instance, if we apply non-relational verifiers (e.g., Zonotopes) to input regions defined by $\varphi_1$ and $\varphi_2$ in isolation, without considering the cross-executional constraint $\varphi_\delta$, we cannot verify the property. In this case, the lower bound of $(o_1 - o_2)$ w.r.t. the input region $\varphi_1$ and the lower bound of $(o_2 - o_1)$ w.r.t. the input region $\varphi_2$ are

$-13.25$ and $-31.44$ respectively when computed independently, which is insufficient to prove the output specification. However, relational verifiers like **RaVeN** use specialized abstract domains, such as **DiffPoly**, to infer linear constraints across the outputs of each layer in $f_{ex}$. This approach captures cross-executional dependencies, enabling the verification of relational properties. Figure 3.17 illustrates how computing these additional linear constraints enhances the precision of the analysis.

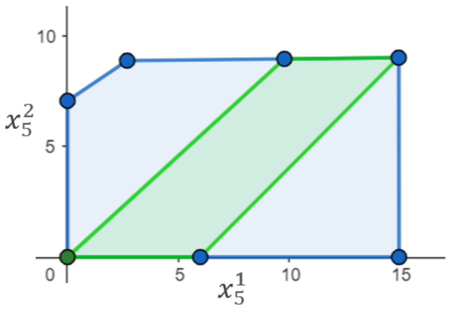


**Figure 3.17:** For the variables $x_5^1$ and $x_5^2$ the convex region (green) obtained with constraints from DiffPoly analysis is more precise than the convex region (blue) formed without the difference constraints.

**Relational verification algorithms.** The first approach to UAP robustness verification, known as I/O Formulation (Zeng *et al.*, 2023), breaks down the relational verification process into two stages. In the first stage, the I/O formulation applies existing non-relational verifiers like DeepPoly to each local input region $\varphi_i$. Through this analysis, it derives a local linear approximation $g_i(\alpha(\varphi_i))$ of the network $f$ for each $\varphi_i$ (see Zeng *et al.*, 2023 for details). In the second stage, the I/O formulation utilizes these linear approximations $g_i(\alpha(\varphi_i))$ and represents the cross-execution constraints $\varphi_\delta$ as a set of linear inequalities, which are then used to construct a Mixed Integer Linear Program (MILP). The MILP instance is subsequently optimized using an off-the-shelf solver (Gurobi Optimization, LLC, 2018) to produce the verification result. However, I/O Formulation tracks cross-execution dependencies only at the input layer. Because the linear approximations $g_i(\alpha(\varphi_i))$ are obtained independently, without capturing dependencies between the executions, it loses precision. Additionally, I/O Formulation is limited to verifying UAP robustness and cannot handle other input-relational properties, such as monotonicity. RaVeN (Banerjee *et al.*, 2024b) addresses both limitations by introducing a new abstract domain called DiffPoly, which includes appropriate abstract transformers for the affine and activation layers of the DNN. In the following sections, we will describe the DiffPoly domain and outline the key steps of RaVeN.

**DiffPoly domain.** RaVeN leverages the DiffPoly domain to track dependencies between the outputs of a DNN across all layers for two executions. Since common relational properties like UAP robustness and monotonicity involve bounded differences between pairs of inputs, the DiffPoly domain focuses on efficiently capturing the difference relationship. DiffPoly operates over the product DNN, containing two copies of the same DNN corresponding to different executions. For a neuron $x_i$ in the original DNN, we use $x_i^1$ and $x_i^2$ to refer to its two copies in the product DNN. DiffPoly analysis not only maintains the symbolic and concrete bounds for each variable, as done in DeepPoly, but also tracks additional bounds on the **difference** between two copies of the same variable (e.g., the difference between two instances $x_5^1, x_5^2$ of $x_5$ in Figure 3.17) to improve the analysis precision. The domain provides specialized abstract transformers for bounding the difference between the outputs of non-linear functions applied to the same neuron in different related executions.

RaVeN uses DiffPoly analysis to infer linear constraints for variables from each execution pair from the set of $k \geq 2$ executions, capturing cross-execution dependencies across all DNN layers. Finally, by leveraging these cross-execution linear constraints, RaVeN formulates a MILP instance and solves it using an off-the-shelf MILP solver (Gurobi Optimization, LLC, 2018) to produce the verification results (see Figure 3.18). Note that regardless of the size of the DNN, RaVeN introduces only a constant number of integer variables per execution in the formulated MILP instance. This design prevents an exponential increase in the optimization time of the MILP instance, enabling RaVeN to scale effectively to large DNNs.

**Relational property guided refinement.** Although RaVeN is much more precise than I/O Formulation, it has two main drawbacks: a)

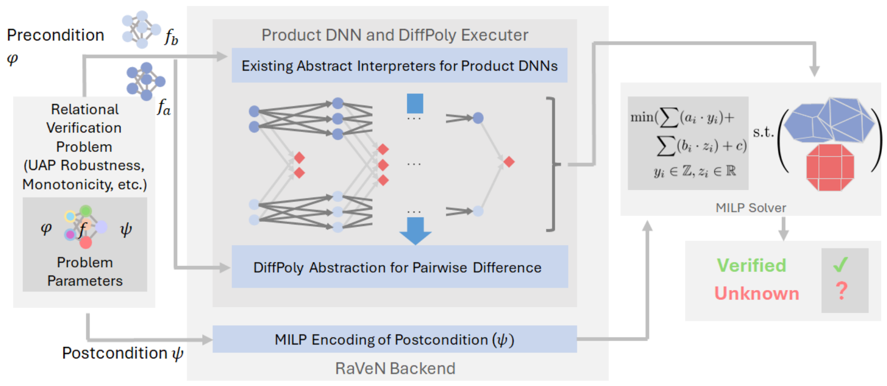

Precondition $\varphi$
$f_b$
$f_a$
Relational Verification Problem (UAP Robustness, Monotonicity, etc.)
$\varphi$ $f$ $\psi$
Problem Parameters
Product DNN and DiffPoly Executer
Existing Abstract Interpreters for Product DNNs
DiffPoly Abstraction for Pairwise Difference
MILP Encoding of Postcondition ($\psi$)
RaVeN Backend
Postcondition $\psi$
$\min(\sum(a_i \cdot y_i)+ \sum(b_i \cdot z_i)+c)$ s.t.
$y_i \in \mathbb{Z}, z_i \in \mathbb{R}$
MILP Solver
Verified ✓
Unknown ?

**Figure 3.18:** The overview of the sound and incomplete relational verifier RaVeN. Given a DNN $f$ and a relational property $(\varphi, \psi)$ relating $k$ DNN inferences we show the flow of RaVeN along with the key steps - (i) constructing the product DNN by duplicating $f$ $k$ times and analyzing the product DNN with an existing DNN abstract interpreter, (ii) computing pairwise differences of outputs of all $k$ inferences at each layer with DiffPoly analysis that uses concrete lower and upper bounds of each variable in the product DNN, (iii) combining DiffPoly analysis and product DNN analysis with an existing DNN abstract interpreter to infer layerwise linear constraints over outputs of all $k$ DNN executions that preserves dependencies between different DNN executions, (iv) encoding the postcondition as a MILP objective and formulate MILP with layerwise linear constraints computed in step (iii). Finally, it uses an off-the-shelf MILP solver (Gurobi Optimization, LLC, 2018) to verify the relational property by solving the corresponding MILP.

While it adds only a constant number of integer variables per execution, it also introduces linear constraints from the DiffPoly analysis for each pair of executions, which can become computationally expensive to handle as the number $k$ of executions increases; b) RaVeN performs a single verification pass for each relational property, and if the generated MILP instance fails to prove the property, it cannot refine the result. Subsequent works, RACoon (Banerjee and Singh, 2024) and RABBit (Suresh *et al.*, 2024), address both of these limitations. RaVeN's computational bottleneck comes from the large number of linear constraints added at each DNN layer. To overcome this, RACoon replaces the static symbolic and concrete bounds used by RaVeN with parametric bounds for variables at each layer in every execution. RACoon then refines these parametric bounds **jointly** over multiple executions to facilitate

verification of the specific relational property being analyzed. The parametric bounds correspond to dual variables from the Lagrangian dual of the LP relaxation of RaVeN's MILP instance. However, instead of optimizing these dual variables directly with an LP/MILP solver, RACoon introduces a gradient-descent-based refinement algorithm, avoiding the computational blowup of an off-the-shelf solver.
Although RACoon improves the scalability of RaVeN, since it only considers the dual of the MILP instance (or, LP relaxed version of the MILP instance) it does not improve on the precision over RaVeN. To address this, RABBit (Suresh *et al.*, 2024) introduces a "Branch and Bound" solver designed for DNNs with piecewise linear activation functions like ReLU. In the branching step, RABBit selects an activation node, decomposes the piecewise linear activations into linear functions, and explores subproblems corresponding to each linear segment. For the bounding step, RABBit, like RACoon, employs parametric bounds and refines them jointly for each subproblem across multiple executions.

## 3.3 Probabilistic Analysis

For deterministic DNNs, both single execution and relational properties can be verified *quantitatively* by computing the subset $\varphi_f$ of $\varphi$ for which the DNN output satisfies the postcondition. The fraction $p_{\varphi,\psi} = \frac{\varphi_f}{\varphi}$ is the probability that for any $x \in \varphi$ $f(x) \in \psi$ holds leading to probabilistic guarantees. Qualitative verification discussed in Sections 3.1 and 3.2 can be seen as a special case of quantitative verification where we check whether $\varphi_f = \varphi$ and, therefore, the probability should be one. Complete quantitative verifiers can compute $p_{\varphi,\psi}$ exactly; however, the problem is undecidable in the general case. A lower bound on the probability can be computed using abstract interpretation-based incomplete verifiers. If the verifier proves the specification for $\varphi$, then the probability is one. Otherwise, one can leverage input splitting, discussed in Section 3.1.5 to find an underapproximation $\varphi_g \subseteq \varphi_f$ of the region where the property holds. The fraction $\frac{\varphi_g}{\varphi}$ is a *sound* lower bound on $p_{\varphi,\psi}$ (Wei *et al.*, 2023). An upper bound on the probability can also be computed with abstract interpretation by finding regions containing only counterexamples. See the work of Dimitrov *et al.* (2022), which leverages DeepPoly.

Another class of probabilistic specifications for deterministic DNNs arises when the precondition $\varphi$ defines a probability distribution over the input space. For example, the transformations $T$ applied to an input $x$ in the local robustness problem can be samples from a noise distribution (e.g., uniform, Gaussian), leading to a probabilistic specification. The verification problem is to find the probability that the DNN output $f(x)$ satisfies $\psi$ given a random input $x \sim \varphi$ (Fazlyab *et al.*, 2019a; Mirman *et al.*, 2021; Pilipovsky *et al.*, 2023; Păsăreanu *et al.*, 2020). One can treat a DNN with probabilistic input as a *probabilistic program*, which represents rules for computing complicated probability distributions as code (Goodman *et al.*, 2008). The example below shows probabilistic local robustness specification with transformations sampled from a noise distribution.

```
def program (x):
    T = NoiseDistribution(parameters)
    x' = T(x)
    return f(x')
```

Transformation $T$ can be a scalar, a vector, or a matrix, where the distribution's parameters control each element perturbation. Commonly, one can choose a Gaussian distribution $\mathcal{N}$ with mean and variance as its parameters. As the program computes with probability distributions rather than with individual values, the underlying analysis needs to keep track of a sound approximation of the distribution function. Abstract interpretation has a rich history of being applied to analyze probabilistic programs, with standard abstract domains, e.g., see Cousot and Monerau (2012), Mardziel *et al.* (2013), and Sankaranarayanan *et al.* (2013). Recent works extend the Box abstraction to track lower and upper bounds on the probability density (Huang *et al.*, 2021; Zhou *et al.*, 2023) or cumulative density functions (Ferson *et al.*, 2015) of an interval of values of a random variable. The works of Bouissou *et al.* (2012), Bouissou *et al.* (2016), and Goubault and Putot (2025) define the probabilistic Zonotope domain that can be used to analyze neural networks. A probabilistic version of the StarSet domain for reasoning about probabilistic specifications over neural networks was introduced in Tran *et al.* (2023). Existing abstract interpreters for symbolically propagating probability distributions through DNNs are less scalable than those for propagating a set of points.

For stochastic DNNs constructed from randomization in learned parameters, such as Bayesian neural networks (BNNs), probabilistic specifications measure the probability that a deterministic DNN $f_w$ corresponding to a random sample from the posterior distribution of learned weights $w$, satisfies a given single execution or relational specification. Abstract interpretation can be leveraged for computing a lower bound on this probability for certain posterior distributions. The probabilistic verifier first constructs an interval neural network (INN) around a sampled $w$. In an INN, the learned weights are not scalars but intervals (Prabhakar and Rahimi Afzal, 2019). Abstract interpreters such as Boxes and DeepPoly/CROWN are adapted to verify whether $(\varphi, \psi)$ holds on an INN. The sets of weight intervals on which the abstract interpreter proves the specification are collected and then converted into a lower bound on the probability. For details, see the works of Wicker *et al.* (2020) and Batten *et al.* (2024). An upper bound can also be obtained using abstract interpreters by modifying the above procedure. For details, see Wicker *et al.* (2023).

For stochastic DNNs based on latent variable sampling, such as variational autoencoders (VAE), each input in $\varphi$ maps to a probability distribution in the output space. $\psi$ in this case defines constraints that each output probability distribution should satisfy. For example, one can require that the expectation of each output distribution satisfies Box constraints. Extending probabilistic abstract interpretation to this setting is an open area of research. We refer the reader to the works of Dvijotham *et al.* (2018a) and Berrada *et al.* (2021) that design verifiers based on Lagrangian optimization.

## 3.4 Incremental Analysis

In this section, we study the iterative development procedure for obtaining fast, accurate, and trustworthy DNNs, which we introduced in Figure 1.1 in the Introduction. The most expensive step in the development workflow is running a DNN verifier. Domain experts usually design a large number of specifications (around 10-100K), typically defined for inputs in the test set. Therefore, the expensive verifier needs to be run several thousand times on the same DNN. While there has been a lot of work in recent years on developing precise and scalable verifiers, they do not *scale* in the deployment setting: they can precisely verify individual specifications in a few seconds or minutes, however, the verification of a large and diverse set of specifications on a single DNN can take multiple days to years or the verifier can run out of memory. Given that multiple networks are generated due to repair or retraining, using existing verifiers for trustworthy development is infeasible. The inefficiency is because the verifier needs to be run from scratch for every new pair of specifications and DNNs. A straightforward approach to overcoming this limitation is to run the verifier on several machines. However, such an approach is not sustainable due to its huge environmental cost (Wu *et al.*, 2022a; Bender *et al.*, 2021). Further, in many cases, large computational resources are not available. For example, to preserve privacy, reduce latency, and increase battery lifetime, DNNs are increasingly employed on edge devices with limited computational power (Wang *et al.*, 2020; Chugh *et al.*, 2021). Therefore, for sustainable, democratic, and trustworthy DNN development, it is essential to develop new general approaches for improving the verifier scalability, when verifying multiple specifications and networks. In recent years, approaches to enable incremental application of abstract interpretation-based incomplete DNN verifiers have been developed to address these challenges based on the novel concepts of *proof sharing* and *proof transfer*. We describe these in detail next. We use the notation $f^{i:j}(T)$ to refer to applying the DNN transformations from layer $i$ till layer $j$ on an input set $T$. We refer the interested readers to Ugare *et al.* (2023) and Tang (2024) for incremental verification of complete verifiers.

### 3.4.1 Proof Sharing Across Specifications on the Same DNN

Proof sharing focuses on improving the efficiency of abstract interpretation-based verifiers when verifying a large number $(r)$ of specifications $(\varphi_i, \psi)$ on a single DNN with different preconditions but the same postcondition. The specifications can be local or global. Examples of this scenario include verifying popular specifications for image classifiers such as robustness against norm-based or geometric perturbations (Singh *et al.*, 2019b; Balunovic *et al.*, 2019) that have the same postcondition but define different local regions.

Proof sharing can be applied for boosting the verification of any DNN for which a baseline abstract interpretation-based verifier $V$ is available. The high-level idea behind proof sharing is shown in Figure 3.19. We focus on a feedforward network with $L$ layers to simplify the presentation. Incremental verification with proof sharing involves two steps:

1. Generate a set $\mathcal{T}$ of $m \in \mathbb{Z}^+$ symbolic shapes as proof templates at an intermediate layer $k < L$ such that each template $T \in \mathcal{T}$ implies the postcondition $\psi$, i.e., network output for the template $T$ satisfies $\psi$. Formally, $\forall T \in \mathcal{T}, f^{k+1:L}(T) \subseteq \psi$.
2. For a new specification $(\varphi, \psi)$, propagate the abstract element $\alpha(\varphi)$ till layer $k$ with the given verifier $V$ computing $S = V(f^{1:k}, \varphi)$. Check whether $S \subseteq T$ holds for one of the templates in $\mathcal{T}$. If yes, then the proof is complete; otherwise, run the verifier till the last layer to compute $S' = V(f^{k+1:L}, S)$ and check $S' \subseteq \psi$.

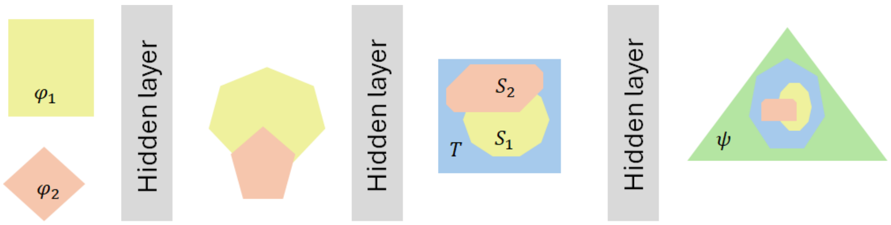


**Figure 3.19:** The concept of proof sharing across multiple specifications on the same network. Here, the DNN output with respect to the template $T$ is $\subseteq \psi$. The symbolic outputs $S_1$ and $S_2$ of the DNN for the preconditions $\varphi_1$ and $\varphi_2$, respectively, are $\subseteq T$. This is sufficient to prove that $f(\varphi_1) \subseteq \psi$ and $f(\varphi_2) \subseteq \psi$ holds.

By construction, it is guaranteed that verification with proof sharing is at least as precise as the baseline analysis without proof sharing. The verifier for template generation and verifying the target specifications can be different. Proof sharing can be generalized to work with templates

at multiple layers. Step 2 above can be adapted such that $S$ is covered by multiple templates from $\mathcal{T}$ with each such template covering $S$ partially.
**Complexity.** Let $\rho \in [0, 1]$ be the rate at which the inclusion checks $S \subseteq T$ for $T \in \mathcal{T}$ succeeds, then the runtime cost of verification with proof sharing across multiple specifications is:

$$t_{PT} = t_{\mathcal{T}} + r(t_S + t_{\subseteq} + (1 - \rho)t_{\psi}) \tag{3.5}$$

where $t_{\mathcal{T}}$ is the cost of generating the template set $\mathcal{T}$, $t_S$ is the cost of computing $S$, $t_{\subseteq}$ is time to check $S \subseteq T$ for $T \in \mathcal{T}$ until a match is found, and $t_{\psi}$ is the time required to check whether $V(f^{k+1,L}, S) \subseteq \psi$ holds. In contrast, the cost of verification without proof sharing is $t_{BL} = r(t_S + t_{\psi})$.
**Reducing runtime.** The cost $t_{PT}$ of verification with proof sharing is smaller than the baseline $t_{BL}$ if $t_S, t_{\mathcal{T}}, t_{\subseteq}$ are significantly smaller than $t_{\psi}$ and $\rho$ is large (i.e., close to 1). However, these requirements are at odds with each other. First, generating optimal $\mathcal{T}$ that maximizes proof sharing for any verifier requires reasoning about all possible verifiers which is not possible with current methods. For a fixed verifier $V$, generating the optimal $\mathcal{T}$ maximizing $\rho$ corresponding to a set of $r$ specifications $\{(\varphi_i, \psi)\}$ can be posed as the solution to the following optimization problem where [.] is the indicator function:

$$\operatorname{argmax}_{\mathcal{T}} \sum_{i=1}^{r} \left[ \bigvee_{T \in \mathcal{T}} V(f^{1:k}, \varphi_i) \subseteq T \right], \text{s.t. } \forall\, T \in \mathcal{T}.\, f^{k+1:L}(T) \subseteq \psi \tag{3.6}$$

Unfortunately, solving (3.6) is still computationally infeasible as it requires computing the pre-image of the non-linear network function $f^{k+1:L}$ with respect to $\psi$. Further, even if optimal $T$ could be somehow computed, the resulting templates have complicated shapes for which the inclusion check $t_{\subseteq}$ is expensive. Therefore while the high-level idea behind proof sharing is simple, actually obtaining speedup requires careful design of new representations for templates, to enable fast inclusion check, and novel algorithms for generating them such that $t_{\mathcal{T}}$ is reduced and $\rho$ is large.
**Results.** Fischer *et al.* (2022) instantiated the proof sharing framework for speeding up the Zonotope analysis (Singh *et al.*, 2018). The results

show that proof sharing enables upto 3x speedup over the vanilla Zonotope analysis for proving the robustness of classifiers for the popular MNIST (LeCun *et al.*, 1989) and CIFAR10 (Krizhevsky, 2009) datasets, based on fully-connected architectures (the largest network had 9 layers with 500 neurons per layer), against challenging patch-based (Chiang *et al.*, 2020) and geometric perturbations (Balunovic *et al.*, 2019). The templates are constructed by considering a relaxed version of (3.6):

$$\begin{gathered}\operatorname{argmax}_{\hat{\varphi}_1,\ldots,\hat{\varphi}_m} \sum_{i=1}^{r} \left[ \bigvee_{j=1}^{m} V(f^{1:k}, \varphi_i) \subseteq T_j \right], \text{where} \\ T_j = \alpha_{\text{Box}}(V(f^{1:k}, \hat{\varphi}_j)), \text{s.t. } V(f^{k+1:L}, T_j) \subseteq \psi\end{gathered} \tag{3.7}$$

In contrast to (3.6), the template generation in (3.7) is tied to a chosen verifier $V$ and a small set $\{(\hat{\phi}_i, \psi) | 1 \leq i \leq m\}$ (with $m << r$) of specifications, different from the target specifications $\{(\varphi_i, \psi)\}$. The templates are generated by first running the vanilla Zonotope analysis on $\{(\hat{\varphi}_i, \psi)\}$ and collecting the zonotopes produced at an intermediate layer. Zonotopes are not ideal for use as templates for proof sharing as matching them against other zonotopes $S$ computed when verifying target specifications is expensive using existing algorithms for inclusion checks (Sadraddini and Tedrake, 2019). Therefore, zonotopes are converted to simpler box shapes $T_j$ via a heuristic function $\alpha_{Box}$ that tries to find the largest box around the zonotope for which the Zonotope analysis can prove that $V(f^{k+1:L}, T_j) \subseteq \psi$.

The layer for template generation is selected by running the proof sharing enabled verifier on a small subset of the target specifications $\{(\varphi_i, \psi)\}$. The empirical observations suggest that the overlap between the shapes produced for different robustness specifications increases for layers later in the network, which enables better matching rates at these layers. Finally, while the templates are generated using the Zonotope analysis, they can be used to enable proof sharing with other verifiers without any modification.

### 3.4.2 Proof Transfer Across Multiple Similar DNNs

We consider specifications of the form considered in Section 3.4.1, but now the goal is to transfer the proof obtained when verifying multiple specifications on one DNN for boosting the verification of a large number of specifications on multiple similar DNNs produced by repeated application of different model repair algorithms before and after deployment (Figure 1.1). Proof transfer focuses on boosting verifier efficiency in this scenario. Figure 3.20 illustrates the workflow of incremental verification based on proof transfer. The set of specifications on the similar DNNs can be the same (before deployment) or different (after deployment in the case of a data shift) from the original DNN. Proof transfer requires that the original and similar networks have the same architecture. Incremental verification with proof transfer involves the following three steps:

1. Generate a set $\mathcal{T}$ of $m$ templates at an intermediate layer $k < L$ on the original network such that the output of similar networks corresponding to each $T \in \mathcal{T}$ is likely to satisfy $\psi$.
2. Transfer $\mathcal{T}$ generated on the original network to a similar network at layer $k$ by incrementally modifying each $T \in \mathcal{T}$ to compute $T^{sim}$, such that the output of the similar network corresponding to $T^{sim}$ satisfies $\psi$, generating a new template set $\mathcal{T}^{sim}$.
3. Run the given verifier $V$ for the target specification $(\varphi, \psi)$ on similar networks till layer $k$ computing $S^{sim} = V(f_{sim}^{1:k}, \varphi)$ and check whether $S^{sim} \subseteq T^{sim}$ holds for one of the templates in $\mathcal{T}^{sim}$. If yes, then the proof is complete; otherwise, run the verifier till the last layer and check accordingly.

