

MATTHEW MIRMAN

CERTIFIED DEEP LEARNING:
VERIFICATION AND TRAINING

DISS. ETH NO. 28391

CERTIFIED DEEP LEARNING:
VERIFICATION AND TRAINING

A dissertation submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

MATTHEW MIRMAN
Dipl., Eidgenössisches Polytechnikum
born on 6 May 1990
citizen of United States of America

accepted on the recommendation of

Prof. Dr. Martin Vechev, examiner
Prof. Dr. Aws Albarghouthi, co-examiner
Prof. Dr. Matthias Hein, co-examiner
Prof. Dr. Zico Kolter, co-examiner
Prof. Dr. Ce Zhang, co-examiner

2022

Matthew Mirman: *Certified Deep Learning: Verification and Training*, © 2022

DOI: 10.3929/ethz-a-

Dedicated to my mother, Michele,
and in memory of my father, Paul.

ABSTRACT

Deep learning models, and in particular neural networks, have become ubiquitous in modern computing. Neural networks can be found in spam and malware detection systems [1], bug fixing tools [2–4], at numerous parts of the stack in self driving cars [5, 6], in aircraft collision avoidance systems [7], facial recognition for authentication [8], health-care [9, 10], in algorithmic trading [11, 12], and even at the base of stacks that power generic upstream code tools such as compilers and SMT solvers [13–15].

As many of these applications are safety-critical, or more generally have no tolerance for error, it is essential that programmers be able to analyze neural networks to ensure their reliability. While historically, neural networks were assumed to be correct and well-behaved if they achieved a low statistical error on a test-set, Szegedy *et al.* [16] demonstrated that this was insufficient: neural networks could be fooled by adversaries presenting intentionally crafted unusual inputs. Such attacks could be made imperceptible to humans [17–27], and could be as easy to deploy as applying a sticker to a road sign [28–30].

The main problem we address in this dissertation is: how can sound, formal guarantees on neural networks be scalably verified. To answer this question, our work is based on a fundamental insight: a verifier needs only to be correct when it determines a property can be *guaranteed*, but it does not always need to make such a determination. Using this insight, we demonstrate the first application of abstract interpretation to neural networks, and show that it can additionally be used for training.

In this dissertation, we first present two extensible frameworks: AI² for sound verification, and DIFFAI which additionally implements provable training. We develop new domains which allow our systems to scale to orders of magnitude larger networks than prior verification systems, and allow us to demonstrate correctness of significantly more properties on those networks. We finally extend DIFFAI with the domain GENPROVE to prove sound, formal probabilistic guarantees and show how this can be used to verify semantic specifications using generative networks.

ZUSAMMENFASSUNG

Deep-Learning-Modelle und insbesondere neuronale Netze sind im modernen Computing allgegenwärtig geworden. Neuronale Netze finden sich in Spam- und Malware-Erkennungssystemen [1], Bugfixing-Tools [2–4], an zahlreichen Stellen der Stacks in selbstfahrenden Autos [5, 6], in Kollisionsvermeidungssystemen in Flugzeugen [7], bei der Gesichtserkennung zur Authentifizierung [8], in der Gesundheitsversorgung [9, 10], im algorithmischen Handel [11, 12] und sogar an der Basis von Stacks welche generische Upstream-Code-Tools wie Compiler und SMT-Solver betreiben [13–15].

Da viele dieser Anwendungen sicherheitskritisch sind, beziehungsweise eine geringe Fehlertoleranz besitzen, ist es von Bedeutung, dass Programmierer dazu in der Lage sind, neuronale Netze zu analysieren, um ihre Zuverlässigkeit sicherzustellen. Während in der Vergangenheit davon ausgegangen wurde, dass neuronale Netze korrekt und vorhersehbar sind, wenn sie einen niedrigen statistischen Fehler in einem Testsatz besitzen, konnte Szegedy *u. a.* [16] zeigen, dass dies nicht ausreicht: Neuronale Netze konnten von Gegnern getäuscht werden, die absichtlich künstliche ungewöhnliche Eingaben präsentierten. Solche Angriffe könnten für Menschen unbemerkbar gemacht werden [17–27] und sind so einfach auszuführen wie das Anbringen eines Aufklebers auf ein Verkehrsschild [28–30].

Das Hauptproblem, das in dieser Dissertation behandelt wird, ist: Wie kann die solide, formale Garantie auf neuronalen Netzen skalierbar verifiziert werden? Um diese Frage zu beantworten, basiert unsere Arbeit auf einer grundlegenden Erkenntnis: Ein Verifizierer muss nur dann richtig liegen, wenn er feststellt, dass eine Eigenschaft *gewährleistet* werden kann. Der Verifizierer muss jedoch nicht immer eine solche Bestimmung durchführen. Mit dieser Erkenntnis demonstrieren wir die erste Anwendung der abstrakten Interpretation von neuronalen Netzen und zeigen, dass sie zusätzlich für das Training verwendet werden kann.

In dieser Dissertation stellen wir zunächst zwei erweiterbare Rahmenordnungen vor: AI^2 für die fundierte Verifikation und DIFFAI, das zusätzlich nachweisbares Training implementiert. Wir entwickeln neue Domänen, die es unseren Systemen ermöglichen, auf die Magnitude grössere Netzwerke zu skalieren als frühere Verifizierungssysteme. Des

Weiteren erlaubt uns dies die Korrektheit von deutlich mehr Eigenschaften in diesen Netzwerken zu demonstrieren. Schließlich erweitern wir DIFFAI um die Domäne GENPROVE, um solide, formale probabilistische Garantien zu beweisen und zu zeigen, wie dies zur Verifizierung semantischer Spezifikationen für generative Netzwerke verwendet werden kann.

PUBLICATIONS

This thesis is based on the following published conference proceedings:

1. **Mirman, M.**, Gehr, T. & Vechev, M. *Differentiable abstract interpretation for provably robust neural networks* in *International Conference on Machine Learning (ICML 2018)*.
2. Gehr, T., **Mirman, M.**, Drachslor-Cohen, D., Tsankov, P., Chaudhuri, S. & Vechev, M. *AI2: Safety and robustness certification of neural networks with abstract interpretation* in *IEEE Symposium on Security and Privacy (SP 2018)*.
3. **Mirman, M.**, Hägele, A., Bielik, P., Gehr, T. & Vechev, M. *Robustness certification with generative models* in *International Conference on Programming Language Design and Implementation (PLDI 2021)*.

The following works were published as part of my PhD and are supplemental to this work, but are not included in this thesis.

1. **Mirman, M.**, Baader, M. & Vechev, M. *The Fundamental Limits of Interval Arithmetic for Neural Networks* in *In Submission (2021)*.
2. Baader, M., **Mirman, M.** & Vechev, M. *Universal Approximation with Certified Networks* in *International Conference on Learning Representations (ICLR 2020)*.
3. **Mirman, M.**, Singh, G. & Vechev, M. *A provable defense for deep residual networks*. *arXiv preprint arXiv:1903.12519 (2019)*.
4. Singh, G., Gehr, T., **Mirman, M.**, Püschel, M. & Vechev, M. *Fast and effective robustness certification* in *International Conference on Neural Information Processing Systems (NeurIPS 2018)*.

ACKNOWLEDGEMENTS

I'd like to use this space to thank everybody who has provided me support during my studies. Firstly, I am eternally grateful to my advisor, Martin Vechev, who taught me the methods and value of thorough detailed research. I am grateful for the time and encouragement of the members of my thesis committee: Prof. Aws Albarghouthi, Prof. Dr. Matthias Hein, Prof. Zico Kolter, and Prof. Dr. Ce Zhang. I have many coauthors in need of thanking: Timon Gehr, Maximilian Baader, Marc Fischer, Petar Tsankov, Swarat Chaudhuri, Gagandeep Singh, Dimitar K. Dimitrov, Dana Drachler-Cohen, and Pavol Bielik.

Indirectly, my studies would not have been possible without the support and encouragement of my friends and family. Firstly, I owe much gratitude to my mother, Michele Mirman, who always encouraged me to pursue science. I'd like to thank Jahlela Hasle, for her patience waiting for me to leave my office. I can not imagine what my thesis would have been without Justine Räber's support and excellent proofreading. Lastly, I would like to thank my sister, Phoebe Mirman, for foregoing a nearby brother.

CONTENTS

1	INTRODUCTION	1
1.1	In Search of Formal Guarantees for Deep Learning	3
1.1.1	Problem Dimensions	3
1.1.2	This Dissertation's Approach	4
1.1.3	Challenges Addressed by This Dissertation	5
1.2	This Dissertation: Sound Guarantees for Deep Learning	6
1.2.1	Systems Overview	7
1.2.2	Impact and Results Beyond The Scope of this Dissertation	7
2	ABSTRACT INTERPRETATION FOR NEURAL NETWORK VERIFICATION	9
2.1	Introduction	9
2.2	Representing Neural Networks for Certification	13
2.3	Background: Abstract Interpretation	19
2.4	AI ² : AI for Neural Networks	23
2.4.1	Abstract Interpretation for CAT Functions	23
2.4.2	CAT Representations for Common Layer Types	27
2.4.3	Neural Network Analysis with AI	29
2.5	Implementation of AI ²	30
2.6	Evaluation of AI ²	31
2.6.1	Experimental Setup	32
2.6.2	Discussion of Results	34
2.7	Comparing Defenses with AI ²	39
2.8	Related Work	40
2.9	Conclusion and Future Work	42
3	ABSTRACT INTERPRETATION FOR TRAINING CERTIFIABLE NETWORKS	43
3.1	Introduction	43
3.2	Background: Verification for Neural Networks	44
3.2.1	Robustness and Sound Approximations	44
3.2.2	Abstract Interpretation	46
3.3	Abstract Transformers for Zonotope	49

3.4	Adversarial Training	53
3.4.1	Approximate Worst-Case Adversarial Loss	53
3.4.2	Line Segments as (Hybrid) Zonotopes	54
3.5	Experimental Setup	55
3.5.1	Training	55
3.5.2	Neural Networks Evaluated	56
3.6	Experimental Results	59
3.6.1	Results against Prior Defenses and Analyzers	59
3.6.2	Segment Training for Higher Accuracy	63
3.7	Outlook	63
3.8	Conclusion	64
4	PROBABILISTIC ABSTRACT INTERPRETATION FOR GENERATIVE MODELS	65
4.1	Introduction	65
4.2	Overview of GENPROVE	68
4.3	Certification of Deterministic Properties	71
4.3.1	GENPROVE for Deterministic Properties	73
4.4	Certification of Probabilistic Properties	74
4.4.1	GENPROVE for Probabilistic Properties	77
4.4.2	Propagation Pseudocode and Example	78
4.4.3	Generalization to Parametric Curves	81
4.5	Evaluation	82
4.5.1	RQ1 - Probabilistic Abstract Interpretation	89
4.5.2	RQ2 - Precision and Scalability	90
4.5.3	RQ3 - Novel Generative Specifications	95
4.6	Related Work	99
4.7	Discussion	100
4.8	Conclusion	100
5	CONCLUSION AND FUTURE WORK	103
5.1	Future Work	103
5.1.1	Verification Outlook	104
5.1.2	Training Outlook	104
5.1.3	Further Application of Our Techniques	106
5.1.4	Utilizing Sound Guarantees	106
A	APPENDIX	107
A.1	Extended DIFFAI Results	107

BIBLIOGRAPHY 115

INTRODUCTION

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [5], malware detection [1], medical diagnosis [10], and aircraft collision avoidance [7]. This adoption can be attributed to the near, and sometimes even super-human accuracy obtained by these models [31]. Despite their success, an essential challenge remains:

*How can one ensure that machine learning models,
and neural networks in particular, behave as intended?*

(Essential Challenge)

In the *traditional software* setting, software engineers have typically approached addressing this challenge two ways: (i) handcrafting proofs of correctness for their code and algorithms, and (ii) utilizing (semi)-automated verifiers and solvers which may not halt, but when they do, always produce entirely correct statements (i.e. are over-approximators of program behavior). Crucially, both of these methods have one thing in common: the programmer knows in advance what properties they would like their code to have.

In the *deep learning* setting, it may be counterintuitive that one would ever want to predetermine such absolute and deterministic properties. After all, neural networks are fundamentally statistical models, most often trained using noisy datasets collected from the physical world (e.g., pictures of handwritten digits). It has been commonly assumed that if a neural network performs well on a randomly chosen subset of the dataset not used for training, the test or validation-set, it will perform well when used in production. This kind of statistical guarantee is sufficient when the stakes are low (e.g., automated story generation [32]), or when guarantees can be made about the production environment (e.g., running in a sandbox on air-gapped hardware in a lab). However, recent research has shown that even highly accurate networks are vulnerable to *adversarial examples*, demonstrating that the typical statistical guarantee of correctness is inadequate.

Classically, Szegedy *et al.* [16] showed that by perturbing pixels a small amount (essentially adding invisible noise), one could cause a network to misclassify an originally correctly classified image. Such adversarial attacks typically generate or modify images in ways that are imperceptible to humans, but induce unexpected behavior by the network [17–27, 33–35].

Adversarial examples can be especially problematic when safety-critical systems rely on neural networks. For instance, it has been shown that attacks can be executed physically (e.g., [28–30]), and against neural networks accessible only as a black box (e.g., [16, 20, 36–40]).

Gu & Rigazio [23] provided an early technique for defending against adversarial examples based on adding concrete noise to the training set and removing it statistically. Goodfellow, Shlens & Szegedy [20] showed an adversarial attack that generated examples that were misclassified on a wide array of networks and then demonstrated a defense against this attack, based on explicit training against perturbations generated by the attack. Madry *et al.* [41] improved on this style of defense by showing that training against an optimal attack would provide a defense against non-optimal attacks as well. While this technique was highly effective in experiments, Carlini *et al.* [42] demonstrated an attack for the safety-critical problem of ground-truthing, where this defense in fact occasionally exacerbated the problem.

Many further works have focused on designing *defenses* that increase robustness by using modified procedures for training the network (e.g., [43–46]). Others have developed approaches that can show non-robustness by under-approximating neural network behaviors [47, 48].

Despite making substantial progress at mitigating the impact of adversarial examples, the aforementioned techniques do not address the agency involved in the generation of adversarial examples: without ensuring guarantees with 100% confidence, a neural network might still be vulnerable to some kind of attack. Without being able to verify such guarantees, ensuring the safety of systems involving neural networks becomes a game of cat-and-mouse.

Widening the focus from attacks, in this dissertation, we assert that the techniques used to analyze and evaluate neural networks need not be restricted to the statistical. We show that despite their scale, valuable and strong guarantees can be made with neural networks. In this dissertation we answer the following fundamental research question:

How can sound formal guarantees be made available to deep learning?

(Fundamental Question)

	Our Work (This Thesis)			
	Reluplex [7] (SMT)	AI ² (Abstract Interpretation)	GENPROVE (Probabilistic Abst. Int.)	DIFFAI (Differentiable Abst. Int.)
Networks	FFNN	FFNN, Conv	FFNN, Conv, Generative	FFNN, Conv, Residual
Activations	ReLU	ReLU, MaxPool	ReLU	ReLU, Softmax, Sigmoid
Properties	Robustness	Robustness	Generative-Semantic	Robustness
Guarantees	Complete	Sound	Complete or Sound	Sound
Runtime ¹	Hours	Minutes	Seconds	Milliseconds
Largest Net	2k Neurons	53k Neurons	200k Neurons	4.5m Neurons ²
Scalable	✗	✓	✓	✓
Training	✗	✗	✗	✓
Deterministic Properties	✓	✓	✓	✓
Probabilistic Properties	✗	✗	✓	✗

¹Runtime is the time to verify a specification on a small convolutional network or, for Reluplex, the largest FFNN tried.

²Result obtained using the DIFFAI verifier with a network trained in our supplemental work: Mirman, Singh & Vechev [49].

TABLE 1.1: Dimensions addressed by our methods compared with prior work.

1.1 IN SEARCH OF FORMAL GUARANTEES FOR DEEP LEARNING

Here we describe the overarching direction of this thesis, the dimensions of the problem we tackle, our high-level approach to addressing these problems, and the challenges our approaches must overcome. In this dissertation, we advance the state-of-the-art in neural network analysis by expanding the following capabilities:

- The nature of *guarantees* that can be scalably verified.
- The types of *properties* that can be scalably verified.
- The *networks* which satisfy, verifiably, these guarantees.

1.1.1 Problem Dimensions

In order for deep learning to be trusted for use in safety-critical systems, it is essential that the kinds of analyses one can perform eventually matches the depth of analyses engineers have come to rely on for traditional code. In particular, this means expanding the guarantees, properties, and model building techniques (outlined in Table 1.1) as follows:

Types of Guarantees. While statistical guarantees have been the default for neural networks, engineers require *sound*, meaning that there is no

chance of the claim being incorrect, guarantees. Sound guarantees can either be deterministic (e.g., “The plane *never* crashes”) or probabilistic (e.g., “The plane crashes *at most* 5% of flights”). Crucially, however, sound guarantees can not be made by statistical sampling, which can at best make claims with high confidence: “I’m 99% confident the plane crashes at most 5% of flights, but there is a 1% chance I’m entirely wrong.” While prior methods have demonstrated lower bounds (e.g., “The plane can crash”), the prior techniques for finding upper bounds only scale to neural networks orders of magnitude smaller than needed for common tasks.

Kinds of Properties. Traditional neural network guarantees are based on sampling a partition of the dataset off-limits to training. Adversarial example research has shown that this is insufficient. Images off of the originally sampled data-manifold can look like those on the data-manifold, and appear in production. This implies that the types of specifications used must be expanded. The first adversarial attacks focused on finding incorrectly classified images in L_p Balls around correctly classified images such that they were imperceptibly different. In this thesis, we observe that the properties verified need not be limited to the imperceptible, but that it may be useful to analyze the compliance to semantic visible properties, such as those produced by generative models.

Network Suitability. It is well known that some programs are significantly easier to prove correct than others, even ones performing the same task. Automated verification suffers similarly from this challenge. We observe that neural networks are no different. While it may be possible to analyze a neural network with a complete system (such as an SMT solver), this is NP-hard in general [7]. We posit that one network may be more suitable than another for verification, even when both may obey the given specification. We thus propose, analogously to quantitative synthesis [50], training neural networks which are *amenable* to verification.

1.1.2 This Dissertation’s Approach

While prior methods were able to provide lower-bound guarantees (by finding attacks), we develop methods that guarantee both sound lower and sound upper-bounds on deterministic and probabilistic properties for large networks. Our analysis methods can be used to ensure that networks are not vulnerable to traditional adversarial attacks, and furthermore to augment statistical claims on semantic accuracy.

When traditional defenses produce networks that are experimentally robust to attack, but too large to be verified by complete systems, we demonstrate how to defend the network so that it can be verified with overapproximation. We address our research goals in the following ways:

- We demonstrate how to soundly *analyze* preexisting networks.
- We demonstrate how to *train* networks amenable to analysis.

1.1.3 Challenges Addressed by This Dissertation

To build scalable neural network verification tools, our systems must overcome the following challenges:

Modern neural networks are massive. Neural networks commonly have millions of neurons, each of which is a non-convexity, that when translated to a SMT formula is represented by a disjunction. Complete verification systems handle disjunctions by enumerating possible branches, and therefore could, in the worst case, run in exponential time. This is unacceptable for neural networks: Huang *et al.* [51], for example, introduced a network with 4.5 million neurons. While such large codebases do exist, they can often be analyzed in more manageable fragments.

Data is high dimensional. Similarly, verification systems have typically been designed to handle low-dimensional inputs. Even in non deep-learning robotic systems, the number of input dimensions is often limited by the number of physical (non-image) sensors. Neural networks on the other hand are commonly used on images. Our smallest dataset, MNIST [52] for example, uses 784-dimensional images. On the other hand, the well-known dataset ImageNet [53] is commonly scaled *down* to 128×128 pixels of three colors, representing nearly 50k input dimensions. While the dimensionality of the inputs represents a similar theoretical challenge as size of the analyzed model, the dimensionality of the input is indicative of *necessary* non-convexity.

Neural networks are imprecise. Fundamentally, neural networks are usually trained on noisy datasets, produced by collection from the real-world. Unlike in traditional software systems, there is often no immediate expectation of “perfection.” It is thus not immediately obvious what kinds of guarantees or specifications one might be able to make. For neural networks, verification systems should be able to make overapproximate claims, and ideally provide probabilistic guarantees.

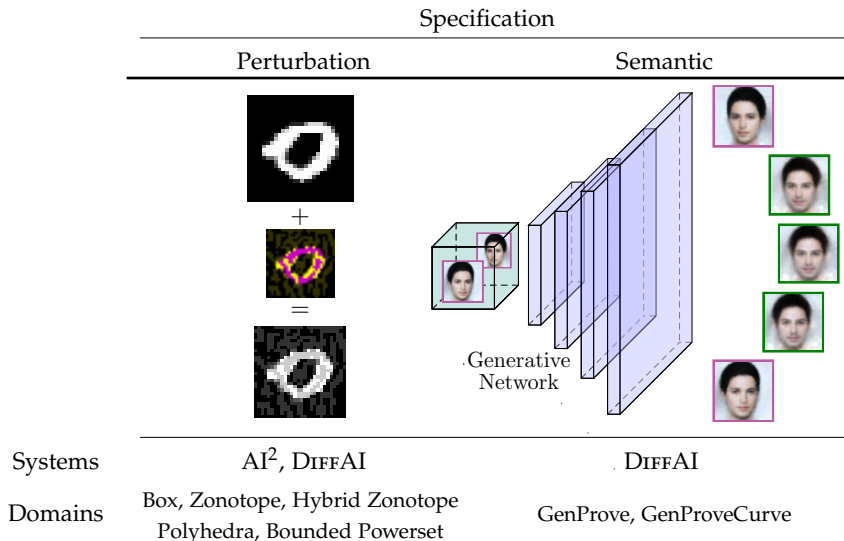


FIGURE 1.1: Outline of the specifications we analyze, and the systems and domains that implement them.

1.2 THIS DISSERTATION: SOUND GUARANTEES FOR DEEP LEARNING

In this thesis, we tackle the problem of introducing sound guarantees into deep learning by designing two systems, AI^2 and DIFFAI each representing a significant jump in capability, bringing neural network verification possibilities from toy-scale to production-scale. The specifications and domains supported by these systems are shown in Fig. 1.1.

Main Achievements. In this dissertation we present the following main technical achievements:

- We designed extensible frameworks, AI^2 and DIFFAI, for verifying properties of neural networks multiple orders of magnitude larger than preexisting systems: up to 200k neurons from 1200 neurons.
- We produced networks with significantly lower verified error than could be previously produced: down to 11.8% from 100% (on normalized MNIST)¹.
- We scaled sound probabilistic verification on generative models so that it could beat (in terms of precision, guarantees, and efficiency) statistical sampling.

¹ Later training schedule refinements reduced this to 4.2% in an unnormalized MNIST. [49]

1.2.1 *Systems Overview*

This dissertation presents three systems, each of which fundamentally extends the capabilities of neural network analysis:

AI² [54], introduced in Chapter 2, is the first scalable system for verifying neural networks. In this chapter, we demonstrate how to leverage the flexible techniques, known as abstract interpretation, developed in the formal methods community to address the increasingly pressing problem of neural network analysis. Specifically, our framework enables the analysis of neural networks through overapproximation by geometric domains which can then be fine-tuned for performance.

DIFFAI [55], introduced in Chapter 3, is the first scalable system for training neural networks to be verifiable. The key insight of DIFFAI is that it is possible to reuse abstract interpretation verification techniques, such as those developed in Chapter 2, as the optimization goal for backpropagation based training.

GENPROVE [56], introduced in Chapter 4, provides a scalable way to verify more complex and semantically meaningful properties based on generative models. We observe that such complex properties are more unlikely to completely hold, and thus extend the abstract interpretation framework to handle probabilistic properties. This allows certification of not just whether a property holds, but also of measuring *how much* it holds.

1.2.2 *Impact and Results Beyond The Scope of this Dissertation*

Since releasing DIFFAI, it has garnered 186 stars on github, and has 23 forks. It has become a standard for producing benchmark networks for comparing verifiers [57] and has also been used industrially [58].

We have also significantly improved DIFFAI’s usability by providing a training DSL which allows easy expression of training schedules and loss functions. We provided an updated evaluation using modern schedules and loss functions [59] in Mirman, Singh & Vechev [49], demonstrating that our core technique scales to networks with up to 4.5 million neurons. Zhang, Albarghouthi & D’Antoni [60] furthermore demonstrated that abstract training could be used to improve training results of NLP tasks.

Finally, DIFFAI has inspired theoretical results: In Baader, Mirman & Vechev [61] we proved that one could construct interval certifiable neural networks to approximate any function with any given interval size. While

it is reassuring to know that the search for certifiable neural networks is not entirely futile, these results do not address the difficulty of improving on the results shown in this thesis. In Mirman, Baader & Vechev [62] we demonstrate that there are fundamental limitations preventing neural networks from being constructed that are certifiable with interval. Namely, we demonstrate that no neural network is completely certifiably robust with the interval domain for even a dataset with only three points. We furthermore show that for any minimum distance between robust regions, there is a dataset that no single hidden layer network can be proven to be robust on with interval analysis.

ABSTRACT INTERPRETATION FOR NEURAL NETWORK VERIFICATION

We begin by presenting AI^2 , the first sound and scalable analyzer for deep neural networks. We first note that the material in this chapter is the result of joint work [54] with Timon Gehr and also appears in his dissertation.

The key insight behind AI^2 is to phrase reasoning about safety and robustness of neural networks in terms of classic abstract interpretation, enabling us to leverage decades of advances in that area. Specifically, AI^2 uses overapproximation to automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks). In this chapter we introduce an interpretation of neural networks into abstract transformers that capture the behavior of fully connected and convolutional network layers with rectified linear unit activations (ReLU), as well as max-pooling layers. This allows us to handle real-world neural networks, which are often built using only these layers.

We present a complete implementation of AI^2 , together with an extensive evaluation on 20 neural networks. Our results demonstrate that: (i) AI^2 is precise enough to prove useful specifications (e.g., robustness), (ii) AI^2 can be used to certify the effectiveness of state-of-the-art defenses for neural networks, (iii) AI^2 is significantly faster than preexisting analyzers that are based on symbolic analysis, which often take hours to verify simple, fully-connected networks, and (iv) AI^2 can handle deep convolutional networks, which are beyond the reach of existing methods.

2.1 INTRODUCTION

Adversarial examples present a pressing concern for safety-critical systems that rely on neural networks. In order to tackle this problem, we will proceed initially by focusing our attention on verifying safety against the first generation of adversarial examples. These are typically obtained by slightly perturbing an input that is correctly classified by the network, such that the network misclassifies the perturbed input. Various kinds of perturbations have been shown to successfully generate adversarial examples (e.g., [17–27]).







Attack	Original	Perturbed	Diff
FGSM [20], $\epsilon = 0.3$			
Brightening, $\delta = 0.085$			

FIGURE 2.1: Attacks applied to MNIST images [52].

Fig. 2.1 illustrates two attacks, FGSM and brightening, against a digit classifier. For each attack, Fig. 2.1 shows an input in the “Original” column, the perturbed input in the “Perturbed” column, and the pixels that were changed in the “Diff” column. Brightened pixels are marked in yellow and darkened pixels are marked in purple. The FGSM [20] attack perturbs an image by adding to it a particular noise vector multiplied by a small number ϵ (in Fig. 2.1, $\epsilon = 0.3$). The brightening attack (e.g., [26]) perturbs an image by changing all pixels above the threshold $1 - \delta$ to the brightest possible value (in Fig. 2.1, $\delta = 0.085$).

To mitigate these issues, recent research has focused on reasoning about neural network robustness, and in particular on *local robustness*. Local robustness (or robustness, for short) requires that all samples in the neighborhood of a given input are classified with the same label [43]. Prior works have introduced methods that decide robustness of small fully connected feed-forward networks [7]. However, no prior sound analyzer handles convolutional networks, one of the most popular architectures.

Key Challenge: Scalability and Precision. The main challenge facing sound analysis of neural networks is scaling to large classifiers while maintaining a precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. For instance, consider the image of the digit 8 in Fig. 2.1 and suppose we would like to prove that no matter how we brighten the value of pixels with intensity above $1 - 0.085$, the network will still classify the image as 8 (in this example we have 84 such pixels, shown in yellow). Assuming 64-bit floating point numbers are used to express pixel intensity, we obtain more than 10^{1154} possible perturbed images. Thus, proving the property by running a network exhaustively on all possible input images and checking if all of them are classified as 8 is

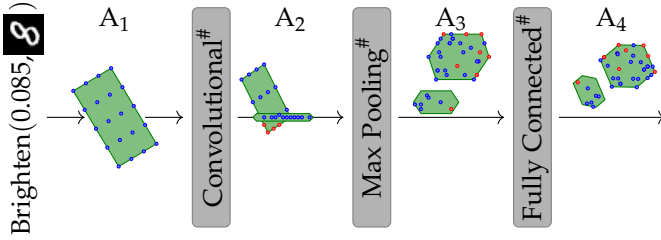


FIGURE 2.2: A high-level illustration of how AI^2 checks that all perturbed inputs are classified the same way. AI^2 first creates an abstract element A_1 capturing all perturbed images (Here, we use a 2-bounded set of zonotopes). It then propagates A_1 through the abstract transformer of each layer, obtaining new shapes. Finally, it verifies that all points in A_4 correspond to outputs with the same classification.

infeasible. To avoid this state space explosion, current methods (e.g., [7, 27, 63]) symbolically encode the network as a logical formula and then check robustness properties with a constraint solver. However, such solutions do not scale to larger (e.g., convolutional) networks, which usually involve many intermediate computations.

Key Concept: Abstract Interpretation for AI. The key insight of our work is to address the above challenge by leveraging the classic framework of abstract interpretation (e.g., [64, 65]), a theory that dictates how to obtain sound, computable, and precise finite approximations of potentially infinite sets of behaviors. Concretely, we leverage numerical abstract domains – a particularly good match, as AI systems tend to heavily manipulate numerical quantities. By showing how to apply abstract interpretation to reason about AI safety, we enable one to leverage decades of research and any future advancements in that area (e.g., in numerical domains [66]). With abstract interpretation, a neural network computation is overapproximated using an *abstract domain*. An abstract domain consists of logical formulas that capture certain shapes (e.g., zonotopes, a restricted form of polyhedra). For example, in Fig. 2.2, the green zonotope A_1 overapproximates the set of blue points (each point represents an image). Of course, sometimes, due to abstraction, a shape may also contain points that will not occur in any concrete execution (e.g., the red points in A_2).

This work: The AI² Analyzer. Based on this insight, we developed a system called AI² (*Abstract Interpretation for Artificial Intelligence*)¹. AI² is the first scalable analyzer that handles common network layer types, including fully connected and convolutional layers with rectified linear unit activations (ReLU) and max-pooling layers.

To illustrate the operation of AI², consider the example in Fig. 2.2, where we have a neural network, an image of the digit 8 and a set of perturbations: brightening with parameter 0.085. Our goal is to prove that the neural network classifies all perturbed images as 8. AI² takes the image of the digit 8 and the perturbation type and creates an abstract element A_1 that captures all perturbed images. In particular, we can capture the entire set of brightening perturbations exactly with a single zonotope. However, in general, this step may result in an abstract element that contains additional inputs (that is, red points). In the second step, A_1 is automatically propagated through the layers of the network. Since layers work on concrete values and not abstract elements, this propagation requires us to define *abstract layers* (marked with #) that compute the effects of the layers on abstract elements. The abstract layers are commonly called the *abstract transformers* of the layers. Defining sound and precise, yet scalable abstract transformers is key to the success of an analysis based on abstract interpretation. We define abstract transformers for all three layer types shown in Fig. 2.2.

At the end of the analysis, the abstract output A_4 is an overapproximation of *all* possible concrete outputs. This enables AI² to verify safety properties such as robustness (e.g., are all images classified as 8?) directly on A_4 . In fact, with a single abstract run, AI² was able to prove that a convolutional neural network classifies all of the considered perturbed images as 8.

We evaluated AI² on important tasks such as verifying robustness and comparing neural networks defenses. For example, for the perturbed image of the digit 0 in Fig. 2.1, we showed that while a non-defended neural network classified the FGSM perturbation with $\epsilon = 0.3$ as 9, this attack is *provably* eliminated when using a neural network trained with Madry *et al.* [41]’s defense. In fact, AI² proved that no attack can generate adversarial examples from this image for any ϵ between 0 and 0.3.

Main Contributions. Our main contributions are:

¹ AI² is available at: <http://ai2.ethz.ch>

- A sound and scalable method for analysis of deep neural networks based on abstract interpretation (Section 2.4).
- AI², an end-to-end analyzer, extensively evaluated on feed-forward and convolutional networks (computing with 53 000 neurons), far exceeding capabilities of current systems (Section 2.6).
- An application of AI² to evaluate *provable robustness* of neural network defenses (Section 2.7).

