Introduction to Neural Network Verification

Other titles in Foundations and Trends® in Programming Languages

Refinement Types: A Tutorial
Ranjit Jhala and Niki Vazou
ISBN: 978-1-68083-884-8

Shape Analysis
Bor-Yuh Evan Chang, Cezara Drăgoi, Roman Manevich, Noam Rinetzky and Xavier Rival
ISBN: 978-1-68083-732-2

Progress of Concurrent Objects
Hongjin Liang and Xinyu Feng
ISBN: 978-1-68083-672-1

QED at Large: A Survey of Engineering of Formally Verified Software
Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric and Zachary Tatlock
ISBN: 978-1-68083-594-6

Reconciling Abstraction with High Performance: A MetaOCaml approach
Oleg Kiselyov
ISBN: 978-1-68083-436-9

Introduction to Neural Network Verification

Aws Albarghouthi
University of Wisconsin–Madison
USA
aws@cs.wisc.edu

the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

A. Albarghoutli. *Introduction to Neural Network Verification*. Foundations and Trends® in Programming Languages, vol. 7, no. 1-2, pp. 1–157, 2021.

ISBN: 978-1-68083-911-1

Foundations and Trends® in Programming Languages
Volume 7, Issue 1-2, 2021

Editorial Board

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract Interpretation
- Compilation and Interpretation Techniques
- Domain Specific Languages
- Formal Semantics, including Lambda Calculi, Process Calculi, and Process Algebra
- Language Paradigms
- Mechanical Proof Checking
- Memory Management
- Partial Evaluation
- Program Logic
- Programming Language Implementation
- Programming Language Security
- Programming Languages for Concurrency
- Programming Languages for Parallelism
- Program Synthesis
- Program Transformations and Optimizations
- Program Verification
- Runtime Techniques for Programming Languages
- Software Model Checking
- Static and Dynamic Program Analysis
- Type Theory and Type Systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2021, Volume 7, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Contents

Introduction to Neural Network Verification

Aws Albarghouthi[1]

[1] *University of Wisconsin–Madison; aws@cs.wisc.edu*

ABSTRACT

Deep learning has transformed the way we think of software and what it can do. But deep neural networks are fragile and their behaviors are often surprising. In many settings, we need to provide formal guarantees on the safety, security, correctness, or robustness of neural networks. This monograph covers foundational ideas from formal verification and their adaptation to reasoning about neural networks and deep learning.

The author's name in native alphabet is
أوس البرغوثي

Aws Albarghouthi (2021), "Introduction to Neural Network Verification", Foundations and Trends® in Programming Languages: Vol. 7, No. 1-2, pp 1–157. DOI: 10.1561/2500000051.

About This Monograph

Why This Monograph?

Over the past decade, a number of hardware and software advances have conspired to thrust deep learning and neural networks to the forefront of computing. Deep learning has created a qualitative shift in our conception of what software is and what it can do: Every day we're seeing new applications of deep learning, from healthcare to art, and it feels like we're only scratching the surface of a universe of new possibilities.

It is thus safe to say that deep learning is here to stay, in one form or another. The line between software 1.0 (that is, manually written code) and software 2.0 (learned neural networks) is getting fuzzier and fuzzier, and neural networks are participating in safety-critical, security-critical, and socially critical tasks. Think, for example, healthcare, self-driving cars, malware detection, etc. But neural networks are fragile and so we need to prove that they are well-behaved when applied in critical settings.

Over the past few decades, the formal methods community has developed a plethora of techniques for automatically proving properties of programs, and, well, neural networks are programs. So there is a great opportunity to port verification ideas to the software 2.0 setting. This monograph offers the first introduction of foundational ideas from automated verification as applied to deep neural networks and deep

learning. I hope that it will inspire verification researchers to explore correctness in deep learning and deep learning researchers to adopt verification technologies.

Who Is This Monograph For?

Given that the monograph's subject matter sits at the intersection of two pretty much disparate areas of computer science, one of my main design goals was to make it as self-contained as possible. This way the monograph can serve as an introduction to the field for first-year graduate students or senior undergraduates, even if they have not been exposed to deep learning or verification. For a comprehensive survey of verification algorithms for neural networks, along with implementations, I direct the reader to Liu *et al.* (2021).

What Does This Monograph Cover?

The monograph is divided into three parts:

Part 1 defines neural networks as data-flow graphs of operators over real-valued inputs. This formulation will serve as our basis for the rest of the monograph. Additionally, we will survey a number of correctness properties that are desirable of neural networks and place them in a formal framework.

Part 2 discusses *constraint-based* techniques for verification. As the name suggests, we construct a system of constraints and solve it to prove (or disprove) that a neural network satisfies some properties of interest. Constraint-based verification techniques are also referred to as *complete verification* in the literature.

Part 3 discusses *abstraction-based* techniques for verification. Instead of executing a neural network on a single input, we can actually execute it on an *infinite* set and show that all of those inputs satisfy desirable correctness properties. Abstraction-based techniques are also referred to as *approximate verification* in the literature.

Parts 2 and 3 are disjointed; the reader may go directly from Part 1 to Part 3 without losing context.

Part I

Neural Networks & Correctness

1

A New Beginning

> He had become so caught up in building sentences that he had almost forgotten the barbaric days when thinking was like a splash of color landing on a page.
>
> —Edward St. Aubyn, *Mother's Milk*

1.1 It Starts With Turing

This monograph is about *verifying* that a *neural network* behaves according to some set of desirable properties. These fields of study, verification and neural networks, have been two distinct areas of computing research with almost no bridges connecting them, until very recently. Intriguingly, however, both fields trace their genesis to a two-year period of Alan Turing's tragically short life.

In 1949, Turing wrote a little-known paper titled *Checking a Large Routine* (Turing, 1949). It was a truly forward-looking piece of work. In it, Turing asks how can we prove that the programs we write do what they are supposed to do? Then, he proceeds to provide a proof of

Quote found in William Finnegan's *Barbarian Days*.

correctness of a program implementing the factorial function. Specifically, Turing proved that his little piece of code always terminates and always produces the factorial of its input. The proof is elegant; it breaks down the program into single instructions, proves a lemma for every instruction, and finally stitches the lemmas together to prove correctness of the full program. Until this day, proofs of programs very much follow Turing's proof style from 1949. And, as we shall see in this monograph, proofs of neural networks will, too.

Just a year before Turing's proof of correctness of factorial, in 1948, Turing wrote a perhaps even more farsighted paper, *Intelligent Machinery*, in which he proposed *unorganized machines*.[1] These machines, Turing argued, mimic the infant human cortex, and he showed how they can *learn* using what we now call a genetic algorithm. Unorganized machines are a very simple form of what we now know as neural networks.

1.2 The Rise of Deep Learning

The topic of training neural networks continued to be studied since Turing's 1948 paper. But it has only exploded in popularity over the past decade, thanks to a combination algorithmic insights, hardware developments, and a flood of data for training.

Modern neural networks are called *deep* neural networks, and the approach to training these neural networks is *deep learning*. Deep learning has enabled incredible improvements in complex computing tasks, most notably in computer vision and natural-language processing, for example, in recognizing objects and people in an image and translating between languages. Everyday, a growing research community is exploring ways to extend and apply deep learning to more challenging problems, from music generation to proving mathematical theorems and beyond.

The advances in deep learning have changed the way we think of what software is, what it can do, and how we build it. Modern software is increasingly becoming a menagerie of traditional, manually

[1] *Intelligent Machinery* is reprinted in Turing (1969).

written code and automatically trained—sometimes constantly learning—neural networks. But deep neural networks can be fragile and produce unexpected results. As deep learning becomes used more and more in sensitive settings, like autonomous cars, it is imperative that we verify these systems and provide formal guarantees on their behavior. Luckily, we have decades of research on program verification that we can build upon, but what exactly do we verify?

1.3 What do We Expect of Neural Networks?

In Turing's proof of correctness of his factorial program, Turing was concerned that we will be programming computers to perform mathematical operations, but we could be getting them wrong. So in his proof he showed that his implementation of factorial is indeed equivalent to the mathematical definition. This notion of program correctness is known as *functional correctness*, meaning that a program is a faithful implementation of some mathematical function. Functional correctness is incredibly important in many settings—think of the disastrous effects of a buggy implementation of a cryptographic primitive or an aircraft controller.

In the land of deep learning, proving functional correctness is an unrealistic task. What does it mean to correctly recognize cats in an image or correctly translate English to Hindi? We cannot mathematically define such tasks. The whole point of using deep learning to do tasks like translation or image recognition is because we cannot mathematically capture what exactly they entail.

So what now? Is verification out of the question for deep neural networks? No! While we cannot precisely capture what a deep neural network should do, we can often characterize some of its desirable or undesirable properties. Let's look at some examples of such properties.

Robustness

The most-studied correctness property of neural networks is *robustness*, because it is generic in nature and deep learning models are infamous for their fragility (Szegedy *et al.*, 2014). Robustness means that small

perturbations to inputs should not result in changes to the output of the neural network. For example, changing a small number of pixels in my photo should not make the network think that I am a cupboard instead of a person, or adding inaudible noise to a recording of my lecture should not make the network think it is a lecture about the Ming dynasty in the 15th century. Funny examples aside, lack of robustness can be a safety and security risk. Take, for instance, an autonomous vehicle following traffic signs using cameras. It has been shown that a light touch of vandalism to a stop sign can cause the vehicle to miss it, potentially causing an accident (Eykholt *et al.*, 2018). Or consider the case of a neural network for detecting malware. We do not want a minor tweak to the malware's binary to cause the detector to suddenly deem it safe to install.

Safety

Safety is a broad class of correctness properties stipulating that a program should not get to a *bad state*. The definition of *bad* depends on the task at hand. Consider a neural-network-operated robot working in some kind of plant. We might be interested in ensuring that the robot does not exceed certain speed limits, to avoid endangering human workers, or that it does not go to a dangerous part of the plant. Another well-studied example is a neural network implementing a collision avoidance system for aircrafts (Katz *et al.*, 2017). One property of interest is that if an intruding aircraft is approaching from the left, the neural network should decide to turn the aircraft right.

Consistency

Neural networks learn about our world via examples, like images. As such, they may sometimes miss basic axioms, like physical laws, and assumptions about realistic scenarios. For instance, a neural network recognizing objects in an image and their relationships might say that object A is on top of object B, B is on top of C, and C is on top of A. But this cannot be! (At least not in the world as we know it.)

For another example, consider a neural network tracking players on the soccer field using a camera. It should not in one frame of video

say that Ronaldo is on the right side of the pitch and then in the next frame say that Ronaldo is on the left side of the pitch—Ronaldo is fast, yes, but he has slowed down in the last couple of seasons.

Looking Ahead

I hope that I have convinced you of the importance of verifying properties of neural networks. In the next two sections, we will formally define what neural networks look like (spoiler: they are ugly programs) and then build a language for formally specifying correctness properties of neural networks, paving the way for verification algorithms to prove these properties.

2

Neural Networks as Graphs

There is no rigorous definition of what deep learning is and what it is not. In fact, at the time of writing this, there is a raging debate in the artificial intelligence community about a clear definition. In this section, we will define neural networks generically as graphs of operations over real numbers. In practice, the shape of those graphs, called the *architecture*, is not arbitrary: Researchers and practitioners carefully construct new architectures to suit various tasks. For example, at the time of writing, neural networks for image recognition typically look different from those for natural language tasks.

First, we will informally introduce graphs and look at some popular architectures. Then, we will formally define graphs and their semantics.

2.1 The Neural Building Blocks

A neural network is a graph where each node performs an operation. Overall, the graph represents a function from vectors of real numbers to vectors of real numbers, that is, a function in $\mathbb{R}^n \to \mathbb{R}^m$. Consider the following very simple graph.

The red node is an *input* node; it just passes input x, a real number, to node v. Node v performs some operation on x and spits out a value

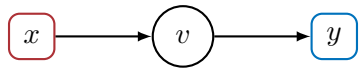


Figure 2.1: A very simple neural network

that goes to the *output* node y. For example, v might simply return $2x + 1$, which we will denote as the function $f_v : \mathbb{R} \to \mathbb{R}$:

$$f_v(x) = 2x + 1$$

In our model, the output node may also perform some operation, for example,

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

Taken together, this simple graph encodes the following function $f : \mathbb{R} \to \mathbb{R}$:

$$f(x) = f_y(f_v(x)) = \max(0, 2x + 1)$$

Transformations and Activations

The function f_v in our example above is *affine*: simply, it multiplies inputs by constant values (in this case, $2x$) and adds constant values (in this case, 1). The function f_y is an *activation* function, because it turns *on* or *off* depending on its input. When its input is negative, f_y outputs 0 (off), otherwise it outputs its input (on). Specifically, f_y, illustrated in Figure 2.2, is called a *rectified linear unit* (ReLU), and it is a very popular activation function in modern deep neural networks (Nair and Hinton, 2010). Activation functions are used to add non-linearity into a neural network.