By construction, the proof transfer framework ensures that verification with proof transfer on similar networks is at least as precise as vanilla verification. Note that $\mathcal{T}$ is generated only once on the original network while template transformation is performed for each network. **Complexity.** Let there be $p$ similar networks, including the original one, and $r_i$ be the number of specifications to be verified on the i-th similar network, $\rho_i \in [0, 1]$ be the rate at which the check $S^{sim} \subseteq T^{sim}$

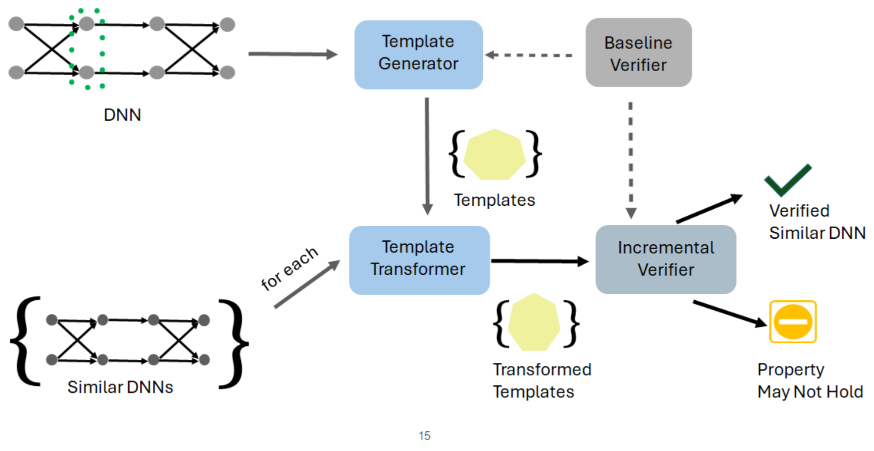


**Figure 3.20:** Workflow of proof transfer from left to right. It consists of three components: template generator, template transformer, and incremental verifier. First, the template generator takes the DNN $f$ as input and creates a set of templates. For each similar DNN, the template transformer transforms the templates and is used by the incremental verifier to verify the similar network. The incremental verifier either successfully verifies the network and generates a certificate or reports that the property may not hold.

succeeds on the i-th network, then the total cost of verification with proof transfer is:

$$t_{PT} = t_{\mathcal{T}} + \sum_{i=1}^{p} t^i_{\mathcal{T}^{sim}} + \sum_{i=1}^{p} r_i(t^i_{\mathcal{S}^{sim}} + t^i_{\subseteq} + (1-\rho_i)t^i_{\psi}) \qquad (3.8)$$

where $t_{\mathcal{T}}$ is the cost of generating templates on the original network, $t^i_{\mathcal{T}^{sim}}$ is the cost of transforming $\mathcal{T}$ to $\mathcal{T}^{sim}$ on the i-th network. $t^i_{\mathcal{S}^{sim}}, t^i_{\subseteq}, t^i_{\psi}$ have the same meaning as in Section 3.4.1 on the i-th network. In contrast, the runtime of vanilla verification is $t_{BL} = \sum_{i=1}^{p} r_i(t^i_{\mathcal{S}^{sim}} + t^i_{\psi})$.

**Improving runtime.** Since templates are generated only once on the original DNN, the cost of generating them can be amortized when the number of similar networks is large. This is the case in the development pipeline of Figure 1.1. The runtime of verification with proof transfer for verifying all $\sum_{i=1}^{p} r_i$ specifications is minimized when the cost of template generation is efficiently amortized, template transformation $t^i_{\mathcal{T}^{sim}}$ and inclusion checks are fast with template matching rates $\rho_i$

close to 1. Generating optimal templates on the original network that are also valid across multiple networks is a harder problem than (3.6) as we now need to reason about multiple DNNs. Optimal templates for similar networks can be significantly larger or smaller than the optimal templates on the original network. Thus, obtaining optimal templates on similar networks with template transformation may be as computationally infeasible as generating them from scratch. Therefore, obtaining speedups across similar networks requires careful design of new algorithms for template generation and transformation.

**Results.** FANC (Ugare *et al.*, 2022) considered similar DNNs obtained after iteratively applying model repair for improving inference speed by popular techniques such as quantization (Gholami *et al.*, 2021) and pruning (Blalock *et al.*, 2020). The specifications involved proving the robustness of classifiers for the MNIST and CIFAR10 datasets, based on fully-connected and convolutional architecture (the largest network had 8 layers and $8,960$ neurons), against challenging patch (Chiang *et al.*, 2020), $L_0$-norm (Ruan *et al.*, 2019), geometric (Balunovic *et al.*, 2019), and brightening perturbations (Pei *et al.*, 2017). The results show that proof transfer makes Zonotope analysis up to 4x faster. Template generation on the original network was verifier-specific (as in Section 3.4.1) and involved computing boxes by running the verifier on a small set of $L_\infty$-norm based specifications. The template transformation expands the box template by joining it with another box generated heuristically. The same layer, determined by a similar method as for proof sharing (Fischer *et al.*, 2022), was used for storing templates on the original DNN and its approximate versions.

# 4

# Training with Differentiable Abstract Interpreters

DNNs trained only to maximize accuracy with standard training (Kingma and Ba, 2015) are often unsafe (Madry *et al.*, 2017). This section describes how the feedback from abstract interpreters can be incorporated into the training loop to obtain DNNs with better safety guarantees. While the description here applies to different safety properties, we focus on local robustness against adversarial attacks, as it is the most common property for abstract interpretation-guided training considered in the literature. We will use the notation $f_w$ to denote a DNN parameterized by the learnable weight parameters $w$ and $\varphi(x) = \{x' \mid x' \in T(x), T \in \mathcal{T}\}$ to represent an adversarial region obtained by applying transformations $T \in \mathcal{T}$ to $x$. The most common transformation for DNN training with abstract interpretation considered in the literature is adding a perturbation $\delta \in \mathcal{B}_p(0, \eta)$ to $x$, where $||.||_p$ represents the $p$-norm and $\mathcal{B}_p(x, \eta) = \{x + \delta \in \mathbb{R}^m \mid ||x + \delta||_p \leq \eta\}$ defines the set of perturbed inputs in an $L_p$-ball of radius $\eta \in \mathbb{R}$ around $x$. The abstract interpretation-guided training methods discussed in this section can also be used to generate larger AI-enabled systems with stronger end-to-end safety guarantees (Yang *et al.*, 2024a).

## 4.1 General Formulation for Deterministic DNNs

We consider robust training of classifiers, which involves defining a differentiable loss function $\mathcal{L}_R$ encoding the robustness specification for each point $x' \in \varphi(x)$ with the property that $\mathcal{L}_R$ at $x'$ is $\leq 0$ iff $x'$ is classified correctly. In other words, in the DNN output $y = f_w(x')$, the score $y_c$ for the correct class $c$ is higher than all other classes $y_i$, i.e., $y_c > y_i$ for all $i$. The DNN is robust iff $\mathcal{L}_R \leq 0$ for all $x' \in \varphi(x)$. An example of $\mathcal{L}_R$ is $\max_{i \neq c} y_i - y_c$ (Mirman *et al.*, 2018).

(4.1) shows the mathematical characterization of the training problem for local robustness. Here, one considers the maximum violation of the robustness loss $\mathcal{L}_R$ within each $\varphi(x)$ corresponding to inputs $x$ from the training distribution $\mathcal{I}$. Note, if we consider average loss within $\varphi(x)$ instead of the maximum, then it will not yield robust DNNs even if the loss was perfectly minimized to 0 during training. The training goal is to learn the weight parameters $w$ so that the expected value of the maximum robustness loss over $\mathcal{I}$ is minimized. This min-max formulation makes robust training a harder optimization problem than standard training. Computing the maximum robust loss for each $\varphi(x)$ exactly requires computing $f_w(\varphi(x))$, which is an undecidable problem in general. Therefore an approximation of $\mathcal{L}_R$ is computed in practice.

$$\begin{aligned}
&\textbf{find } w \\
&\textbf{minimize } \rho(w) \qquad (4.1)\\
&\text{where } \rho(w) = \mathbb{E}_{(x,y)\sim\mathcal{I}}[\mathbf{max}_{x' \in \varphi(x)} \mathcal{L}_R(w, x', y)]
\end{aligned}$$

Adversarial training methods (Madry *et al.*, 2017), a form of counter-example guided learning, compute a lower bound on the worst-case robust loss by heuristically computing a point $x_{adv} \in \varphi(x)$ at which the robust loss is high but not guaranteed to be maximum. This yields a lower bound on the maximum robust loss as $\mathcal{L}_R(w, x_{adv}, y) \leq \max_{x' \in \varphi(x)} \mathcal{L}_R(w, x', y)$. $x'$ is then augmented to the training dataset. Minimizing weight parameters with respect to a lower bound of the maximum loss means that even if the lower bound is $\leq 0$, the actual loss can be $> 0$. As a result, while DNNs trained with adversarial training are harder to attack than those with standard training, they are often not provably robust.

$$\begin{aligned}&\textbf{find } w\\&\textbf{minimize } \rho(w)\\&\text{where } \rho(w)=\mathbb{E}_{(x,y)\sim\mathcal{I}}[\mathbf{max}_{z\in\gamma(g_w(\alpha(\varphi(x))))}\mathcal{L}_R(w,z,y)]\end{aligned}\tag{4.2}$$

On the other hand, certified training methods (Wong *et al.*, 2018; Mirman *et al.*, 2018; Xu *et al.*, 2020; Zhang *et al.*, 2020; Gowal *et al.*, 2018) compute an upper bound on the worst-case robust loss using abstract interpretation-based DNN verifiers. (4.2) shows the formulation of certified training as an optimization problem where the inner maximization differs from adversarial training. While the latter tries to find $x' \in \varphi(x)$ for which the robustness loss computed using $f_w(x')$ is maximized, certified training tries to find a point $z \in \mathbb{R}^n$ in the concretization of the analysis output $g_w(\alpha(\varphi(x)))$ maximizing the loss. Notice that the abstract transformers of the verifier and not the concrete transformers of the DNN are used for computing the loss. Certified training therefore operates on an overapproximation $g_w$ of $f_w$ within the precondition $\varphi(x)$, computed by the abstract interpreter. The abstract computations have the same parameters $w$ as the original DNN. For differentiable optimization, the computations of the abstract interpreter must be expressible as a differentiable function of the weight parameters $w$. Examples of popular differentiable abstract domains include the Box (Gowal *et al.*, 2018; Mirman *et al.*, 2018), Zonotope (Balunovic and Vechev, 2020), and DeepPoly/CROWN (Zhang *et al.*, 2020; Lyu *et al.*, 2021). Figure 4.1 shows the high-level idea behind certified training with differentiable abstract interpreters. The parameter updates during training optimization modify both the concrete and abstract computations.

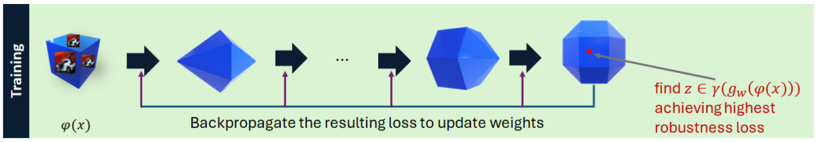

Training
$\varphi(x)$
...
Backpropagate the resulting loss to update weights
find $z \in \gamma(g_w(\varphi(x)))$ achieving highest robustness loss

**Figure 4.1:** Certified training involves computing the point $z \in \gamma(g_w(\alpha(\varphi(x))))$ where the robust loss is maximum. The resulting loss is backpropagated through the verifier code to update the model parameters.

Because $\gamma(g_w(\alpha(\varphi(x)))) \supseteq f_w(\varphi(x))$, the maximum robustness loss computed at $z \in \gamma(g_w(\alpha(\varphi(x))))$ is at least as worse as the maximum $\mathcal{L}_R$ for $f_w$. This results in an upper bound on the true loss. As a result, when this loss is $\leq 0$, the actual loss is also $\leq 0$. DNNs trained with certified training achieve higher robustness guarantees than those trained with adversarial training (Mirman *et al.*, 2018). However, $z$ may not correspond to any output of $f_w$ within the region $\varphi(x)$, resulting in unnecessary over-regularization for robustness. Existing differentiable abstract interpreters essentially compute a linear approximation of the DNN behavior within $\varphi(x)$. The certified training updates, therefore, try to regularize the DNN to behave linearly within the specification region. This makes the resulting networks easier to prove (even imprecise analyzers like Box are quite precise) than adversarial and standard training. However, the linearization of the DNN behavior can conflict with standard accuracy. As a result, certifiably trained DNNs are often less accurate than those trained with standard and adversarial training.

To reduce the loss of standard accuracy, in practice, the robust loss is combined with standard accuracy loss during training using different heuristics (Gowal *et al.*, 2018; Zhang *et al.*, 2020; Mirman *et al.*, 2020; Shi *et al.*, 2021). One would expect that training with precise verifiers yields more accurate and robust DNNs than imprecise ones, as they reduce the gap between the actual output $f_w(\varphi(x))$ and the approximation $\gamma(g_w(\alpha(\varphi(x))))$. Precise differentiable verifiers have a high cost, making them unsuitable for larger networks as the verifier is called thousands of times during each training iteration. Further, even for smaller networks, precise verifiers do not improve the accuracy and robustness tradeoff as the optimization problem for training becomes harder with more complex abstract domains (Jovanovic *et al.*, 2022). In practice, the highly imprecise Box domain performs the best for certified training. The works of Baader *et al.* (2020) and Wang *et al.* (2022b) theoretically show that the Box-based training is quite powerful by showing the existence of two DNNs $f, f'$ such that (i) they have the same accuracy, and (ii) Box analysis achieves the same certification results on $f'$ as a more precise verifier on $f$. This implies that we can always construct provably robust neural networks using the Box domain. Next, we will illustrate how the Box-based certified training, also known in the literature as interval bound propagation (IBP), works on a simple toy example.

## 4.2 Interval Bound Propagation (IBP)

We consider a DNN with the same architecture as in Figure 3.2 with two affine layers and one ReLU layer, each containing two neurons. The precondition $\varphi(x)$ defines the range $[-1, 1]$ for inputs $x_1, x_2$ and the postcondition requires us to prove $o_0 > o_1$. The robustness loss $\mathcal{L}_R$ is $o_1 - o_0$. We focus on a snapshot of the model during training, with weights $w$ shown in the top part of Figure 4.2. The bottom of Figure 4.2 shows the abstract network on which the training operates. This network has the same precondition and parameters $w$ but computes a linear overapproximation $g_w(\alpha(\varphi(x)))$ of $f_w(\varphi(x))$ utilizing the Box abstract transformers handling the affine transformation and the ReLU activation.

$$\begin{aligned}
l_3 &:= \mathbf{max}(0, w_{13}) \cdot l_1 + \mathbf{min}(0, w_{13}) \cdot u_1 + \mathbf{max}(0, w_{23}) \cdot l_2 + \mathbf{min}(0, w_{23}) \cdot u_2 \\
l_4 &:= \mathbf{max}(0, w_{14}) \cdot l_1 + \mathbf{min}(0, w_{14}) \cdot u_1 + \mathbf{max}(0, w_{24}) \cdot l_2 + \mathbf{min}(0, w_{24}) \cdot u_2 \\
l_5 &:= \mathbf{max}(0, l_3) \\
l_6 &:= \mathbf{max}(0, l_4) \\
l_{o_0} &:= \mathbf{max}(0, w_{50}) \cdot l_5 + \mathbf{min}(0, w_{50}) \cdot u_5 + \mathbf{max}(0, w_{60}) \cdot l_6 + \mathbf{min}(0, w_{60}) \cdot u_6 \\
l_{o_1} &:= \mathbf{max}(0, w_{51}) \cdot l_5 + \mathbf{min}(0, w_{51}) \cdot u_5 + \mathbf{max}(0, w_{61}) \cdot l_6 + \mathbf{min}(0, w_{61}) \cdot u_6
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
u_3 &:= \mathbf{min}(0, w_{13}) \cdot l_1 + \mathbf{max}(0, w_{13}) \cdot u_1 + \mathbf{min}(0, w_{23}) \cdot l_2 + \mathbf{max}(0, w_{23}) \cdot u_2 \\
u_4 &:= \mathbf{min}(0, w_{14}) \cdot l_1 + \mathbf{max}(0, w_{14}) \cdot u_1 + \mathbf{min}(0, w_{24}) \cdot l_2 + \mathbf{max}(0, w_{24}) \cdot u_2 \\
u_5 &:= \mathbf{max}(0, u_3) \\
u_6 &:= \mathbf{max}(0, u_4) \\
u_{o_0} &:= \mathbf{min}(0, w_{50}) \cdot l_5 + \mathbf{max}(0, w_{50}) \cdot u_5 + \mathbf{min}(0, w_{60}) \cdot l_6 + \mathbf{max}(0, w_{60}) \cdot u_6 \\
u_{o_1} &:= \mathbf{min}(0, w_{51}) \cdot l_5 + \mathbf{max}(0, w_{51}) \cdot u_5 + \mathbf{min}(0, w_{61}) \cdot l_6 + \mathbf{max}(0, w_{61}) \cdot u_6
\end{aligned} \tag{4.4}$$

The analysis associates the bounds $l_1 = -1, u_1 = 1$ and $l_2 = -1, u_2 = 1$ with $x_1, x_2$ respectively, encoding the precondition region. These remain fixed at each training iteration. (4.3) and (4.4), respectively, show the computations of the lower and upper bounds for each neuron using the Box abstract transformers. It can be seen that each

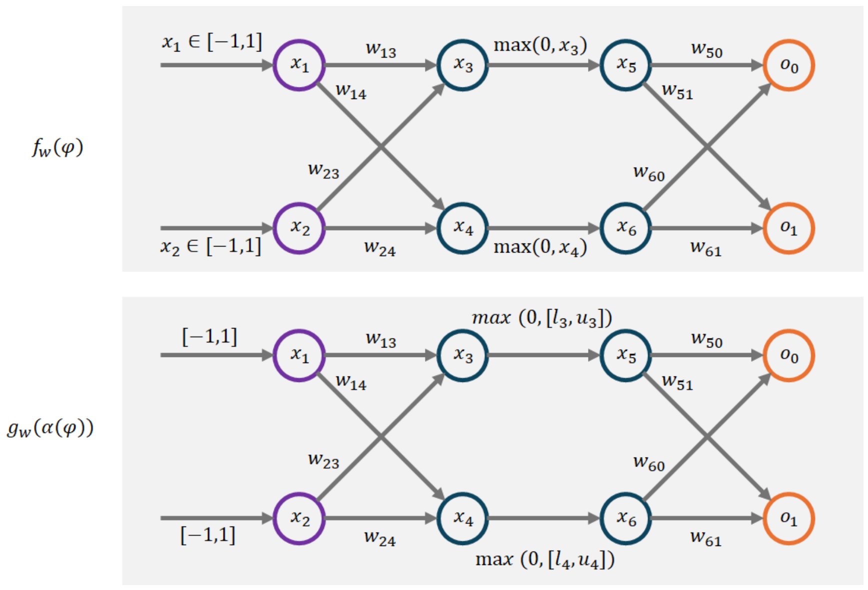


**Figure 4.2:** A toy neural network $f_w$ and its abstract counterpart $g_w$ computed by the Box analysis. Both networks have the same optimizable parameters $w$. $f_w$ operates over concrete values of $x_1, x_2$ from $[-1,1] \times [-1,1]$ using concrete transformers for affine and ReLu layers. $g_w$ operates directly over the intervals for $x_1, x_2$ using the corresponding Box abstract transformers.

bound can be represented as a differentiable function of the trainable parameters $w$. At a given training iteration, the output bounds for $o_0$ and $o_1$ can be computed using the weights at that iteration. Next, we need to solve the inner maximization by finding the point $z \in \gamma(g_w(\alpha(\varphi(x))))$ maximizing the robust loss $\mathcal{L}_R$. We do not need to concretize the analysis output and can show that $z = [l_{o_0}, u_{o_1}]$ maximizes $\mathcal{L}_R$. Therefore $max_{z\in\gamma(g_w(\alpha(\varphi(x))))}\mathcal{L}_\mathcal{R} = u_{o_1} - l_{o_0}$ involves $l_{o_0}, u_{o_1}$ which are differentiable functions of the weight parameters $w$. Next, the gradient of the maximum robustness loss with respect to different weight parameters is computed based on the computations shown in (4.3) and (4.4). The parameters are then updated based on the gradients. For our example, we used the difference of $o_0$ and $o_1$ for defining the robustness loss, but other formulations are also possible, such as cross-entropy. We refer the reader to Gowal *et al.* (2018) and Mirman *et al.* (2018) for details.

As readers may notice, (4.3) and (4.4) are much simpler than the abstract transformers such as DeepPoly/CROWN discussed in Section 3, which allows these bounds to be calculated very efficiently during training. The asymptotic complexity of propagating IBP bounds through a neural network is the same as regular forward propagation on a DNN. Thus, theoretically, any DNN that can be trained using gradient descent can also be trained with IBP. IBP has been applied to train ImageNet-level neural networks (Gowal *et al.*, 2019; Xu *et al.*, 2020). After IBP training, the IBP bounds of the DNN typically become quite tight.

However, challenges still remain during training, as the initialization of DNNs can produce extremely loose IBP bounds, and thus the loss function (4.2) becomes vacuous, unable to provide useful training signals. To address the vacuous bounds at the beginning of training, Gowal *et al.* (2019) propose a warmup procedure where the input preconditions $\varphi(x)$ (typically, pixel-wise perturbations) are scheduled to gradually enlarge very slowly during training. For example, a very small perturbation can be used when training just starts, and the perturbation size will gradually increase during training until it reaches a target. On the other hand, CROWN-IBP (Zhang *et al.*, 2020) utilizes IBP to calculate intermediate layer bounds, while using CROWN to provide a much tighter bounds at the output layer for training, striking a balance between bound tightness and training efficiency. Shi *et al.* (2021) discussed specialized initializations and loss function designs to reduce the training time spent on warmup.

## 4.3 Input Splitting Refinement for Training

To reduce the over-regularization caused by the imprecise Box analysis, abstraction refinement can be employed. In this section, we will discuss, how input splitting, a popular form of refinement introduced in Section 3.1.5 can be leveraged during training to obtain robust and accurate networks. We will describe the method of Yang *et al.* (2023) that is the first work to train DNNs for provable robustness against geometric transformations.

**Robust geometric training.** The key to incorporating geometric robustness guarantees into training lies in formulating verification as part of the loss function. To reduce the overapproximation error, the geometric verifiers (Balunovic *et al.*, 2019; Mohapatra *et al.*, 2020; Yang *et al.*, 2023) heuristically split the range of the input parameters into $K \in \mathbb{N}$ splits. To account for this, the training loss enforces *local* robustness at the level of individual input splits. To certify the network across the entire desired range $\theta \in [\alpha, \beta]$, this local property is enforced across all $K$ splits. Furthermore, the DNN should have high clean accuracy. This yields the following formulation for the *ideal* robust classification loss:

$$\mathcal{L}_{ci}(w, x, y) = \kappa \cdot \mathcal{L}(w, x, y) + \Big(1 - \kappa\Big) \cdot \frac{1}{K} \sum_{k=1}^{K} \mathbf{max}_{z \in \gamma([L_k, U_k])} \mathcal{L}_R(w, z, y) \tag{4.5}$$

where $[L_k, U_k] = g_w(\alpha(P(x, \theta_k)))$ and $\kappa \in [0, 1]$ governs the relative weighting between the clean accuracy loss $\mathcal{L}$ and geometric robustness loss $\mathcal{L}_R$, with higher $\kappa$ prioritizing clean accuracy. Notice that $\mathcal{L}$ is computed on $f_w$ while $\mathcal{L}_R$ is computed on its box approximation $g_w$ sharing the same parameters $w$. Figure 4.3 visualizes this combination. The maximum of $\mathcal{L}_R$ can be computed as $max_{j \neq c} U_j - L_c$ where $c$ is the correct class label. Alternatively, we can use the cross entropy between the ground truth distribution and the distribution obtained after applying softmax on $z$ from (4.6).

$$z_c = L_c \text{ and } z_j = U_j, j \neq c. \tag{4.6}$$

In practice, the loss in (4.5) is too computationally expensive since the runtime scales linearly with the number of splits, which often needs to be large to ensure precise verification. As a remedy, one can enforce the robustness property *stochastically* using data augmentation in conjunction with a randomized sampling of interval splits. The method of Yang *et al.* (2023), called CGT, uniformly samples a scalar perturbation amount $\widetilde{\theta} \sim \mathcal{U}(\alpha, \beta)$ and compute a local interval split $\theta_l = [\widetilde{\theta} - \nu, \widetilde{\theta} + \nu]$, where $\nu$ is a hyperparameter vector governing the interval size of each perturbation parameter. We then compute the *tractable* robust classification loss as:

$$L_{ct}(x, y) = \kappa \cdot \mathcal{L}(w, P(x, \widetilde{\theta}), y) + (1 - \kappa) \cdot \mathbf{max}_{z \in \gamma([L_l, U_l])} \mathcal{L}_R(w, z, y) \tag{4.7}$$

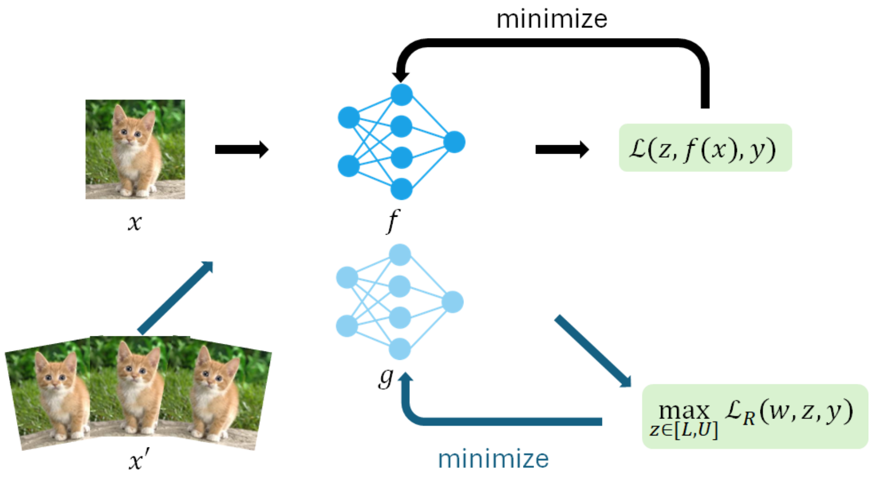

minimize
$x$
$f$
$\mathcal{L}(z, f(x), y)$
$x'$
$g$
$\max_{z \in [L,U]} \mathcal{L}_R(w, z, y)$
minimize

**Figure 4.3:** A robust and accurate neural network $f$ must have high performance on not only the test inputs $x$ but also inputs $x'$ obtained by applying geometric transformations to $x$. To train such a network, one uses the standard accuracy loss $\mathcal{L}(x)$ on $f$ and the robustness loss $\mathcal{L}_R(x')$ on its interval approximation $g$.

where $[L_l, U_l] = g_w(\alpha(P(x, \theta_l)))$. Since CGT samples a different $\widetilde{\theta}$ for each mini-batch of training samples, it will, on average, effectively enforce local robustness over the entire parameter range, hence leading to robustness for the full $P(x, \theta)$.