2.2 REPRESENTING NEURAL NETWORKS FOR CERTIFICATION

In this section, we provide background on feedforward and convolutional neural networks and show how to transform them into a representation amenable to abstract interpretation. This representation helps us simplify the construction and description of our analyzer, which we discuss in later sections. We use the following notation: for a vector $\bar{x} \in \mathbb{R}^n$, x_i denotes its i^{th} entry, and for a matrix $W \in \mathbb{R}^{n \times m}$, W_i denotes its i^{th} row and $W_{i,j}$ denotes the entry in its i^{th} row and j^{th} column.

CAT Functions. We express the neural network as a composition of *conditional affine transformations* (CAT), which are affine transformations guarded by logical constraints. The class of CAT functions, shown in Fig. 2.3, consists of functions $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ for $m, n \in \mathbb{N}$ and is defined recursively. Any affine transformation $f(\bar{x}) = W \cdot \bar{x} + \bar{b}$ is a CAT function, for a matrix W and a vector \bar{b} . Given sequences of conditions E_1, \dots, E_k and CAT functions f_1, \dots, f_k , we write:

$$f(\bar{x}) = \mathbf{case} E_1: f_1(\bar{x}), \dots, \mathbf{case} E_k: f_k(\bar{x}).$$

This is also a CAT function, which returns $f_i(\bar{x})$ for the first E_i satisfied by \bar{x} . The conditions are conjunctions of constraints of the form $x_i \geq x_j$, $x_i \geq 0$ and $x_i < 0$. Finally, any composition of CAT functions is a CAT function. We often write $f'' \circ f'$ to denote the CAT function $f(\bar{x}) = f''(f'(\bar{x}))$.

Layers. Neural networks are often organized as a sequence of layers, such that the output of one layer is the input of the next layer. Layers consist of *neurons*, performing the same function but with different parameters. The output of a layer is formed by stacking the outputs of the neurons into a vector or three-dimensional array. We will define the functionality in terms of entire layers instead of in terms of individual neurons.

$$\begin{aligned}
 f(\bar{x}) & ::= W \cdot \bar{x} + \bar{b} \\
 & \quad | \quad \mathbf{case} \ E_1: f_1(\bar{x}), \dots, \mathbf{case} \ E_k: f_k(\bar{x}) \\
 & \quad | \quad f(f'(\bar{x})) \\
 E & ::= E \wedge E \mid x_i \geq x_j \mid x_i \geq 0 \mid x_i < 0
 \end{aligned}$$

FIGURE 2.3: Definition of CAT functions.

Reshaping of Inputs. Layers often take three-dimensional inputs (e.g., colored images). Such inputs are transformed into vectors by reshaping. A three-dimensional array $\bar{x} \in \mathbb{R}^{m \times n \times r}$ can be reshaped to $\bar{x}^v \in \mathbb{R}^{m \cdot n \cdot r}$ in a canonical way, first by depth, then by column, finally by row. That is, given \bar{x} :

$$\bar{x}^v = (x_{1,1,1} \dots x_{1,1,r} \ x_{1,2,1} \dots x_{1,2,r} \dots x_{m,n,1} \dots x_{m,n,r})^T.$$

Activation Function. Typically, layers in a neural network perform a linear transformation followed by a non-linear activation function. We focus on the commonly used rectified linear unit (ReLU) activation function, which for $x \in \mathbb{R}$ is defined as $\text{ReLU}(x) = \max(0, x)$, and for a vector $\bar{x} \in \mathbb{R}^m$ as $\text{ReLU}(\bar{x}) = (\text{ReLU}(x_1), \dots, \text{ReLU}(x_m))$.

ReLU to CAT. We can express the ReLU activation function as $\text{ReLU} = \text{ReLU}_n \circ \dots \circ \text{ReLU}_1$ where ReLU_i processes the i^{th} entry of the input \bar{x} and is given by:

$$\begin{aligned}
 \text{ReLU}_i(\bar{x}) &= \mathbf{case} \ (x_i \geq 0): \bar{x}, \\
 & \quad \mathbf{case} \ (x_i < 0): I_{i \leftarrow 0} \cdot \bar{x}.
 \end{aligned}$$

$I_{i \leftarrow 0}$ is the identity matrix with the i^{th} row replaced by zeros.

Fully Connected (FC) Layer. An FC layer takes a vector of size m (the m outputs of the previous layer), and passes it to n neurons, each computing a function based on the neuron's *weights* and *bias*, one weight for each component of the input. Formally, an FC layer with n neurons is a function $\text{FC}_{W,\bar{b}}: \mathbb{R}^m \rightarrow \mathbb{R}^n$ parameterized by a weight matrix $W \in \mathbb{R}^{n \times m}$ and a bias $\bar{b} \in \mathbb{R}^n$. For $\bar{x} \in \mathbb{R}^m$, we have:

$$\text{FC}_{W,\bar{b}}(\bar{x}) = \text{ReLU}(W \cdot \bar{x} + \bar{b}).$$

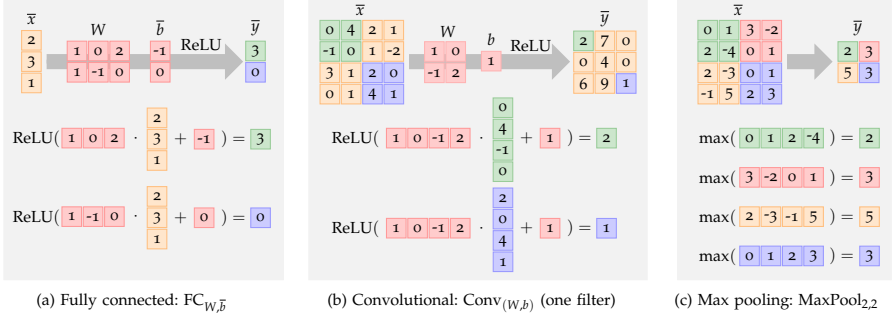


FIGURE 2.4: One example computation for each of the three layer types supported by AI².

Fig. 2.4a shows an FC layer computation for $\bar{x} = (2, 3, 1)$.

Convolutional Layer. A convolutional layer is defined by a series of t filters $F^{p,q} = (F_1^{p,q}, \dots, F_t^{p,q})$, parameterized by the same p and q , where $p \leq m$ and $q \leq n$. A filter $F_i^{p,q}$ is a function parameterized by a three-dimensional array of weights $W \in \mathbb{R}^{p \times q \times r}$ and a bias $b \in \mathbb{R}$. A filter takes a three-dimensional array and returns a two-dimensional array:

$$F_i^{p,q} : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1)}.$$

The entries of the output \bar{y} for a given input \bar{x} are given by:

$$y_{i,j} = \text{ReLU}\left(\sum_{i'=1}^p \sum_{j'=1}^q \sum_{k'=1}^r W_{i',j',k'} \cdot x_{(i+i'-1), (j+j'-1), k'} + b\right).$$

Intuitively, this matrix is computed by sliding the filter along the height and width of the input three-dimensional array, each time reading a slice of size $p \times q \times r$, computing its dot product with W (resulting in a real number), adding b , and applying ReLU. The function Conv_F , corresponding to a convolutional layer with t filters, has the following type:

$$\text{Conv}_F : \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}.$$

As expected, the function Conv_F returns a three-dimensional array of depth t , which stacks the outputs produced by each filter. Fig. 2.4b illustrates a computation of a convolutional layer with a single filter. For example:

$$y_{1,1,1} = \text{ReLU}((1 \cdot 0 + 0 \cdot 4 + (-1) \cdot (-1) + 2 \cdot 0) + 1) = 2.$$

Here, the input is a three-dimensional array in $\mathbb{R}^{4 \times 4 \times 1}$. As the input depth is 1, the depth of the filter’s weights is also 1. The output depth is 1 because the layer has one filter.

Convolutional Layer to CAT. For a convolutional layer $Conv_F$, we define a matrix W^F whose entries are those of the weight matrices for each filter (replicated to simulate sliding), and a bias \bar{b}^F consisting of copies of the filters’ biases. We then treat the convolutional layer $Conv^F$ like the equivalent FC $_{W^F, \bar{b}^F}$. We provide formal definitions of W^F and \bar{b}^F in Section 2.4.2. Here, we provide an intuitive illustration of the translation on the example in Fig. 2.4b. Consider the first entry $y_{1,1} = 2$ of \bar{y} in Fig. 2.4b:

$$y_{1,1} = \text{ReLU}(W_{1,1} \cdot x_{1,1} + W_{1,2} \cdot x_{1,2} + W_{2,1} \cdot x_{2,1} + W_{2,2} \cdot x_{2,2} + b).$$

When \bar{x} is reshaped to a vector \bar{x}^v , the four entries $x_{1,1}, x_{1,2}, x_{2,1}$ and $x_{2,2}$ will be found in x_1^v, x_2^v, x_5^v and x_6^v , respectively. Similarly, when \bar{y} is reshaped to \bar{y}^v , the entry $y_{1,1}$ will be found in y_1^v . Thus, to obtain $y_1^v = y_{1,1}$, we define the first row in W^F such that its 1st, 2nd, 5th, and 6th entries are $W_{1,1}, W_{1,2}, W_{2,1}$ and $W_{2,2}$. The other entries are zeros. We also define the first entry of the bias to be b . For similar reasons, to obtain $y_2^v = y_{1,2}$, we define the second row in W^F such that its 2nd, 3rd, 6th, and 7th entries are $W_{1,1}, W_{1,2}, W_{2,1}$ and $W_{2,2}$ (also $b_2 = b$). By following this transformation, we obtain the matrix $W^F \in \mathbb{R}^9 \times \mathbb{R}^{16}$ and the bias $\bar{b}^F \in \mathbb{R}^9$:

$$W^F = \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{-1} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} \quad \bar{b}^F = \begin{pmatrix} \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \end{pmatrix}$$

To aid understanding, we show the entries from W that appear in the resulting matrix W^F in bold.

Max Pooling (MP) Layer. An MP layer takes a three-dimensional array $\bar{x} \in \mathbb{R}^{m \times n \times r}$ and reduces the height m of \bar{x} by a factor of p and the width n of \bar{x} by a factor of q (for p and q dividing m and n). Depth is kept as-is. Neurons take as input disjoint subrectangles of \bar{x} of size $p \times q$ and return the maximal value in their subrectangle. Formally, the MP layer is

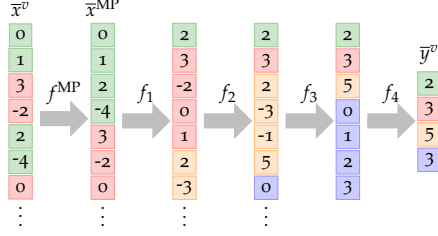


FIGURE 2.5: The operation of the transformed max-pooling layer.

a function $\text{MaxPool}_{p,q}: \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ that for an input \bar{x} returns the three-dimensional array \bar{y} given by:

$$y_{i,j,k} = \max(\{x_{i',j',k} \mid p \cdot (i-1) < i' \leq p \cdot i \\ q \cdot (j-1) < j' \leq q \cdot j\}).$$

Fig. 2.4c illustrates the max-pooling computation for $p = 2$, $q = 2$ and $r = 1$. For example, here we have:

$$y_{1,1,1} = \max(\{x_{1,1,1}, x_{1,2,1}, x_{2,1,1}, x_{2,2,1}\}) = 2.$$

Max Pooling to CAT. Let $\text{MaxPool}'_{p,q}: \mathbb{R}^{m \cdot n \cdot r} \rightarrow \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ be the function that is obtained from $\text{MaxPool}_{p,q}$ by reshaping its input and output: $\text{MaxPool}'_{p,q}(\bar{x}^v) = \text{MaxPool}_{p,q}(\bar{x})^v$. To represent max-pooling as a CAT function, we define a series of CAT functions whose composition is $\text{MaxPool}'_{p,q}$:

$$\text{MaxPool}'_{p,q} = f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r} \circ \dots \circ f_1 \circ f^{MP}.$$

The first function is $f^{MP}(\bar{x}^v) = W^{MP} \cdot \bar{x}^v$, which reorders its input vector \bar{x}^v to a vector \bar{x}^{MP} in which the values of each max-pooling subrectangle of \bar{x} are adjacent. The remaining functions execute standard max-pooling. Concretely, the function $f_i \in \{f_1, \dots, f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r}\}$ executes max-pooling on the i^{th} subrectangle by selecting the maximal value and removing the other values. We provide formal definitions of the CAT functions f^{MP} and f_i in Section 2.4.2.

Here, we illustrate them on the example from Fig. 2.4c, where $r = 1$. The CAT computation for this example is shown in Fig. 2.5. The computation begins from the input vector \bar{x}^v , which is the reshaping of \bar{x} from Fig. 2.4c. The values of the first 2×2 subrectangle in \bar{x} (namely, 0, 1, 2 and -4) are

separated in \bar{x}^v by values from another subrectangle (3 and -2). To make them contiguous, we reorder \bar{x}^v using a permutation matrix W^{MP} , yielding \bar{x}^{MP} . In our example, W^{MP} is:

$$W^{\text{MP}} = \begin{pmatrix} 1000000000000000 \\ 0100000000000000 \\ 0000100000000000 \\ 0000010000000000 \\ 0010000000000000 \\ 0001000000000000 \\ 0000001000000000 \\ 0000000100000000 \\ 0000000010000000 \\ 0000000001000000 \\ 0000000000100000 \\ 0000000000010000 \\ 0000000000001000 \\ 0000000000000100 \\ 0000000000000010 \\ 0000000000000001 \\ 0000000000000001 \end{pmatrix}$$

One entry in each row of W^{MP} is 1, all other entries are zeros. If row i has entry j set to 1, then the j^{th} value of \bar{x}^v is moved to the i^{th} entry of \bar{x}^{MP} . For example, we placed a one in the fifth column of the third row of W^{MP} to move the value x_5^v to entry 3 of the output vector.

Next, for each $i \in \{1, \dots, \frac{m}{p} \cdot \frac{n}{q}\}$, the function f_i takes as input a vector whose values at the indices between i and $i + p \cdot q - 1$ are those of the i^{th} subrectangle of \bar{x} in Fig. 2.4c. It then replaces those $p \cdot q$ values by their maximum:

$$f_i(\bar{x}) = (x_1, \dots, x_{i-1}, x_k, x_{i+p \cdot q}, \dots, x_{m \cdot n - (p \cdot q - 1) \cdot (i-1)}),$$

where the index $k \in \{i, \dots, i + p \cdot q - 1\}$ is such that x_k is maximal. For k given, f_i can be written as a CAT function: $f_i(\bar{x}) = W^{(i,k)} \cdot \bar{x}$, where the rows of the matrix $W^{(i,k)} \in \mathbb{R}^{(m \cdot n - (p \cdot q - 1) \cdot i) \times (m \cdot n - (p \cdot q - 1) \cdot (i-1))}$ are given by the following sequence of standard basis vectors:

$$\bar{e}_1, \dots, \bar{e}_{i-1}, \bar{e}_k, \bar{e}_{i+p \cdot q}, \dots, \bar{e}_{m \cdot n - (p \cdot q - 1) \cdot (i-1)}.$$

For example, in Fig. 2.5, $f_1(\bar{x}^{\text{MP}}) = W^{(1,3)} \cdot \bar{x}^{\text{MP}}$ deletes 0, 1 and -4 . Then it moves the value 2 to the first component, and the values at indices 5, \dots , 16 to components 2, \dots , 13. Overall, $W^{(1,3)}$ is given by:

$$W^{(1,3)} = \begin{pmatrix} 0010000000000000 \\ 0000100000000000 \\ 0000010000000000 \\ 0000001000000000 \\ 0000000100000000 \\ 0000000010000000 \\ 0000000001000000 \\ 0000000000100000 \\ 0000000000010000 \\ 0000000000001000 \\ 0000000000000100 \\ 0000000000000010 \\ 0000000000000001 \\ 0000000000000001 \end{pmatrix}$$

As, in general, k is not known in advance, we need to write f_i as a CAT function with a different case for each possible index k of the maximal value in \bar{x} . For example, in Fig. 2.5:

$$f_1(\bar{x}) = \begin{cases} \text{case } (x_1 \geq x_2) \wedge (x_1 \geq x_3) \wedge (x_1 \geq x_4): & W^{(1,1)} \cdot \bar{x}, \\ \text{case } (x_2 \geq x_1) \wedge (x_2 \geq x_3) \wedge (x_2 \geq x_4): & W^{(1,2)} \cdot \bar{x}, \\ \text{case } (x_3 \geq x_1) \wedge (x_3 \geq x_2) \wedge (x_3 \geq x_4): & W^{(1,3)} \cdot \bar{x}, \\ \text{case } (x_4 \geq x_1) \wedge (x_4 \geq x_2) \wedge (x_4 \geq x_3): & W^{(1,4)} \cdot \bar{x}. \end{cases}$$

In our example, the vector \bar{x}^{MP} in Fig. 2.5 satisfies the third condition, and therefore $f_1(\bar{x}^{\text{MP}}) = W^{(1,3)} \cdot \bar{x}^{\text{MP}}$. Taking into account all four subrectangles, we obtain:

$$\text{MaxPool}'_{2,2} = f_4 \circ f_3 \circ f_2 \circ f_1 \circ f^{\text{MP}}.$$

In summary, each function f_i replaces $p \cdot q$ components of their input by the maximum value among them, suitably moving other values. For \bar{x}^v in Fig. 2.5:

$$\text{MaxPool}'_{2,2}(\bar{x}^v) = W^{(4,7)} \cdot W^{(3,6)} \cdot W^{(2,2)} \cdot W^{(1,3)} \cdot W^{\text{MP}} \cdot \bar{x}^v.$$

Network Architectures. Two popular architectures of neural networks are fully connected (FNN) and convolutional (CNN). An FNN is a sequence of fully connected layers, while a CNN [67] consists of all previously described layer types: convolutional, max-pooling, and fully connected.

2.3 BACKGROUND: ABSTRACT INTERPRETATION

We now provide a short introduction to Abstract Interpretation (AI). AI enables one to prove program properties on a set of inputs *without* actually running the program. Formally, given a function $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$, a set of inputs $X \subseteq \mathbb{R}^m$, and a property $C \subseteq \mathbb{R}^n$, the goal is to determine whether the property holds, that is, whether $\forall \bar{x} \in X. f(\bar{x}) \in C$.

Fig. 2.6 shows a CAT function $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ that is defined as $f(\bar{x}) = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \cdot \bar{x}$ and four input points for the function f , given as

$$X = \{(0, 1), (1, 1), (1, 3), (2, 2)\}.$$

Let the property be $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$, which holds in this example. To reason about all inputs simultaneously, we lift the definition of f to be over a set of inputs X rather than a single input:

$$T_f: \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n), \quad T_f(X) = \{f(\bar{x}) \mid \bar{x} \in X\}.$$

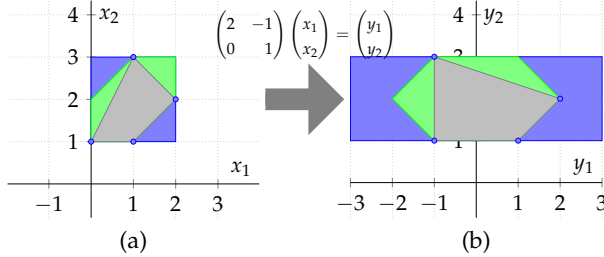


FIGURE 2.6: (a) Abstracting four points with a polyhedron (gray), zonotope (green), and box (blue). (b) The points and abstractions resulting from the affine transformer.

The function T_f is called the *concrete transformer* of f . With T_f , our goal is to determine whether $T_f(X) \subseteq C$ for a given input set X . Because the set X can be very large (or infinite), we cannot enumerate all points in X to compute $T_f(X)$. Instead, AI overapproximates sets with abstract elements (drawn from some abstract domain \mathcal{A}) and then defines a function, called an *abstract transformer* of f , which works with these abstract elements and overapproximates the effect of T_f . Then, the property C can be checked on the resulting abstract element returned by the abstract transformer. Naturally, because AI employs overapproximation, it is sound, but may be imprecise (i.e., may fail to prove the property when it holds). Next, we explain the ingredients of AI in more detail.

Abstract Domains. Abstract domains consist of shapes expressible as sets of logical constraints. A few popular numerical abstract domains are: Box (i.e., Interval), Zonotope, and Polyhedra. These domains provide trade-offs between precision and scalability (e.g., Box’s abstract transformers are far faster than Polyhedra’s abstract transformers, but polyhedra are far more precise than boxes). The Box domain consists of boxes, captured by a set of constraints of the form $a \leq x_i \leq b$, for $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$ and $a \leq b$. A box B contains all points which satisfy all constraints in B . In our example, X can be abstracted by the following box:

$$B = \{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}.$$

Note that B is not very precise since it includes 9 integer points (along with other points), whereas X has only 4 points.

The Zonotope domain [68] consists of *zonotopes*. A zonotope is a center-symmetric convex closed polyhedron $Z \subseteq \mathbb{R}^n$ that can be represented as an affine function:

$$z: [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_m, b_m] \rightarrow \mathbb{R}^n.$$

In other words, z has the form $z(\bar{\epsilon}) = M \cdot \bar{\epsilon} + \bar{b}$ where $\bar{\epsilon}$ is a vector of error terms satisfying interval constraints $\bar{\epsilon}_i \in [a_i, b_i]$ for $1 \leq i \leq m$. The bias vector \bar{b} captures the center of the zonotope, while the matrix M captures the boundaries of the zonotope around \bar{b} . A zonotope z represents all vectors in the image of z (i.e., $z[[a_1, b_1] \times \cdots \times [a_m, b_m]]$). In our example, X can be abstracted by the zonotope $z: [-1, 1]^3 \rightarrow \mathbb{R}^2$:

$$z(\epsilon_1, \epsilon_2, \epsilon_3) = (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3).$$

Zonotope is a more precise domain than Box: for our example, z includes only 7 integer points.

The Polyhedra [69] domain consists of convex closed polyhedra, where a polyhedron is captured by a set of linear constraints of the form $A \cdot \bar{x} \leq \bar{b}$, for some matrix A and a vector \bar{b} . A polyhedron C contains all points which satisfy the constraints in C . In our example, X can be abstracted by the following polyhedron:

$$C = \{x_2 \leq 2 \cdot x_1 + 1, x_2 \leq 4 - x_1, x_2 \geq 1, x_2 \geq x_1\}.$$

Polyhedra is a more precise domain than Zonotope: for our example, C includes only 5 integer points.

To conclude, abstract elements (from an abstract domain) represent large, potentially infinite sets. There are various abstract domains, providing different levels of precision and scalability.

Abstract Transformers. To compute the effect of a function on an abstract element, AI uses the concept of an *abstract transformer*. Given the (lifted) concrete transformer $T_f: \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n)$ of a function $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$, an abstract transformer of T_f is a function over abstract domains, denoted by $T_f^\#: \mathcal{A}^m \rightarrow \mathcal{A}^n$. The superscripts denote the number of components of the represented vectors. For example, elements in \mathcal{A}^m represent sets of vectors of dimension m . This also determines which variables can appear in the constraints associated with an abstract element. For example, elements in \mathcal{A}^m constrain the values of the variables x_1, \dots, x_m .

Abstract transformers have to be *sound*. To define soundness, we introduce two functions: the *abstraction* function α and the *concretization*

function γ . An abstraction function $\alpha^m: \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{A}^m$ maps a set of vectors to an abstract element in \mathcal{A}^m that overapproximates it. For example, in the Box domain:

$$\alpha^2(\{(0,1), (1,1), (1,3), (2,2)\}) = \{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}.$$

A concretization function $\gamma^m: \mathcal{A}^m \rightarrow \mathcal{P}(\mathbb{R}^m)$ does the opposite: it maps an abstract element to the set of concrete vectors that it represents. For example, for Box:

$$\begin{aligned} \gamma^2(\{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}) = & \{(0,1), (0,2), (0,3), \\ & (1,1), (1,2), (1,3), \\ & (2,1), (2,2), (2,3), \dots\}. \end{aligned}$$

This only shows the 9 vectors with integer components. We can now define soundness. An abstract transformer $T_f^\#$ is *sound* if for all $a \in \mathcal{A}^m$, we have $T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^\#(a))$, where T_f is the concrete transformer. That is, an abstract transformer has to overapproximate the effect of a concrete transformer. For example, the transformers illustrated in Fig. 2.6 are sound. For instance, if we apply the Box transformer on the box in Fig. 2.6a, it will produce the box in Fig. 2.6b. The box in Fig. 2.6b includes all points that f could compute in principle when given any point included in the concretization of the box in Fig. 2.6a. Analogous properties hold for the Zonotope and Polyhedra transformers. It is also important that abstract transformers are *precise*. That is, the abstract output should include as few points as possible. For example, as we can see in Fig. 2.6b, the output produced by the Box transformer is less precise (it is larger) than the output produced by the Zonotope transformer, which in turn is less precise than the output produced by the Polyhedra transformer.

Property Verification. After obtaining the (abstract) output, we can check various properties of interest on the result. In general, an abstract output $a = T_f^\#(\alpha^m(X))$ proves a property $T_f(X) \subseteq C$ if $\gamma^n(a) \subseteq C$. If the abstract output proves a property, we know that the property holds for all possible concrete values. However, the property may hold even if it cannot be proven with a given abstract domain. For example, in Fig. 2.6b, for all concrete points, the property $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$ holds. However, with the Box domain, the abstract output violates C , which means that the Box domain is not precise enough to prove the property. In contrast, the Zonotope and Polyhedra domains are precise enough to prove the property.

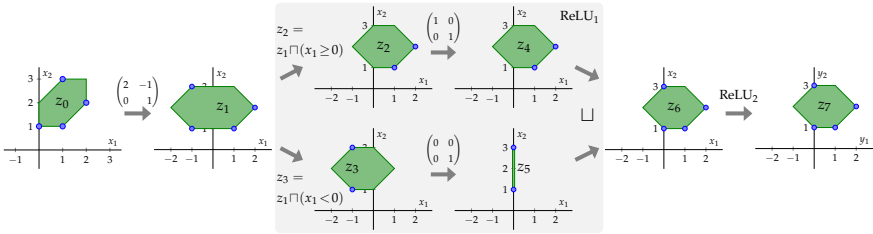


FIGURE 2.7: Illustration of how AI² overapproximates neural network states. Blue circles show the concrete values, while green zonotopes show the abstract elements. The gray box shows the steps in one application of the ReLU transformer (ReLU₁).

In summary, to apply AI successfully, we need to: (a) find a suitable abstract domain, and (b) define abstract transformers that are sound and as precise as possible. In the next section, we introduce abstract transformers for neural networks that are parameterized by the numerical abstract domain. This means that we can explore the precision-scalability trade-off by plugging in different abstract domains.

2.4 AI²: AI FOR NEURAL NETWORKS

In this section we present AI², an abstract interpretation framework for sound analysis of neural networks. We begin by defining abstract transformers for the different kinds of layers. Using these transformers, we then show how to prove robustness properties of neural networks.

2.4.1 Abstract Interpretation for CAT Functions

We now show how to overapproximate CAT functions with AI. We illustrate the method on the example in Fig. 2.7, which shows a simple network that manipulates two-dimensional vectors using a single fully connected layer of the form $f(\bar{x}) = \text{ReLU}_2(\text{ReLU}_1((\begin{smallmatrix} 2 & -1 \\ 0 & 1 \end{smallmatrix}) \cdot \bar{x}))$. Recall

$$\begin{aligned} \text{ReLU}_i(\bar{x}) &= \text{case } (x_i \geq 0): \bar{x}, \\ &\text{case } (x_i < 0): I_{i \leftarrow 0} \cdot \bar{x}, \end{aligned}$$

where $I_{i \leftarrow 0}$ is the identity matrix with the i^{th} row replaced by the zero vector. We overapproximate the network behavior on an abstract input.

The input can be obtained directly (see Sec. 2.4.3) or by abstracting a set of concrete inputs to an abstract element (using the abstraction function α). For our example, we use the concrete inputs (the blue points) from Fig 2.6. Those concrete inputs are abstracted to the green zonotope $z_0: [-1, 1]^3 \rightarrow \mathbb{R}^2$, given as:

$$z_0(\epsilon_1, \epsilon_2, \epsilon_3) = (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3).$$

Due to abstraction, more (spurious) points may be added. In this example, except the blue points, the entire area of the zonotope is spurious. We then apply abstract transformers to the abstract input. Note that, if a function f can be written as $f = f'' \circ f'$, the concrete transformer for f is $T_f = T_{f''} \circ T_{f'}$. Similarly, given abstract transformers $T_{f'}^\#$ and $T_{f''}^\#$, an abstract transformer for f is $T_f^\# \circ T_{f'}^\#$. When a neural network $N = f'_\ell \circ \dots \circ f'_1$ is a composition of multiple CAT functions f'_i of the shape $f'_i(\bar{x}) = W \cdot \bar{x} + \bar{b}$ or $f'_i(\bar{x}) = \mathbf{case} E_1: f_1(\bar{x}), \dots, \mathbf{case} E_k: f_k(\bar{x})$, we only have to define abstract transformers for these two kinds of functions. We then obtain the abstract transformer $T_N^\# = T_{f'_\ell}^\# \circ \dots \circ T_{f'_1}^\#$.

Abstracting Affine Functions. To abstract functions of the form $f(\bar{x}) = W \cdot \bar{x} + \bar{b}$, we assume that the underlying abstract domain supports the operator Aff that overapproximates such functions. Note that for Zonotope and Polyhedra, this operation is supported and exact. Fig. 2.7 demonstrates Aff as the first step taken for overapproximating the effect of the fully connected layer. Here, the resulting zonotope $z_1: [-1, 1]^3 \rightarrow \mathbb{R}^2$ is:

$$\begin{aligned} z_1(\epsilon_1, \epsilon_2, \epsilon_3) = & \\ & (2 \cdot (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2) - (2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3), \\ & \quad 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3) = \\ & (0.5 \cdot \epsilon_1 + \epsilon_2 - 0.5 \cdot \epsilon_3, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3). \end{aligned}$$

Abstracting Case Functions. To abstract functions of the form $f(\bar{x}) = \mathbf{case} E_1: f_1(\bar{x}), \dots, \mathbf{case} E_k: f_k(\bar{x})$, we first split the abstract element a into the different cases (each defined by one of the expressions E_i), resulting in k abstract elements a_1, \dots, a_k . We then compute the result of $T_{f_i}^\#(a_i)$ for each a_i . Finally, we unify the results to a single abstract element.

To split and unify, we assume two standard operators for abstract domains: (1) *meet* with a conjunction of linear constraints and (2) *join*. The meet (\sqcap) operator is an abstract transformer for set intersection: for

For $f(\bar{x}) = W \cdot \bar{x} + \bar{b}$, $T_f^\#(a) = \text{Aff}(a, W, \bar{b})$.

For $f(\bar{x}) = \text{case } E_1: f_1(\bar{x}), \dots, \text{case } E_k: f_k(\bar{y})$,

$$T_f^\#(a) = \bigsqcup_{1 \leq i \leq k} T_{f_i}^\#(a \sqcap E_i).$$

For $f(\bar{x}) = f_2(f_1(\bar{x}))$, $T_f^\#(a) = T_{f_2}^\#(T_{f_1}^\#(a))$.

FIGURE 2.8: Abstract transformers for CAT functions.

an inequality expression E from Fig. 2.3, $\gamma^n(a) \cap \{x \in \mathbb{R}^n \mid x \models E\} \subseteq \gamma^n(a \sqcap E)$. The *join* (\sqcup) operator is an abstract transformer for set union: $\gamma^n(a_1) \cup \gamma^n(a_2) \subseteq \gamma^n(a_1 \sqcup a_2)$. We further assume that the abstract domain contains an element \perp , which satisfies $\gamma^n(\perp) = \{\}$, $\perp \sqcap E = \perp$ and $a \sqcup \perp = a$ for $a \in \mathcal{A}$.

For our example in Fig. 2.7, abstract interpretation continues on z_1 using the meet and join operators. To compute the effect of ReLU_1 , z_1 is split into two zonotopes $z_2 = z_1 \sqcap (x_1 \geq 0)$ and $z_3 = z_1 \sqcap (x_1 < 0)$. One way to compute a meet between a zonotope and a linear constraint is to modify the intervals of the error terms (see [70]). In our example, the resulting zonotopes are $z_2 : [-1, 1] \times [0, 1] \times [-1, 1] \rightarrow \mathbb{R}^2$ such that $z_2(\bar{\epsilon}) = z_1(\bar{\epsilon})$ and $z_3 : [-1, 1] \times [-1, 0] \times [-1, 1] \rightarrow \mathbb{R}^2$ such that $z_3(\bar{\epsilon}) = z_1(\bar{\epsilon})$ for $\bar{\epsilon}$ common to their respective domains. Note that both z_2 and z_3 contain small spurious areas, because the intersections of the respective linear constraints with z_1 are not zonotopes. Therefore, they cannot be captured exactly by the domain. Here, the meet operator \sqcap overapproximates set intersection \cap to get a sound, but not perfectly precise, result.