There are other popular activation functions, for example, sigmoid,

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

whose output is bounded between 0 and 1, as shown in Figure 2.3.

Often, in the literature and practice, the affine functions and the activation function are composed into a single operation. Our graph model of neural networks can capture that, but we usually prefer to

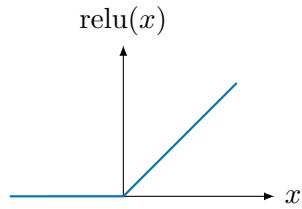


Figure 2.2: Rectified linear unit

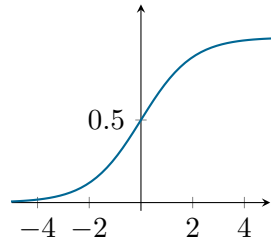


Figure 2.3: Sigmoid function

separate the two operations on to two different nodes of the graph, as it will simplify our life in later sections when we start analyzing those graphs.

Universal Approximation

What is so special about these activation functions? The short answer is they work in practice, in that they result in neural networks that are able to learn complex tasks. It is also very interesting to point out that you can construct a neural network comprised of ReLUs or sigmoids and affine functions to approximate any continuous function. This is known as the *universal approximation theorem* (Hornik *et al.*, 1989), and in fact the result is way more general than ReLUs and sigmoids—nearly any activation function you can think of works, as long as it is not polynomial! (Leshno *et al.*, 1993) For an interactive illustration of universal approximation, I highly recommend Nielsen (2018, Ch.4).

2.2 Layers and Layers and Layers

In general, a neural network can be a crazy graph, with nodes and arrows pointing all over the place. In practice, networks are usually *layered*. Take the graph in Figure 2.4. Here we have 3 inputs and 3

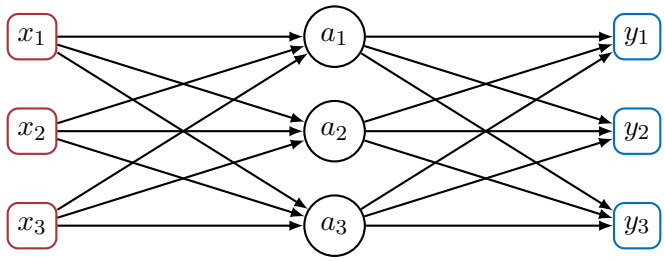


Figure 2.4: A multilayer perceptron

outputs, denoting a function in $\mathbb{R}^3 \to \mathbb{R}^3$. Notice that the nodes of the graph form layers, the input layer, the output layer, and the layer in the middle which is called the *hidden* layer. This form of graph—or architecture—has the grandiose name of *multilayer perceptron* (MLP). Usually, we have a bunch of hidden layers in an MLP; Figure 2.5 shows a MLP with two hidden layers. Layers in an MLP are called *fully connected*

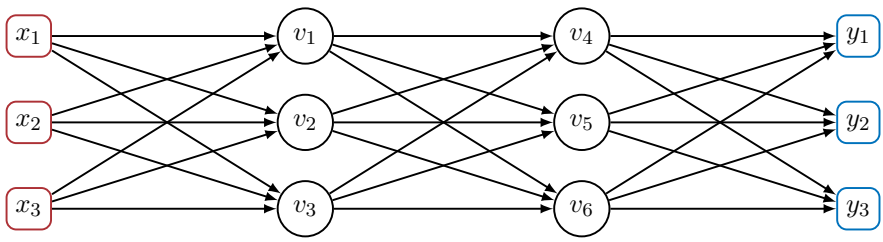


Figure 2.5: A multilayer perceptron with two hidden layers

layers, since each node receives all outputs from the preceding layer.

Neural networks are typically used as *classifiers*: they take an input, e.g., pixels of an image, and predict what the image is about (the image's class). When we are doing classification, the output layer of the MLP represents the probability of each class, for example, y_1 is the

probability of the input being a chair, y_2 is the probability of a TV, and y_3 of a couch. To ensure that the probabilities are normalized, that is, between 0 and 1 and sum up to 1, the final layer employs a *softmax* function. Softmax, generically, looks like this for an output node y_i, where n is the number of classes:

$$f_{y_i}(x_1, \ldots, x_n) = \frac{\exp(x_i)}{\sum_{k=1}^{n} \exp(x_k)}$$

Why does this work? Imagine that we have two classes, i.e., $n = 2$. First, we can verify that

$$f_{y_1}, f_{y_2} \in [0, 1]$$

This is because the numerators and denominators are both positive, and the numerator is $\leqslant$ than the denominator. Second, we can see that $f_{y_1}(x_1, x_2) + f_{y_2}(x_1, x_2) = 1$, because

$$f_{y_1}(x_1, x_2) + f_{y_2}(x_1, x_2) = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} + e^{x_2}} = 1$$

Together, these two facts mean that we have a probability distribution. For an interactive visualization of softmax, please see the excellent online book by Nielsen (2018, Chapter 3).

Given some outputs $(y_1, \ldots, y_n)$ of the neural network, we will use

$$\mathsf{class}(y_1, \ldots, y_n)$$

to denote the index of the largest element (we assume no ties), i.e., the class with the largest probability. For example, $\mathsf{class}(0.8, 0.2) = 1$, while $\mathsf{class}(0.3, 0.7) = 2$.

2.3 Convolutional Layers

Another kind of layer that you will find in a neural network is a *convolutional* layer. This kind of layer is widely used in computer-vision tasks, but also has uses in natural-language processing. The rough intuition is that if you are looking at an image, you want to scan it looking for patterns. The convolutional layer gives you that: it defines an operation, a *kernel*, that is applied to every region of pixels in an image or every

sequence of words in a sentence. For illustration, let's consider an input layer of size 4, perhaps each input defines a word in a 4-word sentence, as shown in Figure 2.6. Here we have a kernel, nodes $\{v_1, v_2, v_3\}$, that is applied to every pair of consecutive words, (x_1, x_2), (x_2, x_3), and (x_3, x_4). We say that this kernel has size 2, since it takes an input in $\mathbb{R}^2$. This kernel is 1-dimensional, since its input is a vector of real numbers. In practice, we work with 2-dimensional kernels or more; for instance, to scan blocks of pixels of a gray-scale image where every pixel is a real number, we can use kernels that are functions in $\mathbb{R}^{10\times 10} \to \mathbb{R}$, meaning that the kernel is applied to every 10×10 sub-image in the input.

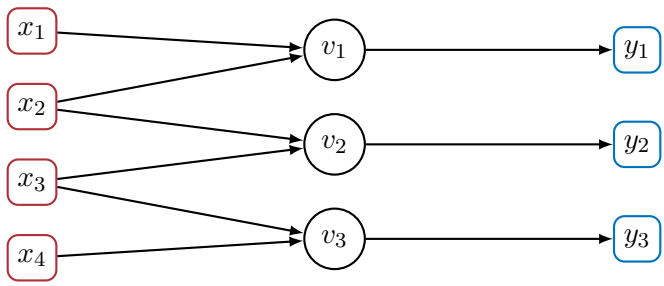


Figure 2.6: 1-dimensional convolution

Typically, a *convolutional neural network* (CNN) will apply a bunch of kernels to an input—and many layers of them—and aggregate (*pool*) the information from each kernel. We will meet these operations in later sections when we verify properties of such networks.[1]

2.4 Where are the Loops?

All of the neural networks we have seen so far seem to be a composition of a number mathematical functions, one after the other. So what about loops? Can we have loops in neural networks? In practice, neural network

[1] Note that there are many parameters that are used to construct a CNN, e.g., how many kernels are applied, how many inputs a kernel applies to, the *stride* or step size of a kernel, etc. These are not of interest to us in this monograph. We're primarily concerned with the core building blocks of the neural network, which will dictate the verification challenges.

graphs are really just directed acyclic graphs (DAG). This makes training the neural network possible using the *backpropagation* algorithm.

That said, there are popular classes of neural networks that appear to have loops, but they are very simple, in the sense that the number of iterations of the loop is just the size of the input. *Recurrent neural networks* (RNN) is the canonical class of such networks, which are usually used for sequence data, like text. You will often see the graph of an RNN rendered as follows, with the self loop on node v.

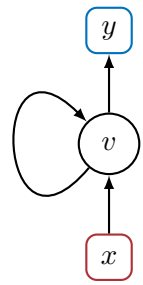


Figure 2.7: Recurrent neural network

Effectively, this graph represents an infinite family of acyclic graphs that unroll this loop a finite number of times. For example, Figure 2.8 is an unrolling of length 3. Notice that this is an acyclic graph that takes 3 inputs and produces 3 outputs. The idea is that if you receive a sentence, say, with n words, you unroll the RNN to length n and apply it to the sentence.

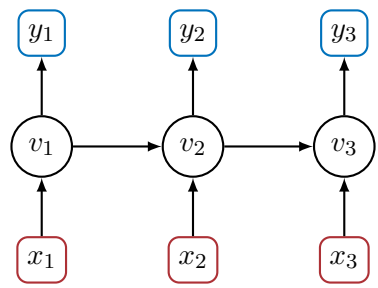


Figure 2.8: Unrolled recurrent neural network

Thinking of it through a programming lens, given an input, we can easily statically determine—i.e., without executing the network—how many loop iterations it will require. This is in contrast to, say, a program where the number of loop iterations is a complex function of its input, and therefore we do not know how many loop iterations it will take until we actually run it. That said, in what follows, we will formalize neural networks as acyclic graphs.

2.5 Structure and Semantics of Neural Networks

We're done with looking at pretty graphs. Let's now look at pretty symbols. We will now formally define neural networks as directed acyclic graphs and discuss some of their properties.

Neural Networks as DAGs

A neural network is a directed acyclic graph $G = (V, E)$, where

- V is a finite set of nodes,
- $E \subseteq V \times V$ is a set of edges,
- $V^{\mathsf{in}} \subset V$ is a non-empty set of input nodes,
- $V^{\mathsf{o}} \subset V$ is a non-empty set of output nodes, and
- each non-input node v is associated with a function $f_v : \mathbb{R}^{n_v} \to \mathbb{R}$, where n_v is the number of edges whose target is node v. The vector of real values $\mathbb{R}^{n_v}$ that v takes as input is all of the outputs of nodes v' such that $(v', v) \in E$. Notice that we assume, for simplicity but without loss of generality, that a node v only outputs a single real value.

To make sure that a graph G does not have any dangling nodes and that semantics are clearly defined, we will assume the following structural properties:

- All nodes are reachable, via directed edges, from some input node.
- Every node can reach an output node.

- There is fixed total ordering on edges E and another one on nodes V.

We will use $\boldsymbol{x} \in \mathbb{R}^n$ to denote an n-ary (row) vector, which we represent as a tuple of scalars $(x_1, \ldots, x_n)$, where x_i is the ith element of $\boldsymbol{x}$.

Semantics of DAGs

A neural network $G = (V, E)$ defines a function in $\mathbb{R}^n \to \mathbb{R}^m$ where

$$n = |V^{\mathsf{in}}| \quad \text{and} \quad m = |V^{\mathsf{o}}|$$

That is, G maps the values of the input nodes to those of the output nodes.

Specifically, for every non-input node $v \in V$, we recursively define the value in $\mathbb{R}$ that it produces as follows. Let $(v_1, v), \ldots, (v_{n_v}, v)$ be an ordered sequence of all edges whose target is node v (remember that we've assumed an order on edges). Then, we define the output of node v as

$$\mathsf{out}(v) = f_v(x_1, \ldots, x_{n_v})$$

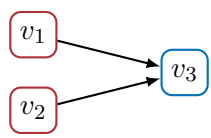
where $x_i = \mathsf{out}(v_i)$, for $i \in \{1, \ldots, n_v\}$.

The base case of the above definition (of out) is input nodes, since they have no edges incident on them. Suppose that we're given an input vector $\boldsymbol{x} \in \mathbb{R}^n$. Let $v_1, \ldots, v_n$ be an ordered sequence of all input nodes. Then,

$$\mathsf{out}(v_i) = x_i$$

A Simple Example

Let's look at an example graph G:



We have $V^{\mathsf{in}} = \{v_1, v_2\}$ and $V^{\mathsf{o}} = \{v_3\}$. Now assume that

$$f_{v_3}(x_1, x_2) = x_1 + x_2$$

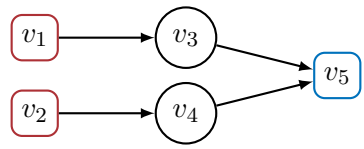
and that we're given the input vector $(11, 79)$ to the network, where node v_1 gets the value 11 and v_2 the value 79. Then, we have

$$\begin{aligned}
\mathsf{out}(v_1) &= 11 \\
\mathsf{out}(v_2) &= 79 \\
\mathsf{out}(v_3) &= f_{v_3}(\mathsf{out}(v_1), \mathsf{out}(v_2)) = 11 + 79 = 90
\end{aligned}$$

Data Flow and Control Flow

The graphs we have defined are known in the field of compilers and program analysis as *data-flow* graphs; this is in contrast to *control-flow* graphs.[2] Control-flow graphs dictate the *order* in which operations need be performed—the flow of who has *control* of the CPU. Data-flow graphs, on the other hand, only tell us what node needs what data to perform its computation, but not how to order the computation. This is best seen through a small example.