While this loss function incorporates only Box analysis to propagate the geometric region through the network, it can be easily adapted to other domains like DeepPoly/CROWN (Zhang *et al.*, 2020) by substituting the $\epsilon$-balls in their loss functions with CGT's formulation of local geometric balls.

CGT was evaluated across multiple datasets, including MNIST, CIFAR10, Tiny ImageNet, and Udacity self-driving car datasets. CGT-trained DNNs consistently achieve state-of-the-art deterministic certified geometric robustness and clean accuracy. Interestingly, the work shows that achieving both high accuracy and robustness on the autonomous driving dataset (Bojarski *et al.*, 2016) is possible. Therefore, in practical scenarios, high accuracy and robustness are achievable, contradicting the hypothesis presented in Tsipras *et al.* (2019).

## 4.4 Combining Certified and Adversarial Training

While refinement based on input splitting yields state-of-the-art models for geometric perturbations, it is infeasible for reducing the over-regularization when considering high-dimensional norm-based perturbations. To handle these cases, several refinements combining adversarial and certified training have been developed. We discuss these next.

**COLT.** Incomplete verifiers based on abstract interpretation can fail to prove a specification $(\varphi(x), \psi)$ on an already trained DNN $f_w$ due to the accumulation of the approximation loss at different layers. If $f_w$ actually satisfies the specification, then it means that there exist *latent adversarial examples* at an intermediate layer $i$. These are points $x'_i \in \gamma(g^i_w(\alpha(\varphi(x)))) \setminus f^i_w(\varphi(x))$ that when propagated from layer $i$ onwards according to the concrete DNN transformers produce an output violating $\psi$, where $f^i_w$ and $g^i_w$ are respectively the DNN and verifier output at layer $i$.

The COLT method (Balunovic and Vechev, 2020) leverages adversarial training to eliminate latent adversarial examples from intermediate layers. It proceeds in a layerwise fashion. Initially, it uses adversarial training to eliminate violating examples at the input layer. However, this does not lead to provability due to latent counter-examples. The next training step shown in Figure 4.4 addresses this issue. The abstract interpreter propagates the precondition through the first layer of the network obtaining the region $C_1 = g^1_w(\alpha(\varphi(x)))$. COLT uses an adversarial attack to find a concrete point $x_1$ inside $C_1$ which produces the maximum loss $\mathcal{L}_R$ when this point is propagated further through the network. The method updates the DNN parameters by backpropagating the robustness loss.

An important design aspect of COLT is that it freezes the first layer and stops backpropagation after updating the second layer. Further, no backpropagation through the verifier code is performed. The verifier is only used to compute the different regions. As a result, the DNN only has to learn to behave well on the concrete points that were found in the region $C_1$. This process is repeated for the other layers. COLT can be instantiated with any abstract domain. However, the latent

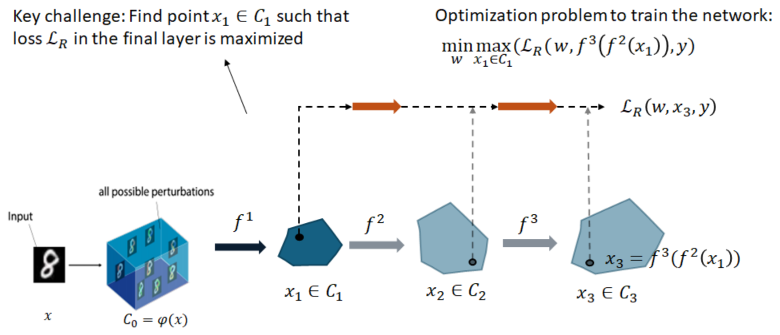

Key challenge: Find point $x_1 \in C_1$ such that loss $\mathcal{L}_R$ in the final layer is maximized
Optimization problem to train the network:
$\min_w \max_{x_1 \in C_1} (\mathcal{L}_R(w, f^3(f^2(x_1)), y)$
$\mathcal{L}_R(w, x_3, y)$
all possible perturbations
Input
$f^1$
$f^2$
$f^3$
$x_3 = f^3(f^2(x_1))$
$x$
$C_0 = \varphi(x)$
$x_1 \in C_1$
$x_2 \in C_2$
$x_3 \in C_3$

**Figure 4.4:** An attack algorithm generates the latent adversarial example $x_1$ inside the abstract shape $C_1$ for the first layer. $x_1$ is propagated till the last layer to compute a lower bound on the maximum robustness loss.

adversarial examples are computed using the PGD attack (Madry *et al.*, 2017) which requires an efficient projection method to the particular abstract shape the method is instantiated with. The paper leverages the Zonotope approximation (Singh *et al.*, 2018) that we discussed in Section 3.1.3 during training by introducing efficient projection methods. We refer the interested readers to Balunovic and Vechev (2020) for more details. The use of Zonotopes makes COLT expensive for training larger models. The procedure was simplified while preserving its effectiveness by Palma *et al.* (2022).

**SABR.** The SABR method aims to reduce the over-regularization caused by the IBP training. The amount of overapproximation depends upon the size of the input region $\varphi(x)$ propagated by the Box analysis. SABR propagates a small but carefully selected subset of $\varphi(x)$, called *propagation regions*. Because it propagates subsets, the Box analysis used during the training is not a sound overapproximation of the DNN behavior within $\varphi(x)$. However, soundness is not needed during training as long as the worst-case robustness loss on the final unsound output is close enough for the true robustness loss over the full region. It is possible that because of overapproximation, the concretization of the Box analysis for the propagation region is close to $f_w(\varphi(\phi))$. The propagation regions are obtained by finding an adversarial example

$x' \in \varphi(x)$ and then heuristically computing a small box region around $x'$ based on a parameter $\tau \in \mathbb{R}$. More details can be found in Müller *et al.* (2023a).

**TAPS.** This method combines IBP and adversarial training by aiming to reduce latent adversarial examples. Like COLT, it first propagates the input region till some intermediate layer using IBP and then conducts adversarial training within the obtained Box approximation to compute the robustness loss. However, unlike COLT, this approach does not freeze layers and propagates gradients also through the verifier computations for computing the intermediate regions. The gradients for adversarial training and verifier computations are combined via a *gradient connector*, that allows training the whole network jointly such that the over-approximation of IBP and under-approximations of adversarial training cancel out. For more details, see Mao *et al.* (2023).

**Combining losses.** The works of Palma *et al.* (2024) and Fan and Li (2020) consider convex combinations of robustness loss formulations for adversarial and certified training. The main challenge is combining the gradients on the DNN for adversarial training with gradients from certified training defined over an abstract interpreter. Fan and Li (2020) update the network parameters in the direction of the angular bisector of the two gradients while Palma *et al.* (2024) leverage scalarizations (Kurin *et al.*, 2022).

This finishes our discussion of certified training methods for single execution properties. We refer the readers to the work of Mao *et al.* (2024) comparing the effectiveness of different training methods introduced in this section. We note that there are certified training methods that do not require differentiable abstract interpreters during training or testing (Zhang *et al.*, 2021a; Leino *et al.*, 2021; Hu *et al.*, 2023a; Anil *et al.*, 2019; Singla and Feizi, 2021; Trockman and Kolter, 2021). These require specific architectures tailored for specific properties, which can reduce their suitability for other specifications (Jiang and Singh, 2024). Next, we will consider training for relational properties, specifically, robustness against universal adversarial perturbations.

## 4.5 Universal Adversarial Perturbations

The training methods presented in the above sections focus on training that is robust against standard input-specific perturbations. In many practical scenarios, attackers cannot feasibly compute and apply single-input adversarial perturbations in real-time. Recent work has shown that input-agnostic attacks, specifically, universal adversarial perturbations (UAPs), are a more realistic threat model (Li *et al.*, 2019a; Li *et al.*, 2019b; Liu *et al.*, 2023). UAPs do not depend on single inputs; instead, they are learned to affect *most* inputs in a data distribution. For these scenarios, it is overly conservative to assume the single-input adversarial region model for verification/certified training. Instead, we would like to ensure safety against universal perturbations. In Section 3.2, we discussed DNN verification for input relational specifications, including robustness against UAPs. In this section, we discuss the use of abstract interpreters to train for UAP robustness where perturbations $\mathbf{u}$ are contained in an $L_\infty$ ball $\mathcal{B}_\infty(0, \eta) = \{\mathbf{u} \in \mathbb{R}^m \mid ||\mathbf{u}||_\infty \leq \eta\}$ of radius $\eta \in \mathbb{R}$. We begin by formally defining the UAP training objective.

### 4.5.1 UAP Robustness Training Objective

For single-input perturbation robustness, we minimize the expected loss over the data distribution $\mathcal{I}$ due to worst-case adversarial perturbations crafted separately for each input. For UAP robustness, we minimize the worst-case expected loss from a single perturbation applied to all points in the data distribution.

$$\mathbf{w} = \arg\min_{w} \max_{\mathbf{u} \in \mathcal{B}_\infty(0,\eta)} \left( \mathop{\mathbb{E}}_{(x,y)\sim\mathcal{I}} [\mathcal{L}(w, \mathbf{x} + \mathbf{u}, y)] \right) \tag{4.8}$$

Here, we assume that $\mathcal{L}$ refers to a standard loss function in which adversarial additive perturbations incur a greater loss than safe perturbations. Since UAPs are input-agnostic, we maximize the expected value over $\mathbf{u} \in \mathcal{B}_\infty(\mathbf{0}, \eta)$. To create an efficient training algorithm for certified UAP robustness, we need an efficiently computable upper bound for the maximization.

### 4.5.2 Certified Training for UAP robustness

In this section, we discuss one way to achieve certified UAP robustness: CITRUS (Xu and Singh, 2024). CITRUS relies on the idea that (4.8) can be upper bounded by computing the worst case expected loss from the set of perturbations that cause at least 2 inputs to be misclassified. The intuition for CITRUS can be seen in Figure 4.5. In the following sections, we describe how this loss can be upper bounded and turned into an algorithm for training.

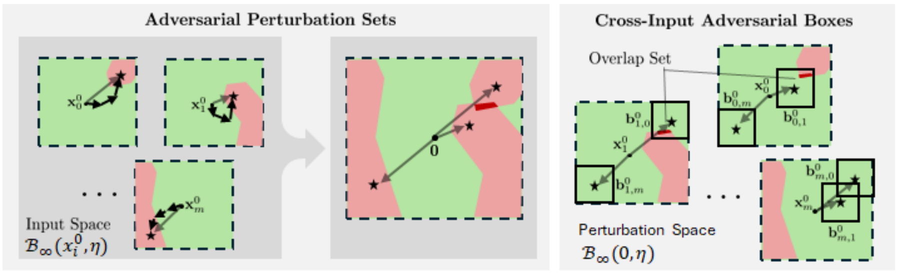

Adversarial Perturbation Sets
Cross-Input Adversarial Boxes
Overlap Set
Input Space
$\mathcal{B}_\infty(x_i^0, \eta)$
Perturbation Space
$\mathcal{B}_\infty(0, \eta)$
0

**Figure 4.5:** Intuition behind CITRUS. UAPs occur where adversarial regions (red boxes) from multiple inputs overlap (dark red boxes). To approximate this, adversarial examples (stars) are computed for each input, $\mathbf{x}_i$, the corresponding perturbation vectors, $\mathbf{v}_i$, (gray arrows), and adversarial regions are collocated to $\mathcal{B}_\infty(\mathbf{0}, \eta)$. To approximate UAP regions, we take cross-input adversarial perturbations and draw $l_\infty$ balls around them, $b_{i,j}^0 = \mathcal{B}_\infty(\mathbf{x}_i + \mathbf{v}_j, \tau)$.

**k-Common perturbations.** We can define a k-common perturbation (k-cp) as a perturbation for which there exists $k$ inputs on which the perturbation is adversarial. Intuitively, single-input perturbation robustness is an upper bound on UAP robustness as UAPs can only exist if single-input perturbations exist for multiple inputs. We can reframe this intuition if we consider the worst-case UAP (i.e., the UAP that affects the most inputs) as a $k^*$-cp. We know this UAP can, at most, affect as many points as there are 1-cps. In fact, for all $k < k^*$ we know that the worst-case UAP can only affect as many points as there are k-cps. This leads us to the conclusion that the expected loss maximizing over the set of k-cps is an upper bound for (4.8).

**CITRUS.** It is still not computationally practical to compute the loss over the set of $k$-cps for training as it is expensive to compute the intersection over single-input adversarial regions (Dimitrov *et al.*, 2022). However, for 2-cps, we can upper bound this loss by considering cross-input adversarial regions. A cross-input adversarial region is a perturbation space that is adversarial for more than one input (see the dark red region in Figure 4.5). Therefore, we can over approximate the UAP robustness of each input. $x_i$, by considering the loss from adversarial regions from other inputs $x_j$ where $j \neq i$. Section 4.4 introduced SABR (Müller *et al.*, 2023a), which showed that taking smaller bounding boxes centered around adversarial examples was an effective way to train networks for standard adversarial robustness. CITRUS utilizes this insight and computes small bounding boxes around the adversarial examples for all other inputs in a batch. Networks trained with CITRUS have SOTA certified UAP robustness while maintaining much higher accuracy than previous certified training methods. For MNIST with $\epsilon = 0.3$, CITRUS achieved 99.04% standard accuracy and 95.61% certified average UAP accuracy, outperforming existing certified training methods like IBP (97.67%/94.76%), SABR (98.75%/95.37%), and TAPS (98.53%/95.24%). On CIFAR-10 with $\epsilon = 8/255$, CITRUS obtained 63.12% standard accuracy—a substantial 10.3% improvement over TAPS (52.82%)—while maintaining competitive certified UAP accuracy of 39.88%. For TinyImageNet with $\epsilon = 1/255$, CITRUS achieved 35.62% standard accuracy and 26.27% certified UAP accuracy, significantly outperforming other methods. For details and proofs, see Xu and Singh (2024).

## 4.6 Certified Training for Variational Autoencoders

The certified training techniques discussed so far focused on training deterministic networks. In this section, we will look at certified training for stochastic networks, specifically for variational autoencoders (VAEs).

### 4.6.1 Variational Autoencoders

Variational Autoencoders (VAEs) are generative DNN architectures that learn latent representations of data. Given a set of inputs $X \subseteq \mathbb{R}^m$ generated via an unknown process with latent variables $Z \subseteq \mathbb{R}^{d_l}$, VAEs learn a latent variable model with joint density $p_w(x, z) = p_w(x|z)p(z)$. Since directly maximizing the likelihood is often intractable, VAEs employ variational inference, learning a conditional likelihood model $p_{w_d}(x|z)$ and an approximated posterior distribution $p_{w_e}(z|x)$. A VAE consists of two main components: an encoder network $f^e : \mathbb{R}^m \to \mathcal{P}(\mathbb{R}^{d_l})$ with parameters $w_e$, and a decoder network $f^d : \mathcal{P}(\mathbb{R}^{d_l}) \to \mathcal{P}(\mathbb{R}^n)$ with parameters $w_d$. Here, $\mathcal{P}(\mathbb{R}^n)$ denotes the set of probability distributions defined over $\mathbb{R}^n$. For standard VAEs, given a single input $z \in \mathbb{R}^{d_l}$, the decoder's output $f^d(z)$ is deterministic. The effectiveness of VAEs has led to their deployment in safety-critical applications, including wireless communications, autonomous driving, and medical diagnosis.

### 4.6.2 VAE Robustness Training Objective

Let $\mathcal{Z}$ denote the set of distributions at the latent layer computed by $f^e$ on an input region $\varphi_t(x)$ around a training data point $x \in \mathbb{R}^m$, i.e., $\mathcal{Z} = \{Z \mid Z = f^e(x'), x' \in \varphi_t(x)\}$. Let $\mathcal{Y}$ be the set of output distributions $\mathcal{Y} = \{Y \mid Y = f^d(Z), Z \in \mathcal{Z}\}$. Each $Y$ and $Z$ are random variables corresponding to specific probability distributions over $\mathbb{R}^{d_l}$ and $\mathbb{R}^n$ respectively. Given a target probability threshold $(1 - \delta)$ and error function $M$, the worst-case error is defined as $\mathcal{L}(w, f^e, f^d, x) = \max_{Y \in \mathcal{Y}} T(Y)$ where $T(Y)$ is defined as:

$$T(Y) = \min_{\epsilon \in \mathbb{R}} \epsilon \quad \text{s.t.} \quad \mathbb{P}(M(Y) \leq \epsilon) \geq (1 - \delta) \tag{4.9}$$

At a high level, for any given output distribution $Y$, $T(Y)$ determines the tightest possible error threshold ensuring that for any sample $y \sim Y$, the corresponding error $M(y)$ is no more than $T(Y)$ with probability at least $(1 - \delta)$.

### 4.6.3 Bounding Worst-Case Loss

For certified training of VAEs, similar to the methods presented in the above sections, exactly computing $T(Y)$ is computationally intractable, so our goal is to find a sound upper bound on the loss. To this end, we introduce the concept of support sets for the set of distributions $\mathcal{Z}$ at the latent layer.

**Definition 4.1** (Support sets). For a set of distributions $\mathcal{Z}$ over $\mathbb{R}^{d_l}$ and a probability threshold $(1-\delta)$, a subset $S \subseteq \mathbb{R}^{d_l}$ is a support for $\mathcal{Z}$ if:

$$\min_{Z \in \mathcal{Z}} \mathbb{P}(z \in S | z \sim Z) \geq (1-\delta) \tag{4.10}$$

For a fixed $(1-\delta)$, we can show that for any support set $S$, the error upper bound $T_{ub}(S) = \max_{z \in S} M(f^d(z))$ serves as a valid upper bound of the worst-case error:

**Theorem 4.1.** For a VAE with encoder $f^e$, decoder $f^d$, local input region $\varphi_t(x)$, error function $M$ and probability threshold $(1-\delta)$, if $\mathcal{Z} = \{Z \mid Z = f^e(x'), x' \in \varphi_t(x)\}$ then for any support set $S$ for $\mathcal{Z}$, the worst-case error $\mathcal{L}(w, f^e, f^d, x) \leq T_{ub}(S)$ where $T_{ub}(S) = \max_{z \in S} M(f^d(z))$.

Since the decoder's output $f^d(z)$ is deterministic for all $z \in S$, computing $T_{ub}(S)$ is equivalent to bounding the error of a deterministic network on an input region, which can be efficiently done using existing techniques like Interval Bound Propagation (IBP).

### 4.6.4 Support Set Computation and Bounding

Given the set of distributions $\mathcal{Z}$ with probability density functions $p_Z(z)$ and a fixed $(1-\delta)$, we aim to find a support set $S = [L, U]$ such that $\forall Z \in \mathcal{Z}, \int_L^U p_Z(z)dz \geq (1-\delta)$. For VAEs, the encoder typically outputs parameters $\mu$ and $\sigma$ for each latent dimension, representing Gaussian distributions. Using deterministic network bounding techniques like IBP, we can approximate the reachable intervals $[\mu_{lb}, \mu_{ub}]$ and $[\sigma_{lb}, \sigma_{ub}]$ for these parameters. For simplicity, let us consider the one-dimensional case where $\mathcal{Z} = \{\mathcal{N}(\mu, \sigma) \mid \mu \in [\mu_{lb}, \mu_{ub}], \sigma \in [\sigma_{lb}, \sigma_{ub}]\}$, see Figure 4.6. We can prove that the distributions specified by $\mu_{lb}, \sigma_{lb}$ and $\mu_{ub}, \sigma_{ub}$ capture the least probability among all possible distributions in $\mathcal{Z}$.

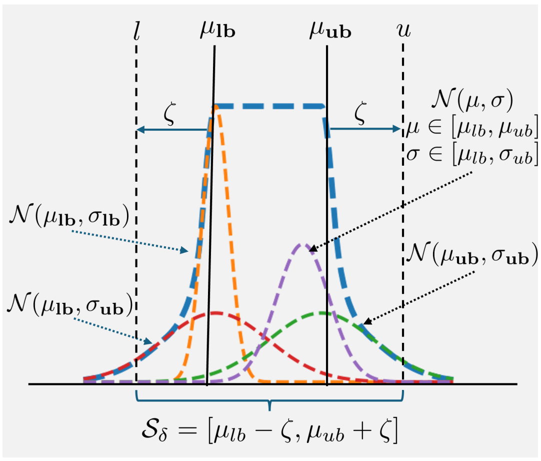

$l$
$\mu_{\mathbf{lb}}$
$\mu_{\mathbf{ub}}$
$u$
$\zeta$
$\zeta$
$\mathcal{N}(\mu, \sigma)$
$\mu \in [\mu_{lb}, \mu_{ub}]$
$\sigma \in [\mu_{lb}, \sigma_{ub}]$
$\mathcal{N}(\mu_{\mathbf{lb}}, \sigma_{\mathbf{lb}})$
$\mathcal{N}(\mu_{\mathbf{ub}}, \sigma_{\mathbf{ub}})$
$\mathcal{N}(\mu_{\mathbf{lb}}, \sigma_{\mathbf{ub}})$
$\mathcal{S}_\delta = [\mu_{lb} - \zeta, \mu_{ub} + \zeta]$

**Figure 4.6:** Support sets. Given a set of distributions $\{\mathcal{N}(\mu, \sigma) | \mu \in [\mu_{lb}, \mu_{ub}], \sigma \in [\sigma_{lb}, \sigma_{ub}]\}$ we define a symmetric support set $S_\delta = [\mu_{lb} - \zeta, \mu_{ub} + \zeta], \zeta \in \mathbb{R}^+$

Therefore, finding a support set $S = [L, U]$ only requires ensuring that both these endpoint distributions capture at least $(1 - \delta)$ probability.

**Theorem 4.2.** Given $\mathcal{Z} = \{\mathcal{N}(\mu, \sigma) \mid \mu \in [\mu_{lb}, \mu_{ub}], \sigma \in [\sigma_{lb}, \sigma_{ub}]\}$ and probability threshold $(1-\delta)$, the interval $[L, U] = [\mu_{lb} - \sigma_{ub}\Phi^{-1}(p_0), \mu_{ub} + \sigma_{ub}\Phi^{-1}(p_0)]$ is a valid support set, where:

$$p_0 = \min_{p \in [(1-\delta), 1]} \left[ \Phi^{-1}(p) + \Phi^{-1}(p - (1 - \delta)) \geq \frac{\mu_{lb} - \mu_{ub}}{\sigma_{ub}} \right] \qquad (4.11)$$

For higher dimensional latent spaces, we can compute the support set for each dimension independently using a probability threshold of $(1 - \delta)^{1/d_l}$.

### 4.6.5 CIVET

CIVET (Certified Interval Variational Autoencoder Training), is a method for training VAEs with certified robustness guarantees building upon the ideas presented above. CIVET computes support sets for multiple probability thresholds and combines them into a weighted loss:

**Definition 4.2** (CIVET Loss). Given a deterministic decoder network $f^d$, input $x \in \mathbb{R}^m$, distribution bounds $\mu_{lb}, \mu_{ub}, \sigma_{ub}$, and a set of thresholds $\{\delta_1, \ldots, \delta_j\}$ (sorted in descending order), the CIVET loss is defined as:

$$\mathcal{L}_{CIVET} = (1-\delta_1)\mathcal{L}_{dec}(f^d, x, S_{\delta_1}) + \sum_{i=2}^{j}(\delta_{i-1}-\delta_i)\mathcal{L}_{dec}(f^d, x, S_{\delta_i}) \quad (4.12)$$

Here, $S_{\delta_i}$ is the support set computed for threshold $(1-\delta_i)$, and $\mathcal{L}_{dec}(f^d, x, S_{\delta_i})$ is the worst-case loss of the decoder over this support set, computed using a deterministic verification method like IBP.

### 4.6.6 Experimental Results

Xu *et al.* (2024) demonstrates CIVET's effectiveness across different domains and perturbation magnitudes. For wireless applications using the FIRE dataset (Liu *et al.*, 2021), CIVET achieves a certified SNR of 13.88-15.02 dB across perturbation budgets of 15-25%, significantly outperforming standard training (-2.35 dB at 25% perturbation) and adversarial training (3.17-6.89 dB). On vision tasks, CIVET reduced certified MSE by up to 93.8% on MNIST and 83.8% on CIFAR-10 compared to standard training, while maintaining competitive baseline performance. When compared against Lipschitz-constrained VAEs (Barrett *et al.*, 2022), CIVET showed superior performance—for CIFAR-10 with $\epsilon = 2/255$, CIVET achieved 0.0055 certified MSE versus 0.0105 for Lipschitz VAEs—while removing architectural constraints. CIVET also demonstrated strong empirical robustness against practical attacks like RAFA (Liu *et al.*, 2023), LSA (Kos *et al.*, 2018), and MDA (Camuto *et al.*, 2021), confirming its real-world effectiveness. For more detailed exploration and results, see Xu *et al.* (2024).

# 5

# Explaining and Interpreting DNNs

To overcome the limitations of black-box construction, several explanation (Samek *et al.*, 2021) and interpretation methods (Räuker *et al.*, 2023) have been developed to generate intuitive explanations and interpretations of DNN behavior. However, popular methods are unreliable and non-robust, and relying on them can lead to a false sense of confidence. In this section, we will discuss how the abstract interpretation framework can generate reliable and robust explanations and interpretations of deterministic DNN classifiers. In this section, we will use $\mathbf{x}$ and $\mathbf{h}$ to refer to a vector of input and hidden neurons, respectively. $x$ and $h$ will represent an input neuron and a hidden neuron that are components of $\mathbf{x}$ and $\mathbf{h}$, respectively.

## 5.1 Explaining DNN Predictions

A popular approach for explaining the predictions of a DNN is identifying the subset of input features that affect the model predictions on a specific input the most. However, most methods (Ribeiro *et al.*, 2016; Qi *et al.*, 2020; Lundberg and Lee, 2017; Ribeiro *et al.*, 2018) do not provide any robustness guarantees that the identified features are sufficient to ensure that the model behavior remains the same within an input region. The field of formal explainable AI has emerged in recent years to overcome this limitation (Marques-Silva and Ignatiev, 2022; Malfa *et al.*, 2021; Ignatiev *et al.*, 2019; Darwiche and Hirth, 2020). Let $S \subseteq \mathcal{X}$ be a subset of input neurons, $A = \mathcal{X} \setminus S$, and $\mathbf{x}^A$ denote input vector containing only the neurons from $A$. We define a local region $\varphi(\mathbf{x}, S, \eta) = \{\mathbf{x}' \in \mathbb{R}^m \mid ||\mathbf{x}^A - \mathbf{x}'^A||_p \leq \eta \wedge \mathbf{x}^S = \mathbf{x}'^S\}$ around an input $\mathbf{x}$, parameterized by $S$, containing inputs $\mathbf{x}'$ such that the values of neurons from $S$ in $\mathbf{x}'$ is the same as in $\mathbf{x}$ while the values of neurons from $A$ varies by at most $\eta \in \mathbb{R}$.