Then, the two cases of ReLU_1 are processed separately. We apply the abstract transformer of $f_1(\bar{x}) = \bar{x}$ to z_2 and we apply the abstract transformer of $f_2(\bar{x}) = I_{0 \leftarrow 0} \cdot \bar{x}$ to z_3 . The resulting zonotopes are $z_4 = z_2$ and $z_5 : [-1, 1]^2 \rightarrow \mathbb{R}^2$ such that $z_5(\epsilon_1, \epsilon_3) = (0, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3)$. These are then joined to obtain a single zonotope z_6 . Since z_5 is contained in z_4 , we get $z_6 = z_4$ (of course, this need not always be the case). Then, z_6 is passed to ReLU_2 . Because $z_6 \sqcap (x_1 < 0) = \perp$, this results in $z_7 = z_6$. Finally, $\gamma^2(z_7)$ is our overapproximation of the network outputs for our initial set of points. The abstract element z_7 is a finite representation of this infinite set.

In summary, we define abstract transformers for every kind of CAT function (summarized in Fig. 2.8). These definitions are general and are compatible with any abstract domain \mathcal{A} which has a minimum element \perp and supports (1) a meet operator between an abstract element and a conjunction of linear constraints, (2) a join operator between two abstract elements, and (3) an affine transformer. We assume that the operations are sound. We note that these operations are standard or definable with standard operations. By definition of the abstract transformers, we get soundness:

Theorem 1. *For any CAT function f with transformer $T_f : \mathcal{P}(\mathbb{R}^m) \rightarrow \mathcal{P}(\mathbb{R}^n)$ and any abstract input $a \in \mathcal{A}^m$,*

$$T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^\#(a)).$$

Theorem 1 is the key to sound neural network analysis with our abstract transformers, as we explain in the next section.

2.4.2 CAT Representations for Common Layer Types

In this section, we provide the formal definitions of the matrices and vectors used to represent the convolutional layer and the max-pooling layer as CAT functions.

Convolutional Layer. Recall that for filters $W^k \in \mathbb{R}^{p \times q \times r}$, $b^k \in \mathbb{R}$ for $1 \leq k \leq t$, we have

$$\begin{aligned} \text{Conv}_F(\bar{x}) &: \mathbb{R}^{n \times m \times r} \rightarrow \mathbb{R}^{(m-p+1) \times (n-q+1) \times t} \\ \text{Conv}_F(\bar{x})_{i,j,k} &= \text{ReLU}\left(\sum_{i'=1}^p \sum_{j'=1}^q \sum_{k'=1}^r W_{i',j',k'}^k \cdot x_{(i+i'-1),(j+j'-1),k'} + b^k\right), \end{aligned}$$

for $1 \leq i \leq m - p + 1$, $1 \leq j \leq n - q + 1$ and $1 \leq k \leq t$. Reshaping both the input and the output vector such that they have only one index, we obtain

$$\begin{aligned} \text{Conv}'_F(\bar{x}) &: \mathbb{R}^{n \cdot m \cdot r} \rightarrow \mathbb{R}^{(m-p+1) \cdot (n-q+1) \cdot t} \\ \text{Conv}'_F(\bar{x})_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k} &= \text{ReLU}\left(\sum_{i'=1}^p \sum_{j'=1}^q \sum_{k'=1}^r W_{i',j',k'}^k \cdot x_{n \cdot r \cdot (i+i'-2) + r \cdot (j+j'-2) + k'} + b^k\right), \end{aligned}$$

for $1 \leq i \leq m - p + 1$, $1 \leq j \leq n - q + 1$ and $1 \leq k \leq t$. The function Conv'_F is ReLU after an affine transformation, therefore there is a matrix $W^F \in \mathbb{R}^{((m-p+1) \cdot (n-q+1) \cdot t) \times (n \cdot m \cdot r)}$ and a vector $\bar{b}^F \in \mathbb{R}^{(m-p+1) \cdot (n-q+1) \cdot t}$ such that

$$\text{Conv}_F(\bar{x})^v = \text{Conv}'_F(\bar{x}^v) = \text{ReLU}(W^F \cdot \bar{x}^v + \bar{b}^F) = \text{FC}_{W^F, \bar{b}^F}(\bar{x}).$$

The entries of W^F and \bar{b}^F are obtained by equating

$$\begin{aligned} \text{FC}(\bar{e}_l)_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k} &= \text{ReLU}(W_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k, l}^F \\ &\quad + b_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k}^F) \text{ with} \\ \text{Conv}'_F(\bar{e}_l)_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k} &= \text{ReLU}\left(\sum_{i'=1}^p \sum_{j'=1}^q \sum_{k'=1}^r W_{i',j',k'}^k \cdot \right. \\ &\quad \left. [l = n \cdot r \cdot (i + i' - 2) \right. \\ &\quad \left. + r \cdot (j + j' - 2) + k'] + b^k\right), \end{aligned}$$

for standard basis vectors \bar{e}_l with $(\bar{e}_l)_i = [l = i]$ for $1 \leq l \leq n$ and $1 \leq i \leq n \cdot m \cdot r$. This way, we obtain

$$W_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k, l}^F = \sum_{i'=1}^p \sum_{j'=1}^q \sum_{k'=1}^r W_{i', j', k'}^k \cdot [l = n \cdot r \cdot (i + i' - 2) + r \cdot (j + j' - 2) + k'] \text{ and}$$

$$b_{(n-q+1) \cdot t \cdot (i-1) + t \cdot (j-1) + k}^F = b^k,$$

for $1 \leq i \leq m - p + 1$, $1 \leq j \leq n - q + 1$ and $1 \leq k \leq t$. Note that here, $[\varphi]$ is an Iverson bracket, which is equal to 1 if φ holds and equal to 0 otherwise.

Max Pooling Layer. Recall that $\text{MaxPool}_{p,q}: \mathbb{R}^{m \times n \times r} \rightarrow \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ partitions the input vector into disjoint blocks of size $p \times q \times 1$ and replaces each block by its maximum value. Furthermore, $\text{MaxPool}'_{p,q}: \mathbb{R}^{m \cdot n \cdot r} \rightarrow \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ is obtained from $\text{MaxPool}_{p,q}$ by reshaping both the input and output: $\text{MaxPool}'_{p,q}(\bar{x}^v) = \text{MaxPool}_{p,q}(\bar{x})^v$. We will represent $\text{MaxPool}'_{p,q}$ as a composition of CAT functions,

$$\text{MaxPool}'_{p,q} = f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r} \circ \dots \circ f_1 \circ f^{\text{MP}}.$$

Here, f^{MP} rearranges the input vector such that values from the same block are adjacent. Values from different blocks are brought into the same order as the output from each block appears in the output vector.

Note that $((i - 1) \bmod p) + 1, ((j - 1) \bmod q) + 1, 1$ are the indices of input value $x_{i,j,k}$ within its respective block and $\left(\left\lfloor \frac{i-1}{p} \right\rfloor + 1, \left\lfloor \frac{j-1}{q} \right\rfloor + 1, k\right)$ are the indices of the unique value in the output vector whose value depends on $x_{i,j,k}$. Recall that the permutation matrix M representing a permutation π is given by $M_{\pi(i)} = \bar{e}_i$.

The CAT function f^{MP} is a linear transformation $f^{\text{MP}}(\bar{x}^v) = W^{\text{MP}} \cdot \bar{x}^v$ where the permutation matrix W^{MP} is given by

$$W_{r \cdot p \cdot q \cdot \left(\frac{n}{q} \left\lfloor \frac{i-1}{p} \right\rfloor + \left\lfloor \frac{i-1}{q} \right\rfloor\right) + p \cdot q \cdot (k-1) + q \cdot ((i-1) \bmod p) + ((j-1) \bmod q) + 1}^{\text{MP}} = \bar{e}_{n \cdot r \cdot (i-1) + r \cdot (j-1) + k'}$$

for $1 \leq i \leq m$, $1 \leq j \leq n$ and $1 \leq k \leq r$.

For each $1 \leq i \leq \frac{m}{p} \cdot \frac{n}{q} \cdot r$, the CAT function f_i selects the maximum value from a $(p \cdot q)$ -segment starting from the i^{th} component of the input

vector. The function f_i consists of a sequence of cases, one for each of the $p \cdot q$ possible indices of the maximal value in the segment:

$$\begin{aligned} f_i(\bar{x}) = & \text{case } (x_i \geq x_{i+1}) \wedge \dots \wedge (x_i \geq x_{i+p \cdot q - 1}): W^{(i,i)} \cdot \bar{x}, \\ & \text{case } (x_{i+1} \geq x_i) \wedge \dots \wedge (x_{i+1} \geq x_{i+p \cdot q - 1}): W^{(i,i+1)} \cdot \bar{x}, \\ & \vdots \\ & \text{case } (x_{i+p \cdot q - 1} \geq x_i) \wedge \dots \wedge (x_{i+p \cdot q - 1} \geq x_{i+p \cdot q - 2}): W^{(i,i+p \cdot q - 1)} \cdot \bar{x}. \end{aligned}$$

The matrix $W^{(i,k)} \in \mathbb{R}^{(m \cdot n \cdot r - (p \cdot q - 1) \cdot i) \times (m \cdot n \cdot r - (p \cdot q - 1) \cdot (i-1))}$ replaces the segment $x_i, \dots, x_{i+p \cdot q - 1}$ of the input vector \bar{x} by the value x_k and is given by

$$W_j^{(i,k)} = \begin{cases} \bar{e}_j, & \text{if } 1 \leq j \leq i - 1 \\ \bar{e}_k, & \text{if } j = i \\ \bar{e}_{j+p \cdot q - 1}, & \text{if } i + 1 \leq j \leq m \cdot n \cdot r - (p \cdot q - 1) \cdot i \end{cases}.$$

2.4.3 Neural Network Analysis with AI

In this section, we explain how to leverage AI with our abstract transformers to prove properties of neural networks. We focus on robustness properties below, however, the framework can be applied to reason about any safety property.

For robustness, we aim to determine if for a (possibly unbounded) set of inputs, the outputs of a neural network satisfy a given condition. A robustness property for a neural network $N: \mathbb{R}^m \rightarrow \mathbb{R}^n$ is a pair $(X, C) \in \mathcal{P}(\mathbb{R}^m) \times \mathcal{P}(\mathbb{R}^n)$ consisting of a robustness region X and a robustness condition C . We say that the neural network N satisfies a robustness property (X, C) if $N(\bar{x}) \in C$ for all $\bar{x} \in X$.

Local Robustness. This is a property (X, C_L) where X is a robustness region and C_L contains the outputs that describe the *same* label L :

$$C_L = \left\{ \bar{y} \in \mathbb{R}^n \mid \arg \max_{i \in \{1, \dots, n\}} (y_i) = L \right\}.$$

For example, Fig. 2.7 shows a neural network and a robustness property (X, C_2) for $X = \{(0, 1), (1, 1), (1, 3), (2, 2)\}$ and $C_2 = \{\bar{y} \mid \arg \max(y_1, y_2) =$

2}. In this example, (X, C_2) holds. Typically, we will want to check that there is *some* label L for which (X, C_L) holds.

We now explain how our abstract transformers can be used to prove a given robustness property (X, C) .

Robustness Proofs using AI. Assume we are given a neural network $N: \mathbb{R}^m \rightarrow \mathbb{R}^n$, a robustness property (X, C) and an abstract domain \mathcal{A} (supporting \sqcup, \sqcap with a conjunction of linear constraints, Aff , and \perp) with an abstraction function α and a concretization function γ . Further assume that N can be written as a CAT function. Denote by $T_N^\#$ the abstract transformer of N , as defined in Fig. 2.8. Then, the following condition is sufficient to prove that N satisfies (X, C) :

$$\gamma^n(T_N^\#(\alpha^m(X))) \subseteq C.$$

This follows from Theorem 1 and the properties of α and γ . Note that there may be abstract domains \mathcal{A} that are not precise enough to prove that N satisfies (X, C) , even if N in fact satisfies (X, C) . On the other hand, if we are able to show that some abstract domain \mathcal{A} proves that N satisfies (X, C) , we know that it holds.

Proving Containment. To prove the property (X, C) given the result $a = T_N^\#(\alpha^m(X))$ of abstract interpretation, we need to be able to show $\gamma^n(a) \subseteq C$. There is a general method if C is given by a CNF formula $\bigwedge_i \bigvee_j l_{i,j}$ where all literals $l_{i,j}$ are linear constraints: we show that the negated formula $\bigvee_i \bigwedge_j \neg l_{i,j}$ is inconsistent with the abstract element a by checking that $a \sqcap \left(\bigwedge_j \neg l_{i,j} \right) = \perp$ for all i .

For our example in Fig. 2.7, the goal is to check that all inputs are classified as 2. We can represent C using the formula $y_2 \geq y_1$. Its negation is $y_2 < y_1$, and it suffices to show that no point in the concretization of the abstract output satisfies this negated constraint. As indeed $z_7 \sqcap (y_2 < y_1) = \perp$, the property is successfully verified. However, note that we would not be able to prove some other true properties, such as $y_1 \geq 0$. This property holds for all concrete outputs, but some points in the concretization of the output z_7 do not satisfy it.

2.5 IMPLEMENTATION OF AI²

The AI² framework is implemented in the D programming language and supports any neural network composed of fully connected, convolutional, and max-pooling layers.

Properties. AI² supports properties (X, C) where X is specified by a zonotope and C by a conjunction of linear constraints over the output vector’s components. In our experiments, we illustrate the specification of local robustness properties where the region X is defined by a box or a line, both of which are precisely captured by a zonotope.

Abstract Domains. The AI² system is fully integrated with all abstract domains supported by Apron [71], a popular library for numerical abstract domains, including: Box [64], Zonotope [68], and Polyhedra [69].

Bounded Powerset. We also implemented bounded powerset domains (disjunctive abstractions [72, 73]), which can be instantiated with any of the above abstract domains. An abstract element in the powerset domain $\mathcal{P}(\mathcal{A})$ of an underlying abstract domain \mathcal{A} is a set of abstract elements from \mathcal{A} , concretizing to the union of the concretizations of the individual elements (i.e., $\gamma(A) = \bigcup_{a \in A} \gamma(a)$ for $A \in \mathcal{P}(\mathcal{A})$).

The powerset domain can implement a precise join operator using standard set union (potentially pruning redundant elements). However, since the increased precision can become prohibitively costly if many join operations are performed, the bounded powerset domain limits the number of abstract elements in a set to N (for some constant N).

We implemented bounded powerset domains on top of standard powerset domains using a greedy heuristic that repeatedly replaces two abstract elements in a set by their join, until the number of abstract elements in the set is below the bound N . For joining, AI² heuristically selects two abstract elements that minimize the distance between the centers of their bounding boxes. In our experiments, we denote by Zonotope N or ZN the bounded powerset domain with bound $N \geq 2$ and underlying abstract domain Zonotope.

2.6 EVALUATION OF AI²

In this section, we present our empirical evaluation of AI². Before discussing the results in detail, we summarize our three most important findings:

- AI² can prove useful robustness properties for convolutional networks with 53 000 neurons and large fully connected feedforward networks with 1 800 neurons.

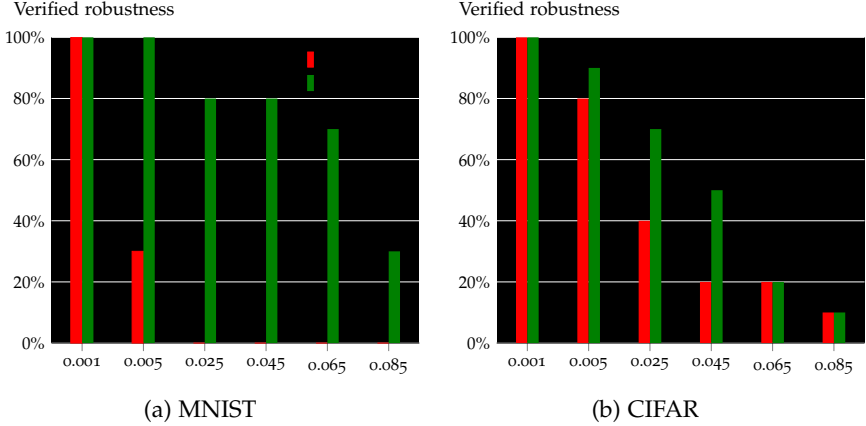


FIGURE 2.9: Verified properties by AI² on the MNIST and CIFAR convolutional networks for each bound $\delta \in \Delta$ (x-axis).

- AI² benefits from more precise abstract domains: Zonotope enables AI² to prove substantially more properties over Box. Furthermore, ZonotopeN, with $N \geq 2$, can prove stronger robustness properties than Zonotope alone.
- AI² scales better than the SMT-based Reluplex [7]: AI² is able to verify robustness properties on large networks with ≥ 1200 neurons within few minutes, while Reluplex takes hours to verify the same properties.

In the following, we first describe our experimental setup. Then, we present and discuss our results.

2.6.1 Experimental Setup

We now describe the datasets, neural networks, and robustness properties used in our experiments.

Datasets. We used two popular datasets: MNIST [52] and CIFAR-10 [74] (referred to as CIFAR from now on). MNIST consists of 60 000 grayscale images of handwritten digits, whose resolution is 28×28 pixels. The images show white digits on a black background.

CIFAR consists of 60 000 colored photographs with 3 color channels, whose resolution is 32×32 pixels. The images are partitioned into 10

different classes (e.g., airplane or bird). Each photograph has a different background (unlike MNIST).

Neural Networks. We trained convolutional and fully connected feedforward networks on both datasets. All networks were trained using accelerated gradient descent with at least 50 epochs of batch size 128. The training completed when each network had a test set accuracy of at least 0.9.

For the convolutional networks, we used the LeNet architecture [75], which consists of the following sequence of layers: 2 convolutional, 1 max-pooling, 2 convolutional, 1 max-pooling, and 3 fully connected layers. We write $n_{p \times q}$ to denote a convolutional layer with n filters of size $p \times q$, and m to denote a fully connected layer with m neurons. The hidden layers of the MNIST network are $8_{3 \times 3}, 8_{3 \times 3}, 14_{3 \times 3}, 14_{3 \times 3}, 50, 50, 10$, and those of the CIFAR network are $24_{3 \times 3}, 24_{3 \times 3}, 32_{3 \times 3}, 32_{3 \times 3}, 100, 100, 10$. The max-pooling layers of both networks have a size of 2×2 . We trained our networks using an open-source implementation [76].

We used 7 different architectures of fully connected feedforward networks (FNNs). We write $l \times n$ to denote the FNN architecture with l layers, each consisting of n neurons. Note that this determines the network's size; e.g., a 4×50 network has 200 neurons. For each dataset, MNIST and CIFAR, we trained FNNs with the following architectures: $3 \times 20, 6 \times 20, 3 \times 50, 3 \times 100, 6 \times 100, 6 \times 200$, and 9×200 .

Robustness Properties. In our experiments, we consider local robustness properties (X, C_L) where the region X captures changes to lighting conditions. This type of property is inspired by the work of [26], where adversarial examples were found by brightening the pixels of an image.

Formally, we consider robustness regions $S_{\bar{x}, \delta}$ that are parameterized by an input $\bar{x} \in \mathbb{R}^m$ and a robustness bound $\delta \in [0, 1]$. The robustness region is defined as:

$$S_{\bar{x}, \delta} = \{\bar{x}' \in \mathbb{R}^m \mid \forall i \in [1, m]. 1 - \delta \leq x_i \leq x'_i \leq 1 \vee x'_i = x_i\}.$$

For example, the robustness region for $\bar{x} = (0.6, 0.85, 0.9)$ and bound $\delta = 0.2$ is given by the set:

$$\{(0.6, x, x') \in \mathbb{R}^3 \mid x \in [0.85, 1], x' \in [0.9, 1]\}.$$

Note that increasing the bound δ increases the region's size.

In our experiments, we used AI^2 to check whether all inputs in a given region $S_{\bar{x},\delta}$ are classified to the label assigned to \bar{x} . We consider 6 different robustness bounds δ , which are drawn from the set

$$\Delta = \{0.001, 0.005, 0.025, 0.045, 0.065, 0.085\}.$$

We remark that our robustness properties are stronger than those considered in [26]. This is because, in a given robustness region $S_{\bar{x},\delta}$, each pixel of the image \bar{x} is brightened independently of the other pixels. We remark that this is useful to capture scenarios where only part of the image is brightened (e.g., due to shadowing).

Other perturbations. Note that AI^2 is not limited to certifying robustness against such brightening perturbations. In general, AI^2 can be used whenever the set of perturbed inputs can be overapproximated with a set of zonotopes in a precise way (i.e., without adding too many inputs that do not capture actual perturbations to the robustness region). For example, the inputs perturbed by an ℓ_∞ attack [25] are captured exactly by a single zonotope. Further, rotations and translations have low-dimensional parameter spaces, and therefore can be overapproximated by a set of zonotopes in a precise way.

Benchmarks. We selected 10 images from each dataset. Then, we specified a robustness property for each image and each robustness bound in Δ , resulting in 60 properties per dataset. We ran AI^2 to check whether each neural network satisfies the robustness properties for the respective dataset. We compared the results using different abstract domains, including Box, Zonotope, and Zonotope N with N ranging between 2 and 128.

We ran all experiments on an Ubuntu 16.04.3 LTS server with two Intel Xeon E5-2690 processors and 512GB of memory. To compare AI^2 to existing solutions, we also ran the FNN benchmarks with Reluplex [7]. We did not run convolutional benchmarks with Reluplex as it currently does not support convolutional networks.

2.6.2 Discussion of Results

In the following, we first present our results for convolutional networks. Then, we present experiments with different abstract domains and discuss how the domain’s precision affects AI^2 ’s ability to verify robustness. We also plot AI^2 ’s running times for different abstract domains to investigate

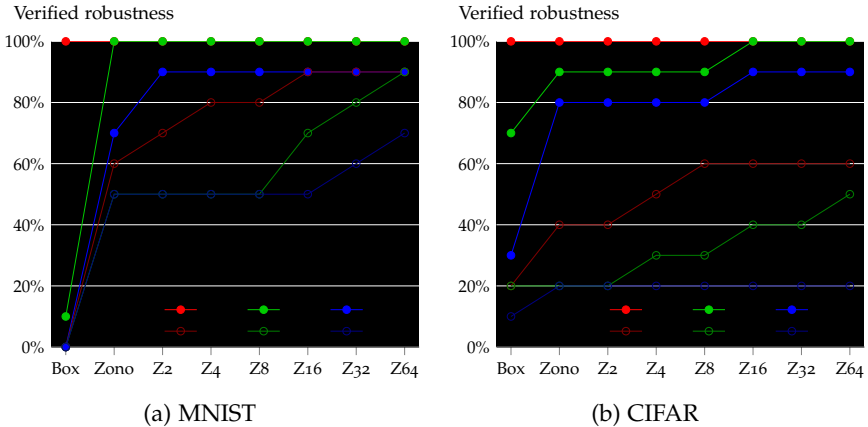


FIGURE 2.10: Verified properties as a function of the abstract domain used by AI² for the 9×200 network. Each point represents the fraction of robustness properties for a given bound (as specified in the legend) verified by a given abstract domain (x -axis).

the trade-off between precision and scalability. Finally, we compare AI² to Reluplex.

Proving Robustness of Convolutional Networks. We start with our results for convolutional networks. AI² terminated within 1.5 minutes when verifying properties on the MNIST network and within 1 hour when verifying the CIFAR network.

In Fig. 2.9, we show the fraction of robustness properties verified by AI² for each robustness bound. We plot separate bars for Box and Zonotope to illustrate the effect of the domain’s precision on AI²’s ability to verify robustness.

For both networks, AI² verified all robustness properties for the smallest bound 0.001 and it verified at least one property for the largest bound 0.085. This demonstrates that AI² can verify properties of convolutional networks with rather wide robustness regions. Further, the number of verified properties converges to zero as the robustness bound increases. This is expected, as larger robustness regions are more likely to contain adversarial examples.

In Fig. 2.9a, we observe that Zonotope proves significantly more properties than Box. For example, Box fails to prove any robustness properties with bounds at least 0.025, while Zonotope proves 80% of the

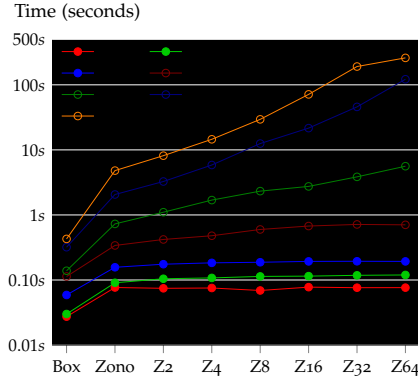


FIGURE 2.11: Average running time of AI^2 when proving robustness properties on MNIST networks as a function of the abstract domain used by AI^2 (x -axis). Axes are scaled logarithmically.

properties with bounds 0.025 and 0.045. This indicates that Box is often imprecise and fails to prove properties that the network satisfies.

Similarly, Fig. 2.9b shows that Zonotope proves more robustness properties than Box also for the CIFAR convolutional network. The difference between these two domains is, however, less significant than that observed for the MNIST network. For example, both Box and Zonotope prove the same properties for bounds 0.065 and 0.085.

Precision of Different Abstract Domains. Next, we demonstrate that more precise abstract domains enable AI^2 to prove stronger robustness properties. In this experiment, we consider our 9×200 MNIST and CIFAR networks, which are our largest fully connected feedforward networks. We evaluate the Box, Zonotope, and the Zonotope N domains for exponentially increasing bounds of N between 2 and 64. We do not report results for the Polyhedra domain, which takes several days to terminate for our smallest networks.

In Fig. 2.10, we show the fraction of verified robustness properties as a function of the abstract domain used by AI^2 . We plot a separate line for each robustness bound. All runs of AI^2 in this experiment completed within 1 hour.

The graphs show that Zonotope proves more robustness properties than Box. For the MNIST network, Box proves 11 out of all 60 robustness properties (across all 6 bounds), failing to prove any robustness properties

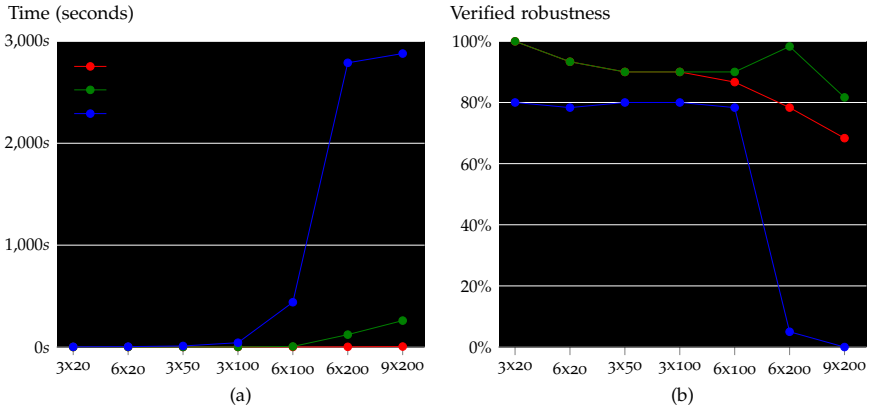


FIGURE 2.12: Comparing the performance of AI² to Reluplex. Each point is an average of the results for all 60 robustness properties for the MNIST networks. Each point in (a) represents the average time to completion, regardless of the result of the computation. While not shown, the result of the computation could be a failure to verify, timeout, crash, or discovery of a counterexample. Each point in (b) represents the fraction of the 60 robustness properties that were verified.

with bounds above 0.005. In contrast, Zonotope proves 43 out of the 60 properties and proves at least 50% of the properties across the 6 robustness bounds. For the CIFAR network, Box proves 25 out of the 60 properties while Zonotope proves 35.

The data also demonstrates that bounded sets of zonotopes further improve AI²'s ability to prove robustness properties. For the MNIST network, Zonotope64 proves more robustness properties than Zonotope for all 4 bounds for which Zonotope fails to prove at least one property (i.e., for bounds $\delta \geq 0.025$). For the CIFAR network, Zonotope64 proves more properties than Zonotope for 4 out of the 5 the bounds. The only exception is the bound 0.085, where Zonotope64 and Zonotope prove the same set of properties.

Trade-off between Precision and Scalability. In Fig. 2.11, we plot the running time of AI² as a function of the abstract domain. Each point in the graph represents the average running time of AI² when proving a robustness property for a given MNIST network (as indicated in the legend). We use a log-log plot to better visualize the trade-off in time.

The data shows that AI^2 can efficiently verify robustness of large networks. AI^2 terminates within a few minutes for all MNIST FNNs and all considered domains. Further, we observe that AI^2 takes less than 10 seconds on average to verify a property with the Zonotope domain.

As expected, the graph demonstrates that more precise domains increase AI^2 's running time. More importantly, AI^2 's running time is polynomial in the bound N of Zonotope_N , which allows one to adjust AI^2 's precision by increasing N .

Comparison to Reluplex. The current state-of-the-art system for verifying properties of neural networks is Reluplex [7]. Reluplex supports FNNs with ReLU activation functions, and its analysis is sound and complete. Reluplex would eventually either verify a given property or return a counterexample.

To compare the performance of Reluplex and AI^2 , we ran both systems on all MNIST FNN benchmarks. We ran AI^2 using Zonotope and Zonotope64. For both Reluplex and AI^2 , we set a 1 hour timeout for verifying a single property.

Fig. 2.12 presents our results: Fig. 2.12a plots the average running time of Reluplex and AI^2 and Fig. 2.12b shows the fraction of robustness properties verified by the systems. The data shows that Reluplex can analyze FNNs with at most 600 neurons efficiently, typically within a few minutes. Overall, both system verified roughly the same set of properties. However, Reluplex crashed during verification of some of the properties. This explains why AI^2 was able to prove slightly more properties than Reluplex on the smaller FNNs.

For large networks with more than 600 neurons, the running time of Reluplex increases significantly and its analysis often times out. In contrast, AI^2 analyzes the large networks within a few minutes and verifies substantially more robustness properties than Reluplex. For example, Zonotope64 proves 57 out of the 60 properties on the 6×200 network, while Reluplex proves 3. Further, Zonotope64 proves 45 out of the 60 properties on the largest 9×200 network, while Reluplex proves none. We remark that while Reluplex did not verify any property on the largest 9×200 network, it did disprove some of the properties and returned counterexamples.

We also ran Reluplex without a predefined timeout to investigate how long it would take to verify properties on the large networks. To this end, we ran Reluplex on properties that AI^2 successfully verified. We observed that Reluplex often took more than 24 hours to terminate. Overall, our

results indicate that Reluplex does not scale to larger FNNs whereas AI² succeeds on these networks.

2.7 COMPARING DEFENSES WITH AI²

In this section, we illustrate a practical application of AI²: evaluating and comparing neural network defenses. A defense is an algorithm whose goal is to reduce the effectiveness of a certain attack against a specific network, for example, by retraining the network with an altered loss function. Since the discovery of adversarial examples, many works have suggested different kinds of defenses to mitigate this phenomenon (e.g., [20, 41, 44]). A natural metric to compare defenses is the average “size” of the robustness region on some test set. Intuitively, the greater this size is, the more robust the defense.

We compared three state-of-the-art defenses:

- **GSS** [20] extends the loss with a regularization term encoding the fast gradient sign method (FGSM) attack.
- **Ensemble** [44] is similar to GSS, but includes regularization terms from attacks on other models.
- **MMSTV** [41] adds, during training, a perturbation layer before the input layer which applies the FGSM^k attack. FGSM^k is a multi-step variant of FGSM, also known as projected gradient descent.

All these defenses attempt to reduce the effectiveness of the FGSM attack [20]. This attack consists of taking a network N and an input \bar{x} and computing a vector $\bar{\rho}_{N,\bar{x}}$ in the input space along which an adversarial example is likely to be found. An adversarial input \bar{a} is then generated by taking a step ϵ along this vector: $\bar{a} = \bar{x} + \epsilon \cdot \bar{\rho}_{N,\bar{x}}$.

We define a new kind of robustness region, called *line*, that captures resilience with respect to the FGSM attack. The line robustness region captures all points from \bar{x} to $\bar{x} + \delta \cdot \bar{\rho}_{N,\bar{x}}$ for some robustness bound δ :

$$L_{N,\bar{x},\delta} = \{\bar{x} + \epsilon \cdot \bar{\rho}_{N,\bar{x}} \mid \epsilon \in [0, \delta]\}.$$

This robustness region is a zonotope and can thus be precisely captured by AI².

We compared the three state-of-the-art defenses on the MNIST convolutional network described in Section 2.6; we call this the Original network. We trained the Original network with each of the defenses, which resulted in 3 additional networks: GSS, Ensemble, and MMSTV. We used

40 epochs for GSS, 12 epochs for Ensemble, and 10 000 training steps for MMSTV using their published frameworks.