Consider the following graph



Viewing this graph as an imperative program, one way to represent it is as follows, where $\leftarrow$ is the assignment symbol.

$$\begin{aligned}
\mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1)) \\
\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2)) \\
\mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))
\end{aligned}$$

This program dictates that the output value of node v_3 is computed before that of node v_4. But this need not be, as the output of v_3 does not depend on that of v_4. Therefore, an equivalent implementation of the same graph can swap the first two operations:

[2] In deep learning frameworks like TensorFlow, they call data-flow graphs *computation graphs*.

$$\begin{aligned}\mathsf{out}(v_4) &\leftarrow f_{v_4}(\mathsf{out}(v_2))\\ \mathsf{out}(v_3) &\leftarrow f_{v_3}(\mathsf{out}(v_1))\\ \mathsf{out}(v_5) &\leftarrow f_{v_5}(\mathsf{out}(v_3), \mathsf{out}(v_4))\end{aligned}$$

Formally, we can compute the values $\mathsf{out}(\cdot)$ in any *topological* ordering of graph nodes. This ensures that all inputs of a node are computed before its own operation is performed.

Properties of Functions

So far, we have assumed that a node v can implement any function f_v it wants over real numbers. In practice, to enable efficient training of neural networks, these functions need be *differentiable* or differentiable *almost everywhere*. The sigmoid activation function, which we met earlier in Figure 2.3, is differentiable. However, the ReLU activation function, Figure 2.2, is differentiable almost everywhere, since at $x = 0$, there is a sharp turn in the function and the gradient is undefined.

Many of the functions we will be concerned with are *linear* or *piecewise linear*. Formally, a function $f : \mathbb{R}^n \to \mathbb{R}$ is linear if it can be defined as follows:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} c_i x_i + b$$

where $c_i, b \in \mathbb{R}$. A function is piecewise linear if it can be written in the form

$$f(\boldsymbol{x}) = \begin{cases} \sum_i c_i^1 x_i + b^1, & \boldsymbol{x} \in S_1 \\ \vdots & \\ \sum_i c_i^m x_i + b^m, & \boldsymbol{x} \in S_m \end{cases}$$

where S_i are mutually disjoint subsets of $\mathbb{R}^n$ and $\cup_i S_i = \mathbb{R}^n$. ReLU, for instance, is a piecewise linear function, as it is of the form:

$$\text{relu}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geqslant 0 \end{cases}$$

Another important property that we will later exploit is *monotonicity*. A function $f : \mathbb{R} \to \mathbb{R}$ is monotonically increasing if for any $x \geqslant y$,

we have $f(x) \geqslant f(y)$. Both activation functions we saw earlier in the section, ReLUs and sigmoids, are monotonically increasing. You can verify this in Figures 2.2 and 2.3: the functions never decrease with increasing values of x.

Looking Ahead

Now that we have formally defined neural networks, we're ready to pose questions about their behavior. In the next section, we will formally define a language for posing those questions. Then, in the sections that follow, we will look at algorithms for answering those questions.

Most discussions of neural networks in the literature use the language of linear algebra—see, for instance, the comprehensive book of Goodfellow *et al.* (2016). Linear algebra is helpful because we can succinctly represent the operation of many nodes in a single layer as a matrix A that applies to the output of the previous layer. Also, in practice, we use fast, parallel implementations of matrix multiplication to evaluate neural networks. Here, we choose a lower-level presentation, where each node is a function in $\mathbb{R}^n \to \mathbb{R}$. While this view is non-standard, it will help make our presentation of different verification techniques much cleaner, as we can decompose the problem into smaller ones that have to do with individual nodes.

The graphs of neural networks we presented are lower-level versions of the computation graphs of deep-learning frameworks like Tensorflow (Abadi *et al.*, 2016) and PyTorch (Paszke *et al.*, 2019)

Neural networks are an instance of a general class of programs called *differentiable programs*. As their name implies, differentiable programs are ones for which we can compute derivatives, a property that is needed for standard techniques for training neural networks. Recently, there have been interesting studies of what it means for a program to be differentiable (Abadi and Plotkin, 2020; Sherman *et al.*, 2021). In the near future, it is likely that people will start using arbitrary differentiable programs to define and train neural networks. Today, this is not the case, most neural networks have one of a few prevalent architectures and operations.

3

Correctness Properties

In this section, we will come up with a *language* for specifying properties of neural networks. The specification language is a formulaic way of making statements about the behavior of a neural network (or sometimes multiple neural networks). Our concerns in this section are solely about specifying properties, not about automatically verifying them. So we will take liberty in specifying complex properties, ridiculous ones, and useless ones. In later parts of the monograph, we will constrain the properties of interest to fit certain verification algorithms—for now, we have fun.

3.1 Properties, Informally

Remember that a neural network defines a function $f : \mathbb{R}^n \to \mathbb{R}^m$. The properties we will consider here are of the form:

> **for any input** x, the neural network **produces an output** that ...

In other words, properties dictate the input–output behavior of the network, but not the internals of the network—how it comes up with the answer.

Sometimes, our properties will be more involved, talking about multiple inputs, and perhaps multiple networks:

> **for any inputs** x, y, ... **that** ... the neural networks **produce outputs** that ...

The first part of these properties, the one talking about inputs, is called the *precondition*; the second part, talking about outputs, is called the *postcondition*. In what follows, we will continue our informal introduction to properties using examples.

Image Recognition

Let's say we have a neural network f that takes in an image and predicts a label from *dog*, *zebra*, etc. An important property that we may be interested in ensuring is *robustness* of such classifier. A classifier is robust if its prediction does not change with small variations (or perturbations) of the input. For example, changing the brightness slightly or damaging a few pixels should not change classification.

Let's fix some image $\boldsymbol{c}$ that is classified as *dog* by f. To make sure that $\boldsymbol{c}$ is not an *adversarial image* of a dog that is designed to fool the neural network, we will check—*prove* or *verify*—the following property:

> for any image $\boldsymbol{x}$ that is slightly brighter or darker than $\boldsymbol{c}$,
> $f(\boldsymbol{x})$ predicts *dog*

Notice here that the precondition specifies a set of images $\boldsymbol{x}$ that are brighter or darker than $\boldsymbol{c}$, and the postcondition specifies that the classification by f remains unchanged.

Robustness is a desirable property: you don't want classification to change with a small movement in the brightness slider. But there are many other properties you desire—robustness to changes in contrast, rotations, Instagram filters, white balance, and the list goes on. This hits at the crux of the specification problem: we often cannot specify every possible thing that we desire, so we have to choose some. (More on this later.)

For a concrete example, see Figure 3.1. The MNIST dataset (LeCun *et al.*, 2010) is a standard dataset for recognizing handwritten digits.

The figure shows a handwritten 7 along with two modified versions, one where brightness is increased and one where a spurious dot is added—perhaps a drip of ink. We would like our neural network to classify all three images as 7.

Figure 3.1: Left: Handwritten 7 from MNIST dataset. Middle: Same digit with increased brightness. Right: Same digit but with a dot added in the top left.

Natural-Language Processing

Suppose now that f takes an English sentence and decides whether it represents a positive or negative sentiment. This problem arises, for example, in automatically analyzing online reviews or tweets. We're also interested in robustness in this setting. For example, say we have fixed a sentence $\boldsymbol{c}$ with positive sentiment, then we might specify the following property:

> for any sentence $\boldsymbol{x}$ that is $\boldsymbol{c}$ with a few spelling mistakes added, $f(\boldsymbol{x})$ should predict positive sentiment

For another example, instead of spelling mistakes, imagine replacing words with synonyms:

> for any sentence $\boldsymbol{x}$ that is $\boldsymbol{c}$ with some words replaced by synonyms, then $f(\boldsymbol{x})$ should predict positive sentiment

For instance, a neural network should classify both of these movie reviews as positive reviews:

> *This movie is delightful*
> *This movie is enjoyable*

We could also combine the two properties above to get a stronger property specifying that prediction should not change in the presence of synonyms or spelling mistakes.

Source Code

Say that our neural network f is a malware classifier, taking a piece of code and deciding whether it is malware or not. A malicious entity may try to modify a malware to sneak it past the neural network by fooling it into thinking that it's a benign program. One trick the attacker may use is adding a piece of code that does not change the malware's operation but that fools the neural network. We can state this property as follows: Say we have piece of malware $\boldsymbol{c}$, then we can state the following property:

> for any program $\boldsymbol{x}$ that is equivalent to $\boldsymbol{c}$ and syntactically similar, then $f(\boldsymbol{x})$ predicts malware

Controllers

All of our examples so far have been robustness problems. Let's now look at a slightly different property. Say you have a controller deciding on the actions of a robot. The controller looks at the state of the world and decides whether to move left, right, forward, or backward. We, of course, do not want the robot to move into an obstacle, whether it is a wall, a human, or another robot. As such, we might specify the following property:

> for any state $\boldsymbol{x}$, if there is an obstacle to the right of the robot, then $f(x)$ should *not* predict right

We can state one such property per direction.

3.2 A Specification Language

Our specifications are going to look like this:

$$\{\ \textit{precondition}\ \}$$
$$\boldsymbol{r} \leftarrow f(\boldsymbol{x})$$
$$\{\ \textit{postcondition}\ \}$$

The *precondition* is a Boolean function (*predicate*) that evaluates to true or false. The precondition is defined over a set of variables which will be used as inputs to the neural networks we're reasoning about. We will use $\boldsymbol{x}_i$ to denote those variables. The middle portion of the specification is a number of calls to functions defined by neural networks; in this example, we only see one call to f, and the return value is stored in a variable r. Generally, our specification language allows a sequence of such assignments, e.g.:

$$\begin{gathered}\{\ \textit{precondition}\ \} \\ \boldsymbol{r_1} \leftarrow f(\boldsymbol{x_1}) \\ \boldsymbol{r_2} \leftarrow g(\boldsymbol{x_2}) \\ \vdots \\ \{\ \textit{postcondition}\ \}\end{gathered}$$

Finally, the postcondition is a Boolean predicate over the variables appearing in the precondition $\boldsymbol{x}_i$ and the assigned variables $\boldsymbol{r}_j$.

The way to read a specification, informally, is as follows:

> for any values of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ that make the precondition true, let $\boldsymbol{r}_1 = f(\boldsymbol{x_1}), \boldsymbol{r}_2 = g(\boldsymbol{x}_2), \ldots$. Then the postcondition is true.

If a correctness property is not true, i.e., the postcondition yields false, we will also say that the property *does not hold*.

Example 3.1. Recall our image brightness example from the previous section, and say $\boldsymbol{c}$ is an actual grayscale image, where each element of $\boldsymbol{c}$ is the intensity of a pixel, from 0 to 1 (black to white). For example, in our MNIST example in Figure 3.1, each digit is represented by 784 pixels (28×28), where each pixel is a number between 0 and 1. Then, we can state the following specification, which informally says that changing the brightness of $\boldsymbol{c}$ should not change the classification (recall the definition of class from Section 2.2):

$$\begin{gathered}\{\ |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}_2 \leftarrow f(\boldsymbol{c}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

Let's walk through this specification:

Precondition Take any image $\boldsymbol{x}$ where each pixel is at most 0.1 away from its counterpart in $\boldsymbol{c}$. Here, both $\boldsymbol{x}$ and $\boldsymbol{c}$ are assumed to be the same size, and the $\leqslant$ is defined pointwise.[1]

Assignments Let $\boldsymbol{r}_1$ be the result of computing $f(\boldsymbol{x})$ and $\boldsymbol{r}_2$ be the result of computing $f(\boldsymbol{c})$.

Postcondition Then, the predicted labels in vectors $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ are the same. Recall that in a classification setting, each element of vector $\boldsymbol{r}_i$ refers to the probability of a specific label. We use class as a shorthand to extract the index of the largest element of the vector.

Counterexamples

A *counterexample* to a property is a valuation of the variables in the precondition (the $\boldsymbol{x}_i$s) that falsifies the postcondition. In Example 3.1, a counterexample would be an image $\boldsymbol{x}$ whose classification by f is different than that of image $\boldsymbol{c}$ and whose distance from $\boldsymbol{c}$, i.e., $|\boldsymbol{x} - \boldsymbol{c}|$, is less than 0.1.