**Definition 5.1** (Provably robust explanation). A subset of neurons $S \subseteq \mathcal{X}$ is a provably robust explanation of $f$'s prediction on an input $\mathbf{x}$ for a given $\eta \in \mathbb{R}$ if $\forall \mathbf{x}' \in \varphi(\mathbf{x}, S, \eta)$ we have that $f(\mathbf{x}) = f(\mathbf{x}')$.

$A = \mathcal{X} \setminus S$ contains irrelevant neurons with respect to $f(\mathbf{x})$ as changing them does not change the DNN prediction. There can be multiple $S$ that constitute provably robust explanations. Additional criteria, such as reducing the size of the robust explanation set can be employed to choose among different candidates.

A brute-force algorithm for computing $S$ satisfying Definition 5.1 involves constructing $\varphi(\mathbf{x}, S, \eta)$ for each $S \subseteq \mathcal{X}$ and then leveraging an abstract interpretation-based verifier $V$ to check whether $f$'s prediction is robust within $\varphi(\mathbf{x}, S, \eta)$. However, the number of possible subsets is $2^{|\mathcal{X}|}$, and each check involves expensive verifier calls. VeriX (Wu *et al.*, 2023) constructs a robust explanation incrementally. It starts by initializing $S = \mathcal{X}$. Next, it selects a neuron $x$ according to a predefined traversal order. If $V$ can prove $f$ to be robust within $\varphi(\mathbf{x}, S \setminus \{x\}, \eta)$, then we set $S = S \setminus \{x\}$. The quality of explanations produced by this method depends on the order in which features are processed and the precision of the verifier (imprecise verifiers produce bigger sets). VeriX+ improves both the generation time and the size of robust explanations by identifying more efficient traversal orders based on abstract interpretation and binary search. The generated explanations are useful for detecting incorrect predictions and out-of-distribution samples. See Wu *et al.* (2024) for details.

## 5.2 Interpreting Robustness Proofs

As we saw in Section 3, to prove DNN safety, abstract interpretation-based DNN verifiers generate abstract elements at different layers, capturing the relationships between neurons. These abstract elements are usually complex, high-dimensional convex shapes defined over thousands of neurons in the DNN. Although these proofs can guarantee the DNN's safety and thus induce reliability in the working of the DNN, they do not provide any human-understandable insights into the semantic meanings of the constituents (neurons) of the generated proofs. This is unlike conventional program verification, where the proofs analyze the program behavior by examining the semantic meanings of the different parts, like inductive invariants, program contracts, etc. In the case of DNN verification, the absence of any human-understandable elements in the generated proofs, it remains unclear whether they are based on any meaningful features learned by the DNN. If the DNN is proven to be safe, but the proof is based on meaningless features not aligned with human intuition, then the DNN behavior cannot be considered fully trustworthy.

While there has been a lot of work on interpreting black-box DNNs, standard methods (Ribeiro *et al.*, 2016; Wong *et al.*, 2021) can only explain the DNN behavior on individual inputs and cannot interpret the complex behavior encoded by the abstract elements capturing DNN behavior on an infinite set of inputs. The main challenge in interpreting DNN proofs is mapping the complex abstract elements to human-understandable interpretations.

The work by Banerjee *et al.* (2024a) is the first to develop a method, called ***ProFIt*** (**Pro**o**F** **I**nterpretation **T**echnique) for interpreting robustness proofs computed by DNN verifiers and generating human-understandable interpretations. The method can interpret proofs computed by different DNN verifiers. The main concept behind the interpretation is to dissect the generated proof into several components called the *proof features*. The proof features are computed by projecting the generated proof, a high-dimensional abstract element onto individual neurons. The proof features can be analyzed independently by generating the corresponding interpretations. The proposed projection creates thousands of proof features. However, since the proof features are to be analyzed manually, analyzing thousands of them can be prohibitive. Further, since some proof features can be more important than others, the algorithm proposes a set of *representative* proof features. The algorithm defines a priority function over the proof features that signify the importance of each proof feature in the complete proof is defined. The method extracts a set of proof features by retaining only the more important parts of the proof that preserve the property.

### 5.2.1 Proof Features

Before we describe the algorithm to find the representative set of proof features to be analyzed, let us define a proof feature.

First, consider the case of DNN inference over a given concrete input, where we wish to analyze the output. Treating the DNN as a collection of individual neurons makes it difficult to analyze all the neurons due to the sheer number of neurons in a DNN. On the other hand, analyzing just the output layer neurons hides the details of how the output was computed. To solve this problem, instead of analyzing the neurons from all layers in the DNN or just the final layer neurons, recent works (Wong *et al.*, 2021; Liao and Cheung, 2022) on analyzing the DNN output partition a DNN into a *deep feature extractor* and a *decision layer*. The output neurons of the penultimate layer are the deep features and the last layer linearly combines these features to compute the final output.

Similarly, in the case of DNN verification, the DNN can be segregated into a deep feature extractor and a decision layer. However, instead of a single input, we are given a local robustness input region $\varphi = \varphi(x) = \{x' \mid ||x' - x||_\infty \leq \eta\}$ containing infinitely many inputs instead of a single input as handled by previous interpretation methods. Similarly, instead of a concrete output, we now have a proof, high-dimensional convex shape generated by the DNN verifier at the penultimate layer. This shape can be projected to the hidden neurons $h$ in the penultimate layer to generate proof features. So, the proof features are a set of intervals of the form $[l_h, u_h]$ containing all possible output values of the corresponding neuron $h$ w.r.t. $\varphi$.

**Definition 5.2** (Proof features). Given a DNN $f$, input region $\varphi$ and DNN verifier (abstract interpreter) $V$, for each neuron $h$ at the penultimate layer of $f$, the proof feature $Q_h$ extracted at that neuron $h$ is an interval $[l_h, u_h]$ such that $\forall \mathbf{x} \in \varphi$, the output at $h$ always lies in the range $[l_h, u_h]$.

Notice that our target is neither to verify the DNN safety from scratch nor to find the more important features in the DNN that make it safe. Rather, it is to find the more important features in a given proof of the DNN safety. Even if a DNN is safe, some verifiers may not be able to prove it. Conversely, if two verifiers are both able to prove a DNN safe, their proofs may have different proof features. So, in Definition 5.2, the computation of proof features is specific to the verifier ($V$).

### 5.2.2 Representative Proof Feature Set

Notice that using Definition 5.2, there are as many proof features as the number of neurons in the penultimate layer. This can be a prohibitively large number to analyze manually. So, we must choose a representative subset of the *more important* proof features that are responsible for the proof to go through.

Let the size of the penultimate layer of an $l$-layered DNN be $d_{l-1}$. We use $\mathcal{Q}$ to denote the set of all proof features at the penultimate layer and $\mathcal{Q}_S$ to denote the proof features corresponding to a subset $S$ of neurons in layer $l-1$, i.e. $\mathcal{Q}_S = \{Q_h \mid h \in S\}$. Note that there are $2^{d_{l-1}}$ possible subsets. Since the number of proof features ($d_{l-1}$) can be very large, enumerating all possible subsets is not an option. Further, many of the proof features, and hence proof feature sets may not be important for the proof. Similar to how DNNs are interpreted w.r.t individual inferences, we want to identify a small set of proof features that are more important for the proof of the property. Let us first describe the expectations from such a proof feature set. These are – (i) Size, (ii) Sufficiency, and (iii) Importance. Since the proof features are to be interpreted manually, we want the size of this set to be as small as possible. Following, we discuss the sufficiency and the importance of the set in detail.

**Sufficiency.** The proof feature set should be sufficient for the proof to go through, i.e., if the proof is restricted to only the proof features from the selected set, the proof should still be able to prove the safety of the DNN. This is an important property expected of the proof feature set because otherwise, it would not make sense to interpret a proof feature set that is not even sufficient to prove the safety of the DNN.

**Definition 5.3** (Proof feature Pruning)**.** Pruning any Proof feature $Q_h \in \mathcal{Q}$ corresponding to a neuron $h$ in the penultimate layer involves setting weights of all of $h$'s outgoing connections to 0 so that given any input $\mathbf{x} \in \varphi$ the final output of $f$ no longer depends on the output of $h$.

Once, a proof feature $Q_h$ is pruned the verifier $V$ no longer uses $Q_h$ to prove the safety property $(\varphi, \psi)$ on the DNN. So, the proof is restricted to the selected set of proof features. Next, we formally define the sufficiency of a proof feature set.

**Definition 5.4** (Sufficient proof feature set)**.** For the proof of safety of $f$ with verifier $V$, a nonempty set $\mathcal{Q}_S \subseteq \mathcal{Q}$ of proof features is sufficient if the property $(\varphi, \psi)$ can still be proven to hold on $f$ by verifier $V$ even when all the proof features not in $\mathcal{Q}_S$ are pruned.

Let $W_l$ be the weight matrix of the final layer of the DNN. Pruning any proof feature $Q_h$ results in setting all weights in $W_l[:, i]$ to 0 where $i$ is the column index of $h$ in $W_l$. For a proof feature set $\mathcal{Q}_S \subseteq \mathcal{Q}$, let $W_l(S)$ be the pruned weight matrix. The proof feature set $\mathcal{Q}_S$ is sufficient if $(\varphi, \psi)$ can be proven to hold on $f$ by $V$ with the pruned weight matrix $W_l(S)$. Let, the verifier $V$ compute an over-approximated output region $g$ of $f$ over the input region $\varphi$. Since the input region $(\varphi)$, the feature extractor (first $l-1$ layers of $f$), the verifier $(V)$ do not change, the output region $g$ of the pruned network only depends on the pruning done at the final layer. Let $g(W_l, S)$ denote the over-approximated output region corresponding to $W_l(S)$. Without loss of generality, we assume that the postcondition is $\psi(Y) = (C^T Y \geq 0)$ where $C \in \mathbb{R}^n$ defines a linear inequality for encoding the robustness property. $f$ can be verified by $V$ with $W_l(S)$ if the value of a $\psi$-specific lower bound function $L_\psi$ applied on $g(W_l, S)$ is $\geq 0$.

**Importance.** Another property expected of the representative feature set is that it contains *important* proof features. The importance of a proof feature $Q_h$ in a proof feature set $\mathcal{Q}_S$ can be thought of as its contribution in the final proof $L_\psi(g(W_l, S))$. Formally, the importance of a proof feature w.r.t a proof feature set $\mathcal{Q}_S$ can be approximated by $\Delta(Q_h, \mathcal{Q}_S)$, where

$$\Delta(Q_h, \mathcal{Q}_S) \triangleq |L_\psi(g(W_l, S)) - L_\psi(g(W_l, S \setminus Q_h))| \tag{5.1}$$

Let $g^{l-1}$ denote the output region computed at the layer $l-1$ (penultimate layer). So,

$$\begin{aligned} \Delta(Q_h, \mathcal{Q}_S) &= |L_\psi(g(W_l, S)) - L_\psi(g(W_l, S \setminus Q_h))| \\ &\leq \max_{\mathbf{h} \in g^{l-1}} |(C^T W_l(S)\mathbf{h} - C^T W_l(S \setminus \{i\})\mathbf{h})| \\ &= \max_{\mathbf{h} \in g^{l-1}} |(C^T W_l[:, i]) \cdot h)| \end{aligned} \tag{5.2}$$

Notice that this definition is dependent on the set $\mathcal{Q}_S$. However, it is useful to compute the importance of the proof feature $Q_h$ independent of the set $\mathcal{Q}_S$. So, we need the maximum contribution of $Q_h$ w.r.t all the sufficient proof feature sets containing $Q_h$ (represented by $\mathbb{S}(Q_h)$). So, the importance is defined formally as

$$P(Q_h) \triangleq \max_{\mathcal{Q}_S \,\in\, \mathbb{S}(Q_h)} \Delta(Q_h, \mathcal{Q}_S) \tag{5.3}$$

### 5.2.3 ProFIt

**Challenges.** Now that we have defined a proof feature, its importance, and the sufficiency of a proof feature set, our target is to find a sufficient proof feature set that is small enough to be interpreted manually and contains the more important proof features. However, there are still the following two challenges.

First, the importance of each proof feature, $P(Q_h)$ as defined in Equation 5.3 is computationally expensive to compute because $\mathbb{S}(Q_h)$ is a huge set - exponential in the total number of proof features, i.e., $\mathbb{S}(Q_h) = \mathcal{O}(2^{d_{l-1}})$. So, we compute the approximate importance of a proof feature - $P_{ub}(Q_h)$. Combining Equations 5.2 and 5.3, $P(Q_h) \leq$

$\max_{\mathbf{h} \in g^{l-1}} |(C^T W_l[:,i]) \cdot h)|$. Further, any proof feature $Q_h = [l_h, u_h]$ computed by $V$ contains all possible values of $h$ where $\mathbf{h} \in g^{l-1}$. So,

$$\begin{aligned} P(Q_h) &\leq \max_{h \in [l_h, u_h]} |(C^T W_l[:,i])| \cdot |h| \\ &= |(C^T W_l[:,i])| \cdot \max(|l_h|, |u_h|) \\ &= P_{ub}(Q_h) \end{aligned} \tag{5.4}$$

Using $P_{ub}(Q_h)$ as an approximation for $P(Q_h)$, the importance of a proof feature can be computed with $O(d_{l-1})$ elementary vector operations and a single verifier call that computes the intervals $[l_h, u_h]$.

The second challenge is that there is an inherent trade-off among the goals. On one hand, the set of all the proof features, $\mathcal{Q}$ is sufficient, but the size is $|d_{l-1}|$. On the other hand, the empty set of proof features is sufficient by definition, however, it does not contain any important proof features. The ProFIt algorithm tackles this challenge.

**Algorithm.** The ProFIt algorithm proceeds by initializing $\boldsymbol{H}_{S_0}$ as an empty set and $\boldsymbol{H}_S$ as $\mathcal{Q}$ and iteratively adds proof features to $\boldsymbol{H}_{S_0}$ and prunes features from $\boldsymbol{H}_S$. The set $\boldsymbol{H}_{S_0}$ contains the features guaranteed to be included in the final output, and $\boldsymbol{H}_S$ contains the candidate features yet to be pruned by the algorithm. At every step, the algorithm ensures that the set $\boldsymbol{H}_S \cup \boldsymbol{H}_{S_0}$ is sufficient and iteratively reduces its size by pruning proof features from $\boldsymbol{H}_S$. The algorithm iteratively prunes the feature $Q_h$ with the lowest value of $P_{ub}(Q_h)$ from $\boldsymbol{H}_S$ while retaining features with higher priorities in $\boldsymbol{H}_S \cup \boldsymbol{H}_{S_0}$. If the feature set $\boldsymbol{H}_{S_0} \cup \boldsymbol{H}_{S_1}$ is sufficient ($\boldsymbol{H}_{S_1}$ is the set containing top-$|S|/2$ features based on $P_{ub}(Q_h)$), ProFIt removes all features in $\boldsymbol{H}_S \setminus \boldsymbol{H}_{S_1}$ from $\boldsymbol{H}_S$ and therefore $\boldsymbol{H}_S$ is updated as $\boldsymbol{H}_{S_1}$ in this step. Otherwise, if $\boldsymbol{H}_{S_0} \cup \boldsymbol{H}_{S_1}$ does not preserve the property $(\varphi,\psi)$, ProFIt adds all feature in $\boldsymbol{H}_{S_1}$ to $\boldsymbol{H}_{S_0}$ (line 16) and replaces $\boldsymbol{H}_S$ with $\boldsymbol{H}_S \setminus \boldsymbol{H}_{S_1}$. The algorithm terminates after all features in $\boldsymbol{H}_S$ are exhausted.

**Optimization and correctness.** Note that checking the sufficiency of an arbitrary proof feature set $\mathcal{Q}_S$ requires expensive verifier invocations. Since only the final layer is modified, incremental verification (Ugare *et al.*, 2023; Ugare *et al.*, 2022) can be used which can efficiently verify the

**Algorithm 1:** Approximate minimum proof feature extraction

```
Input: DNN $f$, property $(\varphi, \psi)$, verifier $V$
Output: Approx. minimum proof features $\boldsymbol{H}_{S_0}$
if $V$ cannot verify $(\varphi, \psi)$ on $f$ then
    return;
end
Calculate all proof features for input region $\varphi$;
Calculate priority $P_{ub}(Q_h)$ for all proof features;
$\boldsymbol{H}_{S_0} = \{\}$, $\boldsymbol{H}_S = \mathcal{Q}$
while $\boldsymbol{H}_S$ is not empty do
    $\boldsymbol{H}_{S_1}$ = top-$|S|/2$ features based on $P_{ub}(Q_h)$
    $\boldsymbol{H}_{S_2} = \boldsymbol{H}_S \setminus \boldsymbol{H}_{S_1}$
    if $\boldsymbol{H}_{S_0} \cup \boldsymbol{H}_{S_1}$ is sufficient using $V$ then
        $\boldsymbol{H}_S = \boldsymbol{H}_{S_1}$
    else
        $\boldsymbol{H}_{S_0} = \boldsymbol{H}_{S_0} \cup \boldsymbol{H}_{S_1}$
        $\boldsymbol{H}_S = \boldsymbol{H}_{S_2}$
    end
end
return proof features $\boldsymbol{H}_{S_0}$
```

property without starting from scratch. However, even removing a single feature in each iteration and checking the sufficiency of the remaining features in the worst case leads to $\mathcal{O}(d_{l-1})$ incremental verification calls which are expensive. So, at each step, from $\boldsymbol{H}_S$ ProFIt greedily picks top-$|S|/2$ features $\boldsymbol{H}_{S_1}$ based on their importance and invokes $V$ to check the sufficiency of $\boldsymbol{H}_{S_0} \cup \boldsymbol{H}_{S_1}$. Since at every step, the size of $\boldsymbol{H}_S$ reduces by half, the algorithm terminates within $\mathcal{O}(\log(d_{l-1}))$ incremental verifier calls. The proof for sufficiency of $\boldsymbol{H}_{S_0}$, a non-trivial theoretical upper bound on its size, and guarantees about the correctness and efficacy of ProFIt can be found in the paper by Banerjee *et al.* (2024a).

**Interpretation of the proof features.** Once the proof features are extracted, the existing local gradient-based visualization techniques

(Smilkov *et al.*, 2017) can be adapted for visualizing the extracted proof features. For any proof feature $Q_h = [l_h, u_h]$ both $l_h, u_h$ can be expressed as differentiable functions $l_h = L_h(l_{x_1}, u_{x_1}, \ldots, l_{x_m}, u_{x_m})$ and $u_h = U_h(l_{x_1}, u_{x_1}, \ldots, l_{x_m}, u_{x_m})$ where $\forall i \in [m]$. $l_{x_i} = x_i - \delta_i$ and $u_{x_i} = x_i + \delta_i$ are the lower and upper bound of the $i$-th input coordinate, $x_i$ is the unperturbed value, $\delta_i$ is the amount of perturbation. To measure the sensitivity of proof feature $Q_h$ w.r.t change in $i$-th input coordinate, we take the gradient $\frac{1}{2} \times (\frac{\partial L_h}{\partial \delta_i} + \frac{\partial U_h}{\partial \delta_i})$ of the mean (also the midpoint) $\frac{(l_h+u_h)}{2}$ of $Q_h$ w.r.t $\delta_i$. This gradient captures the change in the mean value of the proof feature w.r.t the change in $i$-th input coordinate.

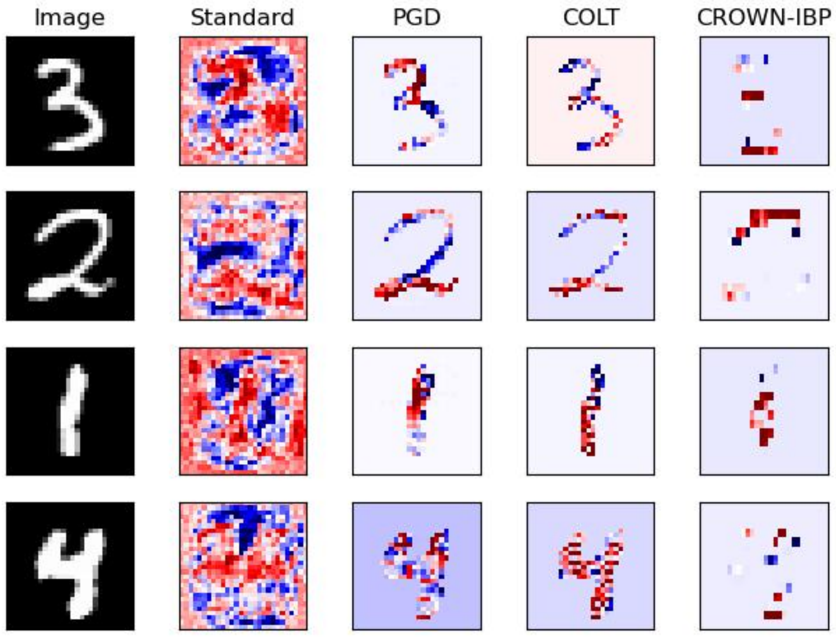


**Figure 5.1:** Gradient map corresponding to the top proof feature for MNIST DNNs trained using different methods discussed in Section 4.

### 5.2.4 Results

In Figures 5.1 and 5.2, a comparison of proof interpretations for DNNs trained with standard and robust training methods (Madry *et al.*, 2017;

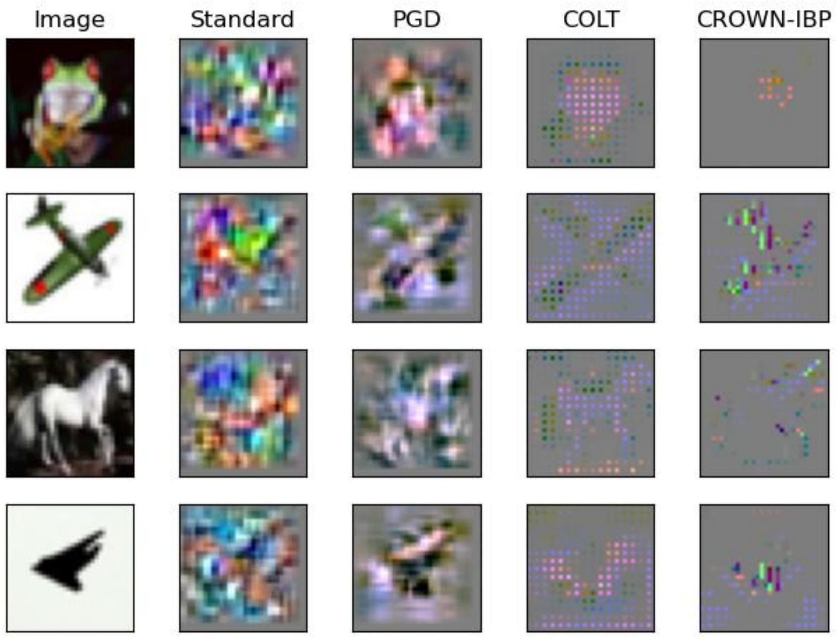


**Figure 5.2:** Gradient map corresponding to the top proof feature for CIFAR-10 DNNs trained using different methods discussed in Section 4.

Zhang *et al.*, 2020; Balunovic and Vechev, 2020) on the popular MNIST (LeCun *et al.*, 1989) and CIFAR10 datasets (Krizhevsky, 2009) shows that the proof features corresponding to the standard networks rely on meaningless input features while the proofs of adversarially trained DNNs – PGD – (Madry *et al.*, 2017) filter out some of these spurious features. In contrast, the networks trained with certified training – CROWN-IBP – (Zhang *et al.*, 2020) produce proofs that do not rely on any spurious features but they also miss out on some meaningful features. Proofs for training methods that combine both empirical and certified robustness – COLT – (Balunovic and Vechev, 2020), not only preserve meaningful features but also selectively filter out spurious ones. These observations have empirically been shown to be not dependent on any specific DNN verifier. These insights suggest that DNNs can satisfy safety properties but their behavior can still be untrustworthy.

# 6

# Analyzing and Verifying Differentiable Programs

Static analysis of differentiable programs opens the door to analyzing a broad new class of formal program properties which go beyond the scope of classical program analysis. However new challenges arise when analyzing a program's derivative that do not exist when analyzing the original program itself. Our goal will be to build a unified and automated framework that gives programmers formal guarantees over derivative properties while removing the burden of needing to know all the intricacies of calculus. Drawing upon both our prior work (Laurel *et al.*, 2022a; Laurel *et al.*, 2022b; Laurel *et al.*, 2023; Laurel *et al.*, 2024; Laurel, 2024) as well as other similar techniques (Zhang *et al.*, 2019; Jordan and Dimakis, 2021; Jordan and Dimakis, 2020; Shi *et al.*, 2022; Deussen, 2021), we will see how to obtain general, precise, and scalable static analysis of differentiable programs.

## 6.1 Differentiable Programming and Automatic Differentiation

Given that computer programs often define mathematical functions, one may ask: can these mathematical functions be differentiable? This question motivates the idea of Differentiable Programming, also called Automatic Differentiation (AD) which is a way to automatically construct programs that compute the mathematical derivatives of other programs. Automatic Differentiation has a long history in Computer Science, going back to at least the 1960s (Wengert, 1964).