We conducted 20 experiments. In each experiment, we randomly selected an image \bar{x} and computed $\bar{\rho}_{N,\bar{x}}$. Then, for each network, our goal was to find the largest bound δ for which AI^2 proves the region $L_{N,\bar{x},\delta}$ robust. To approximate the largest robustness bound, we ran binary search to depth 6 and ran AI^2 with the Zonotope domain for each candidate bound δ . We refer to the largest robustness bound verified by AI^2 as the *verified bound*.

The average verified bounds for the Original, GSS, Ensemble, and MMSTV networks are 0.026, 0.031, 0.042, and 0.209, respectively. Fig. 2.13 shows a box-and-whisker plot which demonstrates the distribution of the verified bounds for the four networks. The bottom and top of each whisker show the minimum and maximum verified bounds discovered during the 20 experiments. The bottom and top of each whisker’s box show the first and third quartiles of the verified bounds.

Our results indicate that MMSTV provides a significant increase in provable robustness against the FGSM attack. In all 20 experiments, the verified bound for the MMSTV network was larger than those found for the Original, GSS, and Ensemble networks. We observe that GSS and Ensemble defend the network in a way that makes it only slightly more provably robust, consistent with observations that these styles of defense are insufficient [41, 77].

2.8 RELATED WORK

In this section, we survey the works closely related to ours.

Adversarial Examples. [16] showed that neural networks are vulnerable to small perturbations on inputs. Since then, many works have focused on constructing adversarial examples. For example, [17] showed how to find adversarial examples without starting from a test point, [18] generated adversarial examples using random perturbations, [78] demonstrated that even intermediate layers are not robust, and [19] generated adversarial examples for malware classification. Other works presented ways to construct adversarial examples during the training phase, thereby increasing the network robustness (see [20–25]). [47] formalized the notion of robustness in neural networks and defined metrics to evaluate the robustness of a neural network. [26] illustrated how to systematically generate adversarial examples that cover all neurons in the network.

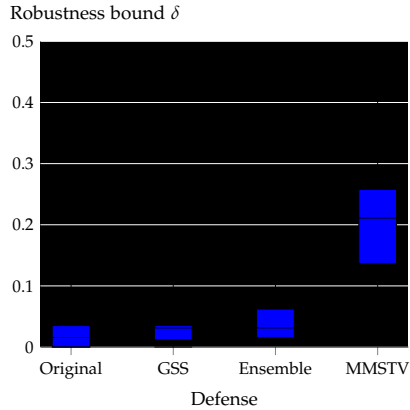


FIGURE 2.13: Box-and-whisker plot of the verified bounds for the Original, GSS, Ensemble, and MMSTV networks. The boxes represent the δ for the middle 50% of the images, whereas the whiskers represent the minimum and maximum δ . The inner-lines are the averages.

Neural Network Analysis. Many works have studied the robustness of networks. [63] presented an abstraction-refinement approach for FNNs. However, this was shown successful for a network with only 6 neurons. [76] introduced a bounded model checking technique to verify safety of a neural network for the Cart Pole system. [27] showed a verification framework, based on an SMT solver, which verified the robustness with respect to a certain set of functions that can manipulate the input and are *minimal* (a notion which they define). However, it is unclear how one can obtain such a set. [7] extended the simplex algorithm to verify properties of FNNs with ReLU. [79] showed lower bounds on the norm of the input manipulation required to fool a network, but is tested only on a one-hidden layer network with 1024 units.

Robustness Analysis of Programs. Many works deal with robustness analysis of programs (e.g., [80–83]). [80] considered a definition of robustness that is similar to the one in our work, and [81] used a combination of abstract interpretation and SMT-based methods to prove robustness of programs. The programs considered in this literature tend to be small but have complex constructs such as loops and array operations. In contrast, neural networks (which are our focus) are closer to circuits, in that they lack high-level language features but are potentially massive.

2.9 CONCLUSION AND FUTURE WORK

We presented AI², the first system able to certify convolutional and large fully connected networks. The key insight behind AI² is to phrase the problem of analyzing neural networks in the classic framework of abstract interpretation. To this end, we defined abstract transformers that capture the behavior of common neural network layers and presented a bounded powerset domain that enables a trade-off between precision and scalability.

Our experimental results showed that AI² can effectively handle neural networks that are beyond the reach of existing methods.

We believe AI², and the approach behind it, is a promising step towards ensuring the safety and robustness of AI systems. While not included in the scope of this thesis, we have extended AI² with additional abstract transformers to support more neural network features, and provided more accurate domains that can scale to larger networks. We believe these extensions would further improve AI²'s applicability and foster future research in AI safety.

ABSTRACT INTERPRETATION FOR TRAINING CERTIFIABLE NETWORKS

In this chapter, we introduce the first scalable method for training certifiably robust neural networks, *DIFFAI*. The key insight in this chapter is that the analysis techniques outlined in Chapter 2 can be used directly as a training goal that may be optimized (trained) with backpropagation.

We first present several abstract transformers that balance efficiency with precision, designed specifically to be used during training, and show that these techniques scale to train large networks that are certifiably robust to adversarial perturbations.

We then present our public implementation of *DIFFAI* and provide a detailed evaluation. Our results show that (i) *DIFFAI* can efficiently train orders of magnitude larger network than prior methods, (ii) *DIFFAI* can quickly verify orders of magnitude larger networks than prior methods, and finally (iii) *DIFFAI* can produce networks which are vastly more provably robust.

3.1 INTRODUCTION

As heuristic defenses are insufficient to ensure safety, works such as Gehr *et al.* [54] provided methods for certifying local robustness properties of neural networks. Wong & Kolter [84] demonstrated a defense against adversarial attacks that provides certificates proving that none of the training examples could be adversarially permuted, as well as providing bounds on the capability of an adversary to influence performance on a test set. This method is based on computing an overapproximation to the *adversarial polytope*, which describes the set of possible neural network outputs given the region of possible inputs. However, this approach incurs significant accuracy and scalability overheads. Prior work by Raghunathan, Steinhardt & Liang [85] provides certifiable robustness, but only for neural networks consisting of two layers. Thus, developing techniques to train large neural networks that can be automatically certified free of robustness violations remains a fundamental challenge.

This Work: DiffAI - Abstract Interpretation for Network Training. We address the above challenge by leveraging the classic framework of *abstract interpretation* [64], a general theory for approximating a potentially infinite set of behaviors with a finite representation, as described in further detail in Chapter 2. This theory has been widely used over the last 40 years to build large-scale automatic code analyzers [86]. We show how to bridge abstract interpretation and gradient-based optimization and how to apply these concepts to train larger networks. Concretely, we compute an approximation to the adversarial polytope and use this approximation as part of our loss function, effectively training the network on entire regions of the input space at once. This *abstract loss* has the advantage of being optimizable via standard techniques such as gradient descent and, as we demonstrate, networks trained in this manner are more provably robust.

Main Contributions. Our main contributions are:

- A new method for training neural networks based on abstract interpretation (Sections 3.2.1 and 3.4).
- Novel abstract transformers for the zonotope domain which are parallelizable and suited for differentiation and gradient descent (Sections 3.2.2 and 3.3).
- A complete implementation of the method in a system called DIFFAI¹ together with an extensive evaluation on a range of datasets and architectures. Our results show that DIFFAI improves provability of robustness and scales to large networks (Section 3.6).

3.2 BACKGROUND: VERIFICATION FOR NEURAL NETWORKS

In this section we review and formally define the concept of robustness and discuss an approach to robustness verification via sound, computable approximations.

3.2.1 Robustness and Sound Approximations

Let $N_\theta: \mathbb{R}^d \rightarrow \mathbb{R}^k$ be a neural network with d input features and k output classes, parameterized by weights θ . The network N_θ assigns the class $i \in \{1, \dots, k\}$ to the point $x \in \mathbb{R}^d$ if $N_\theta(x)_i > N_\theta(x)_j$ for all $j \neq i$.

¹ Available at: <http://diffai.ethz.ch>

Let $B_\epsilon(x)$ denote the ℓ_∞ -ball of radius ϵ around a point $x \in \mathbb{R}^d$. A network N_θ is called ϵ -robust around a point $x \in \mathbb{R}^d$ if N_θ assigns the same class to all points $\tilde{x} \in B_\epsilon(x)$.

More generally, a network N_θ is called π -robust around x if it assigns the same class to all points in $\tilde{x} \in \pi(x)$, where $\pi: \mathbb{R}^d \rightarrow \mathcal{P}(\mathbb{R}^d)$ describes the capabilities of an attacker. In particular, N_θ is ϵ -robust if it is π -robust for $\pi = B_\epsilon$.

Given a set of labeled training examples $\{(x_i, y_i)\}_{i=1}^n$, the goal of *adversarial training* is to find a θ such that: (i) N_θ assigns the correct class y_i to each example x_i , and (ii) N_θ is π -robust around each example x_i .

Definition 3.2.1. Given a loss function $L(z, y)$ which is non-negative if $\arg \max_i z_i \neq y$ and strictly negative if $\arg \max_i z_i = y$, we define the *worst-case adversarial loss* L_N on a labeled example (x, y) with respect to network N :

$$L_N(x, y) = \max_{\tilde{x} \in \pi(x)} L(N(\tilde{x}), y).$$

Intuitively, the worst-case adversarial loss is the maximal loss that an attacker can obtain by perturbing the example x to an arbitrary $\tilde{x} \in \pi(x)$. For a given labeled example (x, y) , we call a point $\tilde{x} \in \pi(x)$ that maximizes $L(N(\tilde{x}), y)$ a *worst-case adversarial perturbation*.

Using the worst-case adversarial loss, we can formulate adversarial training as the following optimization problem:

$$\min_{\theta} \max_i L_{N_\theta}(x_i, y_i).$$

If the value of the solution is negative, N_θ classifies all training examples correctly and is π -robust around all points in the training set.

Optimizing Over Sound Approximations. It is usually very difficult to find a worst-case adversarial perturbation in the ϵ -ball around an example. Madry *et al.* [41] strengthen the network using a heuristic approximation of the worst-case adversarial perturbation and show that the method is practically effective. Wong & Kolter [84] propose an approach where a superset of the possible classifications for a particular example is determined and the optimization problem is stated in terms of this overapproximation. At a high level, we take a similar approach, but we introduce sound approximations that scale to larger networks and are easier to work with.

Definition 3.2.2. A sound approximation of a given function $f: \mathbb{R}^d \rightarrow \mathbb{R}^k$ under perturbations $\pi: \mathbb{R}^d \rightarrow \mathcal{P}(\mathbb{R}^d)$ is a function $\mathcal{A}_{f,\pi}: \mathbb{R}^d \rightarrow \mathcal{P}(\mathbb{R}^k)$, such that for every $x \in \mathbb{R}^d$, we have that $f(\pi(x)) \subseteq \mathcal{A}_{f,\pi}(x)$.

To avoid clutter, we usually write $f(S)$ (as in the above definition) as shorthand for the image of $S \subseteq \mathbb{R}^d$ under f .

Sound approximations can be used to prove robustness properties: for example, if we can show that all values $\tilde{z} \in \mathcal{A}_{N,B_\epsilon}(x)$ share the same $\arg \max_i \tilde{z}_i$, then the network N is ϵ -robust around the point x .

Definition 3.2.3. Given a sound approximation $\mathcal{A}_{N,\pi}$, the *approximate worst-case adversarial loss* L_N^A is given by

$$L_N^A(x, y) = \max_{\tilde{z} \in \mathcal{A}_{N,\pi}(x)} L(\tilde{z}, y).$$

The worst-case adversarial loss L_N can be expressed equivalently as L_N^A using $\mathcal{A}_{N,\pi}(x) = N(\pi(x))$. Therefore, as by definition, we have $N(\pi(x)) \subseteq \mathcal{A}_{N,\pi}(x)$, it follows that $L_N(x, y) \leq L_N^A(x, y)$.

This means that if we choose \mathcal{A} such that: (i) we can compute L_N^A , and (ii) we can find θ where $\max_i L_{N_\theta}^A(x_i, y_i)$ is negative, then we have proven the neural network N_θ correct and ϵ -robust for the entire training set.

While the above does not imply that N_θ is ϵ -robust on the test set, we find that approximating the optimal θ in

$$\min_{\theta} \max_i L_{N_\theta}^A(x_i, y_i)$$

produces networks N_θ that can often be proven robust around previously unseen test examples using the sound approximation \mathcal{A} . That is, provable robustness generalizes.

3.2.2 Abstract Interpretation

We will approximate neural networks using *abstract interpretation* [64]. Abstract interpretation has been recently used to certify robustness of neural networks [54]. It has also been used to find constants in small programs using black-box optimization by Chaudhuri, Clochard & Solar-Lezama [87], who approximate abstract transformers for a particular probabilistic domain by families of continuous functions. We now introduce the necessary general concepts and our specific instantiations.

Definition 3.2.4. An *abstract domain* \mathcal{D} is a set equipped with an *abstraction function* $\alpha: \mathcal{P}(\mathbb{R}^p) \rightarrow \mathcal{D}$ and a *concretization function* $\gamma: \mathcal{D} \rightarrow \mathcal{P}(\mathbb{R}^p)$ for some $p \in \mathbb{N}$.

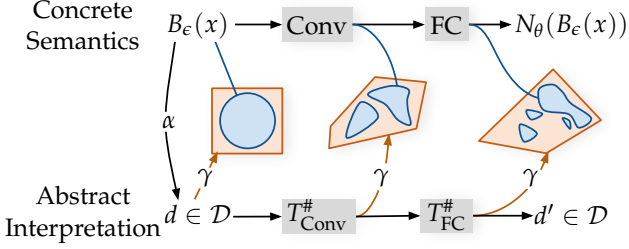


FIGURE 3.1: A visualization of abstract interpretation applied on a neural network with convolutional and fully connected layers.

Intuitively, an element $d \in \mathcal{D}$ corresponds to a set of symbolic constraints over \mathbb{R}^p and $\gamma(d)$ determines the set of points that satisfy the constraints d . The abstraction function α is defined such that $X \subseteq \gamma(\alpha(X))$ for each $X \subseteq \mathbb{R}^p$.

Definition 3.2.5. A (computable) function $T_f^\# : \mathcal{D} \rightarrow \mathcal{D}'$ is called an *abstract transformer* for a function $f : \mathbb{R}^p \rightarrow \mathbb{R}^{p'}$ if $f(\gamma(d)) \subseteq \gamma'(T_f^\#(d))$ for all $d \in \mathcal{D}$.

Intuitively, an abstract transformer $T_f^\#$ overapproximates the behavior of a function f on a set $\gamma(d)$ by operating directly on a symbolic representation $d \in \mathcal{D}$ to produce a new abstract element $d' \in \mathcal{D}'$ whose concretization overapproximates the image of f under $\gamma(d)$.

Abstract transformers compose: If $T_f^\#$ and $T_g^\#$ are abstract transformers for functions f and g , then $T_f^\# \circ T_g^\#$ is an abstract transformer for $f \circ g$. Therefore, it suffices to define abstract transformers for each basic operation in a neural network N . If we then write the neural network as a composition of basic operations, we can immediately derive an abstract transformer $T_N^\#$ for the entire N . This abstract transformer $T_N^\#$ induces a sound approximation $\mathcal{A}_{N,\pi}(x) = \gamma(T_N^\#(\alpha(\pi(x))))$ in the sense of Definition 3.2.2.

We apply abstract interpretation to compute $T_N^\#(\alpha(\pi(x)))$, which describes a superset of the possible outputs of the neural network N under perturbations π . Figure 3.1 illustrates how abstract interpretation overapproximates the behavior of a network on a concrete set of inputs $\pi(x) = B_\epsilon(x)$. For each intermediate abstract result, γ yields a superset of the corresponding intermediate concrete result (analysis always happens in the abstract, γ is only used to ensure soundness).

We next discuss the abstract domains we use in this chapter.

Interval Domain. The simplest domain we consider is the *interval* (also called Box) domain. Abstract interpretation using this domain is equivalent to computation using standard interval arithmetic. Each element of the domain represents a p -dimensional box, described by its center and deviations in each component. We use this representation for two reasons: (i) it makes the transformations for multiplication and addition efficiently parallelizable on a GPU, and (ii) it exposes essential relationships between our presentation of interval and zonotope domains (discussed below).

An element of the domain is a pair $b = \langle b_C, b_B \rangle$ where $b_C \in \mathbb{R}^p$ is the center of the box, while $b_B \in \mathbb{R}_{\geq 0}^p$ describes (non-negative) deviations. The concretization function γ_I is:

$$\gamma_I(b) = \{b_C + \text{diag}(b_B) \cdot \beta \mid \beta \in [-1, 1]^p\}.$$

Here, $\text{diag}(b_B)$ creates a diagonal $p \times p$ matrix where the entries on the diagonal are those of b_B , and β is an *error vector* used to pick a particular element of the concretization.

Definition 3.2.6. The *total error* of the i -th component of a box b is $\epsilon_I(b)_i = (b_B)_i$ and the *interval concretization* of the i -th component of b is given by

$$\iota_I(b)_i = [(b_C)_i - \epsilon_I(b)_i, (b_C)_i + \epsilon_I(b)_i].$$

Zonotope Domain. Interval arithmetic can be imprecise as it does not keep information on how values of variables are related. The *zonotope* domain [68] aims to preserve some of these relationships. Unlike the interval domain, where each error term is associated to a particular component, the zonotope domain freely shares error terms among components. In this way, some amount of dependency information can be encoded at moderate costs. The most important feature of the zonotope domain is that there exists an abstract transformer for affine functions (such as the transition function of a fully connected or convolutional layers) which does not lose precision.

An element of the zonotope domain is a pair $z = \langle z_C, z_E \rangle$ where $z_C \in \mathbb{R}^p$ is the center of the zonotope, while $z_E \in \mathbb{R}^{p \times m}$ describes a linear relationship between the error vector $e \in [-1, 1]^m$ and the output components (for arbitrary m). The concretization function γ_Z is given by

$$\gamma_Z(z) = \{z_C + z_E \cdot e \mid e \in [-1, 1]^m\}.$$

Definition 3.2.7. The *total error* of the i -th component of a zonotope z is $\epsilon_Z(z)_i = \sum_{j=1}^m |(z_E)_{i,j}|$ and the *interval concretization* of the i -th component of z is given by

$$\iota_Z(z)_i = [(z_C)_i - \epsilon_Z(z)_i, (z_C)_i + \epsilon_Z(z)_i].$$

The zonotope domain is strictly more expressive than the interval domain: for a box b , its corresponding zonotope z is given by $z_C = b_C$, $z_E = \text{diag}(b_B)$.

Hybrid Zonotope Domain. While the zonotope domain is more precise than interval, its transformers are less efficient. The *hybrid zonotope domain*, introduced originally as *perturbed affine arithmetic* by Goubault & Putot [88], aims to address this issue: its transformers are more accurate than interval, but more efficient than zonotope.

An element of this domain is a triple $h = \langle h_C, h_B, h_E \rangle$ where $h_C \in \mathbb{R}^p$ is the center, $h_B \in \mathbb{R}_{\geq 0}^p$ contains non-negative deviations for each component (similar to interval domain), and $h_E \in \mathbb{R}^{p \times m}$ describes error coefficients (similar to zonotope domain). The concretization γ_H is given by

$$\gamma_H(h) = \{\widehat{h}(\beta, e) \mid \beta \in [-1, 1]^p, e \in [-1, 1]^m\},$$

where $\widehat{h}(\beta, e) = h_C + \text{diag}(h_B) \cdot \beta + h_E \cdot e$.

Definition 3.2.8. The *total error* of the i -th component of a hybrid zonotope h , is $\epsilon_H(h)_i = (h_B)_i + \sum_{j=1}^m |(h_E)_{i,j}|$, and the *interval concretization* of the i -th component of h is

$$\iota_H(h)_i = [(h_C)_i - \epsilon_H(h)_i, (h_C)_i + \epsilon_H(h)_i].$$

This domain is equally expressive as the zonotope domain but can represent interval constraints more efficiently. Further, the abstract transformers of this domain treat a hybrid zonotope differently than they would treat a zonotope with the same concretization, due to the deviation coefficients.

A box b can be expressed efficiently as a hybrid zonotope h with $h_C = b_C$, $h_B = b_B$ and $m = 0$. A zonotope z can be expressed as a hybrid zonotope h with $h_C = z_C$, $h_B = 0$ and $h_E = z_E$.

3.3 ABSTRACT TRANSFORMERS FOR ZONOTOPE

We now introduce our abstract transformers for the hybrid zonotope domain, specifically ReLU transformers which balance precision with

scalability. The transformers are “point-wise”: they are efficiently executable on a GPU and can benefit both training and analysis of the network.

There are three basic types of abstract transformers, those that: (i) increase the deviations, (ii) introduce new error terms producing a hybrid zonotope h' with $m' > m$, and (iii) handle deviations and error coefficients separately and do not introduce new error terms. We first discuss the transformers of type (iii). For this type, the transformers of the interval and zonotope domains arise as special cases.

Addition. We first consider a function f that replaces the i -th component of the input vector $x \in \mathbb{R}^p$ by the sum of the j -th and k -th components:

$$f(x) = (x_1, \dots, x_{i-1}, x_j + x_k, x_{i+1}, \dots, x_p)^T.$$

The corresponding abstract transformer is given by

$$T_f^\#(h) = \langle M \cdot h_C, M \cdot h_B, M \cdot h_E \rangle,$$

where the matrix $M \in \mathbb{R}^{p \times p}$ is such that $M \cdot x$ replaces the i -th row of x by the sum of the j -th and k -th rows.

Multiplication. Consider a function f that multiplies the i -th component of the input vector $x \in \mathbb{R}^p$ by a known constant $\kappa \in \mathbb{R}$:

$$f(x) = (x_1, \dots, x_{i-1}, \kappa \cdot x_i, x_{i+1}, \dots, x_p)^T.$$

The abstract transformer $T_f^\#$ for this operation is simple and does not lose any precision. We define

$$T_f^\#(h) = \langle M_\kappa \cdot h_C, M_{|\kappa|} \cdot h_B, M_\kappa \cdot h_E \rangle,$$

where $M_\alpha = I + \text{diag}((\alpha - 1) \cdot e_i)$. Here, I is the identity matrix and e_i is the i -th standard basis vector.

Matrix Multiplication and Convolution. Consider a function f that multiplies the input vector $x \in \mathbb{R}^p$ by a known matrix $M \in \mathbb{R}^{p' \times p}$:

$$f(x) = M \cdot x.$$

Combining the insights from the addition and multiplication transformers, the abstract transformer $T_f^\#$ is given by

$$T_f^\#(h) = \langle M \cdot h_C, |M| \cdot h_B, M \cdot h_E \rangle.$$

Here, $|M|$ simply performs component-wise absolute value operation. The above transformer can be easily differentiated and parallelized on the GPU. As convolutions are linear operations, the same approach can be applied for these.

ReLU. ReLU is a simple nonlinear activation function:

$$\text{ReLU}(x) = \max(x, 0).$$

In contrast to the abstract transformers discussed so far, there is no single best ReLU abstract transformer for the hybrid zonotope domain. Instead, there are many possible, pairwise incomparable, abstract transformers $T_{\text{ReLU}}^\#$.

Abstract domains \mathcal{D} usually support a join operator (\sqcup) such that for all abstract elements $d, d' \in \mathcal{D}$, we have

$$\gamma(d) \cup \gamma(d') \subseteq \gamma(d \sqcup d'),$$

and a meet-with-linear-constraint operator (\sqcap) such that for all linear constraints $L(x) = x_i < 0$ or $L(x) = x_i \geq 0$ for $1 \leq i \leq p$, we have

$$\gamma(d) \cap \{x \mid L(x)\} \subseteq \gamma(d \sqcap L).$$

In this case, we can define the abstract transformer for ReLU in a general way. Let $f_i(x) = (x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_p)$ be the function that assigns 0 to the i -th component of the input vector. The abstract transformer for ReLU can then be defined as

$$T_{\text{ReLU}}^\# = T_{\text{ReLU}_p}^\# \circ T_{\text{ReLU}_{p-1}}^\# \circ \dots \circ T_{\text{ReLU}_2}^\# \circ T_{\text{ReLU}_1}^\#,$$

where $T_{\text{ReLU}_i}^\#(d) = d \sqcap (x_i \geq 0) \sqcup T_{f_i}^\#(d \sqcap (x_i < 0))$.

ReLU_i applies ReLU only to the i -th component of the input vector. While Goubault, Le Gall & Putot [89] demonstrate that accurate join and meet are possible for a hybrid zonotope domain, they take $\Omega(m^3 + m^2 \cdot p)$ time in the worst case, which is significant in the case of deep neural networks. Further, ReLU defined in this way is not parallelizable as the application of ReLU in each *component* is dependent on the application in the previous component.

Point-wise Abstract Transformers for ReLU. We define several novel zonotope transformers for the functions ReLU_i . Each modifies the i -th component of the input zonotope and does not depend on any other component (hence, “point-wise”). As a result, the result can be computed in parallel for all components, enabling scalable ReLU analysis.

Definition 3.3.1 (zBox ReLU transformer). For a zonotope z and index i , we define $z' = T_{\text{ReLU}_i}^{\#(\text{zBox})}(z)$ with $m' = m + 1$. If $\min(\iota_i(z)) \geq 0$, then z' is z with an additional unused error term, meaning new entries in z'_E are set to 0. Otherwise

$$\begin{aligned} (z'_X)_t &= (z_X)_t, \text{ for } X \in \{C, E\}, t \neq i, \\ (z'_C)_i &= \text{ReLU}(\tfrac{1}{2} \max(\iota(z)_i)), \\ (z'_E)_{i,l} &= 0, \text{ for } l \leq m, \\ (z'_E)_{i,m+1} &= \text{ReLU}(\tfrac{1}{2} \max(\iota(z)_i)), \\ (z'_E)_{j,m+1} &= 0, \text{ for } j \neq i. \end{aligned}$$

$T_{\text{ReLU}_i}^{\#(\text{zBox})}(z)$ propagates the input zonotope unchanged if it can prove the i -th component is non-negative (then ReLU_i has no effect). Otherwise, it bounds the i -th component of the output by a suitable independent interval using the new error term. In both cases, the number of error terms in z' is the same, allowing for more effective parallelization.

We also define a transformer which uses the above transformer with incomparable precision.

Definition 3.3.2 (zDiag ReLU transformer). Given a zonotope z and index i , we define a ReLU abstract transformer $z' = T_{\text{ReLU}_i}^{\#(\text{zDiag})}(z)$ where $m' = m + 1$. If the condition $\min(\iota(z)_i) < 0 < \max(\iota(z)_i)$ holds, we have

$$\begin{aligned} (z'_X)_t &= (z_X)_t \text{ for } X \in \{C, E\}, t \neq i, \\ (z'_C)_i &= (z_C)_i - \tfrac{1}{2} \min(\iota(z)_i), \\ (z'_E)_{i,l} &= (z_E)_{i,l} \text{ for } l \leq m, \\ (z'_E)_{i,m+1} &= -\tfrac{1}{2} \min(\iota(z)_i), \\ (z'_E)_{j,m+1} &= 0, \text{ for } j \neq i. \end{aligned}$$

Otherwise, $z' = T_{\text{ReLU}_i}^{\#(\text{zBox})}(z)$.

Finally, we define two transformers which combine zBox and zDiag in different ways.

Definition 3.3.3 (zSwitch ReLU transformer). Transformer $T_{\text{ReLU}_i}^{\#(\text{zSwitch})}$ uses $T_{\text{ReLU}_i}^{\#(\text{zBox})}$ if $|\min(\iota(z)_i)| > |\max(\iota(z)_i)|$. Otherwise, it uses $T_{\text{ReLU}_i}^{\#(\text{zDiag})}$.

Definition 3.3.4 (zSmooth ReLU transformer). Transformer $T_{\text{ReLU}_i}^{\#(\text{zSmooth})}$ takes a weighted average of the results of $T_{\text{ReLU}_i}^{\#(\text{zBox})}$ and $T_{\text{ReLU}_i}^{\#(\text{zDiag})}$, with weights $|\min(\iota(z)_i)|$ and $|\max(\iota(z)_i)|$ respectively.

Point-wise ReLU for Hybrid Zonotopes. The hybrid zonotope transformers (hSwitch, hSmooth) operate the same way as the respective zonotope versions (zSwitch, zSmooth), but do not add new error terms. Instead, they accumulate the computed error in the i -th component of h_B .

3.4 ADVERSARIAL TRAINING

We now introduce adversarial training with our domains.

3.4.1 Approximate Worst-Case Adversarial Loss

Let the loss $L(z, y) = \max_{y' \neq y} (z_{y'} - z_y)$. This loss satisfies the requirements for adversarial training from Definition 3.2.1. Let us define our approximate worst-case adversarial loss by instantiating $\mathcal{A}_{N, \pi}(x)$ from Definition 3.2.3 as

$$\mathcal{A}_{N, \pi}(x) = \gamma(T_N^{\#}(\alpha(\pi(x)))).$$

This means that we take the region $\pi(x)$, abstract it so it can be captured in our abstract domain, obtain $\alpha(\pi(x))$, and then apply the neural network transformer $T_N^{\#}$ (as discussed so far) to that result. For $\pi = B_\epsilon$, the expression $\alpha(\pi(x))$ corresponds to a simple interval constraint that can be computed easily for each of the discussed abstract domains. Once we obtain the abstract output of the transformer, we can apply γ to obtain all concrete output points represented by this result. With this instantiation we obtain the loss

$$L_N^A(x, y) = \max_{\tilde{z} \in \gamma(T_N^{\#}(\alpha(\pi(x))))} L(\tilde{z}, y).$$

This is the approximate worst-case adversarial loss when we use abstract interpretation for approximation. We can compute the loss $L_N^A(x, y)$ using

$$L_N^A(x, y) = \max_{y' \neq y} (\max \iota(T_{f_{y'}}^{\#}(T_N^{\#}(\alpha(\pi(x)))))),$$

provided the following conditions hold: (i) an interval concretization function ι exists for which we can compute interval upper bounds $\max \iota(x)$,

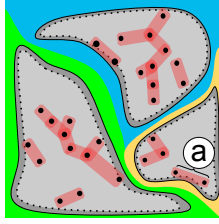


FIGURE 3.2: Visualizing line segment training with 3 classifications.

(ii) we can compute $\alpha(\pi(x_i))$, and (iii) the linear function $f_{y'}(z) = z_{y'} - z_y$ has a precise abstract transformer $T_{f_{y'}}^\#$. All three of these conditions hold for the interval, zonotope and hybrid zonotope domains.

3.4.2 Line Segments as (Hybrid) Zonotopes

We show how to consider input regions in other shapes beyond the standard ℓ_∞ -balls used in local robustness properties. We use the observation that when two points with the same classification in the training set are close enough, the points on the line between them should be classified the same. Figure 3.2 shows an intuitive illustration. Black dots represent data and grey regions represent the ground truth classifications. The network learns to classify points in the yellow, green, and blue regions. Data points only actually appear within a smaller shell within the class boundary (dotted lines). The red lines connecting the points show how these points are grouped into line abstractions. Marker “a” shows a bad scenario, where one of the abstractions comes near the ground truth class boundary. We experimented with encoding the segment between nearby points instead of simple robustness regions, aiming to improve the result of training. In terms of encoding, with a zonotope, we can conveniently represent a line segment between two points x and y as follows:

$$(z_C)_i = \frac{1}{2}(x_i + y_i), (z_E)_{i,1} = \frac{1}{2}|x_i - y_i|.$$

For hybrid zonotopes, we additionally consider a width parameter w :

$$(h_C)_i = \frac{1}{2}(x_i + y_i), (h_B)_i = w, (h_E)_{i,1} = \frac{1}{2}|x_i - y_i|.$$

3.5 EXPERIMENTAL SETUP

We implemented our approach in a system called DIFFAI² and evaluated it extensively across a range of datasets and network architectures. We demonstrate that DIFFAI training scales to networks larger than those of prior work and that networks trained with DIFFAI are more provably robust than those trained with state-of-the-art defenses.

Our system is built on top of PyTorch [90]. For all of our experiments, we used the Adam Optimizer [91], with the default parameters and a learning rate (lr) of 0.0001, unless otherwise specified. Additionally, we used norm-clipping on the weights after every batch with a max-norm of 10,000. For training we use a batch size of 500. For testing, batch sizes vary from network to network depending on the experiment and the GPU used. We ran on a GeForce GTX 1080 Ti, and a K80. For all comparisons on a single network and dataset (shown in the tables), we used the same initial weights and presented the models with the same batches. In all experiments, accuracy and provability is tested on 500 samples.

3.5.1 Training

While we did find that it was often possible to train entirely using the loss function described in Section 3.4.1, we achieved better results by combining it with a standard cross-entropy loss function during training. Furthermore, we found that the loss in Section 3.4.1 continued to provide gradient information to the optimizer long after the example was proven correct, which was detrimental to learning for other examples. We found that applying a smooth version of ReLU – softplus [92] – to this loss helped to avoid this issue. Our combined loss is therefore

$$\mathcal{L}_\theta(x, y) = \lambda \cdot \text{softplus}(L_{N_\theta}^A(x, y)) + H(\text{softmax}(N_\theta(x)), y),$$

where λ is set to 0.1 in all of our experiments (except line training). The total loss for a batch is obtained by *adding* the losses for all examples in the batch.