Example 3.2. Here's a concrete example (not about image recognition, just a simple function that adds 1 to the input):

$$\{\ x \leqslant 0.1\ \}$$
$$r \leftarrow x + 1$$
$$\{\ r \leqslant 1\ \}$$

This property does not hold. Consider replacing x with the value 0.1. Then, $r \leftarrow 1 + 0.1 = 1.1$. Therefore, the postcondition is falsified. So, setting x to 0.1 is a counterexample.

A Note on Hoare Logic

Our specification language looks like specifications written in *Hoare logic* (Hoare, 1969). Specifications in Hoare logic are called *Hoare triples*,

[1] The pointwise operation $|\cdot|$ is known as the ℓ_∞ norm, which we formally discuss in Section 11 and compare it to other norms.

as they are composed of three parts, just like our specifications. Hoare logic comes equipped with deduction rules that allows one to prove the validity of such specifications. For our purposes in this monograph, we will not define the rules of Hoare logic, but many of them will crop up implicitly throughout the monograph.

3.3 More Examples of Properties

We will now go through a bunch of example properties and write them in our specification language.

Equivalence of Neural Networks

Say you have a neural network f for image recognition and you want to replace it with a new neural network g. Perhaps g is smaller and faster, and since you're interested in running the network on a stream of incoming images, efficiency is very important. One thing you might want to prove is that f and g are equivalent; here's how to write this property:

$$\begin{gathered}\{\ \mathsf{true}\ \} \\ \boldsymbol{r_1} \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r_2} \leftarrow g(\boldsymbol{x}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

Notice that the precondition is true, meaning that for any image $\boldsymbol{x}$, we want the predicted labels of f and g to be the same. The true precondition indicates that the inputs to the neural networks ($\boldsymbol{x}$ in this case) are unconstrained. This specification is very strong: the only way it can be true is if f and g agree on the classification on every possible input, which is highly unlikely in practice.

One possible alternative is to state that f and g return the same prediction on some subset of images, plus or minus some brightness, as in our above example. Say S is a finite set of images, then:

$$\begin{gathered}\{\ \boldsymbol{x}_1 \in S,\ |\boldsymbol{x}_1 - \boldsymbol{x}_3| \leqslant \mathbf{0.1},\ |\boldsymbol{x}_1 - \boldsymbol{x}_2| \leqslant \mathbf{0.1}\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}_2) \\ \boldsymbol{r}_2 \leftarrow g(\boldsymbol{x}_3) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

This says the following: Pick an image $\boldsymbol{x}_1$ and generate two variants, $\boldsymbol{x}_2$ and $\boldsymbol{x}_3$, whose brightness differs a little bit from $\boldsymbol{x}_1$. Then, f and g should agree on the classification of the two images.

This is a more practical notion of equivalence than our first attempt. Our first attempt forced f and g to agree on all possible inputs, but keep in mind that most images (combinations of pixels) are meaningless noise, and therefore we don't care about their classification. This specification, instead, constrains equivalence to an infinite set of images that look like those in the set S.

Collision Avoidance

Our next example is one that has been a subject of study in the verification literature, beginning with the pioneering work of Katz *et al.* (2017). Here we have a collision avoidance system that runs on an autonomous aircraft. The system detects intruder aircrafts and decides what to do. The reason the system is run on a neural network is due to its complexity: The trained neural network is much smaller than a very large table of rules. In a sense, the neural network *compresses* the rules into an efficiently executable program.

The inputs to the neural network are the following:

- v_{own}: the aircraft's velocity
- v_{int}: the intruder aircraft's velocity
- a_{int}: the angle of the intruder with respect to the current flying direction
- a_{own}: the angle of the aircraft with respect to the intruder.
- d: the distance between the two aircrafts
- *prev*: the previous action taken.

Given the above values, the neural network decides how to steer: left-/right, strong left/right, or nothing. Specifically, the neural network assigns a score to every possible action, and the action with the lowest score is taken.

As you can imagine, many things can go wrong here, and if they do—disaster! Katz *et al.* (2017) identify a number of properties that they verify. These properties do not account for all possible scenarios, but they are important to check. Let's take a look at one that says if the intruder aircraft is far away, then the score for doing *nothing* should be below some threshold.

$$\{\ d \geqslant 55947,\ v_{own} \geqslant 1145,\ v_{int} \leqslant 60\ \}$$
$$\boldsymbol{r} \leftarrow f(d, v_{own}, v_{int}, \ldots)$$
$$\{\ \text{score of nothing in } \boldsymbol{r} \text{ is below } 1500\ \}$$

Notice that the precondition specifies that the distance between the two aircrafts is more than 55947 feet, that the aircraft's velocity is high, and the intruder's velocity is low. The postcondition specifies that doing nothing should have a low score, below some threshold. Intuitively, we should not panic if the two aircrafts are quite far apart and have moving at very different velocities.

Katz *et al.* (2017) explore a number of such properties, and also consider robustness properties in the collision-avoidance setting. But how do we come up with such specific properties? It's not straightforward. In this case, we really need a domain expert who knows about collision-avoidance systems, and even then, we might not cover all corner cases. A number of people in the verification community, the author included, argue that specification is harder than verification—that is, the hard part is asking the right questions!

Physics Modeling

Here is another example due to Qin *et al.* (2019). We want the neural network to internalize some physical laws, such as the movement of a pendulum. At any point in time, the state of the pendulum is a triple (v, h, w), its vertical position v, its horizontal position h, and its angular velocity w. Given the state of the pendulum, the neural network is to predict the state in the next time instance, assuming that time is divided into discrete steps.

A natural property we may want to check is that the neural network's understanding of how the pendulum moves adheres to the law of conservation of energy. At any point in time, the energy of the pendulum

is the sum of its potential energy and its kinetic energy. (Were you paying attention in high school physics?) As the pendulum goes up, its potential energy increases and kinetic energy decreases; as it goes down, the opposite happens. The sum of the kinetic and potential energies should only decrease over time. We can state this property as follows:

$$\begin{gathered}\{\ \mathsf{true}\ \} \\ v', h', w' \leftarrow f(v, h, w) \\ \{\ E(h', w') \leqslant E(h, w)\ \}\end{gathered}$$

The expression $E(h, w)$ is the energy of the pendulum, which is its potential energy mgh, where m is the mass of the pendulum and g is the gravitational constant, plus its kinetic energy $0.5ml^2w^2$, where l is the length of the pendulum.

Natural-Language Processing

Let's recall the natural language example from earlier in this section, where we wanted to classify a sentence into whether it expresses a positive or negative sentiment. We decided that we want the classification not to change if we replaced a word by a synonym. We can express this property in our language: Let $\boldsymbol{c}$ be a fixed sentence of length n. We assume that each element of vector $\boldsymbol{c}$ is a real number representing a word—called an *embedding* of the word. We also assume that we have a thesaurus T, which given a word gives us a set of equivalent words.

$$\begin{gathered}\{\ 1 \leqslant i \leqslant n,\ w \in T(c_i),\ \boldsymbol{x} = \boldsymbol{c}[i \mapsto w]\ \} \\ \boldsymbol{r}_1 \leftarrow f(\boldsymbol{x}) \\ \boldsymbol{r}_2 \leftarrow f(\boldsymbol{c}) \\ \{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}\end{gathered}$$

The precondition specifies that variable $\boldsymbol{x}$ is just like the sentence $\boldsymbol{c}$, except that some element i is replaced by a word w from the thesaurus. We use the notation $\boldsymbol{c}[i \mapsto w]$ to denote $\boldsymbol{c}$ with the ith element replaced with w and c_i to denote the ith element of $\boldsymbol{c}$.

The above property allows a single word to be replaced by a synonym. We can extend it to two words as follows (I know, it's very ugly, but it works):

$$\{\ 1 \leqslant i, j \leqslant n\ ,\ i \neq j\ ,\ w_i \in T(c_i)\ ,\ w_j \in T(c_j)\ ,\ \boldsymbol{x} = \boldsymbol{c}[i \mapsto w_i, j \mapsto w_j]\ \}$$
$$\boldsymbol{r}_1 \leftarrow f(\boldsymbol{x})$$
$$\boldsymbol{r}_2 \leftarrow f(\boldsymbol{c})$$
$$\{\ \mathsf{class}(\boldsymbol{r}_1) = \mathsf{class}(\boldsymbol{r}_2)\ \}$$

Monotonicity

A standard mathematical property that we may desire of neural networks is monotonicity (Sivaraman *et al.*, 2020), meaning that larger inputs should lead to larger outputs. For example, imagine you're one of those websites that predict house prices using machine learning. You'd expect the machine-learning model used is monotonic with respect to square footage—if you increase the square footage of a house, its price should not decrease, or perhaps increase. Or imagine a model that estimates the risk of complications during surgery. You'd expect that increasing the age of the patient should not decrease the risk. (I'm not a physician, but I like this example.) Here's how you could encode monotonicity in our language:

$$\{\ \boldsymbol{x} > \boldsymbol{x}'\ \}$$
$$\boldsymbol{r} \leftarrow f(\boldsymbol{x})$$
$$\boldsymbol{r}' \leftarrow f(\boldsymbol{x}')$$
$$\{\ \boldsymbol{r}' \geqslant \boldsymbol{r}'\ \}$$

In other words, pick any pair of inputs such that $\boldsymbol{x} > \boldsymbol{x}'$, we want $f(\boldsymbol{x}) \geqslant f(\boldsymbol{x}')$. Of course, we can *strengthen* the property by making the postcondition a strict inequality—that completely depends on the problem domain we're working with.

Looking Ahead

We're done with the first part of the monograph. We have defined neural networks and how to specify their properties. In what follows, we will discuss different ways of verifying properties automatically.

There has been an insane amount of work on robustness problems, particularly for image recognition. Lack of robustness was first observed by Szegedy *et al.* (2014), and since then many approaches to discover

and defend against robustness violations (known as adversarial examples) have been proposed. We will survey those later. The robustness properties for natural-language processing we have defined follow those of Ebrahimi *et al.* (2018) and Huang *et al.* (2019).

Part III

Abstraction-Based Verification

8

Neural Interval Abstraction

In the previous part of the monograph, we described how to precisely capture the semantics of a neural network by encoding it, along with a correctness property, as a formula in first-order logic. Typically, this means that we're solving an NP-complete problem, like satisfiability modulo linear real arithmetic (equivalently, mixed integer linear programming). While we have fantastic algorithms and tools that surprisingly work well for such hard problems, scaling to large neural networks remains an issue.

In this part of the monograph, we will look at approximate techniques for neural-network verification. By approximate, we mean that they overapproximate—or abstract—the semantics of a neural-network, and therefore can produce proofs of correctness, but when they fail, we do not know whether a correctness property holds or not. The approach we use is based on *abstract interpretation* (Cousot and Cousot, 1977), a well-studied framework for defining program analyses. Abstract interpretation is a very rich theory, and the math can easily make you want to quit computer science and live a monastic life in the woods, away from anything that can be considered technology. But fear not, it is a very simple idea, and we will take a pragmatic approach here in defining it and using it for neural-network verification.

8.1 Set Semantics and Verification

Let's focus on the following correctness property, defining robustness of a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ on an input grayscale image $\boldsymbol{c}$ whose classification label is 1.

$$\{\ |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\ \}$$
$$\boldsymbol{r} \leftarrow f(\boldsymbol{x})$$
$$\{\ \mathsf{class}(\boldsymbol{r}) = 1\ \}$$

Concretely, this property makes the following statement: Pick any image $\boldsymbol{x}$ that is like $\boldsymbol{c}$ but is slightly brighter or darker by at most 0.1 per pixel, assuming each pixel is some real number encoding its brightness. Now, execute the network on $\boldsymbol{x}$. The network must predict that $\boldsymbol{x}$ is of class 1.

The issue in checking such statement is that there are infinitely many possible images $\boldsymbol{x}$. Even if there are finitely many images—because, at the end of the day, we're using bits—the number is still enormous, and we cannot conceivably run all those images through the network and ensure that each and every one of them is assigned class 1. But let's just, for the sake of argument, imagine that we can lift the function f to work over *sets of images*. That is, we will define a version of f of the form:

$$f^s : \mathcal{P}(\mathbb{R}^n) \to \mathcal{P}(\mathbb{R}^m)$$

where $\mathcal{P}(S)$ is the powerset of set S. Specifically,

$$f^s(X) = \{\boldsymbol{y} \mid \boldsymbol{x} \in X,\ \boldsymbol{y} = f(\boldsymbol{x})\}$$

Armed with f^s, we can run it with the following input set:

$$X = \{\boldsymbol{x} \mid |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\}$$

which is the set of all images $\boldsymbol{x}$ defined above in the precondition of our correctness property. By computing $f^s(X)$, we get the predictions of the neural network f for all images in X. To verify our property, we simply check that

$$f^s(X) \subseteq \{\boldsymbol{y} \mid \mathsf{class}(\boldsymbol{y}) = 1\}$$

In other words, all runs of f on every image $\boldsymbol{x} \in X$ result in the network predicting class 1.