To describe Automatic Differentiation, we first introduce in Figure 6.1, a core language used to construct programs that represent differentiable functions. Given a differentiable programming language, AD frameworks are built upon techniques like operator overloading or source code transformations, and these frameworks enable automatically applying the rules of calculus (e.g., chain rule) at the level of a program's source code in order to produce a new program that is a valid mathematical derivative of the original program (Griewank and Walther, 2008). For instance, given the simple program `x = input(); y = sin(x)+x; return y;` (shown in Figure 6.2a), AD can produce a new program (shown in Figure 6.2b) that computes $cos(x) + 1$ so that we can obtain $\frac{\partial \mathrm{y}}{\partial \mathrm{x}}$. The original program is often referred to as the *primal*. In this example, we use forward mode AD implemented by source code transformation. New variables corresponding to derivatives of the primal's intermediate variables are inserted by the AD compiler and are denoted with a `_d` suffix. In particular, `y_d` stores the value of $\frac{\partial \mathrm{y}}{\partial \mathrm{x}}$.

$$
\begin{array}{rcl}
P & ::= & P_1; P_2 \mid x_i = \mathit{Expr} \\
\mathit{Expr} & ::= & x_j + x_k \mid x_j - x_k \mid x_j * x_k \\
 & \mid & 1/x_j \mid \log(x_j) \mid \exp(x_j) \\
 & \mid & \cos(x) \mid \sin(x) \mid \sigma(x_j) \mid c \in \mathbb{R}
\end{array}
$$

**Figure 6.1:** Differentiable Function Syntax

AD has two main variants: forward-mode and reverse-mode. In the forward mode (which is used in Figure 6.2), the derivatives are computed side-by-side with the original program variables, whereas in reverse mode AD, the entire original program is first computed before any derivatives are computed (Griewank and Walther, 2008). Forward-mode AD can be thought of as a forward propagation (through the computational graph) of derivatives of intermediate program variables with respect to a fixed input variable. Reverse-mode AD can be thought

```
x = Input();
y = sin(x)+x;
return y;
```

**(a)**

```
x = Input();
x_d = 1;
y = sin(x)+x;
y_d = (cos(x)+1)*x_d;
return y,y_d;
```

**(b)**

**Figure 6.2:** (a) Primal Program and (b) Forward-Mode Differentiated Program

of as a backward propagation (through the computational graph) of the derivative of a fixed output variable with respect to intermediate program variables.

To compute the entire Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ (expressed as a differentiable program), the time complexity of forward-mode AD is proportional to the number of *input* variables: $\mathcal{O}(m)$. In contrast, the time complexity of reverse-mode AD is proportional to the number of *output* variables: $\mathcal{O}(n)$. Beyond computing first derivatives and Jacobians, AD can be iterated to compute higher-derivatives such as Hessians.

### 6.1.1 Differentiable Programming Language Expressivity

Since the language supports function compositions (e.g., exp), multiplication, and division, the computational graph described by a program is naturally differentiable. Indeed, the chain rule, product rule and quotient rule will respectively be applied by the compiler for each of those operations in the original program. While this language may appear restricted, it remains expressive enough to encode important functions and programs from a variety of applications across Machine Learning, Optimization, and Scientific Computing. For instance, this language can easily express deep neural networks (DNNs) and can also encode numerical ODE solvers like Euler and Runge-Kutta solvers.

## 6.2 Formal Properties Defined over Derivatives

The first derivatives specified by the Jacobian matrix form the foundation of many prominent learning paradigms and are used in all facets of the machine learning (ML) pipeline, from training to testing. Beyond ML, derivatives (including higher-derivatives) are used extensively in scientific computing for tasks as diverse as climate modeling (Mametjanov *et al.*, 2012), analyzing differential equations (Bendtsen and Stauning, 1996; Ma *et al.*, 2021) and sensitivity analysis (Hovland *et al.*, 2005).

Since AD allows one to differentiate through complex programs written for such tasks, one can now define and analyze formal properties over those derivatives. Hence, AD allows one to go beyond analyzing properties over the program's original variables and instead analyze properties over those variables' *derivatives*. For this reason, many key formal properties are defined over the derivatives and gradients that AD computes. Hence AD and differentiable programming open the door to new formal methods problems. These new formal methods problems are especially relevant to trustworthy AI. We now describe key formal properties that are defined over the derivatives computed by AD.

**Notation**. In keeping with the notation of Section 1.2, for a set of points in $\mathbb{R}^m$ described by a (precondition) formula $\varphi$ and a differentiable function $f$, we write $f'(\varphi)$ or $\nabla f(\varphi)$ to denote the evaluation of the derivative or gradient of $f$ on all points in $\varphi$. We use $x_i$ to refer to the $i^{th}$ component of a vector $x$. To denote the evaluation of a specific $i^{th}$ partial derivative over all points in $\varphi$, we write $\frac{\partial f(\varphi)}{\partial x_i}$. To denote the evaluation of higher derivatives over $\varphi$, we may write $f''(\varphi)$ or $\frac{\partial^n f(\varphi)}{\partial x_i \ldots \partial x_k}$.

**Lipschitz robustness.** Lipschitz constants offer a natural way to reason about a function's behavior. Lipschitz constants can be used as a metric to compare the stability and smoothness of the output of neural networks prior to deployment, as a network with a smaller constant is often preferable (Lin *et al.*, 2019). Formal bounds on the Lipschitz constant can also be used during training to learn classifiers that are certifiably robust to adversarial perturbations (Tsuzuku *et al.*, 2018), robust to quantizations (Lin *et al.*, 2019), or to improve interpretability by making network explanations themselves more robust (Alvarez-Melis and Jaakkola, 2018). Further, analyzing the Lipschitz constant has direct applications in algorithmic fairness (Dwork *et al.*, 2012) and differential privacy (Dwork *et al.*, 2006), where fairness and privacy are established by certifying a small Lipschitz constant. This formal property can now be stated. For a differentiable function $f(x) : \mathbb{R}^m \to \mathbb{R}$ and local region $\varphi \subset \mathbb{R}^m$ the bound on the gradient norm implies the local Lipschitz constant bound as follows:

$$\max_{x\in\varphi} \|\nabla f(x)\| \leq L \implies \forall x_1, x_2 \in \varphi,\ \|f(x_1) - f(x_2)\| \leq L\|x_1 - x_2\| \tag{6.1}$$

In this setting $L \in \mathbb{R}_{>0}$ is the local Lipschitz constant. Intuitively, the local Lipschitz constant means that the function $f$ can be bounded by a line with slope $L$. We can equivalently formalize this property with the following postcondition $\psi_L = \{v \in \mathbb{R}^m : \|v\| \leq L\}$, hence the full formal specification becomes:

$$\nabla f(\varphi) \subseteq \psi_L \tag{6.2}$$

**Optimization analysis.** Beyond using the Jacobian for formally bounding (local) Lipschitz constants, a Jacobian analysis can also be used to formally reason about the local optimization geometry of ML models (Zhang *et al.*, 2019). As an example, one may wish to certify that for some local region, $\varphi \subset \mathbb{R}^m$, a differentiable function of interest $f(x) : \mathbb{R}^m \to \mathbb{R}$ never attains its extrema. The desired postcondition is $\psi_o = \{v \in \mathbb{R}^m : v \neq \mathbf{0}\}$. This property can now be stated as:

$$\nabla f(\varphi) \subseteq \psi_o \tag{6.3}$$

Existing work (Deussen, 2021) discusses how derivative bounds which provably exclude zero can then be incorporated into branch-and-bound optimization solvers to provably rule out entire (local) regions of the input space. Hence by knowing that a region excludes any optimal values, an optimization solver can avoid paying the computational cost to explore that region.

**Convexity and concavity.** Another class of formal properties that are defined over derivatives are convexity and concavity conditions. Unlike the properties defined in (6.2) and (6.3), the convexity or concavity of a function is defined over *second* derivatives instead of first derivatives. For a function $f : \mathbb{R}^m \to \mathbb{R}$ and a precondition $\varphi \subset \mathbb{R}^m$, $f$ is convex over all points in $\varphi$ if the following holds:

$$\min_{x \in \varphi} x^T \cdot H(f, x) \cdot x \geq 0 \tag{6.4}$$

In this setting $H(f, x)$ is the Hessian, or matrix of all partial second derivatives. Similarly, $f$ is concave over the points captured by precondition $\varphi$ if

$$\max_{x \in \varphi} x^T \cdot H(f, x) \cdot x \leq 0 \tag{6.5}$$

We note that for functions of a single variable $f : \mathbb{R} \to \mathbb{R}$, proving convexity of $f$ over a region given by a precondition $\varphi$, reduces to proving the following postcondition $\psi_{cnvx} = \{v \in \mathbb{R} : v \geq 0\}$, equivalently:

$$f''(\varphi) \subseteq \psi_{cnvx} \tag{6.6}$$

And likewise proving concavity for a univariate $f$ over $\varphi$ reduces to proving the postcondition $\psi_{cncv} = \{v \in \mathbb{R} : v \leq 0\}$, equivalently:

$$f''(\varphi) \subseteq \psi_{cncv} \tag{6.7}$$

A more thorough treatment of convexity properties is found in Deussen (2021). Convexity properties have been used in trustworthy ML to train fair DNNs (Gupta *et al.*, 2021). Relaxed notions of convexity like *pseudoconvexity* can similarly be formulated as checking interval bounds over second derivatives as in Hladík (2018) and Hladík *et al.* (2021).

**Sensitivity analysis.** Sensitivities are often expressed with derivatives. For instance, to understand how sensitive a function $f$ is to the $i^{th}$ input $x_i$, one often computes $\frac{\partial f}{\partial x_i}$. These derivatives are commonly computed by AD (Hovland *et al.*, 2005) which allows one to perform sensitivity analysis for entire programs. AD-based sensitivity analyses have proven especially useful in Scientific ML (Rackauckas *et al.*, 2020; Blondel *et al.*, 2022). In our setting, one may aim to prove that in a region $\varphi \subset \mathbb{R}^m$ that a function's sensitivity is bounded by some $K \in \mathbb{R}_{>0}$. The evaluation of the $i^{th}$ partial derivative on every point in $\varphi$ is denoted as $\frac{\partial f}{\partial x_i}(\varphi)$. The desired postcondition is $\psi_K = \{v \in \mathbb{R} : v \leq K\}$. Hence the formal specification we wish to prove is:

$$\frac{\partial f}{\partial x_i}(\varphi) \subseteq \psi_K \tag{6.8}$$

This specification corresponds to a *robust* sensitivity analysis.

**Explainable ML: feature attributions and interactions** Similar in spirit to sensitivity analysis is Explainable Machine Learning, which aims to interpret how ML models like neural networks make their decisions. In Explainable ML, one may seek to explain which features or input pixels are more important or salient than others. The reason is that one may wish to *attribute* a DNN's output to a particular feature, hence why explanations are often referred to as feature attributions.

Derivatives with respect to the input features are computed to quantify these attributions (Simonyan *et al.*, 2013) and these derivatives can then be compared to rank which features are more salient than others. In our setting, this idea can be formally specified as follows: for a function (e.g., a DNN) $f : \mathbb{R}^m \to \mathbb{R}$, a local region $\varphi \subset \mathbb{R}^m$ and features $x_1$ and $x_2$, we want to verify the following inequality:

$$\forall x \in \varphi, \frac{\partial f(x)}{\partial x_2} < \frac{\partial f(x)}{\partial x_1} \tag{6.9}$$

This specification certifies that feature $x_1$ is always more salient than $x_2$. In practice we can prove (6.9) by verifying that

$$\max_{x \in \varphi} \frac{\partial f(x)}{\partial x_2} < \min_{x \in \varphi} \frac{\partial f(x)}{\partial x_1} \tag{6.10}$$

Hence as long as the upper bound of one feature's attribution is less than the lower bound of another feature's attribution, one can provably rank the importance of the features. Alternatively, we can formalize the proof of (6.10) as verifying that the postcondition $\psi_{>0} = \{v \in \mathbb{R} : v > 0\}$ holds as follows (where "$-$" is the Minkowski difference):

$$\frac{\partial f(\varphi)}{\partial x_1} - \frac{\partial f(\varphi)}{\partial x_2} \subseteq \psi_{>0} \tag{6.11}$$

In Explainable ML, higher derivatives are used to express properties corresponding to the interaction of *multiple* input features (Lerman *et al.*, 2021). Thus for a function $f(x_1, ..., x_m) : \mathbb{R}^m \to \mathbb{R}$, a local region $\varphi$, and $n$ input features of interest $x_i, ..., x_k$ one may wish to find the tightest range $l, u \in \mathbb{R}$ that encloses the *higher* derivatives. The postcondition can be formalized as $\psi_{l,u} = \{v \in \mathbb{R} : l \leq v \leq u\}$, hence the specification for robustly quantifying $n^{th}$-order interactions is:

$$\frac{\partial^n f(\varphi)}{\partial x_i, ..., \partial x_k} \subseteq \psi_{l,u} \tag{6.12}$$

In addition, just as first-derivative attributions can be provably ranked, (6.10) and (6.11), different higher derivative interactions can also be provably ranked. This formal property can be stated as follows:

$$\frac{\partial^n f(\varphi)}{\partial x_i, ..., \partial x_k} - \frac{\partial^n f(\varphi)}{\partial x_j, ..., \partial x_l} \subseteq \psi_{>0} \tag{6.13}$$

**Monotonicity.** In many applications, the formal property one must certify is that a function, such as DNN, behaves monotonically. The monotonicity property has proven useful in high-stakes social settings such as for DNNs that are used to hire candidates or to offer applicants loans. Indeed many algorithmic fairness properties can be formalized as a monotonicity condition (Shi *et al.*, 2022; Sivaraman *et al.*, 2020). For example, one way wish to ensure that for two otherwise equally qualified job candidates, the candidate with more work experience is preferred. Even beyond algorithmic fairness, monotonicity specifications of neural networks (defined over their derivatives) arise in computer systems tasks such as in Wei *et al.* (2023).

The monotonicity property for a univariate differentiable function $f$ (e.g., a DNN) with respect to an input $x$ over a region $\varphi \subset \mathbb{R}$ can be stated as follows: $f$ is monotonically increasing with respect to $x$ if

$$\forall x_1, x_2 \in \varphi, x_1 \leq x_2 \implies f(x_1) \leq f(x_2) \tag{6.14}$$

However (6.14) is both one-dimensional and relational since it is defined over separate inputs $x_1$ and $x_2$. These limitations can be overcome with derivatives. Using derivatives and the postcondition $\psi_{\geq 0} = \{v \in \mathbb{R} : v \geq 0\}$, we can recast the monotonically *increasing* specification into an equivalent non-relational form and generalize it to higher dimensions. This formalization can be stated as follows: $f : \mathbb{R}^m \to \mathbb{R}$ is monotonically increasing with respect to the $i^{th}$ feature over a region $\varphi \subset \mathbb{R}^m$ if

$$\frac{\partial f(\varphi)}{\partial x_i} \subseteq \psi_{\geq 0} \tag{6.15}$$

Similarly, $f$ is monotonically *decreasing* if the postcondition $\psi_{\leq 0} = \{v \in \mathbb{R} : v \leq 0\}$ is satisfied, or equivalently:

$$\frac{\partial f(\varphi)}{\partial x_i} \subseteq \psi_{\leq 0} \tag{6.16}$$

**Range analysis.** One may also need to perform a range analysis on the gradients. For instance, in Misra *et al.* (2023), the authors needed to reason about how small and how large gradients obtained during gradient descent can be so that their compiler can select an appropriate fixed point arithmetic data type with sufficient number of integer bits to avoid gradient overflows and underflows. The range analysis specification can be stated as follows: for a differentiable function $f$, an input region $\varphi \subset \mathbb{R}$ one needs to find the tightest $l, u \in \mathbb{R}$ such that the postcondition $\psi_{l,u} = \{v \in \mathbb{R} : l \leq v \leq u\}$ is still satisfied, or equivalently:

$$f'(\varphi) \subseteq \psi_{l,u} \tag{6.17}$$

**Domain specific properties.** In specific domains, researchers have formalized other properties over derivatives for highly specific classes of models. For instance, Eiras *et al.* (2023) uses derivative bounds to reason about physics-inspired DNNs in scientific ML. Chang *et al.* (2019) uses bounds on Jacobian-vector product to formally guarantee the stability of neural network controllers and Deussen (2021) certifies bounds over higher derivatives to compute sensitivities needed to select where to apply approximations to a program. In addition, Qin *et al.* (2022) certified bounds on derivatives in connection with barrier functions for verifying self-driving control systems.

## 6.2.1 Encoding Derivative Properties by Abstract Interpretation

While we formally defined the properties of interest above, one still needs an automated method to analyze and verify those properties. As a solution, our work uses abstract interpretation as the foundational framework to analyze these derivative properties.

The key insight is that the local regions $\varphi \subset \mathbb{R}^m$ (captured by the preconditions) shown in the equations of Section 6.2 can be encoded with numerical abstract domains like intervals, zonotopes or polyhedra.

Hence checking derivative properties and verifying postconditions of a program (e.g., a DNN) reduces to numerical abstract interpretation over the derivative code generated by an AD tool.

## 6.3 Challenges

While Automatic Differentiation and Abstract Interpretation offer opportunities for synergy, combining these two distinct techniques encounters several challenges. In particular, one must ensure that static analysis of differentiable programs by abstract interpretation attains *generality, precision, and scalability.* While these concerns are also faced by other types of program analyses, the setting of differentiable programming poses unique challenges and opportunities. In this context, *generality* means the ability of an analysis to support multiple different features of AD, such as higher-order derivatives and non-differentiable functions. *Precision* means the analysis should compute as tight of a bound as possible on the derivative expressions, which is a challenging task since most derivative expressions are highly nonlinear. Lastly, *scalability* means the analysis should compute derivative bounds for programs with as many variables as possible - a core necessity for large ML programs like DNNs. We now describe these challenges in more detail.

### 6.3.1 Generality

Formal, compositional reasoning about the semantics of differentiable programs presents challenges because computer programs are often *non-differentiable.* These points of non-differentiability stem from branch statements in the program. These mathematical difficulties in the program mean one thing: to prove formal guarantees about AD code, more generalized types of derivatives are needed.

The need for generality also extends to *higher* derivatives and *richer* abstract domains. As shown in Section 6.2, formal properties are often defined over higher derivatives. Previously, a programmer would have to define an AD semantics for the desired order of derivative and then prove the corresponding abstract semantics sound for a chosen abstract domain. To compute a different order of derivative or use a different abstract domain, all proofs would need to redone which puts a heavy burden on the programmer.

### 6.3.2 Precision

Obtaining precision for AD static analyses remains difficult. This difficulty stems from the large amount of nonlinear operations inherent to AD. Nonlinear operations pose challenges because most abstract interpreters were designed for linear operations (Cousot and Halbwachs, 1978; Singh *et al.*, 2017). Compared to the original program (the *primal*), the derivative program AD computes (called the *adjoint*) can have 2-5$\times$ more non-linear operations (Griewank and Walther, 2008), e.g., for the most common operations:

- Every composition with a *non-linear function* in the primal requires a separate composition with that function's derivative in the adjoint and an additional multiplication, due to the chain rule.
- A single multiplication in the primal leads to 2 separate multiplications in the adjoint due to the product rule.
- A single division in the primal leads to 4 nonlinear operations in the adjoint due to the quotient rule.

As a strategy to tame the imprecision resulting from this increased amount of nonlinearity (compared to other kinds of programs) one could try to design optimal abstractions for groups of operations. However the challenge then becomes how to choose the right level of granularity for a more precise abstraction, a question which lacks an easy answer.

### 6.3.3 Scalability

Since derivative computations in AD typically have 2$\times$-5$\times$ more operations than the original function that was differentiated (Griewank and Walther, 2008), scalability becomes a primary concern, especially when analyzing derivative properties of large neural networks. Furthermore, tackling the precision challenge also affects the scalability, as more precise analyses tend to be more expensive and thus less scalable.

### Combining Generality, Precision and Scalability

The challenges of *generality*, *precision*, and *scalability* do not exist in isolation. In fact these three dimensions mutually influence each other. For instance generality and precision are often competing goals and similarly obtaining more precision often comes at the cost of less scalability. In addition, many concerns exist at the intersection of these dimensions. For example, supporting the analysis of higher-order AD creates challenges for precision since higher-derivatives contain more nonlinearity than first derivatives and creates challenges for generality since new semantics are needed. Hence striking the perfect balance between these three dimensions remains difficult. A visual representation of these three dimensions and the AD-specific concerns that cut across these dimensions is shown in Figure 6.3.

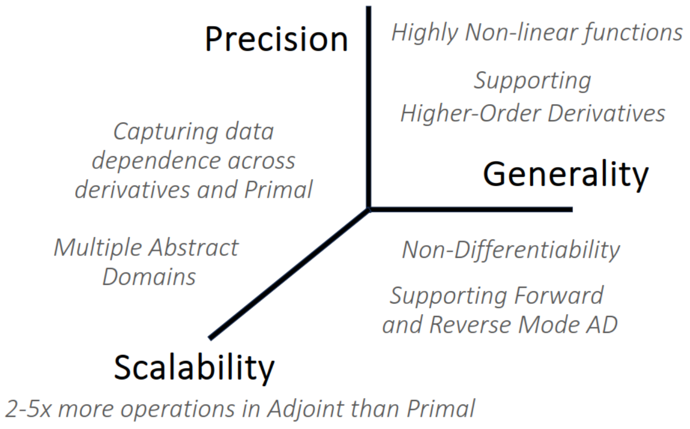

Precision
Highly Non-linear functions
Supporting Higher-Order Derivatives
Capturing data dependence across derivatives and Primal
Generality
Multiple Abstract Domains
Non-Differentiability
Supporting Forward and Reverse Mode AD
Scalability
2-5x more operations in Adjoint than Primal

**Figure 6.3:** Precision, Generality and Scalability dimensions along with their associated analyses concerns which are shared across multiple dimensions.

## 6.4 Synthesizing Precise AD Static Analyzers

We now present Pasado, our technique for synthesizing precise abstract transformers, specialized for AD. Pasado's technique allows us to synthesize precise abstract transformers for the Chain Rule, Product

Rule and Quotient Rule of Calculus, which we will denote $T_{C_f}$, $T_P$, $T_Q$: $\mathcal{D} \to \mathcal{D}$ where $\mathcal{D}$ is the abstract domain which in our work will be the reduced product of Zonotopes and Intervals. This section focuses solely on Pasado's chain rule abstraction, rules for the quotient and product rule abstractions can be found in Laurel *et al.* (2023).

**Function properties.** We require each function $f$ to to be twice differentiable and we require a guaranteed root solver for the second derivative of each $f$, so that we can solve for all $x^* \in [l, u]$ (or certify that none exist inside $[l, u]$) such that $f''(x^*) = c$ for any given $c \in \mathbb{R}$.

**Pasado preliminaries.** Pasado will use standard abstract transformers to abstract the primal computation and $T_{C_f}$, $T_P$, $T_Q$ to abstract the derivative computation. Since both forward-mode AD and reverse-mode AD use these same rules, we can synthesize precise abstractions for each of the core operations for either mode of AD. The only difference between AD modes is the order of application, for instance in forward mode for $a \in \mathcal{D}$, the application order is $T_P(T_*(T_{C_f}(T_f(a))))$ while for reverse mode the order is $T_P(T_{C_f}(T_*(T_f(a))))$ since the entire primal must be abstracted before any derivatives can be. In our setting, the input abstract element $a \in \mathcal{D}$ captures the precondition $\varphi$. Our goal is to verify the final abstract element satisfies a desired postcondition $\psi$.

Pasado's abstract transformer synthesis involves a combination of linear regression at uniformly spaced points and solving a nonlinear optimization problem to ensure soundness.

## 6.4.1 Chain Rule Synthesized Transformer

The first rule of Calculus for which we want to synthesize a precise abstraction is the Chain Rule. For functions $f, g : \mathbb{R} \to \mathbb{R}$, the chain rule is mathematically given as:

$$f(g(u))' = f'(g(u)) \cdot g'(u) \tag{6.18}$$

**Forward-mode chain rule.** In forward-mode AD, this rule is implemented via:

```
z.real = f(x.real);
z.dual = f'(x.real)*x.dual    //chain rule
```

where intuitively, `x.real` $= g(u)$, `x.dual` $= g'(u)$, `z.real` $= f(g(u))$, and `z.dual` $= f(g(u))'$.

**Reverse-mode chain rule.** Likewise in reverse-mode AD, this rule is implemented as:

```
z = f(x); ...
z̄ = ...;
x̄ += f'(x)*z̄;    //chain rule
```

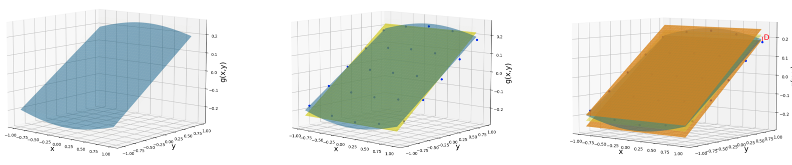
where the "..." at the end of the first line represents the break between the end of the *primal* part of the differentiable program and the start of the *adjoint* part of the same differentiable program, which computes all the derivatives (e.g. $\overline{z}$, $\overline{x}$).

**Chain rule abstraction pattern.** Based on these implementations, the main expression, present in both forward and reverse AD, for which we want to synthesize an abstract transformer, $T_{C_f}$, is:

$$\boxed{g(x,y) = f'(x) \cdot y}$$

The benefit of synthesizing an abstraction for this chain rule pattern is that this pattern could have *multiple* nonlinear operations. For instance, if $f(x) = \sigma(x)$, then $f'(x) = \sigma(x) \cdot (1 - \sigma(x))$, which has a nonlinear multiplication, in addition to the nonlinear multiplication with $y$. Thus naively composing the abstract transformers for each nonlinear operation e.g., $T_*(T_*(T_-(T_\sigma(a))))$ as in Laurel *et al.* (2022b) can lead to imprecision. When using zonotopes, each of those nonlinear operations introduces a new noise symbol which adds additional over-approximation. In contrast, $T_{C_f}$ introduces only a single noise symbol for the entire chain rule derivative expression.