Datasets.

We evaluate DIFFAI on four different datasets: MNIST, CIFAR10, FashionMNIST (F-MNIST) and SVHN in the fashion seen in Table 3.1. Before analysis, we normalize the inputs as described by LeCun *et al.* [95],

² Available at: <http://diffai.ethz.ch>

Dataset	# Train	Inp Dim	μ	σ
MNIST [52]	60000	$1 \times 28 \times 28$	0.1307	0.3081
F-MNIST [93]	60000	$1 \times 28 \times 28$	Not Normalized	Not Normalized
CIFAR ₁₀ [74]	50000	$3 \times 32 \times 32$	[0.4914, 0.4822, 0.4465]	[0.2023, 0.1994, 0.2010]
SVHN [94]	73257	$3 \times 32 \times 32$	[0.5, 0.5, 0.5]	[0.2, 0.2, 0.2]

TABLE 3.1: The datasets we evaluate with. All use 10 classifications.

using an approximated mean μ and standard deviation σ per channel as $\frac{X-\mu}{\sigma}$.

Parameters. For all experiments (except with segment training), we trained with an L_2 regularization constant of 0.01, and a λ of 0.1. We halt training after 200 epochs. For MNIST, we used a learning rate of 10^{-3} . For all other experiments, the learning rate was 10^{-4} . For both testing and training, we used the untargeted PGD attack with $k = 5$ iterations. At testing time, we used 500 examples.

3.5.2 Neural Networks Evaluated

We train six networks: one feed forward, four convolutional (without maxpool), and one with a residual connection. In the following descriptions, we use $\text{Conv}_s C \times W \times H$ to mean a convolutional layer that outputs C channels, with a kernel width of W pixels and height of H , with a stride of s which then applies ReLU to every output. FC n is a fully connected layer which outputs n neurons without automatically applying ReLU.

FFNN. A 5 layer feed forward net with 100 nodes in each and a ReLU after each layer. This network has a ReLU after the last layer.

ConvSmall. Our smallest conv-net with no convolutional padding.

$$x \rightarrow \text{Conv}_2 16 \times 4 \times 4 \rightarrow \text{Conv}_2 32 \times 4 \times 4 \rightarrow \text{FC } 100 \rightarrow z.$$

ConvMed. The same as ConvSmall, but with a padding of 1.

$$x \rightarrow \text{Conv}_2 16 \times 4 \times 4 \rightarrow \text{Conv}_2 32 \times 4 \times 4 \rightarrow \text{FC } 100 \rightarrow z.$$

ConvBig. A significantly larger conv-net with a padding of 1.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 32 \times 3 \times 3 \rightarrow \text{Conv}_2 32 \times 4 \times 4 \\ &\rightarrow \text{Conv}_1 64 \times 3 \times 3 \rightarrow \text{Conv}_2 64 \times 4 \times 4 \\ &\rightarrow \text{FC } 512 \rightarrow \text{ReLU} \rightarrow \text{FC } 512 \rightarrow z. \end{aligned}$$

ConvSuper. Our largest convolutional network with no padding.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 32 \times 3 \times 3 \rightarrow \text{Conv}_1 32 \times 4 \times 4 \\ &\rightarrow \text{Conv}_1 64 \times 3 \times 3 \rightarrow \text{Conv}_1 64 \times 4 \times 4 \\ &\rightarrow \text{FC } 512 \rightarrow \text{ReLU} \rightarrow \text{FC } 512 \rightarrow z. \end{aligned}$$

Skip. Two convolutional networks of different sizes, which are then concatenated together. This network uses no convolutional padding.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 16 \times 3 \times 3 \\ &\rightarrow \text{Conv}_1 16 \times 3 \times 3 \\ &\rightarrow \text{Conv}_1 32 \times 3 \times 3 \rightarrow \text{FC } 200 \rightarrow o_1, \\ x &\rightarrow \text{Conv}_1 32 \times 4 \times 4 \\ &\rightarrow \text{Conv}_1 32 \times 4 \times 4 \rightarrow \text{FC } 200 \rightarrow o_2, \\ \text{CAT}(o_1, o_2) &\rightarrow \text{ReLU} \rightarrow \text{FC } 200 \rightarrow \text{ReLU} \rightarrow z. \end{aligned}$$

Dataset	Model	Type	# Hidden Units	# Parameters	Train Time (s/epoch)		Total Testing Time (s)	
					Baseline	Box	Box	hSwitch
MNIST	FFNN	fully connected	510	119910	0.610	2.964	0.076	0.184
	ConvSmall	convolutional	3604	89606	0.560	4.014	0.056	0.360
	ConvBig	convolutional	34688	893418	1.839	7.229	0.060	7.431
	ConvSuper	convolutional	88500	10985962	4.391	15.743	0.080	12.856
	Skip	residual	71600	6301890	3.703	12.613	0.073	11.313
CIFAR10	FFNN	fully connected	510	348710	1.273	4.145	0.066	1.018
	ConvSmall	convolutional	4852	125318	0.718	3.979	0.065	1.870
	ConvMed	convolutional	6244	214918	1.462	5.200	0.051	1.953
	ConvBig	convolutional	62464	2466858	6.585	21.539	0.062	11.372
	ConvSuper	convolutional	124128	16229418	23.416	74.247	0.089	40.270
	Skip	residual	97730	8760802	14.245	42.482	0.083	25.198

TABLE 3.2: A table showing the size of our networks, the time it takes to train one epoch (averaged over 200 epochs), and the best total testing time for 500 samples, with the maximum batch size allowable by the GPU memory for each domain and network combination. The testing times are for a baseline-trained network, the times for a Box-trained network are similar.

3.6 EXPERIMENTAL RESULTS

In this section, we demonstrate that:

- Training with DIFFAI is efficient and scalable.
- Testing with DIFFAI is efficient and scalable.
- Segment training has potential to improve accuracy.

3.6.1 Results against Prior Defenses and Analyzers

We evaluated the performance of DIFFAI using ℓ_∞ -balls and Box training against standard baseline training. We also trained with the state-of-the-art defense of Madry *et al.* [41], which permutes each batch using the untargeted PGD attack. The *adversarial test error* is the largest error an adversary can achieve by perturbing all examples in the test set. As we cannot efficiently compute the adversarial test error, we instead give lower and upper bounds using the PGD attack and DIFFAI respectively. Some of our results are shown in Table 3.3 (all results are in the supplementary). **Scalability of Training.** To our knowledge, we analyzed and defended the

largest networks considered so far in the context of provable robustness, in terms of both number of neurons and weights. As shown in Table 3.2, we were able to train a network (ConvSuper on CIFAR10) with 124000 neurons and over 16 million weights in under 75 seconds per epoch for a total time of less than 5 hours. The net trained is larger than the largest considered by Wong & Kolter [84], who took 10 hours to train a significantly smaller network, and do not report stand-alone testing speed. Often, DIFFAI’s Box training is even faster than PGD training with 5 iterations. For ConvSuper on CIFAR10 in Table 3.3, Box took under 4.5 hours to train, while PGD took over 8 hours.

Scalability of Testing. DIFFAI can also analyze large networks: for a given example, it can verify ConvSuper on CIFAR10 in under 2×10^{-4} seconds with Box and 0.1 seconds with hSwitch. This is an order of magnitude speed-up over the current state-of-the-art [54].

Dataset	ϵ	lr	Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
							PGD	Box	hSwitch	
MNIST	0.1	10^{-3}	ConvBig	Baseline	367.80	0.8	3.0	100.0	100.0	
				PGD	1847.76	0.2	1.6	100.0	99.8	
				Box	1445.76	1.0	2.4	14.0	3.4	
			ConvSuper	Baseline	878.28	1.6	2.4	100.0	97.2	
				PGD	4867.56	1.2	1.6	100.0	88.8	
				Box	3148.68	1.0	2.8	11.8	3.6	
			Skip	Baseline	731.40	1.4	3.8	100.0	100.0	
				PGD	3935.04	1.0	2.0	100.0	83.4	
				Box	2734.44	1.6	4.4	13.6	5.8	
CIFAR10	0.007	10^{-4}	ConvBig	Baseline	1317.00	32.4	36.2	100.0	100.0	
				PGD	8574.72	31.4	35.8	100.0	100.0	
				Box	4307.88	55.0	58.6	76.4	61.4	
			ConvSuper	Baseline	4683.24	37.4	42.4	100.0	100.0	
				PGD	29828.52	35.4	41.0	100.0	100.0	
				Box	14849.40	52.8	59.2	83.8	64.2	
			Skip	Baseline	802.92	36.8	41.8	100.0	88.0	
				PGD	4828.68	33.6	40.2	100.0	82.8	
				Box	2980.92	38.0	45.4	72.2	47.8	

TABLE 3.3: Results on time, test error, and adversarial bounds after 200 epochs with L_2 regularization constant of 0.01 and 5 PGD iterations.

Applicability to Complex Nets. We also trained and tested a network, Skip, with a residual connection [96] for all datasets. Scalability results for MNIST and CIFAR10 are shown in Table 3.2, and the adversarial performance of Box training is shown in Table 3.3. Additional results can be found in Appendix A.1. While Skip is quite wide, it is only 5 layers deep with only one residual connection, using concatenation instead of addition.

Provability. PGD defense tends to slightly improve accuracy over baseline and those networks are typically less attackable by a PGD attack than box-trained ones. However, box-trained networks are more provably robust (via Box or hSwitch) than PGD-defended networks. Table 3.3 also shows that Box training produces more provably robust networks than baseline and often, with little loss of accuracy.

DIFFAI achieved consistently below 4% test error on the MNIST benchmark for convolutional networks, as can be seen in Table 3.3 (and in supplementary). When trained using Box, ConvSuper achieves 1% test error and DIFFAI can prove an upper bound of 3.6% on the adversarial test error, close to the lower bound of 2.8% given by PGD. In contrast, baseline training produced a network which is less accurate and could not be proved robust for any test example.

In both Table 3.3 and Table 3.4, hSwitch and zSwitch always produce better upper bounds than Box. In theory, these domains are incomparable to Box, however, in practice, they are typically significantly more precise. As testing with Box is essentially free, and hSwitch is quite efficient, we suggest testing with both and selecting the lower value.

Training with Accurate Domains. Occasionally, training with Box led to much lower accuracy. In these cases, we attempted to improve the accuracy by instead training with the more accurate (and more expensive) hSmooth domain. Results can be seen in Table 3.4, where for example the FFNN network for F-MNIST has 91.4% testing error when trained with Box and 15.6% error when trained with hSmooth.

As training with hSmooth is significantly less space efficient and cannot be done with batches on convolutional networks, a smaller network was used, ConvSmall, to demonstrate this point. In CIFAR10 for example, hSmooth produced a network nearly as accurate as ConvSuper trained using baseline. This was accomplished with little reduction to provable robustness (when testing with zSwitch).

Dataset	ϵ	Epochs	Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
							PGD	Box	zSwitch	
F-MNIST	0.1	200	FFNN	Baseline	119	5.4	98.8	100.0	100.0	
				Box	608	91.4	91.4	100.0	100.0	
				hSmooth	4316	15.6	71.8	100.0	79.0	
CIFAR10	0.03	20	ConvSmall	Baseline	572	35.0	54.8	100.0	83.8	
				Box	999	44.2	56.4	77.6	63.0	
				hSmooth	36493	38.0	53.6	99.8	62.6	
SVHN	0.01	20	ConvSmall	Baseline	700	15.8	83.6	100.0	98.0	
				Box	1223	27.0	78.4	92.6	89.8	
				hSmooth	43859	19.6	78.4	98.4	89.0	

TABLE 3.4: Results showing that training with hSmooth can lead to improved accuracy (and upper bounds) over training with Box.

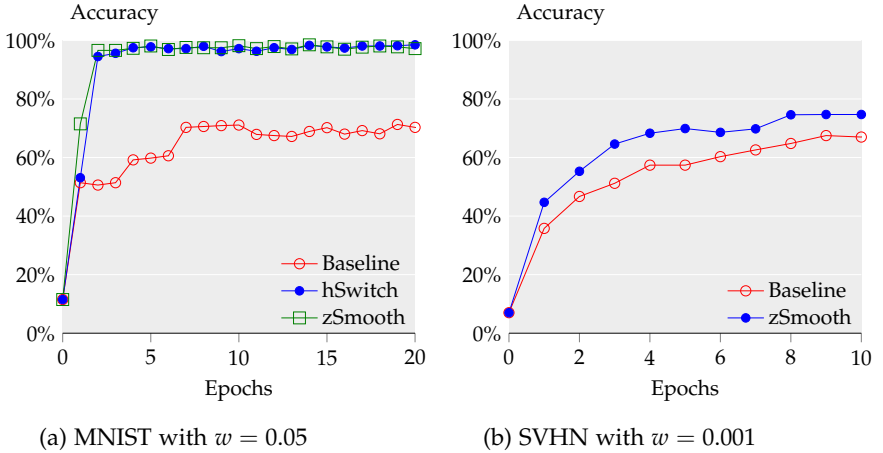


FIGURE 3.3: Accuracy of segment training on FFNN. No regularization and learning rate of 0.001. (a) Batch size 200 for Baseline and 20 for hSwitch and zSmooth; λ of 10^5 . (b) Batch size 150 for Baseline and 30 for zSmooth. λ scheduled with a power of 10 over 10^5 examples starting at 10^{-6} and ending at 10^{-4} .

3.6.2 Segment Training for Higher Accuracy

To test DIFFAI on more complex abstract regions, we trained with line segments connecting examples, as described in Section 3.4.2. For every element in a batch, we built a zonotope connecting it to the nearest (in terms of ℓ_2 distance) other element in the batch with the same class.

The plot in Figure 3.3 demonstrates that line segment training improves accuracy. After 10 epochs and 1.5 hours, hSwitch with line segment training reached the highest accuracy of 74.7% for SVHN. After 20 epochs and 10 hours, zSwitch and hSwitch both reached an accuracy of 97.4% for MNIST, significantly higher than the 70% achieved by the baseline.

3.7 OUTLOOK

At the same time as these techniques were developed, an alternate approach was published in Wong & Kolter [84] which uses duality from optimization theory to find bounds on provability and perform training. Here, and in the later work by Wong *et al.* [97], the convex over-approximation of the feasible set is used in an optimization problem, the

dual of which can also be seen as a neural network. The convex relaxations here are similar to those developed in [98].

Since publishing these works, it was discovered by [59] that significant accuracy and certifiability boosts could be achieved by using the Box domain and using a training schedule and refined loss function. Further works have been developed based mostly on these techniques [99–101].

Since these results were published however, progress has stagnated and provable training methods remain far from being able to produce networks with certifiability anywhere near state of the art accuracy on datasets such as CIFAR10. This dilemma has raised fundamental theoretical questions analogous to those that have been answered for standard training: Do certifiable networks exist? How large they would need to be? Can gradient descent efficiently find them?

Baader, Mirman & Vechev [61] answered the first of these questions by proving an analog to the universal approximation theorem, demonstrating that networks exist which can be certified by interval to a specified degree which approximate any function. Further work by Wang *et al.* [102] showed hardness results for learning interval provably robust classifiers.

One promising approach that potentially circumvents these problems is randomized smoothing developed by Cohen, Rosenfeld & Kolter [103]. Here, the notion of inference with a network is modified such that class label is chosen by vote among outputs made by running the network on inputs drawn from distributions around the input. This allows one to get guarantees with high confidence that the classification was robust. Unlike with over-approximation based methods this guarantee may be wrong with some probability.

3.8 CONCLUSION

In this chapter, we presented DIFFAI and showed how to apply abstract interpretation for defending neural networks against adversarial perturbations in such a way as to also allow efficient certification. We additionally introduced several zonotope transformers which carefully balance precision with scalability. Our results indicate the training approach scales to networks larger than those of prior work and the resulting networks are more provably robust than networks trained with state-of-the-art defenses.

PROBABILISTIC ABSTRACT INTERPRETATION FOR GENERATIVE MODELS

So far we have presented techniques for verifying deterministic guarantees, based on standard abstract interpretation. In this chapter we introduce GENPROVE, the first system to apply *probabilistic* abstract interpretation to neural networks. GENPROVE is also the first system that can verify complex semantically meaningful properties based on generative networks.

Generative networks are powerful models capable of learning a wide range of semantic image transformations such as altering a person’s age or head orientation. In this chapter, we bridge the gap between (i) the well studied but limited norm-based and geometric transformations, and (ii) the rich set of semantic transformations used in practice. This problem is especially hard since the generated images lie on a highly non-convex manifold, preventing the use of existing verifiers. We present a new verifier, called GENPROVE, capable of certifying the rich set of semantic transformations of generative models. GENPROVE provides both sound deterministic and probabilistic guarantees, by capturing non-convex sets of distributions over activation states, while scaling to realistic networks.

4.1 INTRODUCTION

While there has been much progress on certifying deep neural networks for norm-constrained pixel perturbations [7, 54, 59, 85, 97, 98, 104–108] and geometric transformations [57, 109–111], these works only capture a restricted subset of natural changes that can occur in practice. At the same time, to train state-of-the-art deep models, a wide range of rich semantic transformations are often being used to improve accuracy via data augmentation. As such transformations are hard to specify manually, they are often learned directly from data via generative networks [9, 112–116]. As a concrete example, a generative network can be trained so that interpolating between the encodings of the flipped head produces images of intermediate head orientations, as in Figure 4.1. While generative networks provide a compelling way to express such semantic

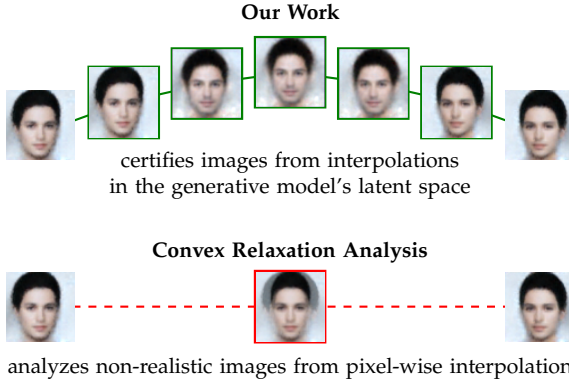


FIGURE 4.1: Example of generated latent space images (□) our work certifies compared to naive pixel-wise interpolation (□).

transformations, they have so far eluded certification due to the scale of non-convexity that they introduce.

This Chapter: GenProve for Generative Specifications. The goal of this chapter is to advance the state-of-the-art in verification by bridging the gap between the norm-based and geometric transformations supported by existing verifiers and the rich set of semantic transformations used in practice. In particular, our verifier can certify a number of rich semantic transformations such as: (i) robustness to addition or removal of image features (e.g., changing shoe color or adding mustache), (ii) baldness is robust to *all* head orientations, and (iii) robustness to higher dimensional specifications that use norm-based perturbations but are applied over the latent space of generative models.

Key Challenge: Non-Convexity of Generative Models. The fundamental technical challenge we address is efficiently handling the non-convexity inherent to generative models, while producing accurate bounds *and* scaling to large networks. We first show that for such complex specifications, deterministic guarantees do not hold frequently enough to warrant certification. We then show that even though it sacrifices soundness, sampling does not scale suitably in this case. We address these challenges by introducing tight approximations when necessary to otherwise exact bound computation. Further, we develop a technique for verifying probabilistic properties, allowing us to produce tight deterministic bounds on the probability of a *probabilistic* specification being

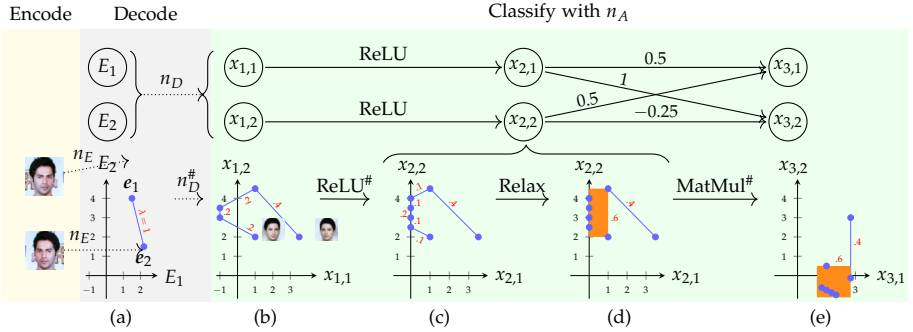


FIGURE 4.2: Using GENPROVE to find probability bounds for latent space interpolation of flipped images. Blue chains represent activation distributions at each layer exactly. The orange boxes represent the relaxation that GENPROVE creates, obviating the need to keep track of the segments it covers. Each segment or box's associated probabilities are shown in red. The inference shown here is faithful to the weights of the toy network in the top row. We provide pseudocode for GENPROVE in Appendix 4.4.2.

satisfied. As we will see in our evaluation, this is a critical component for certifying complex generative specifications for which the equivalent deterministic specification holds only rarely.

Main contributions. Our key contributions are:

- A relaxation technique that handles non-convex behaviors (Section 4.3.1) which allows us to scale to large networks with $\approx 200k$ neurons.
- The first application of probabilistic abstract interpretation [117] to neural networks, which allows us to produce tight deterministic bounds on the probability of a *probabilistic* specification being satisfied (Section 4.4).
- A verifier, GENPROVE, supporting rich semantic transformations, including novel specifications using parametric curves (Section 4.4.3) and higher dimensional specifications (Section 4.5.3).
- A thorough evaluation that shows the practical usability and hardness of the problem – we adapted a number of existing verifiers (Zonotope [54], DeepZono [98], ExactLine [118]) but show they either do not scale to complex settings or their bounds are too imprecise.

4.2 OVERVIEW OF GENPROVE

We start by describing the terminology used throughout our work. Let $N: \mathbb{R}^m \rightarrow \mathbb{R}^n$ be a neural network which classifies an input $x \in \mathbb{R}^m$ (in our case an image) to $\arg \max_i N(x)_i$.

Specification. A robustness specification is a pair (\mathbb{X}, \mathbb{Y}) where $\mathbb{X} \subseteq \mathbb{R}^m$ is a set of input activations and $\mathbb{Y} \subseteq \mathbb{R}^n$ is a set of permissible outputs for those inputs.

Deterministic robustness. Given a specification (\mathbb{X}, \mathbb{Y}) , a neural network N is said to be (\mathbb{X}, \mathbb{Y}) -robust if $\forall x \in \mathbb{X}$, we have $N(x) \in \mathbb{Y}$. In the adversarial robustness literature, \mathbb{X} is usually an l_2 - or l_∞ -ball, and \mathbb{Y} is a set of outputs corresponding to a specific classification. In our case, \mathbb{X} will be represented as a segment (or a parametric curve) connecting two encodings $\bar{e}_1 \bar{e}_2$ produced by a generative model.

Probabilistic robustness. A limitation of deterministic robustness properties is that the result is always binary – the property either fully holds or does not. While useful for certifying single images, when combined with complex generative models it becomes too restrictive. Instead, we would like to compute a lower bound on the robustness of complex transformations (e.g., a lower bound for different head

orientations is 0.9) for cases when existing deterministic techniques only output that the specification does not hold.

Formally, given a distribution μ over \mathbb{X} (e.g., uniform distribution), we call the bounds on the *robustness probability* $\Pr_{x \sim \mu}[N(x) \in \mathbb{Y}]$ the *probabilistic [robustness] bounds*.

GenProve overview. In Figure 4.2 we illustrate how GENPROVE computes exact and probabilistic bounds for the robustness of a classifier based on a latent space image transformation. In this example, the goal is to verify that a classification network n_A is robust (i.e., does not change its prediction) when presented with images of a head from different angles, produced by interpolating encodings in the latent space of an autoencoder. To represent this specification, we first use the encoder, n_E , to produce encodings e_1 and e_2 from the original image and that image flipped horizontally. It is a common technique to use the decoder n_D to get a picture of a head at an intermediate angle, on an interpolated point e , taken from the segment $\overline{e_1 e_2} = \{e_1 + \alpha \cdot (e_2 - e_1) \mid \alpha \in [0, 1]\}$. Decodings for e_1 and e_2 can be seen in Figure 4.2(b). Our goal is to check the property for *all* possible encodings on $\overline{e_1 e_2}$ (not only points e_1 and e_2).

To accomplish this, we propagate lists of line segments and interval (box) constraints through the decoder and classifier, starting with $\overline{e_1 e_2}$. At each layer, we adaptively relax this list by combining segments into interval constraints, in order to reduce the number of points that need to be managed in downstream layers. This relaxation is key, as without it, the number of tracked points could grow exponentially with the number of layers. While Sotoudeh & Thakur [118] demonstrated that this is not a significant concern when propagating through just classifiers, for generative models or decoders, the desired output region will be highly non-convex (with better models producing more segments). One may think of the number of segments produced by the model in such a case as the model’s “generative resolution.”

Example of inference with overapproximation. Consider the simple (instructive) two dimensional input classifier network shown in Figure 4.2, with inputs $x_{1,1}$ and $x_{1,2}$. The possible inputs to this network we would like to consider are the points in the region described by the blue polygonal chain in Figure 4.2(b), whose axes are $x_{1,1}$ and $x_{1,2}$. The chain has coordinates $(1, 2), (-1, 3), (-1, 3.5), (1, 4.5), (3.5, 2)$ with $(1, 2)$ representing $n_D(e_1)$ and $(3.5, 2)$ representing $n_D(e_2)$. The segments of the chain are annotated with weights $\lambda = 0.2, 0.2, 0.2, 0.4$. These weights are such that the distribution produced by picking segment j with probability $\lambda^{(j)}$ and

then picking a point uniformly on that segment is the same as the distribution of $n_D(e)$ given $e \sim U(\overline{e_1 e_2})$ where $U(S)$ is the uniform distribution on S .

After applying the ReLU layer to this chain (marked with ReLU[#]), one can observe in Figure 4.2(c) that the first and third segment of this chain are split in half, resulting in 6 segments, which is 50% more than there were originally. As the segments represent uniform distributions, the weights of the new segments is the proportional weight of that part on the pre-ReLU segment. Here, each part of the new segment obtains half the pre-ReLU segment’s weight.

Because a 50% increase is significant, we now consolidate, moving from exact to approximate yet sound analysis. Here, we use a heuristic (labeled Relax), to choose segments to subsume that are small and close together. As they are quite close together, we pick the first 5 segments, replacing them by the (orange) box that has the smallest corner at $(0, 2)$ and largest corner at $(1, 4.5)$. This box, introduced in Figure 4.2(d), is assigned a weight equivalent to the sum 0.6, of the weights of all removed segments. Whereas each segment represents a uniform distribution, the new box represents a specific but unknown distribution with all its mass in the box. As a box is represented by two points (maximum and minimum), only four points are maintained, a significant reduction.

The last step performs matrix multiplication. As this operation is linear, segments can be transformed by transforming their nodes, without adding new points. Box constraints can be transformed using interval arithmetic, also without adding points. The weights of the regions are preserved, as the probability of selecting each region has not changed, only the regions themselves.

Computing probabilistic bounds. Let $\mathbb{A}^{(j)}$ for $j = 1 \dots k$ represent the regions (either the box and segment, or all 6 segments) shown in Figure 4.2(e), each with weight $\lambda^{(j)}$. Letting $[H]$ be the indicator for predicate H , we bound the probability $P_{t, \overline{e_1 e_2}} = \Pr_{e \in \overline{e_1 e_2}}[\arg \max_i n_D(e)_i = t]$ of class $t = 1$ being selected by classifier n_A as follows:

$$\begin{aligned}
 l &\leq P_{t, \overline{e_1 e_2}} \leq u && \text{where} \\
 l &= \sum_j [\forall x_3 \in \mathbb{A}^{(j)}. x_{3,1} > x_{3,2}] \lambda^{(j)} \\
 u &= \sum_j [\exists x_3 \in \mathbb{A}^{(j)}. x_{3,1} > x_{3,2}] \lambda^{(j)}
 \end{aligned}$$

As an example, we compute the lower bound for the case where we used relaxations. Here, the entirety of the orange box lies within the region

where $x_{3,1} > x_{3,2}$, so its indicator is 1 and we use its weight. On the other hand, the segment contains a point where $x_{3,1} = 2.75$ and $x_{3,2} = 3$ which violates this condition, so its indicator is 0 and its weight is not used. We can thus show a probabilistic lower bound of 0.6. Note that it is possible to provide an exact lower bound by computing the fraction of the segment that satisfies the condition (as described formally in Section 4.4). We now observe that all of the regions which would have been preserved using the exact procedure (in blue) would have contributed the same amount to the lower bound, as they all entirely satisfy the constraint. Exact inference would produce the same lower bound, but uses 50% more points.

4.3 CERTIFICATION OF DETERMINISTIC PROPERTIES

We review concepts from prior work [54] and define GENPROVE for deterministic properties. Our goal is to automatically show that images x from a given set \mathbb{X} of valid inputs are mapped to safe outputs from a set \mathbb{Y} . We write this property as $f[\mathbb{X}] \subseteq \mathbb{Y}$. For example, f might be a decoder, \mathbb{X} a line segment in latent space and \mathbb{Y} the images for which a given classifier detects a desired attribute.

Such properties compose: If we want to show that $h[\mathbb{X}] \subseteq \mathbb{Z}$ for $h(x) = g(f(x))$, it suffices to find a set \mathbb{Y} for which we can show $f[\mathbb{X}] \subseteq \mathbb{Y}$ and $g[\mathbb{Y}] \subseteq \mathbb{Z}$. For example, f could be a decoder and g an attribute detector, where \mathbb{Z} describes the output activations that lead to an attribute being detected.

We assume that we can decompose our network f as a sequence of l layers: $f = L_{l-1} \circ \dots \circ L_0$. To show a property $f[\mathbb{X}] \subseteq \mathbb{Y}$, we will find sets $\mathbb{A}_0, \dots, \mathbb{A}_l$ such that $\mathbb{X} \subseteq \mathbb{A}_0$, $L_i[\mathbb{A}_i] \subseteq \mathbb{A}_{i+1}$ for $0 \leq i < l$ and $\mathbb{A}_l \subseteq \mathbb{Y}$.

We determine the sets in order: We pick \mathbb{A}_0 based on \mathbb{X} such that $\mathbb{X} \subseteq \mathbb{A}_0$ and then for each $0 \leq i < l$, we pick \mathbb{A}_{i+1} such that $L_i[\mathbb{A}_i] \subseteq \mathbb{A}_{i+1}$. At the end, we check if we have $\mathbb{A}_l \subseteq \mathbb{Y}$. If so, the verification succeeds and the property holds. Otherwise, our procedure fails to prove the property.

Abstract interpretation. We automate this analysis using abstract interpretation [64]: we choose the sets $\mathbb{A}_0, \dots, \mathbb{A}_l$ such that they admit a simple symbolic representation in terms of real parameters. An *abstract domain* is a set of such symbolic representations. We write \mathcal{A}_n to denote an abstract domain where each element represents a member of $\mathcal{P}(\mathbb{R}^n)$. In our case, each abstract element $a \in \mathcal{A}_n$ represents a set of vectors of n neural network activations. The concretization function $\gamma_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{R}^n)$, which is specific to each abstract domain, maps a symbolic representation

$a \in \mathcal{A}_n$ to its concrete interpretation as a set $\mathbb{A} \in \mathcal{P}(\mathbb{R}^n)$ of neural network activation vectors. We will sometimes drop subscripts indicating dimensionality when they are irrelevant or clear from context. Our procedure will compute abstract elements a_0, \dots, a_l such that $\mathbb{A}_i = \gamma(a_i)$ for all $0 \leq i \leq l$.

An abstract transformer $T_f^\#: \mathcal{A}_m \rightarrow \mathcal{A}_n$ transforms symbolic representations to symbolic representations, overapproximating the function $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$, which means it has to satisfy the soundness property $f[\gamma_m(a)] \subseteq \gamma_n(T_f^\#(a))$ for all $a \in \mathcal{A}_m$. We will compute $a_{i+1} = T_{L_i}^\#(a_i)$. The soundness property of the abstract transformer ensures that we have $L_i[\mathbb{A}_i] \subseteq \mathbb{A}_{i+1}$, as this is equivalent to $L_i[\gamma(a_i)] \subseteq \gamma(T_{L_i}^\#(a_i))$.

By composing abstract transformers for all layers L_i of the neural network f in this fashion, we obtain an abstract transformer $T_f^\# = T_{L_{l-1}}^\# \circ \dots \circ T_{L_0}^\#$. Abstract interpretation provides a sound, typically incomplete method to certify properties: To show that a neural network $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ satisfies $f[\mathbb{X}] \subseteq \mathbb{Y}$, it suffices to show that $\gamma_n(T_f^\#(a)) \subseteq \mathbb{Y}$, for some abstract element $a \in \mathcal{A}_m$ with $\mathbb{X} \subseteq \gamma_m(a)$.