The above discussion may sound like crazy talk: we cannot simply take a neural network f and generate a version f^s that takes an infinite set of images. In this section, we will see that we actually *can*, but we will often have to lose precision: we will define an abstract version of our theoretical f^s that may return more answers. The trick is to define infinite sets of inputs using data structures that we can manipulate, called *abstract domains*.

In this section, we will meet the *interval* abstract domain. We will focus our attention on the problem of executing the neural network on an infinite set. Later, in Section 11, we come back to the verification problem.

8.2 The Interval Domain

Let's begin by considering a very simple function

$$f(x) = x + 1$$

I would like to evaluate this function on a set of inputs X; that is, I would like to somehow evaluate

$$f^s(X) = \{x + 1 \mid x \in X\}$$

We call f^s the *concrete transformer* of f.

Abstract interpretation simplifies this problem by only considering sets X that have a nice form. Specifically, the *interval abstract domain* considers an interval of real numbers written as $[l, u]$, where $l, u \in \mathbb{R}$ and $l \leqslant u$. An interval $[l, u]$ denotes the potentially infinite set

$$\{x \mid l \leqslant x \leqslant u\}$$

So we can now define a version of our function f^s that operates over an interval, as follows:

$$f^a([l, u]) = [l + 1, u + 1]$$

We call f^a an *abstract transformer* of f. In other words, f^a takes a set of real numbers and returns a set of real numbers, but the sets are

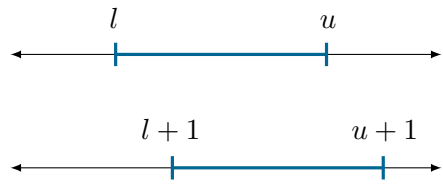


Figure 8.1: Illustration of an abstract transformer of $f(x) = x + 1$.

restricted to those that can be defined as intervals. Observe how we can mechanically evaluate this abstract transformer on an arbitrary interval $[l, u]$: add 1 to l and add 1 to u, arriving at the interval $[l + 1, u + 1]$. Geometrically, if we have an interval on the number line from l to u, and we add 1 to every point in this interval, then the whole interval shifts to the right by 1. This is illustrated in Figure 8.1. Note that the interval $[l, u]$ is an infinite set (assuming $l < u$), and so f^a adds 1 to an infinite set of real numbers!

Example 8.1. Continuing our example,

$$f^a([0, 10]) = [1, 11]$$

If we pass a singleton interval, e.g., $[1, 1]$, we get $f^a([1, 1]) = [2, 2]$—exactly the behavior of f.

Generally, we will use the notation $([l_1, u_1], \ldots, [l_n, u_n])$ to denote an n-dimensional interval, or a hyperrectangular region in $\mathbb{R}^n$, i.e., the set of all n-ary vectors

$$\{\boldsymbol{x} \in \mathbb{R}^n \mid l_i \leqslant x_i \leqslant u_i\}$$

Soundness

Whenever we design an abstract transformer f^a, we need to ensure that it is a *sound* approximation of f^s. This means that its output is a superset of that of the concrete transformer, f^s. The reason is that we will be using f^a for verification, so to declare that a property holds, we cannot afford to miss any behavior of the neural network.

Formally, we define soundness as follows: For any interval $[l, u]$, we have

$$f^s([l, u]) \subseteq f^a([l, u])$$

Equivalently, we can say that for any $x \in [l, u]$, we have

$$f(x) \in f^a([l, u])$$

In practice, we will often find that

$$f^s([l, u]) \subset f^a([l, u])$$

for many functions and intervals of interest. This is expected, as our goal is to design abstract transformers that are easy to evaluate, and so we will often *lose precision*, meaning overapproximate the results of f^s. We will see some simple examples shortly.

The Interval Domain is Non-relational

The interval domain is *non-relational*, meaning that it cannot capture the relations between different dimensions. We illustrate this fact with an example.

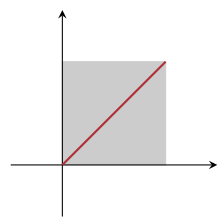
Example 8.2. Consider the set

$$X = \{(x, x) \mid 0 \leqslant x \leqslant 1\}$$

We cannot represent this set precisely in the interval domain. The best we can do is the square between $(0, 0)$ and $(1, 1)$, denoted as the 2-dimensional interval

$$([0, 1], [0, 1])$$

and illustrated as the gray region below:



The set X defines points where higher values of the x coordinate associate with higher values of the y coordinate. But our abstract domain can only represent rectangles whose faces are parallel to the axes. This means that we can't capture the relation between the two dimensions: we simply say that any value of x in $[0, 1]$ can associate with any value of y in $[0, 1]$.

8.3 Basic Abstract Transformers

We now look at examples of abstract transformers for basic arithmetic operations.

Addition

Consider the binary function: $f(x, y) = x + y$. The concrete transformer $f^s : \mathcal{P}(\mathbb{R}^2) \to \mathcal{P}(\mathbb{R})$ is defined as follows:

$$f^s(X) = \{x + y \mid (x, y) \in X\}$$

We define f^a as a function that takes two intervals, i.e., a rectangle, one representing the range of values of x_1 and the other of x_2:

$$f^a([l, u], [l', u']) = [l + l', u + u']$$

The definition looks very much like f, except that we perform addition on the lower bounds and the upper bounds of the two input intervals.

Example 8.3. Consider

$$f^a([1, 5], [100, 200]) = [101, 205]$$

The lower bound, 101, results from adding the lower bounds of x and y $(1 + 100)$; the upper bound, 205, results from adding the upper bounds of x and y $(5 + 200)$.

It is simple to prove soundness of our abstract transformer f^a. Take any

$$(x, y) \in ([l, u], [l', u'])$$

By definition, $l \leqslant x \leqslant u$ and $l' \leqslant y \leqslant u'$. So we have

$$l + l' \leqslant x + y \leqslant u + u'$$

By definition of an interval, we have

$$x + y \in [l + l', u + u']$$

Multiplication

Multiplication is a bit trickier. The reason is that the signs might flip, making the lower bound an upper bound. So we have to be a bit more careful.

Let $f(x, y) = x * y$. If we only consider positive inputs, then we can define f^a just like we did for addition:

$$f^a([l, u], [l', u']) = [l * l', u * u']$$

But consider

$$f^a([-1, 1], [-3, -2]) = [3, -2]$$

We're in trouble: $[3, -2]$ is not even an interval as per our definition—the upper bound is less than the lower bound!

To fix this issue, we need to consider every possible combination of lower and upper bounds as follows:

$$f^a([l, u], [l', u']) = [\min(B), \max(B)]$$

where

$$B = \{l * l',\ l * u',\ u * l',\ u * u'\}$$

Example 8.4. Consider the following abstract multiplication of two intervals:

$$\begin{aligned} f^a([-1, 1], [-3, -2]) &= [\min(B), \max(B)] \\ &= [-3, 3] \end{aligned}$$

where $B = \{3, 2, -3, -2\}$.

8.4 General Abstract Transformers

We will now define general abstract transformers for classes of operations that commonly appear in neural networks.

Affine Functions

For an affine function

$$f(x_1, \ldots, x_n) = \sum_i c_i x_i$$

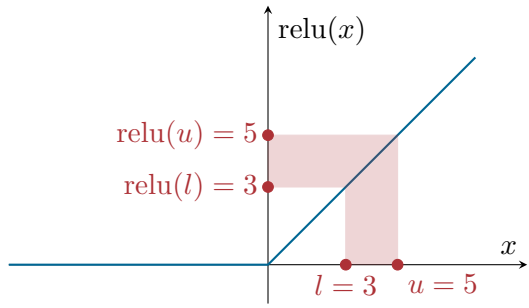


Figure 8.2: ReLU function over an interval of inputs $[l, u]$

where $c_i \in \mathbb{R}$, we can define the abstract transformer as follows:

$$f^a([l_1, u_1], \ldots, [l_n, u_n]) = \left[\sum_i l'_i,\ \sum_i u'_i\right]$$

where $l'_i = \min(c_i l_i, c_i u_i)$ and $u'_i = \max(c_i l_i, c_i u_i)$.

Notice that the definition looks pretty much like addition: sum up the lower bounds and the upper bounds. The difference is that we also have to consider the coefficients, c_i, which may result in flipping an interval's bounds when $c_i < 0$.

Example 8.5. Consider $f(x, y) = 3x + 2y$. Then,

$$\begin{aligned} f([5, 10], [20, 30]) &= [3 \cdot 5 + 2 \cdot 20,\ 3 \cdot 10 + 2 \cdot 30] \\ &= [55, 90] \end{aligned}$$

Monotonic Functions Most activation functions used in neural networks are monotonically increasing, e.g., ReLU and sigmoid. It turns out that it's easy to define an abstract transformer for any monotonically increasing function $f : \mathbb{R} \to \mathbb{R}$, as follows:

$$f^a([l, u]) = [f(l), f(u)]$$

Simply, we apply f to the lower and upper bounds.

Example 8.6. Figure 8.2 illustrates how to apply ReLU to an interval $[3, 5]$. The shaded region shows that any value y in the interval $[3, 5]$ results in a value

$$\text{relu}(3) \leqslant \text{relu}(y) \leqslant \text{relu}(5)$$

that is, a value in the interval $[\text{relu}(3), \text{relu}(5)]$.

Composing Abstract Transformers

Say we have a function composition $f \circ g$—this notation means $(f \circ g)(x) = f(g(x))$. We don't have to define an abstract transformer for the composition: we can simply compose the two abstract transformers of f and g, as $f^a \circ g^a$, and this will be a sound abstract transformer of $f \circ g$.

Composition is very important, as neural networks are a composition of many operations.

Example 8.7. Let

$$\begin{aligned} g(x) &= 3x \\ f(x) &= \text{relu}(x) \\ h(x) &= f(g(x)) \end{aligned}$$

The function h represents a very simple neural network, one that applies an affine function followed by a ReLU on an input in $\mathbb{R}$.

We define

$$h^a([l,u]) = f^a(g^a([l,u]))$$

where f^a and g^a are as defined earlier for monotonic functions and affine functions, respectively. For example, on the input interval $[2,3]$, we have

$$\begin{aligned} h^a([2,3]) &= f^a(g^a([2,3])) \\ &= f^a([6,9]) \\ &= [6,9] \end{aligned}$$

8.5 Abstractly Interpreting Neural Networks

We have seen how to construct abstract transformers for a range of functions and how to compose abstract transformers. We now direct our attention to constructing an abstract transformer for a neural network.

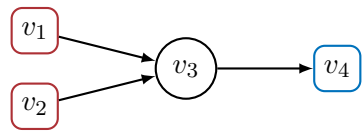
Recall that a neural network is defined as a graph $G = (V, E)$, giving rise to a function $f_G : \mathbb{R}^n \to \mathbb{R}^m$, where $n = |V^{\mathsf{in}}|$ and $m = |V^{\mathsf{o}}|$. Recall that V^{in} are input nodes and V^{o} are output nodes of G. We would

like to construct an abstract transformer f^a_G that takes n intervals and outputs m intervals.

We define $f^a_G([l_1, u_1], \ldots, [l_n, u_n])$ as follows:

- First, for every input node v_i, we define
$$\mathsf{out}^a(v_i) = [l_i, u_i]$$
Recall that we assume a fixed ordering of nodes.
- Second, for every non-input node v, we define
$$\mathsf{out}^a(v) = f^a_v(\mathsf{out}^a(v_1), \ldots, \mathsf{out}^a(v_k))$$
where f^a_v is the abstract transformer of f_v, and v has the incoming edges $(v_1, v), \ldots, (v_k, v)$,
- Finally, the output of f^a_G is the set of intervals $\mathsf{out}^a(v_1), \ldots, \mathsf{out}^a(v_m)$, where $v_1, \ldots, v_m$ are the output nodes.

Example 8.8. Consider the following simple neural network G:

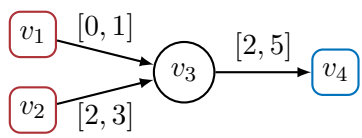


Assume that $f_{v_3}(\boldsymbol{x}) = 2x_1 + x_2$ and $f_{v_4}(x) = \mathrm{relu}(x)$.