**Abstraction.** We now present how to abstract the chain rule pattern. Algorithm 2 presents the abstract transformer $T_{C_f}$ and Figure 6.4 presents a geometric intuition. The core idea is to sample uniformly spaced points that lie within the range of the input intervals and then solve a linear regression problem to find the best linear approximation of $f'(x) \cdot y$ at those points. However, the most critical step for proving soundness is solving a challenging multidimensional, nonconvex opti-



**(a)** **(b)** **(c)**

**Figure 6.4:** Visualization of Pasado's abstract transformer synthesis for the Chain rule pattern $g(x, y) = \sigma(x) \cdot (1 - \sigma(x)) \cdot y$ on $[-1, 1] \times [-1, 1]$. In (a), the blue surface represents $g(x, y)$. In (b), blue dots on the blue surface represent evaluations of $g(x, y)$ at grid sampled points. The yellow hyperplane in (b) is computed by performing linear regression with these blue points and has equation $Ax + By + C$. In (c), the red lines show the difference between $g(x, y)$ and the plane and $D$ represents the maximum such difference. The lower and upper orange planes in (c) are the enclosing linear bounds given by $Ax + By + C \pm D$. The enclosing bounds are parallel for the Zonotope domain and here the maximum difference $D$ occurs at a corner point.

mization problem, to soundly enclose the linear approximation, which we next describe.

**Optimization problem.** The core technical difficulty of the Chain Rule abstract transformer lies in solving the following equation for the maximum deviation between the linear approximation $(Ax + By + C)$ and the function $f'(x) \cdot y$ itself (example shown in Figure 6.4). This maximum deviation is needed to obtain the tightest enclosure around the linear approximation $(Ax+By+C)$ such that this enclosure provably contains the range of $f'(x) \cdot y$. This deviation $D$ is computed as:

$$D = \max_{x \in [l_x, u_x], y \in [l_y, u_y]} |f'(x) \cdot y - (Ax + By + C)| \tag{6.19}$$

Pasado reduces this multivariate, non-convex optimization problem into two simpler univariate problems as well as simply checking the four corner points: $\{l_x, u_x\} \times \{l_y, u_y\}$. For the correctness of our proof which is shown in Theorem 4.1 of Laurel *et al.* (2023), it is a technical requirement that $A \neq 0$. If linear regression obtains $A = 0$, we perturb $A$ by a small quantity, $\delta < 10^{-9}$. To solve the two univariate optimization problems, we compute all $x^* \in [l_x, u_x]$ such that $f''(x^*) = \frac{A}{l_y}$ and all $x^{**} \in [l_x, u_x]$ such that $f''(x^{**}) = \frac{A}{u_y}$. Thus we must also examine the points $(x^*, l_y)$ and $(x^{**}, u_y)$. We can solve for all $x^*$ and $x^{**}$ using

**Algorithm 2:** Chain Rule Abstract Transformer $T_{C_f}$

```
**Input:** Abstract state $a \in \mathcal{D}$ where $\widehat{x} = a[x].\widehat{x}$ and $\widehat{y} = a[y].\widehat{y}$
**Output:** Affine form and optimal interval enclosing $f'(x) \cdot y$
$l_x, u_x \leftarrow Range(a[x])$;
$l_y, u_y \leftarrow Range(a[y])$;
$grid \leftarrow GridSample([l_x, u_x] \times [l_y, u_y])$;
$pts \leftarrow \{f'(x) \cdot y : (x, y) \in grid\}$;
$A, B, C \leftarrow LinearRegression(grid, pts)$;
**if** $A = 0$ **then** $A \leftarrow A + \delta$;
$D \leftarrow \max_{x \in [l_x, u_x], y \in [l_y, u_y]} |f'(x) \cdot y - (Ax + By + C)|$;
$l, u \leftarrow \min_{x \in [l_x, u_x], y \in [l_y, u_y]} f'(x) \cdot y, \max_{x \in [l_x, u_x], y \in [l_y, u_y]} f'(x) \cdot y$ ;
**return** $A\widehat{x} + B\widehat{y} + C + D\epsilon_{new}$ , $[l, u]$
```

the guaranteed root solver that we required in Section 6.4. Hence the optimization problem ultimately reduces to:

$$D = \max_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} |f'(x) \cdot y - (Ax + By + C)|$$

The generality of our approach also stems from expanding this proof technique to other patterns arising from AD. We also show how to adapt this proof to obtain precise interval domain transformers. The key benefit is that we can use virtually the same proof to get the *exact* lower and upper bounds of $f'(x) \cdot y$ for the given input intervals. Hence we can compute optimal lower and upper bounds, $l$ and $u$, as follows:

$$l = \min_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} f'(x) \cdot y \tag{6.20}$$

$$u = \max_{(x,y) \in \left(\{l_{x_1}, u_{x_1}\} \times \{l_{y_1}, u_{y_1}\}\right) \cup \{(x^*, l_y), (x^{**}, u_y)\}} f'(x) \cdot y \tag{6.21}$$

The core benefit of having both zonotope affine forms and separately computed interval lower and upper bounds is that not only does the same proof strategy give us sound abstractions for both domains, but by taking their reduced product, we can always use the interval results to refine the zonotope to enhance precision. Further, Pasado's approach to synthesize linear bounds is applicable to any abstract domain that can exactly represent linear expressions. Hence Pasado's technique would yield sound abstract transformers for other domains like Polynomial Zonotopes (Kochdumper *et al.*, 2023) or DeepPoly (Singh *et al.*, 2019b).

## 6.5 Higher-order AD and AD with Branching

To generalize abstract interpretation of AD to support nondifferentiability, our work also built DeepJ (Laurel *et al.*, 2022a). DeepJ grapples with non-differentiability by defining the first abstract semantics based on Clarke Generalized Jacobians. This generality allows DeepJ to reason about gradient properties for both differentiable and non-differentiable (but Lipschitz continuous) functions. This generality also means DeepJ is the first to obtain Lispchitz robustness guarantees on neural networks (DNNs) that are adversarially perturbed by non-differentiable perturbations like image rotations, a threat model no prior work addressed.

Additionally, the need for generality also extends to *higher* derivatives. Hence, our prior work also developed the first general construction for abstract interpretation of higher-order AD (Laurel *et al.*, 2022b). Our work creates a general framework for building sound abstract interpreters for arbitrary orders of derivatives and general classes of abstract domains. This approach removes the programmer's burden of reformalizing their abstract semantics each time they want to use a different abstract domain or compute a different derivative. Instead, programmers only specify a small set of abstract transformers for primitive functions (e.g. $\exp(x)$) and the highest desired derivative to obtain both a sound concrete and sound abstract semantics which abstractly interprets all derivatives up to that chosen order.

## 6.6 Case Study: Monotonicity Analysis of an Adult Income Network

To highlight a practical use of verifying differentiable programs for safe and trustworthy AI, we show a case study (Laurel *et al.*, 2023). In this case study, we conduct a monotonicity analysis on a multilayer perceptron (MLP) trained on the Adult dataset (Becker and Kohavi, 1996). Our MLP takes 87 input features (where 81 of the 87 features result from one-hot encodings of the original dataset's categorical variables), passes these features through two hidden layers (each containing 10 neurons and applying tanh activation), and outputs a single binary classification score predicting the income level. Our goal is to verify the monotonicity (both increasing and decreasing) of the MLP's output with respect to 5 continuous input features which are: *Age*, *Education-Num*, *Capital Gain*, *Capital Loss*, and *Hours per week*. The monotonicity specifications we verify are the same as those presented in (6.15) and (6.16). Whereas prior work (Shi *et al.*, 2022) varied one feature at a time while holding the value of all other features as fixed, our experiments allow all 5 of the aforementioned continuous features to simultaneously vary within interval bounds. Hence we abstractly interpret the continuous features with a 5D $L_\infty$-ball, with a radius $\epsilon \in [0, 1]$, while holding all the remaining features as fixed. This input ball represents the precondition $\varphi$. Since training data is normalized to have zero mean and unit variance, passing a 5D $L_\infty$-ball with $\epsilon = 0.4$ through the MLP is equivalent to exploring an infinite set of inputs that satisfy $Age \in [33.2, 44.1]$, $Education\text{-}Num \in [9.05, 11.1]$, $Capital\ Gain \in [-1900, 4060]$, $Capital\ Loss \in [-73.7, 249]$, and $Hours\ per\ week \in [35.5, 45.4]$. For this analysis, we used Pasado's reverse-mode AD abstract transformers. Hence in a single (abstract) pass, Pasado computes bounds on the partial derivatives of the output with respect to each of the five input features.

For each $L_\infty$-ball radius $\epsilon$, Pasado abstractly computes bounds on the five partial derivatives when the original input is perturbed by the $L_\infty$-ball for 100 different inputs, computing 500 partial derivative bounds in total. In addition, we compare Pasado against interval AD and zonotope AD. For a given input $L_\infty$-ball, to verify monotonicity with respect to a chosen feature, the partial derivative bound with respect to that feature should provably exclude 0, meaning the interval should be strictly positive (monotonically increasing) or strictly negative (monotonically decreasing). This condition ensures that the MLP is monotonic with respect to that feature for *all* input points in the given $L_\infty$-ball. Hence in Figure 6.5, we show the total number of partial derivative bounds that exclude 0 over 100 test inputs, for different-sized $L_\infty$-balls.

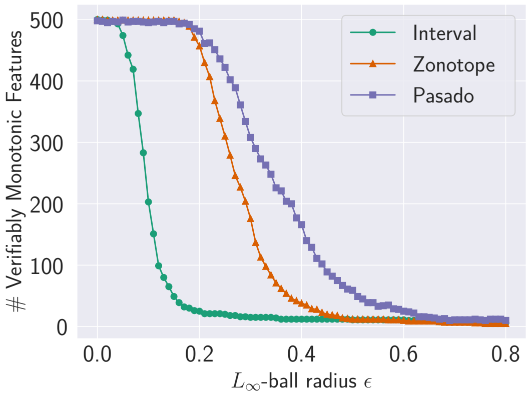


**Figure 6.5:** Counts of verifiably monotonic features of Adult MLP over 100 test-set inputs.

Figure 6.5 shows that the ability of interval AD to prove monotonicity sharply decreases for $\epsilon \geq 0.05$ due to the inherent imprecision of the interval domain. For small $\epsilon$ such as $0 \leq \epsilon \leq 0.2$, zonotope AD and Pasado produce similar counts, meaning both can prove monotonicity. However, their respective performances diverge as $\epsilon$ increases. When $0.2 \leq \epsilon \leq 0.6$, the counts for zonotope AD decline rapidly to nearly zero, whereas the counts for Pasado remain high. Hence, in these cases Pasado can prove monotonicity in significantly more instances. For $\epsilon > 0.6$, all three analyses struggle to prove monotonicity for most continuous input features. The average runtimes for the interval AD, zonotope AD, and Pasado are 0.079, 12, and 39 seconds, respectively. In summary, Pasado is able to prove the most monotonicity specifications across all inputs.

## 6.7 Related Work

While a general formalization of abstract interpretation of automatic differentiation emerged from our work (Laurel *et al.*, 2022a; Laurel *et al.*, 2022b; Laurel *et al.*, 2023; Laurel, 2024), prior work served as an important forerunner.

Using interval analysis for AD goes back to at least the 1990s (Mitchell and Hanrahan, 1992; Bendtsen and Stauning, 1996; Corliss and Rall, 1991), and continues to exist in other, more recent works

(Vassiliadis *et al.*, 2016). Additionally, while they do not bound AD specifically, other works (Misailovic *et al.*, 2014; Mangal *et al.*, 2020) certify interval bounds on derivatives computed symbolically. However these works do not support more expressive abstract domains like zonotopes or polyhedra.

RecurJac (Zhang *et al.*, 2019) first considers neural network verification of properties involving gradients and Jacobians, such as local optimization landscapes and Lipschitz constants, with a customized bound propagation algorithm derived from CROWN (Zhang *et al.*, 2018a). However, its derivation is limited to feedforward ReLU networks only. Shi *et al.* (2022) generalized RecurJac to more general computation graphs with Clarke Jacobians, proposed specialized, precision-enhancing abstract transformers for DNN activations, and also conducted branching for refining bounds on derivatives and Jacobians.

In addition, several papers discussed computing Lipschitz constants (Fazlyab *et al.*, 2019b; Jordan and Dimakis, 2020; Jordan and Dimakis, 2021). These works typically formulate this specific problem as an optimization problem, rather than considering an abstract interpretation on a general computation graph; thus, they are often restricted to the Lipschitz constant scenario and cannot be directly used to verify general properties of derivatives or extended to higher derivatives.

Furthermore, most works (with the exception of Shi *et al.*, 2022; Laurel *et al.*, 2022a; Jordan and Dimakis, 2020; Sherman *et al.*, 2021) lack a method for soundly reasoning about conditional branches, since branches induce nondifferentiability.

Besides Laurel *et al.* (2022b), there is limited work on certifying properties over higher derivatives. One notable work that can still support higher derivatives is Deussen (2021). The author used bounds on second derivatives to aid optimization solvers and bounds on higher derivatives for significance analysis of neural networks. However that work uses only the interval domain instead of other abstract domains.

Lastly, only a few works (Laurel *et al.*, 2023; Shi *et al.*, 2022; Zhang *et al.*, 2019) employ custom or hand-crafted abstract transformers, thus virtually all other works including Bendtsen and Stauning (1996), Vassiliadis *et al.* (2016), and Mangal *et al.* (2020) use standard abstract transformers which leads to imprecision.

# 7

# Conclusion

DNNs are generated directly from data, which makes their inner workings less transparent to potential users than classical programs, hindering their trustworthy deployment in real-world applications. To unlock the transformative benefits of DNN technology, it is essential to look beyond measuring accuracy on standard benchmarks and instead train DNNs that are not only accurate but also trustworthy and transparent. In this monograph, we showed how the classical framework of abstract interpretation, originally designed for analyzing programs, can be successfully leveraged for building state-of-the-art solutions for verifying, training, explaining, and interpreting DNNs. Next, we discuss the limitations of existing works and potential ways to move the field forward.

**Limitations.** While significant progress has been made in recent years, the use of formal methods for trustworthy DNN development is still not as widespread as it should be. This is because:

- **Significant manual effort.** Efficient abstract interpreters must balance the precision/scale tradeoff, which requires substantial expertise in algorithm design, formal methods, and performance optimizations. Existing interpreters are constructed from scratch, requiring developers to design the necessary logic, prove its correctness, and take care of low-level implementation details.

- **Inflexible design.** Existing interpreter implementations are tailored to handle the composition of DNN layers in a particular way to verify specific trustworthy properties (like robustness) on simple DNN architectures (like fully connected, convolutional). In several practical cases, verification requires handling an arbitrary composition that these implementations cannot handle, like when verifying new complex state-of-the-art architectures, or properties beyond robustness or simple combinations of output neurons on existing architectures. This manual construction of specialized implementations for a limited set of DNN architectures and properties cannot keep pace with the rapid development of new DNN architectures and the need to prove more diverse properties in different domains.

- **Lack of scalability.** Existing implementations of abstract interpreters are suboptimal and do not fully exploit available parallelization and performance optimization opportunities on modern hardware specialized for DNN workloads. As a result, even if the abstraction is theoretically efficient, the implementations are often too slow or run out of memory when handling larger models (e.g., transformers or diffusion models).

- **Lack of precision on large models.** Although efficient abstract interpretations such as Box/IBP and DeepPoly/CROWN exist and can theoretically scale to very large models with a good implementation, these methods lack precision and provide vacuous bounds on large models and difficult properties. The fundamental tradeoff between precision and scalability is not fully addressed, which hinders the practical verification of very large DNNs, such as large language models.

- **Inaccessibility.** Even inefficient implementations require writing substantial expert code. This is impractical for many end users of deep learning frameworks who lack the necessary background.

To achieve wider adoption of formal methods in deep learning, minimizing the amount of code an end user needs to write to run an abstract interpreter on their trained model is important. This can be accomplished by having an API that intuitively exposes only the relevant functionality while hiding the algorithmic details. Further, the API should be familiar to the existing practitioners of deep learning frameworks so that they can start using formal methods without requiring extra learning or training.

- **Weak system-level guarantees.** DNNs are often employed inside a larger AI-enabled systems. Existing approaches focusing on verifying end-to-end systems do not efficiently exploit the interactions between DNN and system/program level verifiers and therefore cannot prove complex properties of large systems.

**Future work.** To address some of these limitations, we believe that an optimizing compiler framework can be developed to make it easier to generate efficient implementations (Singh *et al.*, 2024; Singh *et al.*, 2025). In this framework, the developer can specify the logic of the abstract interpreter as a minimal, high-level specification written in a domain-specific language. The compiler can then generate code optimized for specialized hardware to support arbitrary combinations of diverse use cases, application domains, properties, and DNN architectures. To enable stronger system-level guarantees, specialized abstractions for neurosymbolic computations can be designed that efficiency capture the interactions between DNNs and other system components.

# References

Allamigeon, X., S. Gaubert, and É. Goubault. (2008). “Inferring Min and Max Invariants Using Max-Plus Polyhedra”. In: *Static Analysis*. Ed. by M. Alpuente and G. Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg. 189–204.

Alvarez-Melis, D. and T. S. Jaakkola. (2018). “Towards robust interpretability with self-explaining neural networks”. In: *Neural Information Processing Systems*.

Amato, F., A. López, E. M. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel. (2013). “Artificial neural networks in medical diagnosis”. *Journal of Applied Biomedicine*. 11(2).

Anderson, G., S. Pailoor, I. Dillig, and S. Chaudhuri. (2019). “Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness”. In: *Proc. Programming Language Design and Implementation (PLDI)*. 731–744.

Anderson, R., J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma. (2020). “Strong mixed-integer programming formulations for trained neural networks”. *Mathematical Programming*. 183(1): 3–39.

Anil, C., J. Lucas, and R. B. Grosse. (2019). "Sorting Out Lipschitz Function Approximation". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. *Proceedings of Machine Learning Research*. PMLR. 291–301. URL: http://proceedings.mlr.press/v97/anil19a.html.

Baader, M., M. Mirman, and M. Vechev. (2020). "Universal Approximation with Certified Networks". In: *International Conference on Learning Representations*.

Bak, S. (2021). "nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement". In: *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*. Ed. by A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez. Vol. 12673. *Lecture Notes in Computer Science*. Springer. 19–36. DOI: 10.1007/978-3-030-76384-8_2.

Bak, S., T. Dohmen, K. Subramani, A. Trivedi, A. Velasquez, and P. Wojciechowski. (2023). "The Octatope Abstract Domain for Verification of Neural Networks". In: *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*. Ed. by M. Chechik, J. Katoen, and M. Leucker. Vol. 14000. *Lecture Notes in Computer Science*. Springer. 454–472. DOI: 10.1007/978-3-031-27481-7_26.

Bak, S., T. Dohmen, K. Subramani, A. Trivedi, A. Velasquez, and P. Wojciechowski. (2024). "The hexatope and octatope abstract domains for neural network verification". *Form. Methods Syst. Des.* 64(1): 178–199. DOI: 10.1007/s10703-024-00457-y.

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

Bak, S., H. Tran, K. Hobbs, and T. T. Johnson. (2020). "Improved Geometric Path Enumeration for Verifying ReLU Neural Networks". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by S. K. Lahiri and C. Wang. Vol. 12224. *Lecture Notes in Computer Science*. Springer. 66–96. DOI: 10.1007/978-3-030-53288-8_4.

Balunovic, M., M. Baader, G. Singh, T. Gehr, and M. Vechev. (2019). "Certifying Geometric Robustness of Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc.

Balunovic, M. and M. T. Vechev. (2020). "Adversarial Training and Provable Defenses: Bridging the Gap". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Baluta, T., Z. L. Chua, K. S. Meel, and P. Saxena. (2021). "Scalable Quantitative Verification For Deep Neural Networks". In: *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE. 312–323. DOI: 10.1109/ICSE43902.2021.00039.

Banerjee, D., A. Singh, and G. Singh. (2024a). "Interpreting Robustness Proofs of Deep Neural Networks". In: *The Twelfth International Conference on Learning Representations*. URL: https://openreview.net/forum?id=Ev10F9TWML.

Banerjee, D. and G. Singh. (2024). "Relational DNN Verification With Cross Executional Bound Refinement". In: *Forty-first International Conference on Machine Learning*. URL: https://openreview.net/forum?id=HOG80Yk4Gw.

Banerjee, D., C. Xu, and G. Singh. (2024b). "Input-Relational Verification of Deep Neural Networks". *Proc. ACM Program. Lang.* 8(PLDI). DOI: 10.1145/3656377.

Barrett, B., A. Camuto, M. Willetts, and T. Rainforth. (2022). "Certifiably robust variational autoencoders". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 3663–3683.

Batten, B., M. Hosseini, and A. Lomuscio. (2024). "Tight Verification of Probabilistic Robustness in Bayesian Neural Networks". In: *International Conference on Artificial Intelligence and Statistics, 2-4 May 2024, Palau de Congressos, Valencia, Spain*. Ed. by S. Dasgupta, S. Mandt, and Y. Li. Vol. 238. *Proceedings of Machine Learning Research*. PMLR. 4906–4914. URL: https://proceedings.mlr.press/v238/batten24a.html.

Becker, B. and R. Kohavi. (1996). "Adult". UCI Machine Learning Repository.

Bender, E. M., T. Gebru, A. McMillan-Major, and S. Shmitchell. (2021). “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?” In: *FAccT ’21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*. Ed. by M. C. Elish, W. Isaac, and R. S. Zemel. ACM. 610–623.

Bendtsen, C. and O. Stauning. (1996). “FADBAD, a flexible C++ package for automatic differentiation”.

Berrada, L., S. Dathathri, K. Dvijotham, R. Stanforth, R. Bunel, J. Uesato, S. Gowal, and M. P. Kumar. (2021). “Make Sure You’re Unsure: A Framework for Verifying Probabilistic Specifications”. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan. 11136–11147. URL: https://proceedings.neurips.cc/paper/2021/hash/5c5bc7df3d37b2a7ea29e1b47b2bd4ab-Abstract.html.

Blalock, D. W., J. J. G. Ortiz, J. Frankle, and J. V. Guttag. (2020). “What is the State of Neural Network Pruning?” In: *MLSys*. mlsys.org.

Blondel, M., Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. (2022). “Efficient and modular implicit differentiation”. *Advances in neural information processing systems*. 35: 5230–5242.

Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.* (2016). “End to end learning for self-driving cars”. *arXiv preprint arXiv:1604.07316*.

Bonaert, G., D. I. Dimitrov, M. Baader, and M. T. Vechev. (2021). “Fast and precise certification of transformers”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by S. N. Freund and E. Yahav. ACM. 466–481. DOI: 10.1145/3453483.3454056.

Boopathy, A., T.-W. Weng, P.-Y. Chen, S. Liu, and L. Daniel. (2019). "Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. No. 01. 3240–3247.

Bouissou, O., E. Goubault, J. Goubault-Larrecq, and S. Putot. (2012). "A generalization of p-boxes to affine arithmetic". *Computing*. 94(2-4): 189–201. DOI: 10.1007/S00607-011-0182-8.

Bouissou, O., E. Goubault, S. Putot, A. Chakarov, and S. Sankaranarayanan. (2016). "Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by M. Chechik and J. Raskin. Vol. 9636. *Lecture Notes in Computer Science*. Springer. 225–243.

Brix, C., S. Bak, T. T. Johnson, and H. Wu. (2024a). "The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results". *arXiv preprint arXiv:2412.19985*.

Brix, C., S. Bak, T. T. Johnson, and H. Wu. (2024b). "The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results". URL: https://arxiv.org/abs/2412.19985.

Brix, C., S. Bak, C. Liu, and T. T. Johnson. (2023). "The fourth international verification of neural networks competition (VNN-COMP 2023): Summary and results". *arXiv preprint arXiv:2312.16760*.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. (2020). "Language Models are Few-Shot Learners". In: *Proc. Advances in Neural Information Processing Systems (NeurIPS)*.

Brückner, B. and A. Lomuscio. (2024). "Verification of Neural Networks against Convolutional Perturbations via Parameterised Kernels". URL: https://arxiv.org/abs/2411.04594.

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

Camuto, A., M. Willetts, S. Roberts, C. Holmes, and T. Rainforth. (2021). "Towards a theoretical understanding of the robustness of variational autoencoders". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 3565–3573.

Chakravarthy, A., N. Narodytska, A. Rathis, M. Vilcu, M. Sharif, and G. Singh. (2022). "Property-Driven Evaluation of RL-Controllers in Self-Driving Datacenters". In: *Workshop on Challenges in Deploying and Monitoring Machine Learning Systems (DMML)*.

Chang, Y.-C., N. Roohi, and S. Gao. (2019). "Neural lyapunov control". *Advances in Neural Information Processing Systems*. 32.

Chaudhary, I., Q. Hu, M. Kumar, M. Ziyadi, R. Gupta, and G. Singh. (2025). "Certifying Counterfactual Bias in LLMs". In: *The Thirteenth International Conference on Learning Representations*. URL: https://openreview.net/forum?id=HQHnhVQznF.

Chaudhary, I., V. V. Jain, and G. Singh. (2024a). "Decoding Intelligence: A Framework for Certifying Knowledge Comprehension in LLMs". URL: https://arxiv.org/abs/2402.15929.

Chaudhary, I., S. Lin, C. Tan, and G. Singh. (2024b). "Specification Generation for Neural Networks in Systems". *CoRR*. abs/2412.03028. DOI: 10.48550/ARXIV.2412.03028.

Chen, Y., S. Wang, Y. Qin, X. Liao, S. Jana, and D. A. Wagner. (2021). "Learning Security Classifiers with Verified Global Robustness Properties". In: *Proc. Conference on Computer and Communications Security (CCS)*. ACM. 477–494.

Chevalier, S., I. Murzakhanov, and S. Chatzivasileiadis. (2023). "GPU-Accelerated Verification of Machine Learning Models for Power Systems". *arXiv preprint arXiv:2306.10617*.

Chevalier, S., D. Starkenburg, and K. Dvijotham. (2024). "Achieving the Tightest Relaxation of Sigmoids for Formal Verification". URL: https://arxiv.org/abs/2408.10491.

Chiang, P., R. Ni, A. Abdelkader, C. Zhu, C. Studer, and T. Goldstein. (2020). “Certified Defenses for Adversarial Patches”. In: *Proc. International Conference on Learning Representations (ICLR).*

Chugh, U., A. Mitra, A. Deshwal, N. P. Swaroop, A. Saluja, S. Lee, and J. Song. (2021). “An Automated Approach to Accelerate DNNs on Edge Devices”. In: *ISCAS*. IEEE. 1–5.

Cohen, N., M. Ducoffe, R. Boumazouza, C. Gabreau, C. Pagetti, X. Pucel, and A. Galametz. (2024). “Verification for Object Detection – IBP IoU”. URL: https://arxiv.org/abs/2403.08788.