Box domain. If we pick \mathbb{A}_0 as a bounding box of \mathbb{X} , we can compute sets \mathbb{A}_i for $1 \leq i \leq l$ by evaluating the layers L_i using interval arithmetic. The analysis computes a range of possible values for each network activation, i.e., the sets \mathbb{A}_i are boxes. At the end, we check if \mathbb{A}_l 's bounds place it inside \mathbb{Y} .

This interval analysis is an instance of abstract interpretation. An element of the *box domain* \mathcal{B}_n is a box: a pair of vectors (\mathbf{a}, \mathbf{b}) where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$. The concretization function is $\gamma_n(\mathbf{a}, \mathbf{b}) = \prod_{l=1}^n [a_l, b_l]$. Abstract transformers for the box domain propagate bounds using interval arithmetic. While fast, this is imprecise and often fails to prove true properties.

Union domains. A list $a = (a^{(1)}, \dots, a^{(k)})$ of abstract elements (potentially from multiple different abstract domains) can be interpreted as a union with concretization $\gamma(a) = \bigcup_{j=1}^k \gamma(a^{(j)})$. Among other possibilities, we can obtain a union domain abstract transformer $T_f^\#$ by propagating each element of the union independently using an abstract transformer for its abstract domain:

$$T_f^\#(a) = (T_f^{\#(1)}(a^{(1)}), \dots, T_f^{\#(k)}(a^{(k)})).$$

Soundness follows directly from soundness of the component abstract transformers.

For example, we can cover the set \mathbb{X} with boxes and then propagate them through the network independently using interval arithmetic for each box. In the end, we have to show that all resulting boxes are within \mathbb{Y} .

Relaxation. At any point in the analysis, we can choose to replace an abstract element a_i by an element a'_i where $\gamma(a_i) \subseteq \gamma(a'_i)$. This can increase precision or reduce the number of parameters needed to represent a_i .

4.3.1 GENPROVE for Deterministic Properties

GENPROVE for deterministic properties is an analysis with an union domain where each component $a^{(j)}$ represents either a box or a line segment. Let n_i denote the number of neurons in layer i . Formally, $\gamma(a_i) = \bigcup_{j=1}^{k_i} \gamma(a_i^{(j)})$, where for each j either $\gamma(a_i^{(j)}) = \prod_{l=1}^{n_i} [a_l, b_l]$ is a box with given lower bounds \mathbf{a} and upper bounds \mathbf{b} , or $\gamma(a_i^{(j)}) = \overline{x_1 x_2}$ is a segment connecting the given points x_1 and x_2 in \mathbb{R}^{n_i} . We represent a_i as a list of bounds of boxes and a list of pairs of endpoints of segments.

Like Sotoudeh & Thakur [118], we focus on the case where the set \mathbb{X} of input activations is a segment. The work of Sotoudeh & Thakur [118] discusses how to split a given segment into multiple segments that cover it, such that a given neural network is an affine function on each of the new segments. Essentially, it determines the points where the segment crosses decision boundaries of piecewise-linear activation functions and splits it at those points. In order to compute an abstract element a_{i+1} such that $L_i[\gamma(a_i)] \subseteq \gamma(a_{i+1})$, we first split all segments according to this strategy applied to only the current layer L_i . Then, we map the endpoints of the resulting segments to the next layer by applying L_i to all of them. This is valid and captures exactly the image of the segments under L_i , because due to the splits, L_i , restricted to any one of the segments, is always an affine function. Further, we propagate the boxes through L_i by applying interval arithmetic. Note that if we propagate a segment $\overline{x_1 x_2}$ using this strategy alone for all layers, this analysis produces the exact image of \mathbb{X} , which is equivalent to performing the analysis using Sotoudeh & Thakur [118]'s method.

Relaxation. Before applying layer L_i , we may apply relaxation operators to turn a_i into a'_i , such that $\gamma(a_i) \subseteq \gamma(a'_i)$. We use two kinds of relaxation operators: *bounding box* operators remove a single segment \overline{cd} . The removed segment is replaced by its bounding box $\prod_{l=1}^{n_i} [\min(c_l, d_l), \max(c_l, d_l)]$. *Merge* operators replace multiple boxes by

their common bounding box. By carefully applying the relaxation operators, we can explore a rich tradeoff between pure instantiation of Sotoudeh & Thakur [118] and pure interval arithmetic. Our analysis generalizes both: if we never apply relaxation operators, the analysis reduces to Sotoudeh & Thakur [118] and will be exact but potentially slow. If we relax the initial segment into its bounding box, the analysis reduces to interval arithmetic and will be imprecise but fast.

Relaxation heuristic. We define the following heuristic, applied before each convolutional layer. The heuristic is parameterized by a relaxation percentage $p \in [0, 1]$ and a clustering parameter $k \in \mathbb{N}$. Each chain of connected segments with $t > 1000$ nodes is traversed in order, and each segment is turned into its bounding box, until the chain ends, the total number of different segment endpoints visited exceeds t/k or we find a segment whose length is strictly above the p -th percentile, computed over all segment lengths in the chain prior to applying the heuristic. All bounding boxes generated in one such step (from adjacent segments) are then merged, the next segment (if any) is skipped, and the traversal is restarted on the remaining segments of the chain.

4.4 CERTIFICATION OF PROBABILISTIC PROPERTIES

We now define GENPROVE for the probabilistic case. This setting is particularly useful when it is not possible to prove the property deterministically (or it does not hold). Our goal is to automatically show that images x drawn from a given input distribution μ map to desirable outputs \mathbb{D} with a probability in some interval $[l, u]$. We can write this property as $\Pr_{x \sim \mu}[d(x) \in \mathbb{D}] \in [l, u]$. For example, we can choose d to be a decoder, μ to be the uniform distribution on a line segment in its latent space, \mathbb{D} to be the set of images for which a given classifier detects a desired attribute and $[l, u] = [0.95, 1]$. The property then states that for at least a fraction 0.95 of the interpolated points, the classifier detects the desired attribute.

Unlike with the deterministic setting, such probabilistic properties do not compose naturally. We therefore reformulate them by defining sets \mathbb{X} and \mathbb{Y} of *probability distributions* and a distribution transformer f , in analogy to deterministic properties. Let d_* be the *pushforward* of d , formally defined below (intuitively, the pushforward allows a distribution to be mapped through a deterministic function). We let $\mathbb{X} = \{\mu\}$, $\mathbb{Y} = \{\nu \mid \Pr_{y \sim \nu}[\mathbf{y} \in \mathbb{D}] \in [l, u]\}$ and $f = d_*$. That is, f maps a

distribution of inputs to d to the corresponding distribution of outputs of d . Our property again reads $f[\mathbb{X}] \subseteq \mathbb{Y}$ and can be decomposed into properties $(L_i)_*[\mathbb{A}_i] \subseteq \mathbb{A}_{i+1}$ talking about each individual layer. We again overapproximate \mathbb{X} with \mathbb{A}_0 such that $\mathbb{X} \subseteq \mathbb{A}_0$ and push it through each network layer, computing sets $\mathbb{A}_1, \dots, \mathbb{A}_l$. The sets \mathbb{A}_i now contain *distributions* over activation vectors. We automate this analysis using probabilistic abstract interpretation.

Probabilistic abstract interpretation. We denote as \mathbb{D}_n the set of probability measures over \mathbb{R}^n . Probabilistic abstract interpretation is a variant of abstract interpretation where instead of deterministic points from \mathbb{R}^n , we abstract probability measures from \mathbb{D}_n . That is, a probabilistic abstract domain [117] is a set of symbolic representations of sets of measures over program states. We again use subscript notation to determine the number of activations: a probabilistic abstract domain \mathcal{A}_n has elements that each represent an element of $\mathcal{P}(\mathbb{D}_n)$. The probabilistic concretization function $\gamma_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{D}_n)$ maps each abstract element to the set of measures it represents.

For a measurable function $d: \mathbb{R}^m \rightarrow \mathbb{R}^n$, the corresponding pushforward $d_*: \mathbb{D}_m \rightarrow \mathbb{D}_n$ maps each measure $\mu \in \mathbb{D}_m$ to a measure $\nu \in \mathbb{D}_n$, given by

$$\nu(\mathbb{Y}) = \Pr_{x \sim \mu} [d(x) \in \mathbb{Y}] = \mu(d^{-1}(\mathbb{Y})),$$

where \mathbb{Y} ranges over measurable subsets of \mathbb{R}^n .

A probabilistic abstract transformer $T_f^\#: \mathcal{A}_m \rightarrow \mathcal{A}_n$ abstracts the pushforward f_* in the standard way: it satisfies $f_*[\gamma_m(a)] \subseteq \gamma_n(T_f^\#(a))$ for all $a \in \mathcal{A}_m$, analogous to the deterministic setting.

Probabilistic abstract interpretation gives a sound method to compute bounds on robustness probabilities. Namely, to show that $\Pr_{x \sim \mu} [N(x) \in \mathbb{Y}] \in [l, u]$, it suffices to show that $\nu(\mathbb{Y}) \in [l, u]$ for each $\nu \in \gamma_n(T_N^\#(a))$ for some a with $\mu \in \gamma_m(a)$.

Lifting. Note that we can reuse a deterministic abstract domain directly as a probabilistic abstract domain, by ignoring probabilities. More concretely, consider a deterministic abstract domain \mathcal{A}_n with deterministic concretization function $\gamma_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{R}^n)$. We can interpret \mathcal{A}_n as a probabilistic abstract domain by simply defining a probabilistic concretization function $\gamma'_n: \mathcal{A}_n \rightarrow \mathcal{P}(\mathbb{D}^n)$. Namely, for some abstract element $a \in \mathcal{A}_n$ representing a set $\mathbb{A} = \gamma_n(a)$ of (deterministic) activation

vectors, we let $\gamma'_n(a)$ be the set of all probability measures whose support is a subset of \mathbb{A} .¹

The analysis then just propagates abstract elements with the same abstract transformers it would use in the deterministic setting. For example, if we run probabilistic analysis with the box domain, $\gamma(a_i)$ is the set of all probability measures on the box a_i , and the analysis propagates the box constraints using interval arithmetic. Of course, such an analysis is rather limited, as it can at most prove properties with $l = 0$ or $u = 1$. For example, it would be impossible to prove that a probability is between 0.6 and 0.8 using only this kind of lifted analysis. However, interval arithmetic, lifted in this fashion, is a powerful component of GENPROVE for probabilistic properties, detailed below.

Convex combination domains. A formal convex combination $a = \sum_{j=1}^k \lambda^{(j)} \cdot a^{(j)}$ of abstract elements (potentially from multiple different abstract domains) can be interpreted as an abstract element whose concretization $\gamma(a)$ contains all probability measures of the form $\sum_{j=1}^{k_i} \lambda_i^{(j)} \cdot \mu^{(j)}$, where each $\mu^{(j)}$ is some probability measure chosen from the corresponding $\gamma(a^{(j)})$. For example, if the abstract elements $a^{(j)}$ represent disjoint boxes, then a represents all probability measures for which the probability of each box is the corresponding weight $\lambda^{(j)}$. In general, we can think of $\gamma(a)$ as the set of distributions generated by a set of random processes: Each process first randomly selects an index j according to the probabilities $\lambda^{(j)}$ and then samples from some fixed probability measure $\mu^{(j)} \in \gamma(a^{(j)})$. For each $1 \leq j \leq k$, this measure is fixed in advance for each of the random processes.

Similar to unions, we can apply abstract transformers to each abstract element $a^{(j)}$ independently and to form the convex combination of the results using the same weights:

$$T_f^\#(a) = \sum_{j=1}^k \lambda^{(j)} \cdot T_f^\#(a^{(j)}).$$

This is sound because pushforwards are linear functions.

¹ This is subject to some technical constraints: For example, all deterministic concretizations have to be measurable sets.

4.4.1 GENPROVE for Probabilistic Properties

Probabilistic GENPROVE is an analysis with a convex combination domain where each component $a_i^{(j)}$ represents either a lifted box or a single probability measure on a segment. Formally, this means

$$\gamma(a_i) = \left\{ \sum_{j=1}^{k_i} \lambda_i^{(j)} \cdot \mu^{(j)} \mid \mu^{(1)} \in \gamma(a_i^{(1)}), \dots, \mu^{(k_i)} \in \gamma(a_i^{(k_i)}) \right\},$$

where for each j , the concretization $\gamma(a_i^{(j)})$ is either the set of probability measures supported at most on a box $\prod_{l=1}^{n_i} [a_l, b_l]$ with lower bounds \mathbf{a} and upper bounds \mathbf{b} , or $\gamma(a_i^{(j)}) = \{\nu\}$, where ν is a distribution on a segment $\overline{x_1 x_2}$ with endpoints x_1 and x_2 in \mathbb{R}^{n_i} . To automate analysis, we represent $\gamma(a_i)$ as a list of bounds of boxes with associated weights $\lambda_i^{(j)}$, and a list of segments with associated distributions and weights $\lambda_i^{(j)}$. If, as in our evaluation, we consider a restricted case, where distributions on segments are uniform, it suffices to associate a weight to each segment. The weights should be non-negative and sum up to 1.

The element a_i can be propagated through layer L_i to obtain a_{i+1} in a similar fashion as in deterministic analysis. However, when splitting a segment, we now also need to split the distribution associated to it. For example, if we want to split the segment $\mathbb{L} = \overline{cd}$ with distribution ν and weight λ into two segments $\mathbb{L}' = \overline{ce}$ and $\mathbb{L}'' = \overline{ed}$ with $\mathbb{L}' \cup \mathbb{L}'' = \mathbb{L}$, we have to form distributions ν', ν'' and weights λ', λ'' where $\lambda' = \lambda \cdot \Pr_{x \sim \nu}[x \in \mathbb{L}']$, $\lambda'' = \lambda \cdot \Pr_{x \sim \nu}[x \in \mathbb{L}'' \setminus \mathbb{L}']$, ν' is ν conditioned on the event \mathbb{L}' and ν'' is ν conditioned on the event \mathbb{L}'' . If distributions on segments are uniform, this would result in the weight being split according to the relative lengths of the new segments. To propagate a lifted box, we apply interval arithmetic, preserving the box's weight. In practice, this is the same computation used for the deterministic propagation of a box.

We focus on the case where we want to propagate a singleton set containing the uniform distribution on a segment $\mathbb{L} = \overline{x_1 x_2}$ through the neural network. In this case, each distribution on a propagated segment will remain uniform, and it suffices to store a segment's weight without an explicit representation for the corresponding distribution, as noted above. As in the deterministic case, if we apply the analysis to the uniform distribution on a segment without relaxation, the analysis will compute an exact representation of the output distribution. I.e., \mathbb{A}_l will contain only the distribution of outputs obtained when the neural network is applied to inputs distributed uniformly at random on \mathbb{L} .

Relaxation. As this does not scale, we again apply relaxation operators. Similar to the deterministic setting, we can replace a probabilistic abstract element a_i by another probabilistic abstract element a'_i with $\gamma(a_i) \subseteq \gamma(a'_i)$.

Relaxation heuristic. Here, we use the same heuristic described for the deterministic setting. When replacing a segment by its bounding box, we preserve its weight. When merging multiple boxes, their weights are added to give the weight for the resulting box.

Computing bounds. Given the abstract element, a_l , describing a superset of the possible output distributions of the network, we compute bounds on the robustness probabilities $\mathbb{P} = \{\Pr_{y \sim \nu}[y \in \mathbb{D}] \mid \nu \in \gamma(a_l)\}$. The part of the distribution tracked using segments has all its probability mass in determined locations, while the probability mass in a box can be located anywhere within it. We compute bounds as:

$$(l, u) = (\min \mathbb{P}, \max \mathbb{P}) = \left(e + \sum_{j \in \mathbb{L}} \lambda_l^{(j)}, e + \sum_{j \in \mathbb{U}} \lambda_l^{(j)} \right),$$

where e is the probability of the output being on a segment. If \mathbb{D} is given as a set of linear constraints, we compute e by splitting the segments to not cross the constraints and summing up all weights of resulting segments contained in \mathbb{D} . \mathbb{L} is the set of indices of lifted boxes contained in \mathbb{D} and \mathbb{U} is the set of indices of lifted boxes that intersect with \mathbb{D} .

4.4.2 Propagation Pseudocode and Example

Here we show the pseudocode for the full propagation algorithm for GENPROVE, and provide an example of propagation using it. Here, we only show linear probabilistic computation, and do not demonstrate how the final output is verified against a constraint.

We will walk through Algorithm 1 using an example beginning with a line segment in two dimensions $a = (1, 0)$ and $b = (0, 1)$, and a 1 layer neural network with the following weights and biases:

$$M_1 = \begin{pmatrix} 2 & 2 & 3 \\ -1 & 1 & 0 \end{pmatrix} \quad B_1 = (-1, 0, 1)$$

The algorithm first constructs a list D containing the single line segment from a to b with weight one. In the first iteration of the loop, there is only

one iteration of first inner loop, where $i = 1$, and thus D_1 is a segment so we proceed there. We create new start and end nodes for this segment, $a := D_{i,3}M_1 + B_1$ and $b := D_{i,4}M_1 + B_1$. Specifically,

$$a = (1,0) \begin{pmatrix} 2 & 2 & 3 \\ -1 & 1 & 0 \end{pmatrix} + (-1,0,1) = (1,2,4)$$

$$b = (0,1) \begin{pmatrix} 2 & 2 & 3 \\ -1 & 1 & 0 \end{pmatrix} + (-1,0,1) = (-1,1,1).$$

We then fill a T with sorted zero-axis intersection times, starting with 0 and 1. Specifically, for each dimension d we calculate the time t_d such that $(b_d - a_d)t_d + a_d = 0$. We can compute this as $t_d = -a_d / (b_d - a_d)$. We only include this time if t_d is strictly between 0 and 1. In the example, we compute $T = [0, 0.5, 1]$ as the intersection times for dimension $d = 2$ and $d = 3$ fall outside of 0 and 1.

For each time in this list, we compute the start, \tilde{a} , and end nodes, \tilde{b} for a new segment, and the probability p corresponding to that segment. The nodes of the segments are computed by interpolating between a and b using the times in T whereas the probability for each segment is the difference between the times of the nodes, multiplied by the probability of the original segment between a and b . As T has three nodes we calculate two segments. The first from $(1,2,4)$ to $(0,1.5,2.5)$ with $p = 0.5$ and the second from $(0,1.5,2.5)$ to $(-1,1,1)$ with $p = 0.5$. We apply ReLU to each dimension of the nodes of the segments, and add these to a currently empty list, or domain element, \tilde{D} , of segments produced at that layer:

$$\tilde{D} = [(\text{Segment}, 0.5, (1,2,4), (0,1.5,2.5)), \\ (\text{Segment}, 0.5, (0,1.5,2.5), (0,1,1))].$$

Next, the algorithm determines which segments to merge using the chosen Relax heuristic, which returns a list of relaxed boxes. Pedagogically, assume this returns a single box which contains both segments entirely. This is a box that goes from a minimal point $(0,1,1)$ to a maximal point $(1,2,4)$, or as we work with in the algorithm, has a center point $(0.5, 1.5, 2.5)$ and radius $(0.5, 0.5, 1.5)$. Finally, the segments that are contained within this box are deleted. We add this box to the list \tilde{D} with associated probability equivalent to the sum of the deleted elements probabilities. We note that this process also applies to merging boxes that are already part of \tilde{D} .

Algorithm 1 Pseudocode for inference with GENPROVE

Input: k network layers with weights and biases M_i, B_i , and a line segment a, b in the input space.

Output: D a list of boxes and segments describing the probabilities of possible regions of the output space.

$D = [(\text{Segment}, 1, a, b)]$.

for $l = 1$ **to** $k - 1$ **do**

$\tilde{D} = []$

for $i = 1$ **to** $|D|$ **do**

if $D_i == \text{Segment}$ **then**

$a = D_{i,3}M_l + B_l$

$b = D_{i,4}M_l + B_l$

$T = [0, 1]$

for $d = 1$ **to** $|b_l|$ **do**

$t_d = \frac{-a_d}{|(b-a)_d|}$

if $0 < t_d < 1$ **then**

$T.\text{push}(t_d)$

end if

end for

$T.\text{sort}()$

for $t = 2$ **to** $|T|$ **do**

$p = T_t - T_{t-1}$

$\tilde{a} = (b - a) * T_{t-1} + a$

$\tilde{b} = (b - a) * T_t + a$

$\tilde{D}.\text{push}((\text{Segment}, D_{i,2} * p, \text{ReLU}(\tilde{a}), \text{ReLU}(\tilde{b})))$

end for

else

$c = D_{i,3}M_l + b_l$

$r = D_{i,4}|M_l|_p$

$\tilde{c} = \text{ReLU}(c + r) + \text{ReLU}(c - r)$

$\tilde{r} = \text{ReLU}(c + r) - \text{ReLU}(c - r)$

$\tilde{D}.\text{push}((\text{Box}, D_{i,2}, 0.5 * \tilde{c}, 0.5 * \tilde{r}))$

end if

end for

$\tilde{P} = \text{Relax}(\tilde{D})$

for $p = 1$ **to** $|\tilde{P}|$ **do**

for $i = 1$ **to** $|\tilde{D}|$ **do**

if $\gamma(\tilde{D}_i) \subseteq \gamma(\tilde{P}_p)$ **then**

$\tilde{P}_{p,2} = \tilde{P}_{p,2} + \tilde{D}_{i,2}$

delete D_i

end if

end for

end for

$D = \tilde{D} + \tilde{P}$

end for

4.4.3 Generalization to Parametric Curves

The approaches presented so far relied on a number of operations on line segments: We needed to form their image under affine transformations, we had to determine splitting points based on ReLU decision boundaries, and we had to be able to split a line segment into multiple segments whose union is the original segment. For the probabilistic case, we additionally tracked probability measures on those segments.

Therefore, we develop `GENPROVECURVE` which generalizes our analysis to handle other one-dimensional shapes for which these operations can be supported. Let $\gamma: [l, u] \rightarrow \mathbb{R}^n$ be a continuous function given by

$$\gamma(t) = \mathbf{a}^{(0)} + \sum_{i=1}^k \mathbf{a}^{(i)} \cdot \eta^{(i)}(t),$$

for one-dimensional continuous functions $\eta^{(i)}: [l, u] \rightarrow \mathbb{R}$ and vectors $\mathbf{a}^{(i)} \in \mathbb{R}^n$. The function γ represents the curve $\gamma[[l, u]]$ in \mathbb{R}^n . For the probabilistic case, we additionally consider a probability measure μ on $[l, u]$ describing the distribution of the curve parameter. (The probability measure in \mathbb{R}^n describing our probabilistic curve is then implicitly given by $\nu(X) = \mu(\gamma^{-1}(X))$.)

We can symbolically form the image of the shape $\gamma[[l, u]]$ under an affine transformation $f(\mathbf{x}) = A \cdot \mathbf{x} + \mathbf{b}$ as the set $f[\gamma[[l, u]]] = (f \circ \gamma)[[l, u]]$, where $f \circ \gamma$ is given by

$$f(\gamma(t)) = A \cdot \gamma(t) + \mathbf{b} = (A \cdot \mathbf{a}^{(0)} + \mathbf{b}) + \sum_{i=1}^k (A \cdot \mathbf{a}^{(i)}) \cdot \eta^{(i)}(t).$$

I.e., to apply an affine transformation to our curve, it suffices to transform the coefficient vectors $\mathbf{a}^{(i)}$.

To find splitting points for ReLU decision boundaries, we need to solve the equation $\gamma(t)_j = 0$ for t for each component $j \in \{1, \dots, n\}$. For example, if we consider quadratic parametric curves $\gamma: [l, u] \rightarrow \mathbb{R}^n$ of the form

$$\gamma(t) = \mathbf{a}^{(0)} + \mathbf{a}^{(1)} \cdot t + \mathbf{a}^{(2)} \cdot t^2,$$

we have to solve a quadratic equation for each component, yielding at most two splitting points per component, where we ignore solutions outside $[l, u]$. We can split the curve at those points by restricting it to segments between subsequent splitting points in sorted order. In the probabilistic case, we further restrict the measure to the same segments and associate the resulting measures to the new curves.

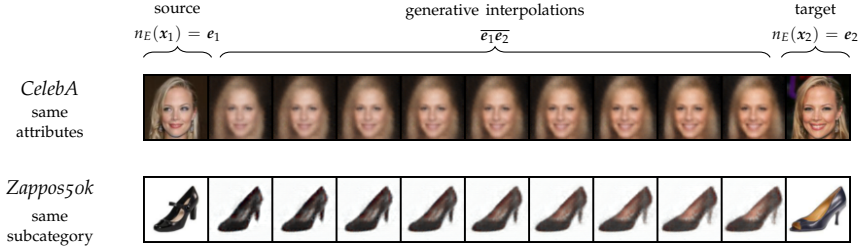


FIGURE 4.3: Example of a generative specification used in our work. Here, x_1 and x_2 are original images with corresponding embeddings e_1 and e_2 , respectively. For this specification, the images are chosen such that they contain the same attributes (for CelebA) or belong to the same subcategory (for Zappos50k). The goal is to verify, deterministically or probabilistically, that the network under test does not change its prediction when presented with the interpolated images $\overline{e_1 e_2}$.

4.5 EVALUATION

In this section, we demonstrate the benefits of GENPROVE and the techniques presented in our work. In particular, our goal is to answer the following three research questions:

- RQ1 Is probabilistic abstract interpretation necessary for handling complex generative specifications (as compared to traditional abstract interpretation)?
- RQ2 Does GENPROVE produce tight bounds and scale to realistic large networks (unlike existing methods)?
- RQ3 What novel specifications can be verified using GENPROVE (beyond what is currently possible)?

We first answer RQ1 by showing that our application of probabilistic abstract interpretation is key for analyzing generative specifications. Concretely, we demonstrate that: (i) using deterministic verification is a limiting factor for *all* non-trivial benchmarks and networks, (ii) probabilistic interpretation proposed in our work improves the fraction of samples for which tight bounds can be computed from 0.5% to up to 76.2%, and (iii) our combination of probabilistic analysis with relaxations can produce non-trivial verified bounds for 100% of samples.

		% of samples w/ non-trivial verified bounds			
		Exact Verification		Verification with Relaxations	
		(Deterministic)	(Probabilistic)	(Deterministic)	(Probabilistic)
DATASET	NETWORK	BASELINE	GENPROVE ⁰	GENPROVE ^{DET0.02} ₁₀₀	GENPROVE ^{0.02} ₁₀₀
<i>CelebA</i>	ConvSmall	91.2%	$\xrightarrow{+8.8\%}$ 100%	78.6%	$\xrightarrow{+21.4\%}$ 100%
	ConvMed	9.5%	$\xrightarrow{+0.5\%}$ 10%	23.8%	$\xrightarrow{+76.2\%}$ 100%
<i>Zappos50k</i>	ConvSmall	56%	$\xrightarrow{+44\%}$ 100%	56%	$\xrightarrow{+44\%}$ 100%
	ConvMed	8%	$\xrightarrow{+3\%}$ 11%	58%	$\xrightarrow{+42\%}$ 100%
		<i>Prior Work</i>		<i>Our Work</i>	

TABLE 4.1: Comparing deterministic analysis with probabilistic analysis. The number is the percentage of 100 samples evaluated on average consistency $\hat{\mathcal{C}}$ that did not return the full interval $l = 0$ and $u = 1$. We note that the poor performance of BASELINE on ConvMed is due to out of memory errors, where the full interval is returned.

We then answer RQ2 by demonstrating that GENPROVE scales to realistically large networks with 200k neurons while producing bounds that are very tight and close to zero (e.g., $5.7 \cdot 10^{-5}$). In contrast, we show that all prior methods fail – either because they are imprecise and produce extremely loose bounds close to 1, or they exhaust the ample GPU memory and crash. Note that, as we will show later in this section, only increasing the GPU memory is not a scalable solution and a fundamentally different approach, like the one proposed in our work, is needed.

To answer RQ3, we show the versatility of GENPROVE in certifying the novel class of generative specifications in five ways: (i) we show how GENPROVE can be used to certify and specify the higher dimensional specification where a generative network defines a continuous set of images that a classifier should categorize correctly under any possible L_∞ attack (ii) certifying robustness to different head orientations, (iii) certifying attribute independence via adding previously absent attributes (e.g., changing the hair color as illustrated in Figure 4.3), (iv) certifying attribute independence over input regions that curve through areas with previously absent attributes. and finally (v) certifying out of distribution detection with non-uniform specifications.

Experimental setup. We certify robustness of generative models using a variety of different approaches:

- GENPROVE_k^p which implements both the probabilistic and deterministic (denoted as $\text{GENPROVE}^{\text{DET}}$) verifier proposed in our work. Here, p is the relaxation percentage and k is the clustering parameter. Note that setting the relaxation percentage to zero (denoted as GENPROVE^0) instantiates our approach without relaxations, thus producing exact results.
- **BASELINE** is the deterministic approach proposed by Sotoudeh & Thakur [118], producing exact results. We note that we use our own, more scalable and GPU-optimized, implementation of this approach. The original implementation supports only computations on CPUs and takes prohibitively large amounts of time when run on the large networks we use for evaluation.
- A wide range of existing convex abstract domains for neural networks: Box [54], Zonotope [54], DeepZono [98], and HybridZono [55]. We adapted all of them to certification of generative models by representing the initial segment $\bar{e}_1\bar{e}_2$. Note that for every domain but Box, this step is exact and does not lose precision.

We implement GENPROVE in the DiffAI [55] framework, taking advantage of the GPU parallelization provided by PyTorch [90]. Our implementation will be made available on GitHub along with all the models used for testing.

For a fair comparison of runtime and scalability, the verifiers used are also implemented with GPU support. Our experiments are performed on a machine with a Titan RTX GPU with 24 GB of GPU memory.

Generative models. We use 3 datasets of increasing complexity – MNIST [119], Zappos50k [120, 121], and CelebA [122]. For each, we trained a VAE [123] autoencoder with the architectures and training schemes described in full detail below. For all datasets, the decoder and generator have each 74 128 neurons, unless otherwise specified.

Target networks. For each dataset, we trained a variety of attribute detectors or classifiers:

- *CelebA*: We trained attribute detectors on the scaled 64×64 images with three different architectures – ConvSmall, ConvMed, and ConvLarge, with 24 676, 63 804 and 123 180 neurons respectively. The

attribute detectors are trained to recognize the 40 attributes provided by CelebA (e.g., bald, bangs, blond) [122]. Here, an attribute i is detected in the image if the i -th component of the network output is strictly positive.

- *Zappos50k*: We trained classifiers on the 64×64 images with the same three architectures as for CelebA. The classifiers are trained to recognize the 21 subcategories (e.g. heels, boots) from the Zappos50k dataset [120, 121].
- *MNIST*: We used a classifier with 175 816 neurons trained with three different techniques to recognize digits [119]. Specifically, we used the publicly available ConvBiggest architecture from [55] and trained it using standard training, using Box with DiffAI, as well as FGSM [20] with $\epsilon = 0.1$.

Network Architecture Details.

Our experiments use two different encoder architectures (Encoder and EncoderSmall), two decoder architectures (Decoder and DecoderSmall), and four different classifier/attribute detector architectures (ConvSmall, ConvMed, ConvLarge, ConvBiggest). These are described in detail here.

Here we use $\text{Conv}_s C \times W \times H$ to denote a convolution which produces C channels, with a kernel width of W pixels and height of H , with a stride of s and padding of 1. $\text{FC } n$ is a fully connected layer which outputs n neurons. $\text{ConvT}_{s,p} C \times W \times H$ is a transposed convolutional layer [124] with a kernel width and height of W and H respectively and a stride of s and padding of 1 and out-padding of p , which produces C output channels. l refers to the number of latent dimensions, and o refers to either the number of attributes or number of classes. For CelebA and Zappos50k use 64 latent dimensions, while the VAE for MNIST uses 50 latent dimensions.