Say we want to evaluate $f^a_G([0, 1], [2, 3])$. We can do this as follows, where $f^a_{v_3}$ and $f^a_{v_4}$ follow the definitions we discussed above for affine and monotonically increasing functions, respectively.

$$\begin{aligned}
\mathsf{out}^a(v_1) &= [0, 1] \\
\mathsf{out}^a(v_2) &= [2, 3] \\
\mathsf{out}^a(v_3) &= [2 * 0 + 2,\ 2 * 1 + 3] = [2, 5] \\
\mathsf{out}^a(v_4) &= [\mathrm{relu}(2), \mathrm{relu}(5)] = [2, 5]
\end{aligned}$$

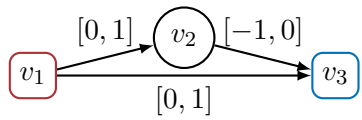
It's nice to see the outputs of every node written on the edges of the graph as follows:



Limitations of the Interval Domain

The interval domain, as described, seems infallible. We will now see how it can, and often does, overshoot: compute wildly overapproximating solutions. The primary reason for this is that the interval domain is non-relational, meaning it cannot keep track of relations between different values, e.g., the inputs and outputs of a function.

Example 8.9. Consider the following, admittedly bizarre, neural network:



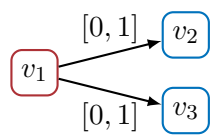
where

$$f_{v_2}(x) = -x$$
$$f_{v_3}(\boldsymbol{x}) = x_1 + x_2$$

Clearly, for any input x, $f_G(x) = 0$. Therefore, ideally, we can define our abstract transformer simply as $f^a_G([l, u]) = [0, 0]$ for any interval $[l, u]$.

Unfortunately, if we follow the recipe above, we get a much bigger interval than $[0, 0]$. For example, on the input $[0, 1]$, f^a_G returns $[-1, 1]$, as illustrated on the graph above. The reason this happens is because the output node, v_3, receives two intervals as input, not knowing that one is the negation of the other. In other words, it doesn't know the *relation* between, or provenance of, the two intervals.

Example 8.10. Here's another simple network, G, where f_{v_2} and f_{v_3} are ReLUs. Therefore, $f_G(x) = (x, x)$ for any positive input x.



Following our recipe, we have $f^a_G([0,1]) = ([0,1],[0,1])$. In other words, the abstract transformer tells us that, for inputs between 0 and 1, the neural network can output any pair (x, y) where $0 \leqslant x, y \leqslant 1$. But that's too loose an approximation: we should expect to see only outputs (x, x) where $0 \leqslant x \leqslant 1$. Again, we have lost the *relation* between the two output nodes. They both should return the same number, but the interval domain, and our abstract transformers, are not strong enough to capture that fact.

Looking Ahead

We've seen how interval arithmetic can be used to efficiently evaluate a neural network on a set of inputs, paying the price of efficiency with precision. Next, we will see more precise abstract domains.

The abstract interpretation framework was introduced by Cousot and Cousot (1977) in their seminal paper. Abstract interpretation is a general framework, based on lattice theory, for defining and reasoning about program analyses. In our exposition, we avoided the use of lattices, because we do not aim for generality—we just want to analyze neural networks. Nonetheless, the lattice-based formalization allows us to easily construct the most-precise abstract transformers for any operation.

Interval arithmetic is an old idea that predates program analysis, even computer science: it is a standard tool in the natural sciences for measuring accumulated measurement errors. For neural-network verification, interval arithmetic first appeared in a number of papers starting in 2018 (Gehr *et al.*, 2018; Gowal *et al.*, 2018; Wang *et al.*, 2018). To implement interval arithmetic for real neural networks efficiently, one needs to employ parallel matrix operations (e.g., using a GPU). Intuitively, an operation like matrix addition can be implemented with two matrix additions for interval arithmetic, one for upper bounds and one for lower bounds.

There are also powerful techniques that employ the interval domain (or any means to bound the output of various nodes of the network) with search. We did not cover this combination here but I would encourage you to check out FastLin approach (Weng *et al.*, 2018) and its successor, CROWN (Zhang *et al.*, 2018a). (Both are nicely summarized by Li *et al.*, 2019).

One interesting application of the interval domain is as a quick-and-dirty way for speeding up constraint-based verification. Tjeng *et al.*, 2019b propose using something like the interval domain to bound the interval of values taken by a ReLU for a range of inputs to the neural network. If the interval of inputs of a ReLU is above or below 0, then we can replace the ReLU with a linear function, $f(x) = x$ or $f(x) = 0$, respectively. This simplifies the constraints for constraint-based verification, as there's no longer a disjunction.

9

Neural Zonotope Abstraction

In the previous section, we defined the interval abstract domain, which allows us to succinctly capture infinite sets in $\mathbb{R}^n$ by defining lower and upper bounds per dimension. In $\mathbb{R}^2$, an interval defines a rectangle; in $\mathbb{R}^3$, an interval defines a box; in higher dimensions, it defines hyperrectangles.

The issue with the interval domain is that it does not *relate* the values of various dimensions—it just bounds each dimension. For example, in $\mathbb{R}^2$, we cannot capture the set of points where $x = y$ and $0 \leqslant x \leqslant 1$. The best we can do is the square region $([0, 1], [0, 1])$. Syntactically speaking, an abstract element in the interval domain is captured by constraints of the form:

$$\bigwedge_i l_i \leqslant x_i \leqslant u_i$$

where every inequality involves a single variable, and therefore no relationships between variables are captured. So the interval domain is called *non-relational*. In this section, we will look at a *relational* abstract domain, the *zonotope domain*, and discuss its application to neural networks.

9.1 What the Heck is a Zonotope?

Let's begin with defining a 1-dimensional *zonotope*. We assume we have a set of m real-valued *generator* variables, denoted $\epsilon_1, \ldots, \epsilon_m$. A 1-dimensional zonotope is the set of all points

$$\left\{ c_0 + \sum_{i=1}^{m} c_i \cdot \epsilon_i \;\middle|\; \epsilon_i \in [-1, 1] \right\}$$

where $c_i \in \mathbb{R}$.

If you work out a few examples of the above definition, you'll notice that a 1-dimensional zonotope is just a convoluted way of defining an interval. For example, if we have one generator variable, ϵ, then a zonotope is the set

$$\{c_0 + c_1\epsilon \mid \epsilon \in [-1, 1]\}$$

which is the interval $[c_0 - c_1, c_0 + c_1]$, assuming $c_1 \geqslant 0$. Note that c_0 is the *center* of the interval.

Zonotopes start being more expressive than intervals in $\mathbb{R}^2$ and beyond. In n-dimensions, a zonotope with m generators is the set of all points

$$\left\{ \left(\underbrace{c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \epsilon_i}_{\text{first dimension}} \;\ldots,\; \underbrace{c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \epsilon_i}_{n\text{th dimension}} \right) \;\middle|\; \epsilon_i \in [-1, 1] \right\}$$

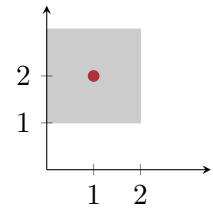
This is best illustrated through a series of examples in $\mathbb{R}^2$.[1]

Example 9.1. Consider the following two-dimensional zonotope with two generators.

$$(1 + \epsilon_1,\ 2 + \epsilon_2)$$

where we drop the set notation for clarity. Notice that in the first dimension the coefficient of ϵ_2 is 0, and in the second dimension the coefficient of ϵ_1 is 0. Since the two dimensions do not share generators, we get the following box shape whose center is $(1, 2)$.

[1] In the VR edition of the monograph, I take the reader on a guided 3D journey of zonotopes; since you cheaped out and just downloaded the free pdf, we'll have to do with $\mathbb{R}^2$.



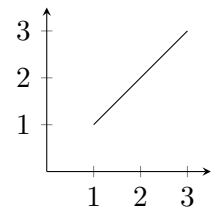
Observe that the center of the zonotope is the vector of constant coefficients of the two dimensions, $(1, 2)$, as illustrated below:

$$(\underbrace{1}_{} + \epsilon_1, \underbrace{2}_{} + \epsilon_2)$$

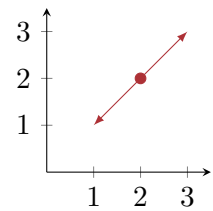
Example 9.2. Now consider the following zonotope with 1 generator:

$$(2 + \epsilon_1,\ 2 + \epsilon_1)$$

Since the two dimensions share the same expression, this means that two dimensions are equal, and so we get we get a line shape centered at $(2, 2)$:



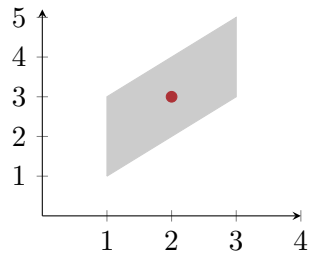
The reason ϵ_1 is called a generator is because we can think of it as a constructor of a zonotope. In this example, starting from the center point (2,2), the generator ϵ_1 *stretches* the point (2,2) to (3,3), by adding (1,1) (the two coefficients of ϵ_1) and stretches the center to (1,1) by subtracting (1,1). See the following illustration:



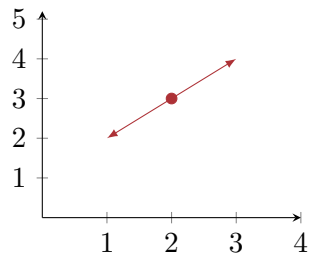
Example 9.3. Now consider the following zonotope with 2 generators,

$$(2 + \epsilon_1,\ 3 + \epsilon_1 + \epsilon_2)$$

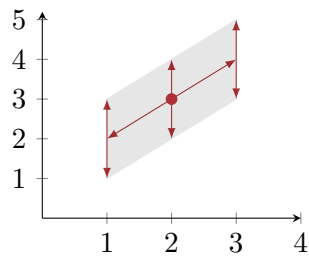
which is visualized as follows, with the center point (2,3) in red.



Let's see how this zonotope is generated in two steps, by considering one generator at a time. The coefficients of ϵ_1 are (1,1), so it stretches the center point (2,3) along the (1,1) vector, generating a line:



Next, the coefficients of ϵ_2 are (0,1), so it stretches *all* points along the $(0, 1)$ vector, resulting in the zonotope we plotted earlier:



You may have deduced by now that adding more generators adds more faces to the zonotope. For example, the right-most zonotope in Figure 9.1 uses three generators to produce the three pairs of parallel faces.

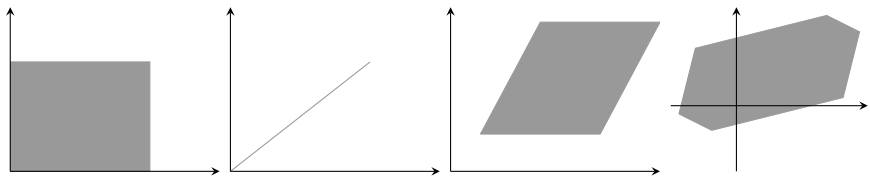


Figure 9.1: Examples of zonotopes in $\mathbb{R}^2$

A Compact Notation

Going forward, we will use a compact notation to describe an n-dimensional zonotope with m generator variables:

$$\left\{ \left(c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \epsilon_i,\ \ldots,\ c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \epsilon_i \right) \middle|\ \epsilon_i \in [-1, 1] \right\}$$

Specifically, we will define it as a tuple of vectors of coefficients:

$$(\langle c_{10}, \ldots, c_{1m} \rangle, \ldots, \langle c_{n0}, \ldots, c_{nm} \rangle)$$

For an even more compact presentation, will also use

$$(\langle c_{1i} \rangle_i, \ldots, \langle c_{ni} \rangle_i)$$

where i ranges from 0 to m, the number of generators; we drop the index i when it's clear from context.

We can compute the upper bound of the zonotope (the largest possible value) in the j dimension by solving the following optimization problem:

$$\begin{aligned} \max\ & c_{j0} + \sum_{i=1}^{m} c_{ji}\epsilon_i \\ \text{s.t.}\ & \epsilon_i \in [-1, 1] \end{aligned}$$

This can be easily solved by setting ϵ_i to 1 if $c_{ji} > 0$ and -1 otherwise.

Similarly, we can compute the lower bound of the zonotope in the jth dimension by minimizing instead of maximizing, and solving the optimization problem by setting ϵ_i to -1 if $c_{ji} > 0$ and 1 otherwise.

Example 9.4. Recall our parallelogram from Example 9.3:

$$(2 + \epsilon_1,\ 3 + \epsilon_1 + \epsilon_2)$$

In our compact notation, we write this as

$$(\langle 2, 1, 0\rangle, \langle 3, 1, 1\rangle)$$

The upper bound in the vertical dimension, $3 + \epsilon_1 + \epsilon_2$, is

$$3 + 1 + 1 = 5$$

where ϵ_1 and ϵ_2 are set to 1.

9.2 Basic Abstract Transformers

Now that we have seen zonotopes, let's define some abstract transformers over zonotopes.