Corliss, G. F. and L. B. Rall. (1991). “Computing the range of derivatives”. *IMACS Annals on Computing and Applied Mathematics,(to appear).*

Cousot, P. (2021). *Principles of abstract interpretation*. MIT Press.

Cousot, P. and R. Cousot. (1977). “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM. 238–252.

Cousot, P. and N. Halbwachs. (1978). “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. ACM Press. 84–96.

Cousot, P. and M. Monerau. (2012). “Probabilistic abstract interpretation”. In: *European Symposium on Programming*. Springer. 169–193.

Darwiche, A. and A. Hirth. (2020). “On the Reasons Behind Decisions”. In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang. Vol. 325. *Frontiers in Artificial Intelligence and Applications*. IOS Press. 712–720. DOI: 10.3233/FAIA200158.

Dathathri, S., K. Dvijotham, A. Kurakin, A. Raghunathan, J. Uesato, R. Bunel, S. Shankar, J. Steinhardt, I. J. Goodfellow, P. Liang, and P. Kohli. (2020). "Enabling certification of verification-agnostic networks via memory-efficient semidefinite programming". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*

Demarchi, S., D. Guidotti, L. Pulina, A. Tacchella, N. Narodytska, G. Amir, G. Katz, and O. Isac. (2023). "Supporting Standardization of Neural Networks Verification with VNNLIB and CoCoNet." In: *FoMLAS@ CAV.* 47–58.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 248–255.

Deussen, J. (2021). "Global Derivatives". *PhD thesis.*

Dimitrov, D. I., G. Singh, T. Gehr, and M. T. Vechev. (2022). "Provably Robust Adversarial Examples". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022.* OpenReview.net. URL: https://openreview.net/forum?id=UMfhoMtIaP5.

Duong, H., L. Li, T. Nguyen, and M. B. Dwyer. (2023). "A DPLL(T) Framework for Verifying Deep Neural Networks". *CoRR.* abs/2307.10266. DOI: 10.48550/ARXIV.2307.10266.

Dvijotham, K., M. Garnelo, A. Fawzi, and P. Kohli. (2018a). "Verification of deep probabilistic models". URL: https://arxiv.org/abs/1812.02795.

Dvijotham, K., R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli. (2018b). "A Dual Approach to Scalable Verification of Deep Networks". In: *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018.* Ed. by A. Globerson and R. Silva. AUAI Press. 550–559. URL: http://auai.org/uai2018/proceedings/papers/204.pdf.

Dwork, C., M. Hardt, T. Pitassi, O. Reingold, and R. S. Zemel. (2012). "Fairness through awareness". In: *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. ACM. 214–226.

Dwork, C., F. McSherry, K. Nissim, and A. Smith. (2006). "Calibrating noise to sensitivity in private data analysis". In: *Theory of cryptography conference*.

Eiras, F., A. Bibi, R. R. Bunel, K. D. Dvijotham, P. Torr, and M. P. Kumar. (2023). "Efficient Error Certification for Physics-Informed Neural Networks". In: *Forty-first International Conference on Machine Learning*.

Fan, J. and W. Li. (2020). "Adversarial Training and Provable Robustness: A Tale of Two Objectives". *CoRR*. abs/2008.06081. URL: https://arxiv.org/abs/2008.06081.

Fazlyab, M., M. Morari, and G. J. Pappas. (2019a). "Probabilistic Verification and Reachability Analysis of Neural Networks via Semidefinite Programming". URL: https://arxiv.org/abs/1910.04249.

Fazlyab, M., A. Robey, H. Hassani, M. Morari, and G. Pappas. (2019b). "Efficient and accurate estimation of lipschitz constants for deep neural networks". *Advances in neural information processing systems*. 32.

Ferrari, C., M. N. Mueller, N. Jovanović, and M. Vechev. (2022). "Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=l_amHf1oaK.

Ferson, S., V. Kreinovich, L. Grinzburg, D. Myers, and K. Sentz. (2015). "Constructing probability boxes and Dempster-Shafer structures". *Sandia journal manuscript; Not yet accepted for publication*. May. URL: https://www.osti.gov/biblio/1427258.

Fischer, M., C. Sprecher, D. I. Dimitrov, G. Singh, and M. T. Vechev. (2022). "Shared Certificates for Neural Network Verification". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*. Vol. 13371. *Lecture Notes in Computer Science*. Springer. 127–148.

Gehr, T., M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev. (2018). "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 3–18. DOI: 10.1109/SP.2018.00058.

Geng, C., N. Le, X. Xu, Z. Wang, A. Gurfinkel, and X. Si. (2022). "Towards Reliable Neural Specifications". In: *International Conference on Machine Learning*. URL: https://api.semanticscholar.org/CorpusID:253224120.

Geng, C., Z. Wang, H. Ye, S. Liao, and X. Si. (2024). "Learning Minimal NAP Specifications for Neural Network Verification". *CoRR*. abs/2404.04662. DOI: 10.48550/ARXIV.2404.04662.

Gholami, A., S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. (2022). "A survey of quantization methods for efficient neural network inference". In: *Low-Power Computer Vision*. Chapman and Hall/CRC. 291–326.

Gholami, A., S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. (2021). "A Survey of Quantization Methods for Efficient Neural Network Inference". *CoRR*. abs/2103.13630.

Ghorbal, K., E. Goubault, and S. Putot. (2009). "The Zonotope Abstract Domain Taylor1+". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by A. Bouajjani and O. Maler. Vol. 5643. *Lecture Notes in Computer Science*. Springer. 627–633.

Goan, E. and C. Fookes. (2020). "Bayesian Neural Networks: An Introduction and Survey". In: *Case Studies in Applied Bayesian Data Science*. Springer International Publishing. 45–87. DOI: 10.1007/978-3-030-42553-1_3.

Goldberg, A. V. and R. E. Tarjan. (1989). "Finding minimum-cost circulations by canceling negative cycles". *J. ACM*. 36(4): 873–886. DOI: 10.1145/76359.76368.

Goodfellow, I. J., J. Shlens, and C. Szegedy. (2014). "Explaining and harnessing adversarial examples". *arXiv preprint arXiv:1412.6572*.

Goodman, N., V. Mansinghka, D. M. Roy, K. Bonawitz, and J. Tenenbaum. (2008). "Church: a language for generative models with non-parametric memoization and approximate inference". In: *Uncertainty in Artificial Intelligence*. 165.

Gopinath, D., H. Converse, C. S. Pasareanu, and A. Taly. (2020). "Property Inference for Deep Neural Networks". URL: https://arxiv.org/abs/1904.13215.

Goubault, E., S. Palumby, S. Putot, L. Rustenholz, and S. Sankaranarayanan. (2021). "Static Analysis of ReLU Neural Networks with Tropical Polyhedra". In: *Static Analysis*. Ed. by C. Drăgoi, S. Mukherjee, and K. Namjoshi. Cham: Springer International Publishing. 166–190.

Goubault, E. and S. Putot. (2022). "Rino: Robust inner and outer approximated reachability of neural networks controlled systems". In: *International Conference on Computer Aided Verification*. Springer. 511–523.

Goubault, E. and S. Putot. (2024). "A Zonotopic Dempster-Shafer Approach to the Quantitative Verification of Neural Networks". In: *International Symposium on Formal Methods*. Springer. 324–342.

Goubault, E. and S. Putot. (2025). "A Zonotopic Dempster-Shafer Approach to the Quantitative Verification of Neural Networks". In: *Formal Methods*. Ed. by A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi. Cham: Springer Nature Switzerland. 324–342.

Gowal, S., K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli. (2018). "On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models". *CoRR*. abs/1810.12715.

Gowal, S., K. D. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. Mann, and P. Kohli. (2019). "Scalable verified training for provably robust image classification". In: *Proc. IEEE/CVF International Conference on Computer Vision (ICCV)*. 4842–4851.

Griewank, A. and A. Walther. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

Gupta, A., L. Marla, R. Sun, N. Shukla, and A. Kolbeinsson. (2021). "Pender: Incorporating shape constraints via penalized derivatives". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. No. 13. 11536–11544.

Gurobi Optimization, LLC. (2018). "Gurobi Optimizer Reference Manual".

Habeeb, P., D. D'Souza, K. Lodaya, and P. Prabhakar. (2024). "Interval Image Abstraction for Verification of Camera-Based Autonomous Systems". *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 43(11): 4310–4321. DOI: 10.1109/TCAD.2024.3448306.

Henriksen, P. and A. Lomuscio. (2021). "DEEPSPLIT: An Efficient Splitting Method for Neural Network Verification via Indirect Effect Analysis". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Z.-H. Zhou. International Joint Conferences on Artificial Intelligence Organization. 2549–2555. DOI: 10.24963/ijcai.2021/351.

Henriksen, P. and A. R. Lomuscio. (2020). "Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search". In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*. Ed. by G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang. Vol. 325. *Frontiers in Artificial Intelligence and Applications*. IOS Press. 2513–2520. DOI: 10.3233/FAIA200385.

Heo, J., S. Joo, and T. Moon. (2019). "Fooling Neural Network Interpretations via Adversarial Model Manipulation". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2921–2932.

Hladík, M. (2018). "Testing pseudoconvexity via interval computation". *Journal of Global Optimization*. 71(3): 443–455.

Hladík, M., L. V. Kolev, and I. Skalna. (2021). "Linear interval parametric approach to testing pseudoconvexity". *Journal of Global Optimization*. 79: 351–368.

Hovland, P. D., B. Norris, M. M. Strout, S. Bhowmick, and J. Utke. (2005). "Sensitivity analysis and design optimization through automatic differentiation". In: *Journal of Physics: Conference Series*.

Hu, K., A. Zou, Z. Wang, K. Leino, and M. Fredrikson. (2023a). "Unlocking Deterministic Robustness Certification on ImageNet". In: *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. URL: http://papers.nips.cc/paper%5C_files/paper/2023/hash/863da9d40547f1d1b18859519ce2dee4-Abstract-Conference.html.

Hu, K., A. Zou, Z. Wang, K. Leino, and M. Fredrikson. (2023b). "Unlocking deterministic robustness certification on imagenet". *Advances in Neural Information Processing Systems*. 36: 42993–43011.

Huang, Z., S. Dutta, and S. Misailovic. (2021). "Aqua: Automated quantized inference for probabilistic programs". In: *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*. Springer. 229–246.

Hückelheim, J., Z. Luo, S. H. K. Narayanan, S. Siegel, and P. D. Hovland. (2018). "Verifying properties of differentiable programs". In: *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*. 205–222.

Ignatiev, A., N. Narodytska, and J. Marques-Silva. (2019). "Abduction-based explanations for Machine Learning models". In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence. AAAI'19/IAAI'19/EAAI'19*. Honolulu, Hawaii, USA: AAAI Press. DOI: 10.1609/aaai.v33i01.33011511.

Ivanov, R., J. Weimer, R. Alur, G. J. Pappas, and I. Lee. (2019). "Verisig: Verifying Safety Properties of Hybrid Systems with Neural Network Controllers". In: *Proc. Hybrid Systems: Computation and Control (HSCC)*. 169–178.

Jaderberg, M., K. Simonyan, A. Zisserman, and K. Kavukcuoglu. (2015). "Spatial Transformer Networks". In: *Proc. Neural Information Processing Systems (NeurIPS)*. 2017–2025.

Jia, R., A. Raghunathan, K. Göksel, and P. Liang. (2019). "Certified Robustness to Adversarial Word Substitutions". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Ed. by K. Inui, J. Jiang, V. Ng, and X. Wan. Association for Computational Linguistics. 4127–4140.

Jiang, E. and G. Singh. (2024). "Towards Universal Certified Robustness with Multi-Norm Training". URL: https://arxiv.org/abs/2410.03000.

Jin, S., F. Y. Yan, C. Tan, A. Kalia, X. Foukas, and Z. M. Mao. (2024). "AutoSpec: Automated Generation of Neural Network Specifications". URL: https://arxiv.org/abs/2409.10897.

Jordan, M. and A. Dimakis. (2021). "Provable Lipschitz certification for generative models". In: *International Conference on Machine Learning*. PMLR. 5118–5126.

Jordan, M. and A. G. Dimakis. (2020). "Exactly Computing the Local Lipschitz Constant of ReLU Networks". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/5227fa9a19dce7ba113f50a405dcaf09-Abstract.html.

Jovanovic, N., M. Balunovic, M. Baader, and M. T. Vechev. (2022). "On the Paradox of Certified Training". *Trans. Mach. Learn. Res.* 2022.

Kabaha, A. and D. Drachsler-Cohen. (2022). "Boosting Robustness Verification of Semantic Feature Neighborhoods". In: *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*. Vol. 13790. *Lecture Notes in Computer Science*. Springer. 299–324.

Kabaha, A. and D. Drachsler-Cohen. (2024). "Verification of Neural Networks' Global Robustness". *Proc. ACM Program. Lang.* 8(OOPSLA1): 1010–1039. DOI: 10.1145/3649847.

Katz, G., C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. (2017). "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 97–117.

Katz, G., D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. (2019). "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: 443–452.

Kingma, D. P. and J. Ba. (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kingma, D. P. and M. Welling. (2019). "An Introduction to Variational Autoencoders". *Foundations and Trends® in Machine Learning*. 12(4): 307–392. DOI: 10.1561/2200000056.

Ko, C., Z. Lyu, L. Weng, L. Daniel, N. Wong, and D. Lin. (2019). "POPQORN: Quantifying Robustness of Recurrent Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. *Proceedings of Machine Learning Research*. PMLR. 3468–3477. URL: http://proceedings.mlr.press/v97/ko19a.html.

Kochdumper, N. and M. Althoff. (2021). "Sparse Polynomial Zonotopes: A Novel Set Representation for Reachability Analysis". *IEEE Transactions on Automatic Control*. 66(9): 4043–4058. DOI: 10.1109/TAC.2020.3024348.

Kochdumper, N., C. Schilling, M. Althoff, and S. Bak. (2023). "Open- and Closed-Loop Neural Network Verification Using Polynomial Zonotopes". In: *NASA Formal Methods*. Springer Nature Switzerland. 16–36. DOI: 10.1007/978-3-031-33170-1_2.

König, M., A. W. Bosman, H. H. Hoos, and J. N. van Rijn. (2024a). "Critically Assessing the State of the Art in Neural Network Verification". *Journal of Machine Learning Research*. 25(12): 1–53. URL: http://jmlr.org/papers/v25/23-0119.html.

König, M., X. Zhang, H. H. Hoos, M. Kwiatkowska, and J. N. van Rijn. (2024b). "Automated Design of Linear Bounding Functions for Sigmoidal Nonlinearities in Neural Networks". In: *Machine Learning and Knowledge Discovery in Databases. Research Track - European Conference, ECML PKDD 2024, Vilnius, Lithuania, September 9-13, 2024, Proceedings, Part VII.* Ed. by A. Bifet, J. Davis, T. Krilavicius, M. Kull, E. Ntoutsi, and I. Zliobaite. Vol. 14947. *Lecture Notes in Computer Science.* Springer. 383–398. DOI: 10.1007/978-3-031-70368-3_23.

Kos, J., I. Fischer, and D. Song. (2018). "Adversarial examples for generative models". In: *2018 ieee security and privacy workshops (spw).* IEEE. 36–42.

Kotha, S., C. Brix, Z. Kolter, K. Dvijotham, and H. Zhang. (2023). "Provably Bounding Neural Network Preimages". *CoRR.* abs/2302.01404. DOI: 10.48550/ARXIV.2302.01404.

Krizhevsky, A. (2009). "Learning Multiple Layers of Features from Tiny Images".

Kurakin, A., I. J. Goodfellow, and S. Bengio. (2017). "Adversarial examples in the physical world". In: *ICLR (Workshop).* OpenReview.net.

Kurin, V., A. D. Palma, I. Kostrikov, S. Whiteson, and M. P. Kumar. (2022). "In Defense of the Unitary Scalarization for Deep Multi-Task Learning". In: *Advances in Neural Information Processing Systems.* Ed. by A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho. URL: https://openreview.net/forum?id=wmwgLEPjL9.

Ladner, T. and M. Althoff. (2023). "Automatic Abstraction Refinement in Neural Network Verification using Sensitivity Analysis". In: *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control. HSCC '23.* San Antonio, TX, USA: Association for Computing Machinery. DOI: 10.1145/3575870.3587129.

Ladner, T. and M. Althoff. (2024). "Exponent Relaxation of Polynomial Zonotopes and Its Applications in Formal Neural Network Verification". In: *AAAI.* 21304–21311. URL: https://doi.org/10.1609/aaai.v38i19.30125.

Ladner, T., M. Eichelbeck, and M. Althoff. (2024). “Formal Verification of Graph Convolutional Networks with Uncertain Node Features and Uncertain Graph Structure”. URL: https://arxiv.org/abs/2404.15065.

Lan, J., Y. Zheng, and A. Lomuscio. (2022). “Tight Neural Network Verification via Semidefinite Relaxations and Linear Reformulations”. In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022,* AAAI Press. 7272–7280.

Laurel, J. (2024). “Static Analysis of Differentiable Programs”. *PhD thesis.* University of Illinois at Urbana-Champaign.

Laurel, J., S. B. Qian, G. Singh, and S. Misailovic. (2023). “Synthesizing Precise Static Analyzers for Automatic Differentiation”. *Proc. ACM Program. Lang.* (OOPSLA2).

Laurel, J., S. B. Qian, G. Singh, and S. Misailovic. (2024). “Abstract Interpretation of Automatic Differentiation”. In: *Languages for Inference Workshop (LAFI).*

Laurel, J., R. Yang, G. Singh, and S. Misailovic. (2022a). “A dual number abstraction for static analysis of Clarke Jacobians”. *Proc. ACM Program. Lang.* 6(POPL): 1–30.

Laurel, J., R. Yang, S. Ugare, R. Nagel, G. Singh, and S. Misailovic. (2022b). “A general construction for abstract interpretation of higher-order automatic differentiation”. *Proc. ACM Program. Lang.* 6(OOPSLA2): 1007–1035.

LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. (1989). “Handwritten Digit Recognition with a Back-Propagation Network”. In: *NIPS.* 396–404.

Leino, K., Z. Wang, and M. Fredrikson. (2021). “Globally-Robust Neural Networks”. *CoRR.* abs/2102.08452. URL: https://arxiv.org/abs/2102.08452.

Lemesle, A., J. Lehmann, and T. L. Gall. (2024). “Neural Network Verification with PyRAT”. URL: https://arxiv.org/abs/2410.23903.

Lerman, S., C. Venuto, H. Kautz, and C. Xu. (2021). “Explaining local, global, and higher-order interactions in deep learning”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision.* 1224–1233.

Li, J., S. Qu, X. Li, J. Szurley, J. Z. Kolter, and F. Metze. (2019a). "Adversarial Music: Real world Audio Adversary against Wake-word Detection System". In: *Proc. Neural Information Processing Systems (NeurIPS)*. 11908–11918.

Li, J., F. R. Schmidt, and J. Z. Kolter. (2019b). "Adversarial camera stickers: A physical camera-based attack on deep learning systems". In: *Proc. International Conference on Machine Learning, ICML*. Vol. 97. 3896–3904.

Li, L., X. Qi, T. Xie, and B. Li. (2020). "SoK: Certified Robustness for Deep Neural Networks". *CoRR*. abs/2009.04131. URL: https://arxiv.org/abs/2009.04131.

Liang, T., J. Glossner, L. Wang, S. Shi, and X. Zhang. (2021). "Pruning and quantization for deep neural network acceleration: A survey". *Neurocomputing*. 461: 370–403.

Liao, Z. and M. Cheung. (2022). "Automated Invariance Testing for Machine Learning Models Using Sparse Linear Layers". In: *ICML 2022: Workshop on Spurious Correlations, Invariance and Stability*. URL: https://openreview.net/forum?id=VP8ATzLGyQx.

Lin, J., C. Gan, and S. Han. (2019). "Defensive Quantization: When Efficiency Meets Robustness". In: *International Conference on Learning Representations*.

Lin, S., H. He, T. Wei, K. Xu, H. Zhang, G. Singh, C. Liu, and C. Tan. (2024). "NN4SysBench: Characterizing Neural Network Verification for Computer Systems". *Advances in Neural Information Processing Systems*. 37: 91390–91404.

Liu, Z., G. Singh, C. Xu, and D. Vasisht. (2021). "FIRE: enabling reciprocity for FDD MIMO systems". In: *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking. MobiCom '21*. New Orleans, Louisiana: Association for Computing Machinery. 628–641. DOI: 10.1145/3447993.3483275.

Liu, Z., C. Xu, Y. Xie, E. Sie, F. Yang, K. Karwaski, G. Singh, Z. L. Li, Y. Zhou, D. Vasisht, *et al.* (2023). "Exploring practical vulnerabilities of machine learning-based wireless systems". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1801–1817.

Lu, J. and M. P. Kumar. (2019). "Neural Network Branching for Neural Network Verification". URL: https://arxiv.org/abs/1912.01329.

Lundberg, S. M. and S. Lee. (2017). "A unified approach to interpreting model predictions". *CoRR*. abs/1705.07874. URL: http://arxiv.org/abs/1705.07874.

Lyu, Z., M. Guo, T. Wu, G. Xu, K. Zhang, and D. Lin. (2021). "Towards Evaluating and Training Verifiably Robust Neural Networks". URL: https://arxiv.org/abs/2104.00447.

Lyu, Z., C.-Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. (2020). "Fastened crown: Tightened neural network robustness certificates". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. No. 04. 5037–5044.

Ma, Y., V. Dixit, M. J. Innes, X. Guo, and C. Rackauckas. (2021). "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions". In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 1–9.

Ma, Z. (2023). "Verifying Neural Networks by Approximating Convex Hulls". In: *Formal Methods and Software Engineering - 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, November 21-24, 2023, Proceedings*. Ed. by Y. Li and S. Tahar. Vol. 14308. *Lecture Notes in Computer Science*. Springer. 261–266. DOI: 10.1007/978-981-99-7584-6_17.

Ma, Z., J. Li, and G. Bai. (2024). "ReLU Hull Approximation". *Proc. ACM Program. Lang.* 8(POPL). DOI: 10.1145/3632917.

Madry, A., A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. (2017). "Towards deep learning models resistant to adversarial attacks". *arXiv preprint arXiv:1706.06083*.

Malfa, E. L., R. Michelmore, A. M. Zbrzezny, N. Paoletti, and M. Kwiatkowska. (2021). "On Guaranteed Optimal Robust Explanations for NLP Models". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Z. Zhou. ijcai.org. 2658–2665. DOI: 10.24963/IJCAI.2021/366.

Mametjanov, A., B. Norris, X. Zeng, B. Drewniak, J. Utke, M. Anitescu, and P. Hovland. (2012). "Applying automatic differentiation to the Community Land Model". In: *Recent Advances in Algorithmic Differentiation*.

Mangal, R., K. Sarangmath, A. V. Nori, and A. Orso. (2020). "Probabilistic Lipschitz Analysis of Neural Networks". In: *International Static Analysis Symposium*.

Mao, Y., S. Balauca, and M. Vechev. (2024). "CTBENCH: A Library and Benchmark for Certified Training". URL: https://arxiv.org/abs/2406.04848.

Mao, Y., M. N. Mueller, M. Fischer, and M. Vechev. (2023). "Connecting Certified and Adversarial Training". In: *Thirty-seventh Conference on Neural Information Processing Systems*. URL: https://openreview.net/forum?id=T2lM4ohRwb.

Mardziel, P., S. Magill, M. Hicks, and M. Srivatsa. (2013). "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation". *J. Comput. Secur.* 21(4): 463–532.

Marques-Silva, J. and A. Ignatiev. (2022). "Delivering Trustworthy AI through Formal XAI". In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press. 12342–12350. DOI: 10.1609/AAAI.V36I11.21499.

Miné, A. (2002). "A Few Graph-Based Relational Numerical Abstract Domains". In: *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*. Ed. by M. V. Hermenegildo and G. Puebla. Vol. 2477. *Lecture Notes in Computer Science*. Springer. 117–132. DOI: 10.1007/3-540-45789-5_11.

Miné, A. (2006). "The octagon abstract domain". *High. Order Symb. Comput.* 19(1): 31–100.

Miné, A. (2017). "Tutorial on static inference of numeric invariants by abstract interpretation". *Foundations and Trends® in Programming Languages*. 4(3-4): 120–372.

Mirman, M., T. Gehr, and M. Vechev. (2018). “Differentiable abstract interpretation for provably robust neural networks”. In: *Proc. International Conference on Machine Learning (ICML)*. 3578–3586.

Mirman, M., A. Hägele, P. Bielik, T. Gehr, and M. Vechev. (2021). “Robustness certification with generative models”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021*. Virtual, Canada: Association for Computing Machinery. 1141–1154. DOI: 10.1145/3453483.3454100.

Mirman, M., G. Singh, and M. Vechev. (2020). “A Provable Defense for Deep Residual Networks”. URL: https://arxiv.org/abs/1903.12519.

Misailovic, S., M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. (2014). “Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels”. *ACM Sigplan Notices*. 49(10): 309–328.

Misra, A., J. Laurel, and S. Misailovic. (2023). “ViX: Analysis-driven Compiler for Efficient Low-Precision Variational Inference”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

Mitchell, D. and P. Hanrahan. (1992). “Illumination from curved reflectors”. *SIGGRAPH Comput. Graph.* 26(2). DOI: 10.1145/142920.134082.

Mitra, S., C. S. Pasareanu, P. Prabhakar, S. A. Seshia, R. Mangal, Y. Li, C. Watson, D. Gopinath, and H. Yu. (2024). “Formal Verification Techniques for Vision-Based Autonomous Systems - A Survey”. In: *Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*. Ed. by N. Jansen, S. Junges, B. L. Kaminski, C. Matheja, T. Noll, T. Quatmann, M. Stoelinga, and M. Volk. Vol. 15262. *Lecture Notes in Computer Science*. Springer. 89–108. DOI: 10.1007/978-3-031-75778-5_5.

Mohapatra, J., T. Weng, P. Chen, S. Liu, and L. Daniel. (2020). “Towards Verifying Robustness of Neural Networks Against A Family of Semantic Perturbations”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE. 241–249.