- *EncoderSmall* is a standard convolutional neural network with 74128 neurons. It is used for encoding MNIST and Zappos50k. It is trained with Adam [91] with a learning rate of 0.001 and a batch size of 128. The network was trained for 300 epochs for Zappos50k and 20 epochs for MNIST.

$$\begin{aligned}
 x &\rightarrow \text{Conv}_2 16 \times 4 \times 4 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_2 32 \times 4 \times 4 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 100 && \rightarrow l.
 \end{aligned}$$

- *Encoder* is also a standard convolutional neural network, but significantly larger with 246784 neurons, used only for encoding CelebA. It is trained with Adam with a learning rate of 0.0001 and a batch size of 100 for 20 epochs.

$$\begin{aligned}
 x &\rightarrow \text{Conv}_1 32 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_2 32 \times 4 \times 4 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_1 64 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_2 64 \times 4 \times 4 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 512 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 512 && \rightarrow l.
 \end{aligned}$$

- *Decoder* is a transposed convolutional network which has 74128 neurons used for decoding every dataset in nearly every experiment, unless otherwise specified. Of course, the training parameters are the same as the respective encoders.

$$\begin{aligned}
 l &\rightarrow \text{FC } 400 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 2048 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{ConvT}_{2,1} 16 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{ConvT}_{1,0} 3 \times 3 \times 3 && \rightarrow x.
 \end{aligned}$$

- *DecoderSmall* is a smaller transposed convolutional network which has 41160 neurons used for decoding CelebA for testing GENPROVECURVE. The training parameters are the same as the respective encoders.

$$\begin{aligned}
 l &\rightarrow \text{FC } 200 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 2048 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{ConvT}_{2,1} 8 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{ConvT}_{1,0} 3 \times 3 \times 3 && \rightarrow x.
 \end{aligned}$$

- *ConvSmall* is a convolutional network which has 24676 neurons. The convolutions use a padding of 1. It is only used for a toy parameter

comparison on CelebA. It was trained for 300 epochs with a batch size of 100 using Adam with a learning rate of 0.0001. It had a test-set accuracy of 89.87%.

$$\begin{aligned} x &\rightarrow \text{Conv}_2 16 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_2 32 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{FC } 100 && \rightarrow o. \end{aligned}$$

- *ConvMed* is a convolutional network which has 63804 neurons. Here, the convolutions use a padding of 1. This is used as a classifier and attribute detector for Zappos50k and CelebA experiments. For both experiments it was trained with a batch size of 128 using Adam with a learning rate of 0.001. For Zappos50k it was trained for 5 epochs and achieved a test-set accuracy of 79.40%. For CelebA it was trained for 10 epochs and achieved a test-set accuracy of 89.87%.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 12 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_2 16 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{FC } 500 \rightarrow \text{FC } 200 \rightarrow \text{FC } 100 \rightarrow o. \end{aligned}$$

- *ConvLarge* is a convolutional network which has 123180 neurons. Here, the convolutions use a padding of 1. This is used as a classifier and attribute detector for Zappos50k and CelebA experiments. For both experiments it was trained with a batch size of 128 using Adam with a learning rate of 0.001. For Zappos50k it was trained for 5 epochs and achieved a test-set accuracy of 82.20%. For CelebA it was trained for 10 epochs and achieved a test-set accuracy of 89.86%.

$$\begin{aligned} x &\rightarrow \text{Conv}_1 16 \times 3 \times 3 && \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_2 16 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_1 32 \times 3 \times 3 && \rightarrow \text{ReLU} \\ &\rightarrow \text{Conv}_2 32 \times 4 \times 4 && \rightarrow \text{ReLU} \\ &\rightarrow \text{FC } 200 \rightarrow \text{FC } 100 \rightarrow o. \end{aligned}$$

- *ConvBiggest* is a convolutional network which has 175816 neurons. Here, the convolutions use a padding of 1. This is used as a classifier

detector for MNIST experiments. When trained with DiffAI the training schedule suggested by Goyal *et al.* [59] is used. Each method trains it for 30 epochs.

$$\begin{aligned}
 x &\rightarrow \text{Conv}_1 64 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_1 64 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_2 128 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_1 128 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{Conv}_1 128 \times 3 \times 3 && \rightarrow \text{ReLU} \\
 &\rightarrow \text{FC } 200 && \rightarrow o.
 \end{aligned}$$

Evaluation metrics. We show the precision of our system by verifying that a given model (denoted as n_A for an attribute detector, and n_C for a classifier) is robust to the transformations learned by the generative model.

We formalize this concept with a metric called *consistency*: for a point picked uniformly between the encodings e_1 and e_2 of ground truth inputs, we determine the probability that its decoding (computed by a decoder n_D) will have (or not) the same attribute. As a concrete example, Figure 4.3 shows generative interpolations for both CelebA and Zappos50k. Formally, consistency is defined for attribute detectors as

$$\mathcal{C}_{i,n_A,n_D}(e_1, e_2) = \Pr_{e \sim U(\overline{e_1 e_2})} [\text{sign } n_A(n_D(e))_i = t],$$

and for classifiers as

$$\mathcal{C}_{n_C,n_D}(e_1, e_2) = \Pr_{e \sim U(\overline{e_1 e_2})} [\arg \max_i n_C(n_D(e))_i = t].$$

Suppose P is a set of pairs $\{\mathbf{a}, \mathbf{b}\}$ from the data and it holds that $\text{sign } a_{A,i} = \text{sign } b_{A,i}$ for every attribute, where $a_{A,i}$ is the label of attribute i for \mathbf{a} . We compute bounds on the *average consistency* as $\hat{\mathcal{C}}_P = \text{mean}_{\mathbf{a}, \mathbf{b} \in P, i} \mathcal{C}_{i,n_A,n_D}(n_E(\mathbf{a}), n_E(\mathbf{b}))$ for attribute detectors, and $\hat{\mathcal{C}}_P = \text{mean}_{\mathbf{a}, \mathbf{b} \in P} \mathcal{C}_{n_C,n_D}(n_E(\mathbf{a}), n_E(\mathbf{b}))$ for classifiers, where n_E is the encoding network.

We compute a probabilistic bound, $[l, u]$, for each method in our evaluation such that $l \leq \hat{\mathcal{C}} \leq u$. We call $u - l$ its width.

GenProve Refinement Schedule.

While many refinement schemes start with an imprecise approximation and progressively tighten it, we observe that being only occasionally memory limited and rarely time limited, it conserves more time to start with the most precise approximation we have determined usually works, and progressively try less precise approximations as we determine that more precise ones can not fit into GPU memory. Thus, we start searching for a probabilistic robustness bound with GENPROVE_N^P and if we run out of memory, try $\text{GENPROVE}_{\max(0.95N,5)}^{\min(1.5p,1)}$ for schedule A, and $\text{GENPROVE}_{\max(0.95N,5)}^{\min(3p,1)}$ for schedule B. This procedure is repeated until a solution is found, or time has run out.

4.5.1 RQ1 - Probabilistic Abstract Interpretation

We start by addressing RQ1 and demonstrate that while traditional deterministic verification methods may be precise for deterministic specifications, they are of limited utility when tasked with verifying probabilistic specifications. To understand why, recall that for the deterministic domains, there are only three possible outputs for the lower and upper bounds: $[0, 0]$ meaning that none of the specification was correct, $[1, 1]$ meaning that the specification was entirely correct, or least usefully, the full interval $[0, 1]$ implying that the technique was unable to verify one way or the other how much of the specification was correct. This severely limits the usefulness of the verification, especially for cases with imperfect networks and imperfect specifications.

Table 4.1 demonstrates the limited applicability of deterministic domains. Here, we report the fraction of specifications where the bounds were strictly tighter than $[0, 1]$. Based on the results, we can immediately see that GENPROVE provides useful bounds for 100% of the specifications for every network and dataset. At the same time, the deterministic methods BASELINE and $\text{GENPROVE}^{\text{DET}}$ rarely return useful bounds for the consistency specification. In particular, BASELINE proves at best 91.2%, and at worst 8% of the specifications. This is because even on the large network, GENPROVE^0 and BASELINE run out of memory. One should also observe that GENPROVE^0 performs better than BASELINE and when it does not run out of memory, performs better than $\text{GENPROVE}^{\text{DET}}$. We note that GENPROVE not only always provides useful results, but is also precise and achieved an average width ($u - l$) of at worst 0.0001 for CelebA (ConvMed). In comparison, the next-best domain (BASELINE on ConvSmall) produced

DATASET	DOMAIN	average consistency \hat{C} bound width ($u - l$)						
		($\approx 25k$ neurons)	($\approx 64k$ neurons)	($\approx 123k$ neurons)	Precise	Scalable		
		ConvSmall	ConvMed	ConvLarge				
CelebA	Prior	Box [54]	0.98	0.98	0.98	-	✓	
		HybridZono [55]	0.97	0.97	0.97	-	✓	
		DeepZono [98]	1.0	1.0	1.0	-	-	
		Zonotope [54]	1.0	1.0	1.0	-	-	
		Our	GENPROVE ⁰	0.0	0.9	0.95	✓	-
			GENPROVE ^{0.02} ₁₀₀	$1.8 \cdot 10^{-5}$	$1.1 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$	✓	✓
Zappos50k	Prior	Box [54]	1.0	1.0	1.0	-	✓	
		HybridZono [55]	1.0	1.0	1.0	-	✓	
		DeepZono [98]	1.0	1.0	1.0	-	-	
		Zonotope [54]	1.0	1.0	1.0	-	-	
		Our	GENPROVE ⁰	0.0	0.89	0.99	✓	-
			GENPROVE ^{0.02} ₁₀₀	$3.3 \cdot 10^{-5}$	$4.5 \cdot 10^{-5}$	$5.7 \cdot 10^{-5}$	✓	✓

TABLE 4.2: Scalability and precision of our method compared to a wide range of prior convex abstract domains. All methods are lifted probabilistically. We report average consistency \hat{C} bound widths (lower is better).

a width of at best 0.1748 (not shown in Table 4.1), which is a full four orders of magnitude worse on a smaller network.

4.5.2 RQ2 - Precision and Scalability

Next, we address RQ2 by comparing the precision and scalability of probabilistic GENPROVE to a variety of existing convex abstract domains, as well as sampling. To study the scalability of all domains, we certify the robustness of three networks of increasing complexity – ConvSmall with $\approx 25k$ neurons, ConvMed with $\approx 64k$ neurons and ConvLarge with $\approx 123k$ neurons. Further, to provide variety, we certify robustness using two datasets: CelebA and Zappos50k.

Precision. Table 4.2 shows the result of running the verifiers using the same $|P| = 100$ pairs of images with either matching attributes (for CelebA) or the same class (for Zappos50k). We report the bound $[0, 1]$ if memory is exhausted.

One can first observe that Box, HybridZono, Zonotope, and DeepZono are unable to certify any samples as they almost always produce a probabilistic interval with width 1. We posit that this is due to

DATASET	DOMAIN	peak GPU memory in GB / OOM (%)			runtime in seconds		
		ConvSmall	ConvMed	ConvLarge	ConvSmall	ConvMed	ConvLarge
<i>CelebA</i>	GENPROVE ⁰	7 GB / 0%	22.7 GB / 90%	23.1 GB / 95%	11 sec	OOM	OOM
	GENPROVE ^{0,02} ₁₀₀	3.5 GB / 0%	6.8 GB / 0%	9.4 GB / 0%	13 sec	25 sec	41 sec
<i>Zappos50k</i>	GENPROVE ⁰	6.5 GB / 0%	22.7 GB / 89%	23.6 GB / 99%	11 sec	OOM	OOM
	GENPROVE ^{0,02} ₁₀₀	6.4 GB / 0%	6.6 GB / 0%	7.1 GB / 0%	15 sec	25 sec	32 sec

TABLE 4.3: Comparison of the memory usage and runtime of our GENPROVE with and without relaxations.

non-convexity being highly important for these kinds of specifications. Zonotope and DeepZono run out of memory for all the samples, even for the smallest network ConvSmall.

While GENPROVE⁰ is theoretically complete, it also predominantly failed to provide useful bounds as it frequently ran out of GPU memory. However, we can see that for the small network ConvSmall, where it does scale, it does produce exact results: the width of all bounds is 0.

GENPROVE^{0,02}₁₀₀ is the only approach that is both scalable and precise. The bounds are tight even for the largest network ConvLarge: $5.7 \cdot 10^{-5}$ for Zappos50k for example. Significantly, the bounds remain tight as the network size increases. For example, when the network size increased by 500% (from ConvMed to ConvLarge), the bound width increased from $3.3 \cdot 10^{-5}$ only to $5.7 \cdot 10^{-5}$.

While the speed of each method can be seen in Table 4.6, these numbers can be misleading: despite Box and HybridZono’s apparent speed, they fail to provide any useful information for any specifications due to aforementioned imprecision. Similarly, Zonotope, DeepZono and GENPROVE⁰ also appear very fast while failing to provide useful information. These fail however due to running out of GPU memory. In contrast, GENPROVE takes 41.4 seconds on the most complicated specification and network here, but produces extremely tight and useful bounds in every case.

Scalability. Table 4.3 shows the average runtime, peak GPU memory, and the fraction of samples that resulted in out-of-memory (OOM) errors. For the CelebA dataset, while the network size increased 5 \times , the memory usage of GENPROVE^{0,02}₁₀₀ increased only 2.7 \times . The improvement in memory usage is even better for the Zappos50k dataset, where increasing the size 2 \times leads to an increase in memory usage of only 1.08 \times . This shows

	DOMAIN	bound width ($u - l$)	
		<i>CelebA</i>	<i>Zappos50k</i>
<i>Verified Correctness</i>	GENPROVE ₁₀₀ ^{0.02}	$1.6 \cdot 10^{-4}$	$5.7 \cdot 10^{-5}$
<i>99.999% Confidence</i>	Sampling	$2.1 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$

TABLE 4.4: Comparison of the precision of our method to sampling. Our method not only provides results that are guaranteed to be sound, but also leads to tighter bounds. Note that the runtime of both methods (not shown) is the same.

that the relaxation technique proposed in our work successfully reduces the memory usage while still achieving tight bounds. The results for runtime are similar: runtime increases sublinearly depending on network size. Overall, the verification is fast and takes on average ≈ 40 seconds for ConvLarge and ≈ 11 seconds for ConvSmall.

The results in Table 4.3 also detail why GENPROVE⁰ does not scale: the needed GPU memory increases significantly with network size. For complex specifications, the number of segments that must be tracked often increases exponentially.

To improve the scalability of both GENPROVE⁰ and GENPROVE₁₀₀^{0.02} further, it is possible to split the specification into smaller parts. In our case, this corresponds to partitioning the initial segment (or other one-dimensional shapes) into multiple smaller segments that are verified sequentially and then merged together. However, while useful for avoiding the memory limitations, it comes at the cost of increased runtime. Given that the number of segments can grow exponentially with the network size, we believe that developing and incorporating techniques like the relaxation proposed in our work is critical for scaling to state-of-the-art networks.

Comparison to sampling.

In our next experiment, shown in Table 4.4, we compare to a sampling method, where samples are drawn from the uniform distribution over the initial segment. We report the Clopper-Pearson interval with a confidence of 99.999%. Notably, the probabilistic bound returned by sampling is only guaranteed to be correct 99.999% of the time (in cases it reaches the desired confidence), whereas for other analyses it is guaranteed to always be correct.

The results show that the sampling does scale and also produces bound widths with a reasonable precision. However, not only does GENPROVE


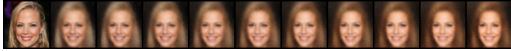

	SPECIFICATION	ILLUSTRATION	CERTIFIED RESULT
(a)	head orientation		bound width ($u - l$) $1.0 \cdot 10^{-4}$
(b)	adding "brown hair"		robust attributes 32/40
(c)	quadratic curve		robust attributes 29/40

TABLE 4.5: An illustration of the various specifications we support and what we are able to certify about them. This is in addition to the specifications shown in Figure 4.3 and certification of adversarial regions around a generative specification (not shown).

produce bounds that are guaranteed to be correct, it also produces bounds are up to two orders of magnitude tighter (i.e., by up to $\approx 2500\%$). These results were consistent across all the networks and datasets we evaluated, the full version of which is included in Table 4.6.

Dataset	Network	Neurons	Domain	Width ($u - l$)	GPU Memory			
					Seconds	OOM (%)	Peak (GB)	
CelebA	ConvSmall	24676	<i>Prior Work</i>	Box	0.98	0.0031	0	0.07
				HybridZono	0.9703	0.0042	0	0.07
				DeepZono	1.0	1.3485	100	23.62
			Zonotope	1.0	1.3345	100	23.62	
			<i>Our Work</i>	GENPROVE ⁰	0.0	10.7657	0	7.05
				GENPROVE ₁₀₀ ^{0.02}	1.78×10^{-5}	12.7403	0	3.51
	99.999% Confidence			Sampling	3.73×10^{-4}	14.7877	0	0.62
	ConvMed	63804	<i>Prior Work</i>	Box	0.98	0.0046	0	0.16
				HybridZono	0.97	0.0059	0	0.16
				DeepZono	1.0	1.6848	100	23.62
			Zonotope	1.0	1.5015	100	23.62	
			<i>Our Work</i>	GENPROVE ⁰	0.9	1.2914	90	22.77
				GENPROVE ₁₀₀ ^{0.02}	1.10×10^{-4}	25.3728	0	6.83
	99.999% Confidence			Sampling	3.11×10^{-4}	26.4949	0	0.70
	ConvLarge	123180	<i>Prior Work</i>	Box	0.98	0.0040	0	0.08
				HybridZono	0.97	0.0202	0	0.08
				DeepZono	1.0	0.9907	100	23.62
			Zonotope	1.0	0.9560	100	23.62	
			<i>Our Work</i>	GENPROVE ⁰	0.95	0.7649	95	23.14
				GENPROVE ₁₀₀ ^{0.02}	1.61×10^{-4}	41.3746	0	9.38
	99.999% Confidence			Sampling	2.06×10^{-4}	42.2111	0	0.71
	ConvSmall	24676	<i>Prior Work</i>	Box	1.0	0.0039	0	0.04
				HybridZono	1.0	0.0048	0	0.04
				DeepZono	1.0	1.3525	100	23.62
Zonotope			1.0	1.3428	100	23.62		
<i>Our Work</i>			GENPROVE ⁰	0.0	11.1377	0	6.54	
			GENPROVE ₁₀₀ ^{0.02}	3.26×10^{-5}	14.8033	0	6.39	
99.999% Confidence			Sampling	1.59×10^{-3}	15.2826	0	0.59	
Zappos50k	ConvMed	63804	<i>Prior Work</i>	Box	1.0	0.0053	0	0.13
				HybridZono	1.0	0.0380	0	0.13
				DeepZono	1.0	1.3513	100	23.62
			Zonotope	1.0	1.3402	100	23.62	
			<i>Our Work</i>	GENPROVE ⁰	0.89	3.4067	89	22.67
				GENPROVE ₁₀₀ ^{0.02}	4.53×10^{-5}	25.1927	0	6.61
99.999% Confidence			Sampling	1.13×10^{-3}	27.0847	0	0.67	
ConvLarge	123180	<i>Prior Work</i>	Box	1.0	0.005	0	0.046	
			HybridZono	1.0	0.038	0	0.046	
			DeepZono	1.0	1.359	100	23.623	
		Zonotope	1.0	1.350	100	23.623		
		<i>Our Work</i>	GENPROVE ⁰	0.99	0.414	99	23.581	
			GENPROVE ₁₀₀ ^{0.02}	5.7×10^{-5}	32.058	0	7.160	
99.999% Confidence			Sampling	1.53×10^{-3}	32.124	0	0.679	

TABLE 4.6: Average consistency \hat{C} bound widths, runtime, and memory usage. For these metrics, lower values are better. Additionally, the percentage of runs which ran out of memory is reported as OOM. Unacceptably large widths, which mean the analysis failed to provide useful bounds, are written in red.

4.5.3 RQ3 - Novel Generative Specifications

So far, we have shown that GENPROVE can verify generative specifications like those shown in Figure 4.3, which interpolate between two images with identical attributes. We now address RQ3 and demonstrate how our method applies to five additional generative specifications.

Certifying robustness to head orientation. As shown by Dumoulin & Visin [124], VAEs can generate images of intermediate poses from flipped images. An example of this transformation is shown in Table 4.5 (a). We evaluated line specifications between encodings of horizontally flipped images. For a head, ideally the intermediate reconstructions will be of the intermediate 3D orientations. As pose is not a provided CelebA attribute, the attribute detector should recognize the same attributes for all interpolations. We evaluated $\text{GENPROVE}_{100}^{0.02}$ on images from CelebA dataset and successfully produced tight bounds for all evaluated images. The average bound width was only $1.0 \cdot 10^{-4}$, with average lower bound $l = 0.8433$ and average upper bound $u = 0.8434$. That is, we verified that on average, the target network is robust to 84% of generated interpolations. The results for other method follow the results shown in Table 4.2. That is, they either do not scale or provide bound width close to 1.

Certifying attribute independence for CelebA. We use GENPROVE to demonstrate that attribute detection for one feature is invariant to transformation of an independent feature. Specifically, we verify for a single image the effect of adding a different hair color, as shown in Table 4.5 (b). To achieve this, we find the attribute vector m for “BrownHair” using the 80k training images in the manner described by [125], and compute probabilistic bounds for $\mathcal{C}_j(n_E(o), n_E(o) + 3m, o_{A,j})$ for $j \neq 11$ and the image o . Here, we used the ConvMed attribute detector. Using GENPROVE we are able to prove that 32 out of the 40 attributes are entirely robust to brown hair addition, and 8 of them were not robust. Among the attributes which can be proven to be robust was $i = 39$ for “young” for example. We are able to find that attribute $i = 9$ for “BlondHair” is not entirely robust to the addition of the BrownHair vector, which is expected. Here, our approach is able to find tight lower and upper bounds on the robustness probability of $[0.6038, 0.6039]$ for that attribute. The average interval width for all attributes was $7.87 \cdot 10^{-6}$. One can observe in Table 4.5 (b) that this matches visually what the interpolated images show: the first 6 or so reconstructions appear to have blond hair, whereas the rest have brown hair.

TRAINING	verification of adversarial generative interpolations			
	standard accuracy	adversarial accuracy (PGD [41])	provable accuracy (Box)	bound width ($u - l$)
Standard	99.2%	54.5%	0.0%	0.9999
FGSM [20]	99.5%	97.1%	0.0%	1.0
DiffAI [55]	99.1%	97.7%	92.5%	0.0990

TABLE 4.7: Average number of fully verified images for interpolations of images in the same MNIST class using adversarial region width $\epsilon = 0.1$ for ConvBiggest with 175 816 neurons trained in three ways.

Certifying curved specifications.

We demonstrate the first exact analysis (both deterministic and probabilistic) of a non-convex smooth input for neural networks. Given three encoding vectors, e_0, e_1, e_2 , we create the following quadratic curve that passes through them at $t = 0, 0.5, 1$ respectively:

$$\gamma(t) = e_0 + (4e_1 - e_2 - 3e_0) \cdot t + 2(e_2 + e_0 - 2e_1) \cdot t^2.$$

We use the encoding of an image of a head for e_0 , the encoding of the flipped head for e_2 , and the midpoint of these two encodings perturbed by a scaled moustache attribute vector, m (found as described earlier for the BrownHair attribute vector, but for attribute 22) for $e_1 := 0.5(e_0 + e_2) + 4m$. We visualize this specification in Table 4.5 (c).

We used GENPROVECURVE to demonstrate attribute independence for 29 out of the 40 different attributes. As GENPROVECURVE is exact, it produced a bound width of 0. The average probability of attribute consistency is 0.85. Here, we used a smaller generator architecture, DecoderSmall with only 41 160 neurons, the usual ConvSmall attribute detector. Even though it is exact, GENPROVECURVE was able to verify the non-linear specification in only 12.6 seconds.

Certifying adversarial regions around generative output. Unlike any other pre-existing methods, GENPROVE can be easily applied to higher-dimensional specifications. Specifically, we use generative models to construct a base specification, which we additionally want to be adversarially robust. We define adversarial consistency $\mathcal{C}_{\epsilon, i, n_A, n_D}^{\text{adv}}(e_1, e_2)$ as:

$$\Pr_{e \sim U(\overline{e_1 e_2})} [\forall a \in B_{\infty, \epsilon}(n_D(e)), \arg \max_i n_A(a)_i = i].$$

Here, $B_{\infty, \epsilon}(n_D(e))$ refers to the L_∞ adversarial region of size ϵ around the output of the generator. To handle this we propagate the interval using

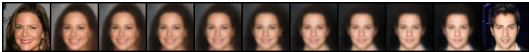

MODEL	INTERPOLATION	UPPER BOUND	BOUND WIDTH
(a) VAE		0.4528	0
(b) FactorVAE		0.32	3.5×10^{-5}
(c) ACAI		0.29	8.8×10^{-5}

TABLE 4.8: Using an interpolation specification with an arcsin distribution between unrelated images to compare the realism of images produced by various generative models. We compute an upper bound on the probability that the out-of-distribution detector (a GAN discriminator here) successfully determines that the generated image is fake.

GENPROVE through the decoder n_D to produce a list of segments and boxes. We compute a box around each segment, and then enlarge each box in the entire list, in every dimension, by ϵ . We then propagate the boxes through n_A . Crucially, these operations all fall under the framework developed in our work, and so this specification can be seen as an instance of GENPROVE.

Because no other method is both capable of handling generative specifications without adversarial regions, or easily extensible to handle this specification, we only use GENPROVE to demonstrate the benefit of DiffAI training. Table 4.7 shows the result of applying GENPROVE to solve this specification on regularly trained networks, FGSM-trained networks [20], and DiffAI-trained networks [55] for the MNIST dataset. We report, in addition to the standard accuracy, the accuracy against the PGD adversary [41] with 5 iterations, and the provability using Box. We finally report the bound width on the generative adversarial specification using GENPROVE. One can see that on a DiffAI-trained network, we are able to provide tight bounds on the adversarial consistency.

Certifying complex specifications.

Finally, we demonstrate the full capabilities of GENPROVE for certifying non-uniform specifications involving naive out-of-distribution detection using a GAN discriminator, and autoencoders specifically trained for disentanglement and interpolation as shown in Table 4.8. Here, we trained two more VAEs on CelebA: (i) ACAI [126] which is designed specifically to

produce realistic encoding interpolations, and (ii) FactorVAE [127] which is designed to learn a latent encoding where each dimension represents an independent disentangled feature. For out-of-distribution detection, we used the discriminator from a vanilla GAN [128]. Each has been modified to use MSE as their reconstruction loss to avoid sigmoids. We use Decoder for all decoders, Encoder for the encoders and the ACAI critic, and EncoderSmall for the GAN discriminator. Further, FactorVAE uses a small feedforward network 5 layers deep (each layer has 100 neurons) as its factorization critic. Each network was trained for 100 epochs, with a batch size of 64. The autoencoders used 64 latent dimensions while the GAN used 128. All other hyperparameters are as in the respective papers.

To demonstrate non-uniform distributions, we use the arcsine distribution over the interpolation specification. Table 4.8 compares the upper bound of an interpolation specification between two unrelated images. A small number means that the discriminator was fooled by the generator in question. We can see that the most successful generator is ACAI, which is specifically trained to produce realistic interpolations.

4.6 RELATED WORK

Next, we review work most closely related to ours.

Certifying generative models. Dvijotham *et al.* [111] verifies lower bounds on a probabilistic property for all inputs in a specification for variational autoencoders with a latent random variable using a dual approach. In contrast, GENPROVE finds tight upper *and* lower bounds on the probability that a property is satisfied given a distribution over a specification. Further, our approach scales to networks that are orders of magnitude larger – we successfully certify CelebA networks with nearly 200k neurons compared to the network used in Dvijotham *et al.* [111] which has only 3 hidden layers of 64 units each.

Convex relaxations. PROVEN [129] proposes a technique to infer confidence intervals on the probability of misclassification from preexisting convex relaxation methods that find linear constraints on outputs. In our evaluation, we show that for interpolations of generative models, convex relaxation methods are unable to prove meaningful bounds. This implies that the linear lower bound function used by PROVEN would be bounded above by 0, and thus because $F_{gt}^L(0.5) \geq F_0^L(0.5)$ and $F_0^L(0.5) = 1$, the lower bound, γ_L , that their system should derive would be $\gamma_L = 0$. This is because even the most precise convex relaxation over the generated images might include many images that are not realistic. For example, the convex hull includes the pixel-wise average of the generated endpoint images, as in Figure 4.1.

Adversarial defenses. Another line of work, smoothing, provides a defense with statistical guarantees [103, 130–133]. In our evaluation, we compared to a variant of this technique, sampling, and demonstrated that GENPROVE computes two orders of magnitude tighter bounds across all the datasets and models. Further, our work provides provable guarantees compared to sampling whose bounds are correct only with some probability (e.g., with 99% confidence).

At the same time, a number of recent adversarial defenses started to incorporate generative models as core component or their approach [134–136]. For all of those, incorporating techniques presented in our work is a natural next step required to certify that the defense is provably correct.

Certifying line segments. Of particular note, the work of Sotoudeh & Thakur [118] also restricts the network inputs to line segments. They

used this method to certify non norm-based properties [137] and to improve Integrated Gradients [138]. In our work we built on the results of Sotoudeh & Thakur [118] and extended them in several major aspects by: (i) computing tight deterministic bounds on the probability of a probabilistic specification, (ii) introducing relaxations that enable scaling to large networks, (iii) ensuring the correctness of the probabilistic guarantees in the presence of these relaxations, and (iv) exploring novel specifications including parametric curves and higher dimensional specifications.

4.7 DISCUSSION

In this chapter, we demonstrated GENPROVE’s use to certify transformations given by generative models. While generative models are intended to represent the underlying data distribution, physical limitations imply they actually generate slightly different distributions. Unfortunately, it is usually not possible to certify how much a given generative model differs from the ground truth. This is because for most real-world applications, the ground truth is only approximated from data. In such cases, our domain is useful for either verifying that the generative model satisfies some property given by a trusted downstream classifier, or verifying that the downstream classifier obeys a property specified by a trusted generator. In this chapter, we predominantly consider verifying classifiers based on a trusted generator. The experiment shown in Table 4.8 is an example where one might consider the converse case: we can evaluate the generator against a trusted classifier that judges whether the produced images appear to be real.

4.8 CONCLUSION

We presented GENPROVE, a scalable non-convex relaxation approach to certify neural network properties when subjected to transformations learned by generative models. Our method supports both deterministic and probabilistic certification and is able to verify, for the first time, interesting visual transformation properties based on latent space interpolation, beyond the reach of prior work.

We provided an evaluation which demonstrated: (i) that probabilistic abstract interpretation was necessary for analyzing complex semantic specifications, (ii) that GENPROVE scales to large networks (200k neurons),

even beating sampling, and (iii) that GENPROVE enables verification of novel semantic specifications.

CONCLUSION AND FUTURE WORK

In this thesis, we developed two frameworks, AI^2 and DIFFAI , and extended DIFFAI to handle semantic probabilistic specifications with generative models with GENPROVE . These frameworks constitute the first *scalable* systems for making sound guarantees on neural networks.

Doing this involved three key insights: (i) we can use the flexible and well researched techniques of overapproximation by abstract interpretation for neural network verification, (ii) we can introduce abstract interpretation into the training pipeline to produce efficiently verifiable networks, and (iii) we can use probabilistic abstract interpretation to verify semantic properties with generative models. Through these contributions we demonstrated that sound verification *is* feasible for production-scale neural networks, thus opening the gates to a plethora of verification possibilities that before were considered either too costly, or too restrictive for usage in deep learning.

The systems presented here have had significant impact on neural network verification: At the time of writing, DIFFAI has 186 stars on github and 23 forks, and has been used in industrial applications [58]. Further research has been developed extending the concept of optimizing a primal overapproximation with backpropagation [99, 139]. Furthermore, since we introduced AI^2 and DIFFAI , much work has been developed extending the idea of verified, provable learning to new networks and properties, and with new methods [57, 60, 106, 109, 110, 140–143].

5.1 FUTURE WORK

The work of this thesis points towards four main directions of future work: (i) improvements in pure verification, (ii) understanding, and fixing, the *provable training gap*, (iii) applying our techniques to other areas, and (iv) utilizing the guarantees.

5.1.1 *Verification Outlook*

While AI² made neural network verification practical for the first time, verification has yet to tackle nets such as GPT-2 [144], and is still mostly limited to adversarial ball type queries. Here we discuss possible areas for future research.

Expanding on Sound Guarantees. While perturbation radius style verification addresses the first generation of adversarial examples, a common concern is that these kinds of specifications are too limited in terms of what they consider adversarial. With GENPROVE we began to expand the types of guarantees that neural network verification systems can make; however it still largely addresses point-wise verification. An important step for neural network verification would be in handling domain invariants, where guarantees are made about all inputs. While methods that verify or find Lipschitz constants for neural networks [145] present global verification in a sense, the scale of invariants such as change of perspective, lighting, and even image translation, are often far beyond what would be meaningful here.

Paulsen, Wang & Wang [146] demonstrated a promising direction with a system that could compare the relative capabilities of neural networks. This technique, however, has yet to be shown to extend to convolutional networks. A possible way to extend this kind of work to larger networks would be by applying the methods of DIFFAI.

Improving Verification Speed and Accuracy. Since releasing AI² we have improved the speed and accuracy of the zonotope domain by introducing new transformers [147]. Further improvements have since been made constructing abstract domains and transformers specifically for neural networks [57]. As probabilistic abstract interpretation is a new concept in neural network analysis, we believe that similar improvements can be made to construct domains for generative models like GENPROVE.

5.1.2 *Training Outlook*

Since releasing DIFFAI, it was discovered by [59] that significantly better accuracy and certifiability could be achieved by using the Box domain and using a training schedule and refined loss function. While not included in the scope of this thesis, we have since significantly improved the usability of this system and provided an updated evaluation [49] using this training

schedule and loss function. We furthermore demonstrated that DIFFAI could scale to deep residual networks with up to 4.5 million neurons, or up to 25 million parameters.