Addition

For addition, $f(x, y) = x + y$, we will define the abstract transformer f^a that takes a two-dimensional zonotope defining a set of values of (x, y). We will assume a fixed number of generators m. So, for addition, its abstract transformer is of the form

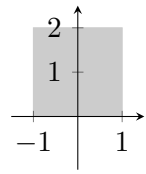
$$f^a(\langle c_{10}, \ldots, c_{1m}\rangle, \langle c_{20}, \ldots, c_{2m}\rangle)$$

Compare this to the interval domain, where $f^a([l_1, u_1], [l_2, u_2])$

It turns out that addition over zonotopes is straightforward: we just sum up the coefficients:

$$f^a(\langle c_{10}, \ldots, c_{1m}\rangle, \langle c_{20}, \ldots, c_{2m}\rangle) = \langle c_{10} + c_{20}, \ldots, c_{1m} + c_{2m}\rangle$$

Example 9.5. Consider the simple zonotope $(0 + \epsilon_1, 1 + \epsilon_2)$. This represents the following box:



The set of possible values we can get by adding the x and y dimensions in this box is the interval between -1 and 3. Following the definition of the abstract transformer for addition:

$$f^a(\langle 0, 1, 0\rangle, \langle 1, 0, 1\rangle) = \langle 1, 1, 1\rangle$$

That is the output zonotope is the set

$$\{1 + \epsilon_1 + \epsilon_2 \mid \epsilon_1, \epsilon_2 \in [-1, 1]\}$$

which is the interval $[-1, 3]$.

Affine Functions

For an affine function

$$f(x_1, \ldots, x_n) = \sum_j a_j x_j$$

where $a_j \in \mathbb{R}$, we can define the abstract transformer as follows:

$$f^a(\langle c_{1i}\rangle, \ldots \langle c_{ni}\rangle) = \left\langle \sum_j a_j c_{j0}, \ldots, \sum_j a_j c_{jm} \right\rangle$$

Intuitively, we apply f to the center point and coefficients of ϵ_1, ϵ_2, etc.

Example 9.6. Consider $f(x, y) = 3x + 2y$. Then,

$$\begin{aligned} f^a(\langle 1, 2, 3\rangle, \langle 0, 1, 1\rangle) &= \langle f(1, 0), f(2, 1), f(3, 1)\rangle \\ &= \langle 3, 8, 11\rangle \end{aligned}$$

9.3 Abstract Transformers of Activation Functions

We now discuss how to construct an abstract transformer for the ReLU activation function.

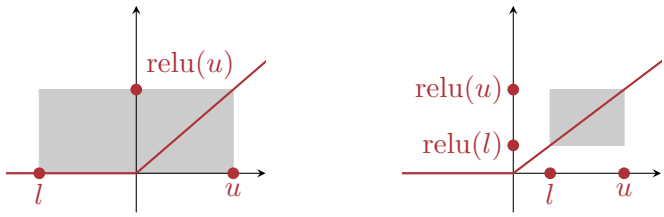
Limitations of the Interval Domain

Let's first recall the interval abstract transformer of ReLU:

$$\text{relu}^a([l,u]) = [\text{relu}(l), \text{relu}(u)]$$

The issue with the interval domain is we don't know how points in the output interval $\text{relu}^a([l,u])$ relate to the input interval $[l,u]$—i.e., which inputs are responsible for which outputs.

Geometrically, we think of the interval domain as approximating the ReLU function with a box as follows:



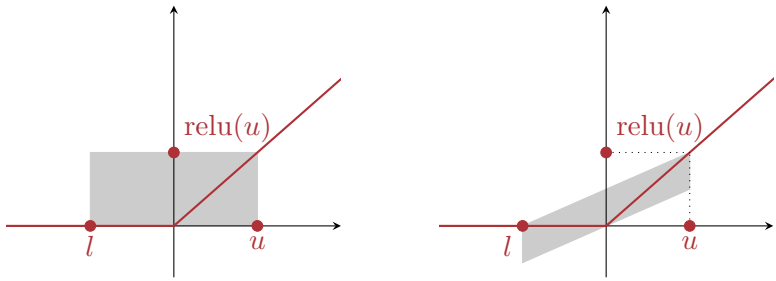
The figure on the left shows the case where the lower bound is negative and the upper bound is positive; the right figure shows the case where the lower bound is positive.

A Zonotope Transformer for ReLU

Let's slowly build the ReLU abstract transformer for zonotopes. We're given a 1-dimensional zonotope $\langle c_i \rangle_i$ as input. We will use u to denote the upper bound of the zonotope and l the lower bound.

$$\text{relu}^a(\langle c_i \rangle_i) = \begin{cases} \langle c_i \rangle_i & \text{for } l \geqslant 0 \\ \langle 0 \rangle_i & \text{for } \ u \leqslant 0 \\ ? & \text{otherwise} \end{cases}$$

If $l \geqslant 0$, then we simply return the input zonotope back; if $u \leqslant 0$, then the answer is 0; when the zonotope has both negative and positive values, there are many ways to define the output, and so I've left it as a question mark. The easy approach is to simply return the interval $[l,u]$ encoded as a zonotope. But it turns out that we can do better: since zonotopes allow us to relate inputs and outputs, we can *shear* a



box into a parallelogram that fits the shape of ReLU more tightly, as follows:

The approximation on the right has a smaller area than the approximation afforded by the interval domain on the left. The idea is that a smaller area results in a better approximation, albeit an incomparable one, as the parallelogram returns negative values, while the box doesn't. Let's try to describe this parallelogram as a zonotope.

The bottom face of the zonotope is the line

$$y = \lambda x$$

for some slope λ. It follows that the top face must be

$$y = \lambda x + u(1 - \lambda)$$

If we set $\lambda = 0$, we get two horizontal faces, i.e., the interval approximation shown above. The higher we crank up λ, the tighter the parallelogram gets. But, we can't increase λ past $u/(u - l)$; this ensures that the parallelogram covers the ReLU along the input range $[l, u]$. So, we will set

$$\lambda = \frac{u}{u - l}$$

It follows that the distance between the top and bottom faces of the parallelogram is $u(1 - \lambda)$,. Therefore, the center of the zonotope (in the vertical axis) must be the point

$$\eta = \frac{u(1 - \lambda)}{2}$$

With this information, we can complete the definition of relu^a as follows:

$$\text{relu}^a(\langle c_1, \ldots, c_m\rangle) = \begin{cases} \langle c_i\rangle_i, & \text{for } l \geqslant 0 \\ \langle 0\rangle_i, & \text{for } u \leqslant 0 \\ \langle \lambda c_1, \ldots, \lambda c_m, 0\rangle + \langle \eta, 0, 0, \ldots, \eta\rangle & \text{otherwise} \end{cases}$$

There are two non-trivial things we do here:

- First, we add a new generator, ϵ_{m+1}, in order to stretch the parallelogram in the vertical axis; its coefficient is η, which is half the hight of the parallelogram.
- Second, we add the input zonotope scaled by λ with coefficient 0 for the new generator; this ensures that we capture the relation between the input and output.

Let's look at an example for clarity:

Example 9.7. Say we invoke relu^a with the interval between $l = -1$ and $u = 1$, i.e.,

$$\text{relu}^a(\langle 0, 1\rangle)$$

Here, $\lambda = 0.5$ and $\eta = 0.25$. So the result of relu^a is the following zonotope:

$$\langle 0, 0.5, 0\rangle + \langle 0.25, 0, 0.25\rangle = \langle 0.25, 0.5, 0.25\rangle$$

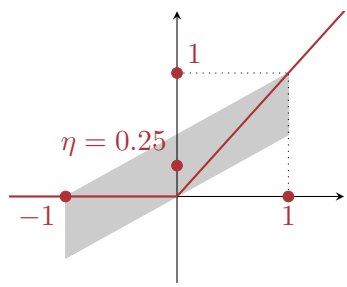
The 2-dimensional zonotope composed of the input and output zonotopes of relu^a is

$$(\langle 0, 1, 0\rangle,\ \langle 0.25, 0.5, 0.25\rangle)$$

or, explicitly,

$$(0 + \epsilon_1,\ 0.25 + 0.5\epsilon_1 + 0.25\epsilon_2)$$

This zonotope, centered at $(0, 0.25)$, is illustrated below:



Other Abstract Transformers

We saw how to design an abstract transformer for ReLU. We can follow a similar approach to design abstract transformers for sigmoid. It is indeed a good exercise to spend some time designing a zonotope transformer for sigmoid or tanh—and don't look at the literature! (Singh *et al.*, 2018)

It is interesting to note that as the abstract domain gets richer—allowing crazier and crazier shapes—the more incomparable abstract transformers you can derive (Sharma *et al.*, 2014). With the interval abstract domain, which is the simplest you can go without being trivial, the best you can do is a box to approximate a ReLU or a sigmoid. But with zonotopes, there are infinitely many shapes that you can come up with. So designing abstract transformers becomes an art, and it's hard to predict which transformers will do well in practice.

9.4 Abstractly Interpreting Neural Networks with Zonotopes

We can now use our zonotope abstract transformers to abstractly interpret an entire neural network in precisely the same way we did intervals. We review the process here for completeness.

Recall that a neural network is defined as a graph $G = (V, E)$, giving rise to a function $f_G : \mathbb{R}^n \to \mathbb{R}^m$, where $n = |V^{\mathsf{in}}|$ and $m = |V^{\mathsf{o}}|$. We would like to construct an abstract transformer f^a_G that takes an n-dimensional zonotope and outputs an m-dimensional zonotope.

We define $f^a_G(\langle c_{1j}\rangle, \ldots \langle c_{nj}\rangle)$ as follows:

- First, for every input node v_i, we define

$$\mathsf{out}^a(v_i) = \langle c_{ij}\rangle_j$$

  Recall that we assume a fixed ordering of nodes.

- Second, for every non-input node v, we define

$$\mathsf{out}^a(v) = f^a_v(\mathsf{out}^a(v_1), \ldots, \mathsf{out}^a(v_k))$$

  where f^a_v is the abstract transformer of f_v, and v has the incoming edges $(v_1, v), \ldots, (v_k, v)$,

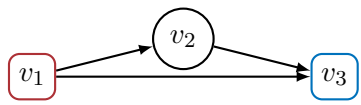
- Finally, the output of f_G^a is the m-dimensional zonotope

$$(\mathsf{out}^a(v_1), \ldots, \mathsf{out}^a(v_m))$$

where $v_1, \ldots, v_m$ are the output nodes.

One thing to note is that some abstract transformers (for activation functions) add new generators. We can assume that all of these generators are already in the input zonotope but with coefficients set to 0, and they only get non-zero coefficients in the outputs of activation function nodes.

Example 9.8. Consider the following neural network, which we saw in the last section,



where

$$f_{v_2}(x) = -x$$
$$f_{v_3}(\boldsymbol{x}) = x_1 + x_2$$

Clearly, for any input x, $f_G(x) = 0$. Consider any input zonotope $\langle c_i \rangle$. The output node, v_3, receives the two-dimensional zonotope

$$(\langle -c_i \rangle, \langle c_i \rangle)$$

The two dimensions cancel each other out, resulting in the zonotope $\langle 0 \rangle$, which is the singleton set $\{0\}$.

In contrast, with the interval domain, given input interval $[0, 1]$, you get the output interval $[-1, 1]$.

Looking Ahead

We've seen the zonotope domain, an efficient extension beyond simple interval arithmetic. Next, we will look at full-blown polyhedra.

To my knowledge, the zonotope domain was first introduced by Girard (2005) in the context of hybrid-system model checking. In the

context of neural-network verification, Gehr *et al.* (2018) were the first to use zonotopes, and introduced precise abstract transformers (Singh *et al.*, 2018), one of which we covered here. In practice, we try to limit the number of generators to keep verification fast. This can be done by occasionally *projecting out* some of the generators heuristically as we're abstractly interpreting the neural network.

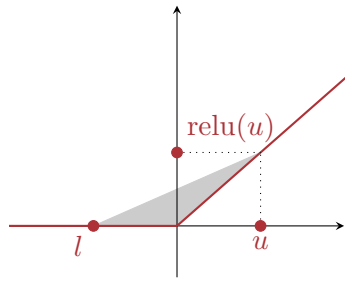
A standard optimization in program analysis is to combine program operations and construct more precise abstract transformers for the combination. This allows us to extract more relational information. In the context of neural networks, this amounts to combining activation functions in a layer of the network. Singh *et al.* (2019a) showed how to elegantly do this for zonotopes.

10

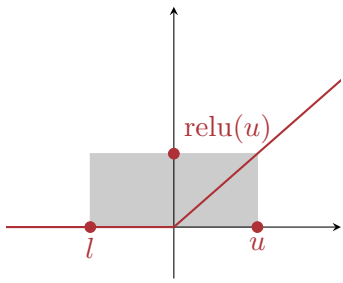
Neural Polyhedron Abstraction

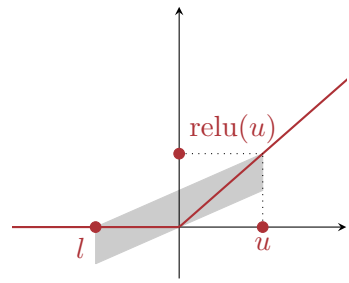
In the previous section, we saw the zonotope abstract domain, which is more expressive than the interval domain. Specifically, instead of approximating functions using a hyperrectangle, the zonotope domain allows us to approximate functions using a zonotope, e.g., a parallelogram, capturing relations between different dimensions.