Mohr, S., K. Drainas, and J. Geist. (2021). "Assessment of Neural Networks for Stream-Water-Temperature Prediction". *CoRR*. abs/2110.04254. URL: https://arxiv.org/abs/2110.04254.

Müller, C., F. Serre, G. Singh, M. Püschel, and M. T. Vechev. (2021a). "Scaling Polyhedral Neural Network Verification on GPUs". In: *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org.

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

Müller, M. N., F. Eckert, M. Fischer, and M. T. Vechev. (2023a). "Certified Training: Small Boxes are All You Need". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Müller, M. N., M. Fischer, R. Staab, and M. T. Vechev. (2023b). "Abstract Interpretation of Fixpoint Iterators with Applications to Neural Networks". *Proc. ACM Program. Lang.* 7(PLDI): 786–810.

Müller, M. N., G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. (2021b). "Precise Multi-Neuron Abstractions for Neural Network Certification". *arXiv preprint arXiv:2103.03638*.

Munakata, S., C. Urban, H. Yokoyama, K. Yamamoto, and K. Munakata. (2023). "Verifying Attention Robustness of Deep Neural Networks Against Semantic Perturbations". In: *NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings*. Vol. 13903. *Lecture Notes in Computer Science*. Springer. 37–61.

Nielson, F., H. R. Nielson, and C. Hankin. (2005). *Principles of program analysis*. Springer.

Palma, A. D., H. S. Behl, R. Bunel, P. H. S. Torr, and M. P. Kumar. (2021a). "Scaling the Convex Barrier with Active Sets". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Palma, A. D., R. Bunel, A. Desmaison, K. Dvijotham, P. Kohli, P. H. S. Torr, and M. P. Kumar. (2021b). "Improved Branch and Bound for Neural Network Verification via Lagrangian Decomposition". *CoRR*. abs/2104.06718. URL: https://arxiv.org/abs/2104.06718.

Palma, A. D., R. Bunel, K. Dvijotham, M. P. Kumar, and R. Stanforth. (2022). "IBP Regularization for Verified Adversarial Robustness via Branch-and-Bound". *CoRR*. abs/2206.14772. DOI: 10.48550/ARXIV.2206.14772.

Palma, A. D., R. Bunel, K. ( Dvijotham, M. P. Kumar, R. Stanforth, and A. Lomuscio. (2024). "Expressive Losses for Verified Robustness via Convex Combinations". In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. URL: https://openreview.net/forum?id=mzyZ4wzKlM.

Pan, L., A. Albalak, X. Wang, and W. Y. Wang. (2023). "Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning". *CoRR*. abs/2305.12295. DOI: 10.48550/ARXIV.2305.12295.

Păsăreanu, C., H. Converse, A. Filieri, and D. Gopinath. (2020). "On the probabilistic analysis of neural networks". In: *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '20*. Seoul, Republic of Korea: Association for Computing Machinery. 5–8. DOI: 10.1145/3387939.3391594.

Paulsen, B. and C. Wang. (2022). "LinSyn: Synthesizing Tight Linear Bounds for Arbitrary Neural Network Activation Functions". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by D. Fisman and G. Rosu. Vol. 13243. *Lecture Notes in Computer Science*. Springer. 357–376. DOI: 10.1007/978-3-030-99524-9_19.

Paulsen, B., J. Wang, and C. Wang. (2020). “ReluDiff: differential verification of deep neural networks”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE ’20.* Seoul, South Korea: Association for Computing Machinery. 714–726. DOI: 10.1145/3377811.3380337.

Pei, K., Y. Cao, J. Yang, and S. Jana. (2017). “DeepXplore: Automated Whitebox Testing of Deep Learning Systems”. In: *Proc. Symposium on Operating Systems Principles (SOSP)*. 1–18.

Pilipovsky, J., V. Sivaramakrishnan, M. Oishi, and P. Tsiotras. (2023). “Probabilistic Verification of ReLU Neural Networks via Characteristic Functions”. In: *Proceedings of The 5th Annual Learning for Dynamics and Control Conference*. Ed. by N. Matni, M. Morari, and G. J. Pappas. Vol. 211. *Proceedings of Machine Learning Research*. PMLR. 966–979. URL: https://proceedings.mlr.press/v211/pilipovsky23a.html.

Prabhakar, P. and Z. Rahimi Afzal. (2019). “Abstraction based Output Range Analysis for Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/5df0385cba256a135be596dbe28fa7aa-Paper.pdf.

Pulina, L. and A. Tacchella. (2010). “An Abstraction-Refinement Approach to Verification of Artificial Neural Networks”. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by T. Touili, B. Cook, and P. B. Jackson. Vol. 6174. *Lecture Notes in Computer Science*. Springer. 243–257. DOI: 10.1007/978-3-642-14295-6_24.

Qi, Z., S. Khorram, and F. Li. (2020). “Visualizing Deep Networks by Optimizing with Integrated Gradients”. In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press. 11890–11898. DOI: 10.1609/AAAI.V34I07.6863.

Qin, Z., T.-W. Weng, and S. Gao. (2022). “Quantifying safety of learning-based self-driving control using almost-barrier functions”. In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 12903–12910.

Rackauckas, C., Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, A. Ramadhan, and A. Edelman. (2020). “Universal differential equations for scientific machine learning”. *arXiv preprint arXiv:2001.04385*.

Ranzato, F., C. Urban, and M. Zanella. (2021). “Fairness-Aware Training of Decision Trees by Abstract Interpretation”. In: *CIKM ’21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. ACM. 1508–1517.

Räuker, T., A. Ho, S. Casper, and D. Hadfield-Menell. (2023). “Toward Transparent AI: A Survey on Interpreting the Inner Structures of Deep Neural Networks”. URL: https://arxiv.org/abs/2207.13243.

Ribeiro, M. T., S. Singh, and C. Guestrin. (2016). “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 1135–1144.

Ribeiro, M. T., S. Singh, and C. Guestrin. (2018). “Anchors: High-Precision Model-Agnostic Explanations”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press. 1527–1535. DOI: 10.1609/AAAI.V32I1.11491.

Rival, X. and K. Yi. (2020). *Introduction to static analysis: an abstract interpretation perspective*. MIT Press.

Rober, N., S. M. Katz, C. Sidrane, E. Yel, M. Everett, M. J. Kochenderfer, and J. P. How. (2023). “Backward reachability analysis of neural feedback loops: Techniques for linear and nonlinear systems”. *IEEE Open Journal of Control Systems*. 2: 108–124.

Ruan, W., M. Wu, Y. Sun, X. Huang, D. Kroening, and M. Kwiatkowska. (2019). “Global Robustness Evaluation of Deep Neural Networks with Provable Guarantees for the Hamming Distance”. In: *Proc. International Joint Conference on Artificial Intelligence, IJCAI.* Ed. by S. Kraus. 5944–5952.

Ryou, W., J. Chen, M. Balunovic, G. Singh, A. Dan, and M. Vechev. (2021). “Scalable Polyhedral Verification of Recurrent Neural Networks”. In: *International Conference on Computer Aided Verification.* Springer. 225–248.

Sadraddini, S. and R. Tedrake. (2019). “Linear Encodings for Polytope Containment Problems”. In: *Proc. Conference on Decision and Control (CDC).*

Salman, H., G. Yang, H. Zhang, C.-J. Hsieh, and P. Zhang. (2019). “A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks”. In: *Advances in Neural Information Processing Systems.* Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2019/file/246a3c5544feb054f3ea718f61adfa16-Paper.pdf.

Samek, W., G. Montavon, S. Lapuschkin, C. J. Anders, and K. Müller. (2021). “Explaining Deep Neural Networks and Beyond: A Review of Methods and Applications”. *Proc. IEEE.* 109(3): 247–278.

Sankaranarayanan, S., A. Chakarov, and S. Gulwani. (2013). “Static analysis for probabilistic programs: inferring whole program properties from finitely many paths”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13.* Ed. by H. Boehm and C. Flanagan. 447–458.

Shafique, M., M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi. (2020). “Robust Machine Learning Systems: Challenges,Current Trends, Perspectives, and the Road Ahead”. *IEEE Design Test.* 37(2): 30–57.

Sherman, B., J. Michel, and M. Carbin. (2021). “LambdaS: computable semantics for differentiable programming with higher-order functions and datatypes”. *Proceedings of the ACM on Programming Languages.* 5(POPL): 1–31.

Shi, Z., Q. Jin, Z. Kolter, S. Jana, C.-J. Hsieh, and H. Zhang. (2024). "Neural Network Verification with Branch-and-Bound for General Nonlinearities". URL: https://arxiv.org/abs/2405.21063.

Shi, Z., Y. Wang, H. Zhang, Z. Kolter, and C.-J. Hsieh. (2022). "Efficiently Computing Local Lipschitz Constants of Neural Networks via Bound Propagation". In: *Advances in Neural Information Processing Systems*.

Shi, Z., Y. Wang, H. Zhang, J. Yi, and C.-J. Hsieh. (2021). "Fast Certified Robust Training with Short Warmup". URL: https://arxiv.org/abs/2103.17268.

Sidrane, C., A. Maleki, A. Irfan, and M. J. Kochenderfer. (2022). "Overt: An algorithm for safety verification of neural network control policies for nonlinear systems". *Journal of Machine Learning Research*. 23(117): 1–45.

Sill, J. (1997). "Monotonic Networks". In: *Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997]*. The MIT Press. 661–667.

Simon, A. and A. King. (2010). "The two variable per inequality abstract domain". *High. Order Symb. Comput.* 23(1): 87–143. URL: https://doi.org/10.1007/s10990-010-9062-8.

Simonyan, K., A. Vedaldi, and A. Zisserman. (2013). "Deep inside convolutional networks: Visualising image classification models and saliency maps". *arXiv preprint arXiv:1312.6034*.

Singh, A., Y. Sarita, C. Mendis, and G. Singh. (2024). "ConstraintFlow: A DSL for Specification and Verification of Neural Network Analyses". *CoRR*. abs/2403.18729. DOI: 10.48550/ARXIV.2403.18729.

Singh, A., Y. C. Sarita, C. Mendis, and G. Singh. (2025). "Automated Verification of Soundness of DNN Certifiers". *Proc. ACM Program. Lang.* 9(OOPSLA1). DOI: 10.1145/3720509.

Singh, G., R. Ganvir, M. Püschel, and M. Vechev. (2019a). "Beyond the single neuron convex barrier for neural network certification". In: *Advances in Neural Information Processing Systems*.

Singh, G., T. Gehr, M. Mirman, M. Püschel, and M. Vechev. (2018). "Fast and effective robustness certification". *Advances in Neural Information Processing Systems*. 31.

Singh, G., T. Gehr, M. Püschel, and M. Vechev. (2019b). “An abstract domain for certifying neural networks”. *Proceedings of the ACM on Programming Languages*. 3(POPL).

Singh, G., T. Gehr, M. Püschel, and M. Vechev. (2019c). “Boosting Robustness Certification of Neural Networks”. In: *International Conference on Learning Representations*.

Singh, G., T. Gehr, M. Püschel, and M. Vechev. (2019d). “Robustness Certification with Refinement”. In: *International Conference on Learning Representations*.

Singh, G., M. Püschel, and M. T. Vechev. (2017). “Fast polyhedra abstract domain”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. 46–59.

Singla, S. and S. Feizi. (2021). “Skew Orthogonal Convolutions”. URL: https://arxiv.org/abs/2105.11417.

Sivaraman, A., G. Farnadi, T. Millstein, and G. Van den Broeck. (2020). “Counterexample-guided learning of monotonic neural networks”. *Neural Information Processing Systems*.

Smilkov, D., N. Thorat, B. Kim, F. Viégas, and M. Wattenberg. (2017). “Smoothgrad: removing noise by adding noise”. *arXiv preprint arXiv:1706.03825*.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. (2014). “Dropout: a simple way to prevent neural networks from overfitting”. *J. Mach. Learn. Res.* 15(1): 1929–1958.

Suresh, T., D. Banerjee, and G. Singh. (2024). “Relational Verification Leaps Forward with RABBit”. In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. URL: https://openreview.net/forum?id=W5U3XB1C11.

Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. (2014). “Intriguing properties of neural networks”. In: *ICLR (Poster)*.

Tang, X. (2024). “Improved Incremental Verification for Neural Networks”. In: *Theoretical Aspects of Software Engineering*. Ed. by W.-N. Chin and Z. Xu. Cham: Springer Nature Switzerland. 392–409.

Tang, X., Y. Zheng, and J. Liu. (2023). “Boosting Multi-neuron Convex Relaxation for Neural Network Verification”. In: *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*. Ed. by M. V. Hermenegildo and J. F. Morales. Vol. 14284. *Lecture Notes in Computer Science*. Springer. 540–563. DOI: 10.1007/978-3-031-44245-2_23.

Tjandraatmadja, C., R. Anderson, J. Huchette, W. Ma, K. K. Patel, and J. P. Vielma. (2020). “The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification”. *Advances in Neural Information Processing Systems*. 33: 21675–21686.

Tran, H.-D., S. Bak, W. Xiang, and T. T. Johnson. (2020a). “Verification of Deep Convolutional Neural Networks Using ImageStars”. In: *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*. 18–42.

Tran, H.-D., S. Choi, H. Okamoto, B. Hoxha, G. Fainekos, and D. Prokhorov. (2023). “Quantitative Verification for Neural Networks using ProbStars”. In: *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control. HSCC ’23*. San Antonio, TX, USA: Association for Computing Machinery. DOI: 10.1145/3575870.3587112.

Tran, H., D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. (2019a). “Star-Based Reachability Analysis of Deep Neural Networks”. In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by M. H. ter Beek, A. McIver, and J. N. Oliveira. Vol. 11800. *Lecture Notes in Computer Science*. Springer. 670–686. DOI: 10.1007/978-3-030-30942-8_39.

Tran, H.-D., D. Manzanas Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson. (2019b). “Star-Based Reachability Analysis of Deep Neural Networks”. In: *Formal Methods – The Next 30 Years*. Cham: Springer International Publishing. 670–686.

Tran, H., N. Pal, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. (2021). "Verification of piecewise deep neural networks: a star set approach with zonotope pre-filter". *Formal Aspects Comput.* 33(4-5): 519–545.

Tran, H., X. Yang, D. M. Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson. (2020b). "NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I.* Ed. by S. K. Lahiri and C. Wang. Vol. 12224. *Lecture Notes in Computer Science.* Springer. 3–17.

Trockman, A. and J. Z. Kolter. (2021). "Orthogonalizing Convolutional Layers with the Cayley Transform". *CoRR.* abs/2104.07167. URL: https://arxiv.org/abs/2104.07167.

Tsipras, D., S. Santurkar, L. Engstrom, A. Turner, and A. Madry. (2019). "Robustness May Be at Odds with Accuracy". In: *proc. International Conference on Learning Representations, ICLR.* OpenReview.net.

Tsuzuku, Y., I. Sato, and M. Sugiyama. (2018). "Lipschitz-margin training: scalable certification of perturbation invariance for deep neural networks". In: *Neural Information Processing Systems.*

Ugare, S., D. Banerjee, S. Misailovic, and G. Singh. (2023). "Incremental Verification of Neural Networks". *Proc. ACM Program. Lang.* 7(PLDI).

Ugare, S., G. Singh, and S. Misailovic. (2022). "Proof transfer for fast certification of multiple approximate neural networks". *Proc. ACM Program. Lang.* 6(OOPSLA1): 1–29.

Urban, C., M. Christakis, V. Wüstholz, and F. Zhang. (2020a). "Perfectly parallel fairness certification of neural networks". *Proc. ACM Program. Lang.* 4(OOPSLA): 185:1–185:30. DOI: 10.1145/3428253.

Urban, C., M. Christakis, V. Wüstholz, and F. Zhang. (2020b). "Perfectly parallel fairness certification of neural networks". *Proc. ACM Program. Lang.* 4(OOPSLA). DOI: 10.1145/3428253.

Vassiliadis, V., J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann. (2016). “Towards automatic significance analysis for approximate computing”. In: *2016 IEEE/ACM International Symposium on Code Generation and Optimization.* No. CGO.

Wang, S., K. Pei, J. Whitehouse, J. Yang, and S. Jana. (2018). “Efficient formal safety analysis of neural networks”. In: *Advances in Neural Information Processing Systems.*

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

Wang, X., M. Hersche, B. Tömekce, B. Kaya, M. Magno, and L. Benini. (2020). “An Accurate EEGNet-based Motor-Imagery Brain-Computer Interface for Low-Power Edge Computing”. In: *IEEE International Symposium on Medical Measurements and Applications, (MeMeA).* IEEE. 1–6.

Wang, Z., C. Huang, and Q. Zhu. (2022a). “Efficient global robustness certification of neural networks via interleaving twin-network encoding”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 1087–1092.

Wang, Z., A. Albarghouthi, G. Prakriya, and S. Jha. (2022b). “Interval universal approximation for neural networks”. *Proc. ACM Program. Lang.* 6(POPL): 1–29. DOI: 10.1145/3498675.

Webb, S., T. Rainforth, Y. W. Teh, and M. P. Kumar. (2019). “A Statistical Approach to Assessing Neural Network Robustness”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.* OpenReview.net. URL: https://openreview.net/forum?id=S1xcx3C5FX.

Wei, T., Z. Jia, C. Liu, and C. Tan. (2023). “Building Verified Neural Networks for Computer Systems with Ouroboros”. In: *Sixth Conference on Machine Learning and Systems.* Sixth Conference on Machine Learning and Systems.

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

Wengert, R. E. (1964). "A simple automatic derivative evaluation program". *Communications of the ACM*. 7(8): 463–464.

Wicker, M., L. Laurenti, A. Patane, and M. Kwiatkowska. (2020). "Probabilistic Safety for Bayesian Neural Networks". In: *Proceedings of the Thirty-Sixth Conference on Uncertainty in Artificial Intelligence, UAI 2020, virtual online, August 3-6, 2020*. Ed. by R. P. Adams and V. Gogate. Vol. 124. *Proceedings of Machine Learning Research*. AUAI Press. 1198–1207. URL: http://proceedings.mlr.press/v124/wicker20a.html.

Wicker, M., A. Patane, L. Laurenti, and M. Kwiatkowska. (2023). "Adversarial Robustness Certification for Bayesian Neural Networks". *CoRR*. abs/2306.13614. DOI: 10.48550/ARXIV.2306.13614. arXiv: 2306.13614.

Wong, E. and J. Z. Kolter. (2018). "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope". In: *Proc. International Conference on Machine Learning, ICML*. Vol. 80. *Proceedings of Machine Learning Research*. PMLR. 5283–5292.

Wong, E., S. Santurkar, and A. Madry. (2021). "Leveraging Sparse Linear Layers for Debuggable Deep Networks". In: *Proceedings of the 38th International Conference on Machine Learning, ICML*. Vol. 139. *Proceedings of Machine Learning Research*. PMLR. 11205–11216.

Wong, E., F. R. Schmidt, J. H. Metzen, and J. Z. Kolter. (2018). "Scaling provable adversarial defenses". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 8410–8419. URL: https://proceedings.neurips.cc/paper/2018/hash/358f9e7be09177c17d0d17ff73584307-Abstract.html.

Wu, C., R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. A. Behram, J. Huang, C. Bai, M. Gschwind, A. Gupta, M. Ott, A. Melnikov, S. Candido, D. Brooks, G. Chauhan, B. Lee, H. S. Lee, B. Akyildiz, M. Balandat, J. Spisak, R. Jain, M. Rabbat, and K. Hazelwood. (2022a). "Sustainable AI: Environmental Implications, Challenges and Opportunities". In: *MLSys*. mlsys.org.

Wu, H., C. Barrett, M. Sharif, N. Narodytska, and G. Singh. (2022b). "Scalable Verification of GNN-Based Job Schedulers". *Proc. ACM Program. Lang.* 6(OOPSLA2).

Wu, M., X. Li, H. Wu, and C. Barrett. (2024). "Better Verified Explanations with Applications to Incorrectness and Out-of-Distribution Detection". URL: https://arxiv.org/abs/2409.03060.

Wu, M., H. Wu, and C. Barrett. (2023). "VeriX: towards verified explainability of deep neural networks". *Advances in neural information processing systems*. 36: 22247–22268.

Xu, C., D. Banerjee, D. Vasisht, and G. Singh. (2024). "Support is All You Need for Certified VAE Training". In: *The Thirteenth International Conference on Learning Representations*.

Xu, C. and G. Singh. (2024). "Cross-Input Certified Training for Universal Perturbations". In: *Computer Vision - ECCV 2024 - 18th European Conference, Milan, Italy, September 29-October 4, 2024, Proceedings, Part LXV*. Ed. by A. Leonardis, E. Ricci, S. Roth, O. Russakovsky, T. Sattler, and G. Varol. Vol. 15123. *Lecture Notes in Computer Science*. Springer. 233–250. DOI: 10.1007/978-3-031-73650-6_14.

Xu, K., Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. (2020). "Automatic perturbation analysis for scalable certified robustness and beyond". In: *Proc. Neural Information Processing Systems (NeurIPS)*. 1129–1141.

Xu, K., H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh. (2021). "Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers". In: *International Conference on Learning Representations*.

Xue, X. and M. Sun. (2024). "Optimal Solution Guided Branching Strategy for Neural Network Branch and Bound Verification". In: *Engineering of Complex Computer Systems - 28th International Conference, ICECCS 2024, Limassol, Cyprus, June 19-21, 2024, Proceedings*. Vol. 14784. *Lecture Notes in Computer Science*. Springer. 67–87. DOI: 10.1007/978-3-031-66456-4_4.

Yang, C. and S. Chaudhuri. (2022). "Safe Neurosymbolic Learning with Differentiable Symbolic Execution". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. URL: https://openreview.net/forum?id=NYBmJN4MyZ.

Yang, C., D. Saxena, R. Dwivedula, K. Mahajan, S. Chaudhuri, and A. Akella. (2024a). "C3: Learning Congestion Controllers with Formal Certificates". *CoRR*. abs/2412.10915. DOI: 10.48550/ARXIV.2412.10915.

Yang, L., H. Dai, Z. Shi, C.-J. Hsieh, R. Tedrake, and H. Zhang. (2024b). "Lyapunov-stable neural control for state and output feedback: A novel formulation for efficient synthesis and verification". *arXiv preprint arXiv:2404.07956*.

Yang, P., R. Li, J. Li, C. Huang, J. Wang, J. Sun, B. Xue, and L. Zhang. (2021). "Improving Neural Network Verification through Spurious Region Guided Refinement". In: *Tools and Algorithms for the Construction and Analysis of Systems TACAS*. Vol. 12651. *Lecture Notes in Computer Science*. Springer. 389–408.

Yang, R., J. Laurel, S. Misailovic, and G. Singh. (2023). "Provable Defense Against Geometric Transformations". In: *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yang, Y. and M. C. Rinard. (2019). "Correctness Verification of Neural Networks". *CoRR*. abs/1906.01030. URL: http://arxiv.org/abs/1906.01030.

Yang, Z., K. Xu, B. Li, and H. Zhang. (2024c). "Improving Branching in Neural Network Verification with Bound Implication Graph". URL: https://openreview.net/forum?id=mMh4W72Hhe.

Yin, B., L. Chen, J. Liu, and J. Wang. (2022). "Efficient Complete Verification of Neural Networks via Layerwised Splitting and Refinement". *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41(11): 3898–3909. DOI: 10.1109/TCAD.2022.3197534.

Zelazny, T., H. Wu, C. Barrett, and G. Katz. (2022). "On Optimizing Back-Substitution Methods for Neural Network Verification". URL: https://arxiv.org/abs/2208.07669.

Zeng, Y., Z. Shi, M. Jin, F. Kang, L. Lyu, C.-J. Hsieh, and R. Jia. (2023). "Towards Robustness Certification Against Universal Perturbations". In: *The Eleventh International Conference on Learning Representations*. URL: https://openreview.net/forum?id=7GEvPKxjtt.

Zhang, B., T. Cai, Z. Lu, D. He, and L. Wang. (2021a). "Towards Certifying L-infinity Robustness using Neural Networks with L-inf-dist Neurons". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by M. Meila and T. Zhang. Vol. 139. *Proceedings of Machine Learning Research*. PMLR. 12368–12379. URL: http://proceedings.mlr.press/v139/zhang21b.html.

Zhang, H., H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh. (2020). "Towards stable and efficient training of verifiably robust neural networks". In: *Proc. International Conference on Learning Representations (ICLR)*.

Zhang, H., S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Z. Kolter. (2022). "General Cutting Planes for Bound-Propagation-Based Neural Network Verification". In: *Advances in Neural Information Processing Systems*. Ed. by A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho. URL: https://openreview.net/forum?id=5haAJAcofjc.

Zhang, H., T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel. (2018a). "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2018/file/d04863f100d59b3eb688a11f95b0ae60-Paper.pdf.

Zhang, H., P. Zhang, and C. Hsieh. (2019). “RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications”. In: *The 33rd AAAI Conference on Artificial Intelligence, (AAAI)*.

Zhang, X., A. Solar-Lezama, and R. Singh. (2018b). “Interpreting Neural Network Judgments via Minimal, Stable, and Symbolic Corrections”. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 4879–4890. URL: https://proceedings.neurips.cc/paper/2018/hash/300891a62162b960cf02ce3827bb363c-Abstract.html.

Zhang, X., B. Wang, M. Kwiatkowska, and H. Zhang. (2024). “PREMAP: A Unifying PREiMage APproximation Framework for Neural Networks”. URL: https://arxiv.org/abs/2408.09262.

Zhang, Y., A. Albarghouthi, and L. D’Antoni. (2021b). “Certified Robustness to Programmable Transformations in LSTMs”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Ed. by M. Moens, X. Huang, L. Specia, and S. W. Yih. Association for Computational Linguistics. 1068–1083.

Zhou, D., C. Brix, G. A. Hanasusanto, and H. Zhang. (2024). “Scalable Neural Network Verification with Branch-and-bound Inferred Cutting Planes”. In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. URL: https://openreview.net/forum?id=FwhM1Zpyft.

Zhou, Z., Z. Huang, and S. Misailovic. (2023). “Aquasense: Automated sensitivity analysis of probabilistic programs via quantized inference”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 288–301.