An Open Problem: The Provable Training Gap. While there have been many later works improving on the training results shown here [59, 97, 99, 139], progress has plateaued and provable training methods remain far from being able to produce networks with certifiability anywhere near state-of-the-art accuracy on relatively simple datasets such as CIFAR10. COLT [148] for example, which is based on different methods entirely, achieves a record breaking certified robustness of 60.5% on a radius of $2/255$ on CIFAR10, compared with 45.5% for DIFFAI. Compared with a standard state-of-the-art accuracy of above 95% for CIFAR10, the lack of progress presents a barrier to practical adoption of provable training.

This dilemma has raised fundamental theoretical questions analogous to those that have been answered for standard training: Do certifiable networks exist? How large would they need to be? Can gradient descent efficiently find them?

In Baader, Mirman & Vechev [61] we answer the first of these questions by proving an analog to the universal approximation theorem: we demonstrate that networks exist that can be certified, with interval, to robustly approximate any function. Wang *et al.* [149] extends this idea by showing that such networks can be constructed in a constant number of layers.

While it is reassuring to know that a search for provable networks is not in vain, this result runs counter to what is observed in practice [150]. We therefore prove in Mirman, Baader & Vechev [62] that there is a fundamental theoretical barrier to training such networks with known techniques (beyond the known barriers to training standard and robust networks) and that provably robust networks must be deeper than their simply robust counterparts.

On the other hand, an entirely different direction is possible: designing new kinds of models which are certifiable by construction. Recent works such as Cohen, Rosenfeld & Kolter [103] take this approach: instead of providing sound guarantees with 100% certainty on a standard model, they provide high certainty guarantees on a model with a modified notion of inference. Other works have taken similar approaches [130, 151, 152].

5.1.3 *Further Application of Our Techniques*

In this thesis we demonstrated that formal verification could scale to systems of unprecedented size. We furthermore demonstrated that even at that scale, it is possible to *train* networks to fit verification goals. As neural networks can be considered program templates (they can even literally be used to learn explicit algorithms [153–155]), one can consider DIFFAI a form of program synthesis [156–160], and in particular a way to solve a kind of quantitative synthesis problem [50].

A natural direction is thus to determine if the methods which apply to building certifiable neural networks transfer to other domains, such as for generating programs that are built to formal specifications in human readable target languages, or even learning analytical networks (e.g., NTMs) which satisfy specifications. Similarly, the theoretical results that have arisen may also generalize to fundamental questions in program synthesis.

5.1.4 *Utilizing Sound Guarantees*

Finally, perhaps the least explored direction is to use these newfound neural network certification capabilities as part of larger systems. As seen in Chapter 4, sound techniques can sometimes outperform even sampling. This suggests that they might be used in place of sampling in some systems. Many systems might benefit from robustness tests, such as those in environments where a user is well aware of the presence of a network (e.g., awareness checking in self-driving cars). How should such systems behave when it is discovered that robust decisions can not be shown?

A

APPENDIX

A.1 EXTENDED DIFFAI RESULTS

Here we include further results for scalability and training efficacy of DIFFAI.

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	121.92	2.4	40.8	100.0	100.0	
	PGD	368.76	1.0	3.8	100.0	100.0	
	Box	592.80	5.6	13.6	33.0	53.0	
ConvSmall	Baseline	123.36	1.2	2.4	100.0	49.8	
	PGD	515.64	0.8	1.8	100.0	22.2	
	Box	690	2.4	4.4	17.8	5.8	
ConvBig	Baseline	367.80	0.8	3.0	100.0	100.0	
	PGD	1847.76	0.2	1.6	100.0	99.8	
	Box	1445.76	1.0	2.4	14.0	3.4	
ConvSuper	Baseline	878.28	1.6	2.4	100.0	97.2	
	PGD	4867.56	1.2	1.6	100.0	88.8	
	Box	3148.68	1.0	2.8	11.8	3.6	
Skip	Baseline	731.40	1.4	3.8	100.0	100.0	
	PGD	3935.04	1.0	2.0	100.0	83.4	
	Box	2734.44	1.6	4.4	13.6	5.8	

TABLE A.1: MNIST with $\epsilon = 0.1$

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	120.84	3.0	99.0	100.0	100.0	
	PGD	357.48	2.0	10.2	100.0	100.0	
	Box	570.84	13.4	50.6	77.8	84.4	
ConvSmall	Baseline	124.44	1.6	20.2	100.0	100.0	
	PGD	518.64	1.8	4.6	100.0	100.0	
	Box	678.36	3.2	9.0	28.2	19.4	
ConvBig	Baseline	368.76	2.4	18.2	100.0	100.0	
	PGD	1863.12	1.6	4.0	100.0	100.0	
	Box	1436.40	3.4	6.2	23.2	18.0	
ConvSuper	Baseline	895.56	1.2	15.0	100.0	100.0	
	PGD	5021.28	1.0	1.0	100.0	100.0	
	Box	3216.72	2.8	8.0	19.0	23.0	

TABLE A.2: MNIST with $\epsilon = 0.3$

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	254.52	53.4	73.8	100.0	100.0	
	PGD	956.64	45.4	53.6	100.0	100.0	
	Box	829.08	48.6	57.8	84.4	65.4	
ConvMed	Baseline	292.44	40.2	44.0	100.0	54.0	
	PGD	1472.16	38.8	43.6	100.0	52.4	
	Box	1040.04	42.8	46.4	74.6	47.8	
ConvBig	Baseline	1317.00	32.4	36.2	100.0	100.0	
	PGD	8574.72	31.4	35.8	100.0	100.0	
	Box	4307.88	55.0	58.6	76.4	61.4	
ConvSuper	Baseline	4683.24	37.4	42.4	100.0	100.0	
	PGD	29828.52	35.4	41.0	100.0	100.0	
	Box	14849.40	52.8	59.2	83.8	64.2	
Skip	Baseline	802.92	36.8	41.8	100.0	88.0	
	PGD	4828.68	33.6	40.2	100.0	82.8	
	Box	2980.92	38.0	45.4	72.2	47.8	

TABLE A.3: CIFAR10 with $\epsilon = 0.007$

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	263.16	57.6	96.8	100.0	100.0	
	PGD	1000.68	46.6	46.6	61.4	100.0	
	Box	849.00	52.2	68.8	90.0	82.6	
ConvMed	Baseline	98.20	40.4	61.8	100.0	100.0	
	PGD	1546.92	42.2	54.8	100.0	97.8	
	Box	1061.52	45.8	60.0	85.8	64.8	
ConvBig	Baseline	1329.48	37.2	61.6	100.0	100.0	
	PGD	8733.84	41.6	56.2	100.0	100.0	
	Box	4355.76	51.6	61.4	83.2	75.8	
ConvSuper	Baseline	4724.76	39.8	66.2	100.0	100.0	
	PGD	31140.72	39.6	56.2	100.0	100.0	
	Box	15314.76	54.2	64.6	83.8	87.6	
Skip	Baseline	805.32	39.6	69.4	100.0	100.0	
	PGD	4916.04	37.0	54.0	100.0	100.0	
	Box	2789.52	52.6	68.6	90.8	78.0	

TABLE A.4: CIFAR10 with $\epsilon = 0.03$

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	151.50	22.6	13.8	100.0	93.8	
	PGD	459.61	19.2	24.0	100.0	88.6	
	Box	722.46	43.8	10.0	66.2	32.2	
ConvMed	Baseline	144.32	15.0	19.8	100.0	54.8	
	PGD	624.59	14.0	4.4	100.0	40.0	
	Box	838.21	23.6	9.0	66.2	32.2	
ConvBig	Baseline	533.31	11.6	9.6	100.0	98.4	
	PGD	2745.82	14.0	3.6	100.0	96.4	
	Box	2065.11	22.2	7.4	57.4	20.8	
Skip	Baseline	969.48	13.8	10.4	100.0	96.8	
	PGD	5844.00	13.6	5.0	100.0	86.8	
	Box	3352.24	25.0	6.6	53.6	23.0	

TABLE A.5: SVHN with $\epsilon = 0.01$

Model	Train Method	Train Time (s)	Test Error %	Lower Bound %		Upper Bound %	
				PGD	Box	hSwitch	
FFNN	Baseline	121.08	3.4	99.2	100.0	100.0	
	PGD	345.24	3.6	13.8	100.0	100.0	
	Box	568.68	99.1	91.0	100.0	100.0	
ConvMed	Baseline	123.36	1.6	61.6	100.0	100.0	
	PGD	507.84	1.0	5.4	100.0	100.0	
	Box	668.04	3.4	14.8	28.6	28.2	
ConvBig	Baseline	368.40	1.6	48	100.0	100.0	
	PGD	1819.08	1.2	3.0	100.0	100.0	
	Box	1429.56	1.8	4.2	11.8	22.0	
Skip	Baseline	687.72	1.4	58.6	100.0	100.0	
	PGD	3791.76	1.2	3.4	100.0	100.0	
	Box	2310.12	3.2	10.0	20.6	29.4	

TABLE A.6: F-MNIST with $\epsilon = 0.1$

BIBLIOGRAPHY

1. Yuan, Z., Lu, Y., Wang, Z. & Xue, Y. *Droid-Sec: deep learning in android malware detection* in *ACM SIGCOMM 2014 Conference* (2014).
2. Gupta, R., Pal, S., Kanade, A. & Shevade, S. *Deepfix: Fixing common c language errors by deep learning* in *AAAI* (2017).
3. Pradel, M. & Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages (PACMPL)* (2018).
4. Chen, Z., Komrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D. & Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).
5. Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J. & Zieba, K. End to End Learning for Self-Driving Cars. *arXiv preprint arxiv:1604.07316* (2016).
6. Maqueda, A. I., Loquercio, A., Gallego, G., Garcia, N. & Scaramuzza, D. *Event-based vision meets deep learning on steering prediction for self-driving cars* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018).
7. Katz, G., Barrett, C., Dill, D. L., Julian, K. & Kochenderfer, M. J. *Reluplex: An efficient SMT solver for verifying deep neural networks* in *International Conference on Computer Aided Verification (CAV)* (2017).
8. Singh, N. S., Hariharan, S. & Gupta, M. in *Advances in Data Sciences, Security and Applications* 375 (Springer, 2020).
9. Yi, X., Walia, E. & Babyn, P. Generative adversarial network in medical imaging: A review. *Medical image analysis* (2019).
10. Amato, F., López, A., Peña-Méndez, E. M., Vañhara, P., Hampl, A. & Havel, J. Artificial neural networks in medical diagnosis. *J Appl Biomed* **11** (2013).
11. Becker, S., Cheridito, P. & Jentzen, A. Pricing and hedging American-style options with deep learning. *Journal of Risk and Financial Management* **13**, 158 (2020).

12. Horvath, B., Muguruza, A. & Tomas, M. Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance* **21**, 11 (2021).
13. Balunovic, M., Bielik, P. & Vechev, M. T. *Learning to Solve SMT Formulas*. in *NeurIPS* (2018).
14. Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T. & Leather, H. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536* (2020).
15. Wang, Z. & O'Boyle, M. Machine learning in compiler optimization. *Proceedings of the IEEE* **106**, 1879 (2018).
16. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J. & Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).
17. Nguyen, A. M., Yosinski, J. & Clune, J. *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images* in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015).
18. Tabacof, P. & Valle, E. *Exploring the space of adversarial images* in *International Joint Conference on Neural Networks (IJCNN)* (2016).
19. Grosse, K., Papernot, N., Manoharan, P., Backes, M. & McDaniel, P. D. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. *arXiv preprint arxiv:1606.04435* (2016).
20. Goodfellow, I. J., Shlens, J. & Szegedy, C. *Explaining and harnessing adversarial examples* in *ICLR* (2015).
21. Moosavi-Dezfooli, S., Fawzi, A. & Frossard, P. *DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks* in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).
22. Huang, R., Xu, B., Schuurmans, D. & Szepesvári, C. Learning with a Strong Adversary. *arXiv preprint arxiv:1511.03034* (2015).
23. Gu, S. & Rigazio, L. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068* (2014).
24. Shaham, U., Yamada, Y. & Negahban, S. Understanding Adversarial Training: Increasing Local Stability of Neural Nets through Robust Optimization. *arXiv preprint arxiv:1511.05432* (2015).
25. Carlini, N. & Wagner, D. A. *Towards Evaluating the Robustness of Neural Networks* in *Symposium on Security and Privacy (SP)* (2017).

26. Pei, K., Cao, Y., Yang, J. & Jana, S. *DeepXplore: Automated whitebox testing of deep learning systems in Symposium on Operating Systems Principles (SOSP)* (2017).
27. Huang, X., Kwiatkowska, M., Wang, S. & Wu, M. *Safety verification of deep neural networks in International Conference on Computer Aided Verification (CAV)* (2017).
28. Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., Rahmati, A. & Song, D. Robust Physical-World Attacks on Machine Learning Models. *arXiv preprint arXiv:1707.08945* (2017).
29. Kurakin, A., Goodfellow, I. J. & Bengio, S. Adversarial examples in the physical world. *arXiv preprint arxiv:1607.02533* (2016).
30. Athalye, A. & Sutskever, I. Synthesizing robust adversarial examples. *arXiv preprint arXiv:1707.07397* (2017).
31. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *ImageNet Classification with Deep Convolutional Neural Networks in NeurIPS* (2012).
32. Fan, A., Lewis, M. & Dauphin, Y. *Hierarchical Neural Story Generation in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)* (2018).
33. Wong, E., Schmidt, F. & Kolter, Z. *Wasserstein adversarial examples via projected sinkhorn iterations in ICML* (2019).
34. Croce, F. & Hein, M. *Minimally distorted adversarial examples with a fast adaptive boundary attack in ICML* (2020).
35. Croce, F. & Hein, M. *Sparse and imperceivable adversarial attacks in Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (2019).
36. Tramèr, F., Papernot, N., Goodfellow, I. J., Boneh, D. & McDaniel, P. D. The Space of Transferable Adversarial Examples. *arXiv preprint arxiv:1704.03453* (2017).
37. Papernot, N., McDaniel, P. & Goodfellow, I. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277* (2016).
38. Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B. & Swami, A. *Practical black-box attacks against machine learning in Asia Conference on Computer and Communications Security* (2017).

39. Shukla, S. N., Sahu, A. K., Willmott, D. & Kolter, J. Z. Black-box Adversarial Attacks with Bayesian Optimization. *arXiv preprint arXiv:1909.13857* (2019).
40. Andriushchenko, M., Croce, F., Flammarion, N. & Hein, M. *Square attack: a query-efficient black-box adversarial attack via random search in European Conference on Computer Vision* (2020).
41. Madry, A., Makelov, A., Schmidt, L., Tsipras, D. & Vladu, A. Towards deep learning models resistant to adversarial attacks (2018).
42. Carlini, N., Katz, G., Barrett, C. & Dill, D. L. Ground-Truth Adversarial Examples. *arXiv preprint arXiv:1709.10207* (2017).
43. Papernot, N., McDaniel, P. D., Wu, X., Jha, S. & Swami, A. *Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks in IEEE Symposium on Security and Privacy (SP)* (2016).
44. Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I., Boneh, D. & McDaniel, P. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204* (2017).
45. Wong, E., Rice, L. & Kolter, J. Z. *Fast is better than free: Revisiting adversarial training in ICLR* (2019).
46. Stutz, D., Hein, M. & Schiele, B. *Confidence-calibrated adversarial training: Generalizing to unseen attacks in ICML* (2020).
47. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A. V. & Criminisi, A. *Measuring Neural Net Robustness with Constraints in NeurIPS* (2016).
48. Croce, F. & Hein, M. *Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks in ICML* (2020).
49. Mirman, M., Singh, G. & Vechev, M. A provable defense for deep residual networks. *arXiv preprint arXiv:1903.12519* (2019).
50. Bloem, R., Chatterjee, K., Henzinger, T. A. & Jobstmann, B. *Better quality in synthesis through quantitative objectives in International Conference on Computer Aided Verification (CAV)* (2009).
51. Huang, G., Liu, Z., van der Maaten, L. & Weinberger, K. Q. *Densely connected convolutional networks in CVPR* (2017).
52. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* (1998).

53. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. & Fei-Fei, L. *Imagenet: A large-scale hierarchical image database* in *Conference on Computer Vision and Pattern Recognition (CVPR)* (2009).
54. Gehr, T., Mirman, M., Tsankov, P., Drachler Cohen, D., Vechev, M. & Chaudhuri, S. *AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation* in *Symposium on Security and Privacy (SP)* (2018).
55. Mirman, M., Gehr, T. & Vechev, M. *Differentiable Abstract Interpretation for Provably Robust Neural Networks* in *ICML* (2018).
56. Mirman, M., Hägele, A., Bielik, P., Gehr, T. & Vechev, M. *Robustness certification with generative models* in *PLDI* (2021).
57. Singh, G., Gehr, T., Püschel, M. & Vechev, M. *An abstract domain for certifying neural networks* in *POPL* (2019).
58. Ayers, E. W., Eiras, F., Hawasly, M. & Whiteside, I. *PaRoT: a practical framework for robust deep neural network training* in *NASA Formal Methods Symposium* (2020).
59. Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Mann, T. & Kohli, P. *On the Effectiveness of Interval Bound Propagation for Training Verifiably Robust Models*. *arXiv preprint arXiv:1810.12715* (2018).
60. Zhang, Y., Albarghouthi, A. & D'Antoni, L. *Robustness to programmable string transformations via augmented abstract training* in *ICML* (2020).
61. Baader, M., Mirman, M. & Vechev, M. *Universal Approximation with Certified Networks* in *International Conference on Learning Representations (ICLR)* (2020).
62. Mirman, M., Baader, M. & Vechev, M. *The Fundamental Limits of Interval Arithmetic for Neural Networks* in *In Submission* (2021).
63. Pulina, L. & Tacchella, A. *An Abstraction-Refinement Approach to Verification of Artificial Neural Networks* in *International Conference on Computer Aided Verification (CAV)* (2010).
64. Cousot, P. & Cousot, R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints* in *Symposium on Principles of Programming Languages (POPL)* (1977).
65. Cousot, P. & Cousot, R. *Abstract Interpretation Frameworks*. *Journal of Logic and Computation* **2**, 511 (1992).

66. Singh, G., Püschel, M. & Vechev, M. T. *Fast polyhedra abstract domain in Symposium on Principles of Programming Languages (POPL)* (2017).
67. Hubel, D. H. & Wiesel, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology* **160**, 106 (1962).
68. Ghorbal, K., Goubault, E. & Putot, S. *The zonotope abstract domain taylor₁₊ in International Conference on Computer Aided Verification (CAV)* (2009).
69. Cousot, P. & Halbwachs, N. *Automatic Discovery of Linear Restraints Among Variables of a Program in Symposium on Principles of Programming Languages (POPL)* (1978).
70. Ghorbal, K., Goubault, E. & Putot, S. *A Logical Product Approach to Zonotope Intersection in International Conference on Computer Aided Verification (CAV)* (2010).
71. Jeannet, B. & Miné, A. *Apron: A Library of Numerical Abstract Domains for Static Analysis in International Conference on Computer Aided Verification (CAV)* (2009).
72. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I. & Gupta, A. *Static Analysis in Disjunctive Numerical Domains in International Symposium on Static Analysis (SAS)* (2006).
73. Popeea, C. & Chin, W.-N. *Inferring Disjunctive Postconditions in Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues (ASIAN)* (Tokyo, Japan, 2007).
74. Krizhevsky, A. Learning multiple layers of features from tiny images (2009).
75. Lecun, Y., Jackel, L., Boser, B. E., Denker, J., Graf, H., Guyon, I., Henderson, D., Howard, R. E. & Hubbard, W. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine* **27** (1989).
76. Scheibler, K., Winterer, L., Wimmer, R. & Becker, B. *Towards Verification of Artificial Neural Networks in Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)* (2015).
77. He, W., Wei, J., Chen, X., Carlini, N. & Song, D. *Adversarial Example Defense: Ensembles of Weak Defenses are not Strong in USENIX (WOOT 17)* (2017).

78. Sabour, S., Cao, Y., Faghri, F. & Fleet, D. J. Adversarial Manipulation of Deep Representations. *arXiv preprint arxiv:1511.05122* (2015).
79. Hein, M. & Andriushchenko, M. *Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation in NeurIPS* (2017).
80. Majumdar, R. & Saha, I. *Symbolic robustness analysis in Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)* (2009).
81. Chaudhuri, S., Gulwani, S., Lubliner, R. & Navidpour, S. *Proving programs robust in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)* (2011).
82. Chaudhuri, S., Gulwani, S. & Lubliner, R. *Continuity Analysis of Programs in Proceedings of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL)* (2010).
83. Goubault, E. & Putot, S. *Robustness Analysis of Finite Precision Implementations in Programming Languages and Systems - 11th Asian Symposium (APLAS)* (2013).
84. Wong, E. & Kolter, Z. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope (2018).
85. Raghunathan, A., Steinhardt, J. & Liang, P. *Certified Defenses against Adversarial Examples in ICLR* (2018).
86. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D. & Rival, X. *A static analyzer for large safety-critical software in Programming Language Design and Implementation (PLDI)* (2003).
87. Chaudhuri, S., Clochard, M. & Solar-Lezama, A. *Bridging boolean and quantitative synthesis using smoothed proof search in POPL* (2014).
88. Goubault, E. & Putot, S. Perturbed affine arithmetic for invariant computation in numerical program analysis. *arXiv preprint arXiv:0807.2961* (2008).
89. Goubault, E., Le Gall, T. & Putot, S. An accurate join for zonotopes, preserving affine input/output relations. *Electronic Notes in Theoretical Computer Science* (2012).
90. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. Automatic differentiation in PyTorch (2017).

91. Kingma, D. P. & Ba, J. *Adam: A method for stochastic optimization* in *ICLR* (2015).
92. Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C. & Garcia, R. *Incorporating second-order functional knowledge for better option pricing* in *Advances in Neural Information Processing Systems (NIPS)* (2001).
93. Xiao, H., Rasul, K. & Vollgraf, R. *Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms*. *arXiv preprint arXiv:1708.07747* (2017).
94. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B. & Ng, A. Y. *Reading Digits in Natural Images with Unsupervised Feature Learning* in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011).
95. LeCun, Y. A., Bottou, L., Orr, G. B. & Müller, K.-R. in *Neural networks: Tricks of the trade 9* (Springer, 2012).
96. He, K., Zhang, X., Ren, S. & Sun, J. *Deep Residual Learning for Image Recognition* in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).
97. Wong, E., Schmidt, F., Metzen, J. H. & Kolter, J. Z. *Scaling provable adversarial defenses* in *NeurIPS* (2018).
98. Singh, G., Gehr, T., Mirman, M., Püschel, M. & Vechev, M. *Fast and effective robustness certification* in *NeurIPS* (2018).
99. Zhang, H., Chen, H., Xiao, C., Gowal, S., Stanforth, R., Li, B., Boning, D. & Hsieh, C.-J. *Towards stable and efficient training of verifiably robust neural networks* in *ICLR* (2020).
100. Chiang, P.-Y., Ni, R., Abdelkader, A., Zhu, C., Studer, C. & Goldstein, T. *Certified defenses for adversarial patches* in *ICLR* (2020).
101. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.-W., Huang, M., Kailkhura, B., Lin, X. & Hsieh, C.-J. *Automatic perturbation analysis for scalable certified robustness and beyond* in *NeurIPS* (2020).
102. Wang, Z., Albarghouthi, A., Prakriya, G. & Jha, S. *Interval universal approximation for neural networks* in *POPL* (2022).
103. Cohen, J., Rosenfeld, E. & Kolter, Z. *Certified adversarial robustness via randomized smoothing* in *ICML* (2019).
104. Tjeng, V., Xiao, K. & Tedrake, R. *Evaluating Robustness of Neural Networks with Mixed Integer Programming* in *ICLR* (2019).

105. Dvijotham, K., Gowal, S., Stanforth, R., Arandjelovic, R., O'Donoghue, B., Uesato, J. & Kohli, P. Training verified learners with learned verifiers. *arXiv preprint arXiv:1805.10265* (2018).
106. Salman, H., Yang, G., Zhang, H., Hsieh, C.-J. & Zhang, P. *A convex relaxation barrier to tight robustness verification of neural networks in NeurIPS* (2019).
107. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T. A. & Kohli, P. *A Dual Approach to Scalable Verification of Deep Networks. in UAI* (2018).
108. Wang, S., Pei, K., Whitehouse, J., Yang, J. & Jana, S. *Efficient formal safety analysis of neural networks in NeurIPS* (2018).
109. Balunovic, M., Baader, M., Singh, G., Gehr, T. & Vechev, M. *Certifying Geometric Robustness of Neural Networks in NeurIPS* (2019).
110. Liu, C., Tomioka, R. & Cevher, V. *On Certifying Non-uniform Bound against Adversarial Attacks in ICML* (2019).
111. Dvijotham, K., Garnelo, M., Fawzi, A. & Kohli, P. Verification of deep probabilistic models. *arXiv preprint arXiv:1812.02795* (2018).
112. Gautam, A., Sit, M. & Demir, I. Realistic River Image Synthesis using Deep Generative Adversarial Networks. *arXiv preprint arXiv:2003.00826* (2020).
113. Liu, J., Ni, B., Yan, Y., Zhou, P., Cheng, S. & Hu, J. *Pose transferrable person re-identification in CVPR* (2018).
114. Ge, Y., Li, Z., Zhao, H., Yin, G., Yi, S., Wang, X., et al. *Fd-gan: Pose-guided feature distilling gan for robust person re-identification in NeurIPS* (2018).
115. Qian, X., Fu, Y., Xiang, T., Wang, W., Qiu, J., Wu, Y., Jiang, Y.-G. & Xue, X. *Pose-normalized image generation for person re-identification in ECCV* (2018).
116. Wang, X., Man, Z., You, M. & Shen, C. Adversarial generation of training examples: applications to moving vehicle license plate recognition. *arXiv preprint arXiv:1707.03124* (2017).
117. Cousot, P. & Monerau, M. *Probabilistic Abstract Interpretation in Programming Languages and Systems* (ed Seidl, H.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), 169.
118. Sotoudeh, M. & Thakur, A. V. Computing Linear Restrictions of Neural Networks. *arXiv preprint arXiv:1908.06214* (2019).

119. LeCun, Y., Cortes, C. & Burges, C. MNIST handwritten digit database. *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist> 2 (2010).
120. Yu, A. & Grauman, K. *Semantic Jitter: Dense Supervision for Visual Comparisons via Synthetic Images* in ICCV (2017).
121. Yu, A. & Grauman, K. *Fine-Grained Visual Comparisons with Local Learning* in CVPR (2014).
122. Liu, Z., Luo, P., Wang, X. & Tang, X. *Deep Learning Face Attributes in the Wild* in ICCV (2015).
123. Kingma, D. P. & Welling, M. *Auto-encoding variational bayes* in ICLR (2013).
124. Dumoulin, V. & Visin, F. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).
125. Larsen, A. B. L., Sønderby, S. K., Larochelle, H. & Winther, O. *Autoencoding beyond pixels using a learned similarity metric* in ICML (2016).
126. Berthelot, D., Raffel, C., Roy, A. & Goodfellow, I. *Understanding and Improving Interpolation in Autoencoders via an Adversarial Regularizer* in ICLR (2018).
127. Kim, H. & Mnih, A. *Disentangling by factorising* in ICML (2018).
128. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. C. & Bengio, Y. *Generative Adversarial Nets* in *NeurIPS* (2014).
129. Weng, T.-W., Chen, P.-Y., Nguyen, L. M., Squillante, M. S., Oseledets, I. & Daniel, L. *PROVEN: Certifying Robustness of Neural Networks with a Probabilistic Approach* in ICML (2019).
130. Lecuyer, M., Atlidakis, V., Geambasu, R., Hsu, D. & Jana, S. *Certified robustness to adversarial examples with differential privacy in S&P* (2019).
131. Liu, X., Cheng, M., Zhang, H. & Hsieh, C.-J. *Towards robust neural networks via random self-ensemble* in ECCV (2018).
132. Li, B., Chen, C., Wang, W. & Carin, L. *Second-order adversarial attack and certifiable robustness*. *arXiv preprint arXiv:1809.03113* (2018).
133. Cao, X. & Gong, N. Z. *Mitigating evasion attacks to deep neural networks via region-based classification* in ACSAC (2017).

134. Li, Y., Bradshaw, J. & Sharma, Y. *Are Generative Classifiers More Robust to Adversarial Attacks?* in *ICML* (2019).
135. Samangouei, P., Kabkab, M. & Chellappa, R. *Defense-GAN: Protecting Classifiers Against Adversarial Attacks Using Generative Models* in *ICLR* (2018).
136. Song, Y., Kim, T., Nowozin, S., Ermon, S. & Kushman, N. *PixelDefend: Leveraging Generative Models to Understand and Defend against Adversarial Examples* in *ICLR* (2018).
137. Julian, K. D., Kochenderfer, M. J. & Owen, M. P. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics* **42**, 598 (2018).
138. Sundararajan, M., Taly, A. & Yan, Q. *Axiomatic attribution for deep networks* in *ICML* (2017).
139. Wang, S., Chen, Y., Abdou, A. & Jana, S. Mixtrain: Scalable training of verifiably robust neural networks. *arXiv preprint arXiv:1811.02625* (2018).
140. Zhang, Y., Albarghouthi, A. & D'Antoni, L. *Certified Robustness to Programmable Transformations in LSTMs* in *EMNLP* (2021).
141. Croce, F., Andriushchenko, M. & Hein, M. Provable robustness of relu networks via maximization of linear regions. *arXiv preprint arXiv:1810.07481* (2018).
142. Croce, F. & Hein, M. *Provable robustness against all adversarial l_p -perturbations for $p \geq 1$* in *ICLR* (2019).
143. Jia, R., Raghunathan, A., Göksel, K. & Liang, P. *Certified Robustness to Adversarial Word Substitutions*. in *EMNLP/IJCNLP (1)* (2019).
144. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. & Sutskever, I. Language Models are Unsupervised Multitask Learners (2019).
145. Fazlyab, M., Robey, A., Hassani, H., Morari, M. & Pappas, G. J. *Efficient and accurate estimation of lipschitz constants for deep neural networks* in *NeurIPS* (2019).
146. Paulsen, B., Wang, J. & Wang, C. *Reludiff: Differential verification of deep neural networks* in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 714.
147. Singh, G., Gehr, T., Mirman, M., Püschel, M. & Vechev, M. *Fast and effective robustness certification* in *International Conference on Neural Information Processing Systems (NeurIPS 2018)*.

148. Balunovic, M. & Vechev, M. *Adversarial training and provable defenses: Bridging the gap in ICLR* (2019).
149. Wang, Z., Albarghouthi, A., Prakriya, G. & Jha, S. *Interval Universal Approximation for Neural Networks in POPL* (2020).
150. Salman, H., Yang, G., Zhang, H., Hsieh, C.-J. & Zhang, P. *A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks in NeurIPS* (2019).
151. Salman, H., Yang, G., Li, J., Zhang, P., Zhang, H., Razenshteyn, I. & Bubeck, S. *Provably Robust Deep Learning via Adversarially Trained Smoothed Classifiers in NeurIPS* (2019).
152. Lin, W.-Y., Sheikholeslami, F., Rice, L., Kolter, J. Z., et al. *Certified robustness against physically-realizable patch attack via randomized cropping in ICLR* (2021).
153. Graves, A., Wayne, G. & Danihelka, I. Neural Turing machines. *arXiv preprint arXiv:1410.5401* (2014).
154. Kaiser, Ł. & Sutskever, I. Neural GPU learn algorithms. *arXiv preprint arXiv:1511.08228* (2015).
155. Mirman, M., Dimitrov, D., Djordjevic, P., Gehr, T. & Vechev, M. *Training Neural Machines with Trace-Based Supervision in ICML* (2018).
156. Lezama, A. S. *Program synthesis by sketching* PhD thesis (PhD thesis, EECS Department, University of California, Berkeley, 2008).
157. Huang, J., Smith, C., Bastani, O., Singh, R., Albarghouthi, A. & Naik, M. *Generating Programmatic Referring Expressions via Program Synthesis in ICML* (2020).
158. Emerson, T. & Burstein, M. H. *Development of a constraint-based airlift scheduler by program synthesis from formal specifications in 14th IEEE International Conference on Automated Software Engineering* (1999).
159. Manna, Z. & Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1980).
160. Vechev, M., Yahav, E. & Yorsh, G. *Abstraction-guided synthesis of synchronization in Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2010).

CURRICULUM VITAE

PERSONAL DATA

Name	Matthew Mirman
Date of Birth	May 6, 1990
Place of Birth	New York, USA
Citizen of	United States of America

EDUCATION

2012 – 2014	Carnegie Mellon University, Pittsburgh, PA, USA <i>Final degree: MScS</i>
2009 – 2012	Carnegie Mellon University, Pittsburgh, PA, USA <i>Final degree: BS in Computer Science</i>