In this section, we look at an even more expressive abstract domain, the *polyhedron domain*. Unlike the zonotope domain, the polyhedron domain allows us to approximate functions using arbitrary *convex polyhedra*. A polyhedron in $\mathbb{R}^n$ is a region made of straight (as opposed to curved) faces; a convex shape is one where the line between any two points in the shape is completely contained in the shape. Convex polyhedra can be specified as a set of linear inequalities. Using convex polyhedra, we approximate a ReLU as follows:



This is the smallest convex polyhedron that approximates ReLU. You can visually check that it is convex. This approximation is clearly more precise than that afforded by the interval and zonotope domains, as it is fully contained in the approximations of ReLU in those domains:





10.1 Convex Polyhedra

We will define a polyhedron in a manner analogous to a zonotope, using a set of m generator variables, $\epsilon_1, \ldots, \epsilon_m$. With zonotopes the generators are bounded in the interval $[-1, 1]$; with polyhedra, generators are bounded by a set of linear inequalities.

Let's first revisit and generalize the definition of a zonotope. A zonotope in $\mathbb{R}^n$ is a set of points defined as follows:

$$\left\{ \left(c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \epsilon_i, \ \ldots, \ c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \epsilon_i \right) \middle| \ F(\epsilon_1, \ldots, \epsilon_m) \right\}$$

where F is a Boolean function that evaluates to true iff all of its arguments are between -1 and 1.

With polyhedra, we will define F as a set (conjunction) of linear inequalities over the generator variables, e.g.,

$$0 \leqslant \epsilon_1 \leqslant 5 \ \wedge \ \epsilon_1 = \epsilon_2$$

(equalities are defined as two inequalities). We will always assume that F defines a bounded polyhedron, i.e., gives a lower and upper bound for each generator; e.g., $\epsilon_1 \leqslant 0$ is not allowed, because it does not enforce a lower bound on ϵ_1.

In the 1-dimensional case, a polyhedron is simply an interval. Let's look at higher dimensional examples:

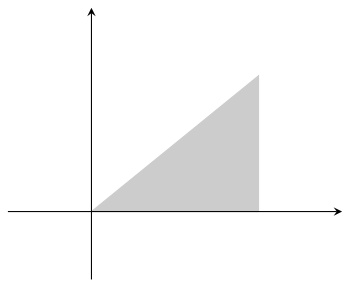
Example 10.1. Consider the following 2-dimensional polyhedron:

$$\{(\epsilon_1, \epsilon_2) \mid F(\epsilon_1, \epsilon_2)\}$$

where

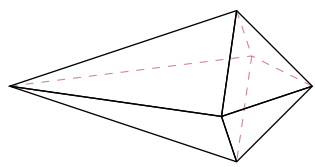
$$F \equiv 0 \leqslant \epsilon_1 \leqslant 1 \ \wedge \ \epsilon_2 \leqslant \epsilon_1 \ \wedge \ \epsilon_2 \geqslant 0$$

This polyhedron is illustrated as follows:



Clearly, this shape is not a zonotope, because its faces are not parallel.

Example 10.2. In 3 dimensions, a polyhedron may look something like this[1]



[1] Adapted from Westburg (2017).

One can add more faces by adding more linear inequalities to F.

From now on, given a polyhedron

$$\left\{ \left(c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \epsilon_i,\ \ldots,\ c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \epsilon_i \right) \middle|\ F(\epsilon_1, \ldots, \epsilon_m) \right\}$$

we will abbreviate it as the tuple:

$$(\langle c_{1i} \rangle_i, \ldots \langle c_{ni} \rangle_i, F)$$

10.2 Computing Upper and Lower Bounds

Given a polyhedron $(\langle c_{1i} \rangle_i, \ldots \langle c_{ni} \rangle_i, F)$, we will often want to compute the lower and upper bounds of one of the dimensions. Unlike with the interval and zonotope domains, this process is not straightforward. Specifically, it involves solving a linear program, which takes polynomial time in the number of variables and constraints.

To compute the lower bound of the jth dimension, we solve the following linear programming problem:

$$\begin{aligned} &\min c_{j0} + \sum_{i=1}^{m} c_{ji}\epsilon_i \\ &\text{s.t. } F \end{aligned}$$

Similarly, we compute the upper bound of the jth dimension by maximizing instead of minimizing.

Example 10.3. Take our triangle shape from Example 10.1, defined using two generators:

$$(\langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle, F)$$

where

$$F \equiv 0 \leqslant \epsilon_1 \leqslant 1 \ \wedge\ \epsilon_2 \leqslant \epsilon_1 \ \wedge\ \epsilon_2 \geqslant 0$$

To compute the upper bound of first dimension, we solve

$$\begin{aligned} &\max \epsilon_1 \\ &\text{s.t. } F \end{aligned}$$

The answer here is 1, which is obvious from the constraints.

10.3 Abstract Transformers for Polyhedra

We're now ready to go over some abstract transformers for polyhedra.

Affine Functions

For affine functions, it is really the same transformer as the one for the zonotope domain, except that we carry around the set of linear inequalities F—for the zonotope domain, F is fixed throughout.

Specifically, for an affine function

$$f(x_1, \ldots, x_n) = \sum_j a_j x_j$$

where $a_j \in \mathbb{R}$, we can define the abstract transformer as follows:

$$f^a(\langle c_{1i} \rangle, \ldots \langle c_{ni} \rangle, F) = \left(\left\langle \sum_j a_j c_{j0}, \ldots, \sum_j a_j c_{jm} \right\rangle, F \right)$$

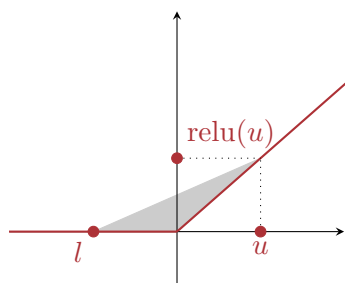
Notice that the set of linear inequalities does not change between the input and output of the function—i.e., there are no new constraints added.

Example 10.4. Consider $f(x, y) = 3x + 2y$. Then,

$$f^a(\langle 1, 2, 3 \rangle, \langle 0, 1, 1 \rangle, F) = (\langle 3, 8, 11 \rangle, F)$$

Rectified Linear Unit

Let's now look at the abstract transformer for ReLU, which we illustrated earlier in the section:



This is the tightest convex polyhedron we can use to approximate the ReLU function. We can visually verify that tightening the shape any further will either make it not an approximation or not convex—e.g., by bending the top face downwards, we get a better approximation but lose convexity.

Let's see how to formally define relu^a. The key point is that the top face is the line

$$y = \frac{u(x-l)}{u-l}$$

This is easy to check using vanilla geometry. Now, our goal is to define the shaded region, which is bounded by $y = 0$ from below, $y = x$ from the right, and $y = \frac{u(x-l)}{u-l}$ from above.

We therefore define relu^a as follows:

$$\text{relu}^a(\langle c_i \rangle_i, F) = (\langle \underbrace{0, 0, \ldots, 0}_{m}, 1 \rangle, F')$$

where

$$\begin{aligned} F' \equiv F \wedge \epsilon_{m+1} &\leqslant \frac{u(\langle c_i \rangle - l)}{(u-l)} \\ \wedge\, \epsilon_{m+1} &\geqslant 0 \\ \wedge\, \epsilon_{m+1} &\geqslant \langle c_i \rangle \end{aligned}$$

There are a number of things to note here:

- l and u are the lower and upper bounds of the input polyhedron, which can be computed using linear programming.
- $\langle c_i \rangle_i$ is used to denote the full term $c_0 + \sum_{i=1}^{m} c_i \epsilon_i$.
- Observe that we've added a new generator, ϵ_{m+1}. The new set of constraints F' relate this new generator to the input, effectively defining the shaded region.

Example 10.5. Consider the 1-dimensional polyhedron

$$(\langle 0, 1 \rangle,\ -1 \leqslant \epsilon_1 \leqslant 1)$$

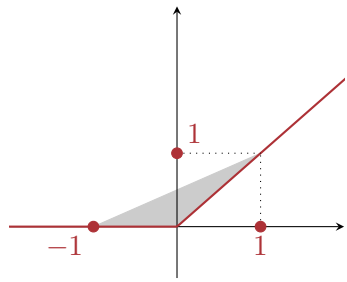
which is the interval between -1 and 1. Invoking

$$\text{relu}^a(\langle 0, 1 \rangle, -1 \leqslant \epsilon_1 \leqslant 1)$$

results in $(\langle 0, 0, 1\rangle, F')$, where

$$\begin{aligned} F' \equiv & -1 \leqslant \epsilon_1 \leqslant 1 \\ & \wedge \epsilon_2 \leqslant \frac{\epsilon_1 + 1}{2} \\ & \wedge \epsilon_2 \geqslant 0 \\ & \wedge \epsilon_2 \geqslant \epsilon_1 \end{aligned}$$

If we plot the region defined by F', using ϵ_1 as the x-axis and ϵ_2 as the y-axis, we get the shaded region



Other Activation Functions

For ReLU, the transformer we presented is the most precise. For other activation functions, like sigmoid, there are many ways to define abstract transformers for the polyhedron domain. Intuitively, one can keep adding more and more faces to the polyhedron to get a more precise approximation of the sigmoid curve.

10.4 Abstractly Interpreting Neural Networks with Polyhedra

We can now use our abstract transformers to abstractly interpret an entire neural network, in precisely the same way we did for zonotopes, except that we're now carrying around a set of constraints. We review the process here for completeness.

Recall that a neural network is defined as a graph $G = (V, E)$, giving rise to a function $f_G : \mathbb{R}^n \to \mathbb{R}^m$, where $n = |V^{\mathsf{in}}|$ and $m = |V^{\mathsf{o}}|$.

We would like to construct an abstract transformer f^a_G that takes an n-dimensional polyhedron and outputs an m-dimensional polyhedron.

We define $f^a_G(\langle c_{1j}\rangle, \ldots \langle c_{nj}\rangle, F)$ as follows:

- First, for every input node v_i, we define

$$\mathsf{out}^a(v_i) = (\langle c_{ij}\rangle_j, F)$$

  Recall that we assume a fixed ordering of nodes.

- Second, for every non-input node v, we define

$$\mathsf{out}^a(v) = f^a_v\left(p_1, \ldots, p_k, \bigwedge_{i=1}^{k} F_k\right)$$

  where f^a_v is the abstract transformer of f_v, v has the incoming edges $(v_1, v), \ldots, (v_k, v)$, and

$$\mathsf{out}^a(v_i) = (p_i, F_i)$$

  Observe what is happening here: we're combining (with $\wedge$) the constraints from the incoming edges. This ensures that we capture the relations between incoming values.

- Finally, the output of f^a_G is the m-dimensional polyhedron

$$\left(p_1, \ldots, p_m, \bigwedge_{i=1}^{m} F_i\right)$$

  where $v_1, \ldots, v_m$ are the output nodes and $\mathsf{out}^a(v_i) = (p_i, F_i)$

Some abstract transformers (for activation functions) add new generators. We can assume that all of these generators are already in the input polyhedron but with coefficients set to 0, and they only get non-zero coefficients in the outputs of activation function nodes.

Looking Ahead

We looked at the polyhedron abstract domain, which was first introduced by Cousot and Halbwachs (1978). To minimize the size of the

constraints, Singh *et al.* (2019b) use a specialized polyhedron restriction that limits the number of constraints, and apply it to neural-network verification. Another representation of polyhedra, with specialized abstract transformers for convolutional neural networks, is ImageStars (Tran *et al.*, 2020a). For a good description of efficient polyhedron domain operations and representations, for general programs, please consult Singh *et al.* (2017).

11

Verifying with Abstract Interpretation

We have seen a number of abstract domains that allow us to evaluate a neural network on an infinite set of inputs. We will now see how to use this idea for verification of specific properties. While abstract interpretation can be used, in principle, to verify any property in our language of correctness properties, much of the work in the literature is restricted to specific properties of the form:

$$\{\ \textit{precondition}\ \}$$
$$\boldsymbol{r} \leftarrow f(\boldsymbol{x})$$
$$\{\ \textit{postcondition}\ \}$$

where the precondition defines a set of possible values for $\boldsymbol{x}$, the inputs of the neural network, and the postcondition defines a set of possible correct values of $\boldsymbol{r}$, the outputs of the neural network. To verify such properties with abstract interpretation, we need to perform three tasks:

1. Soundly represent the set of values of $\boldsymbol{x}$ in the abstract domain.
2. Abstractly interpret the neural network f on all values of $\boldsymbol{x}$, resulting in an overapproximation of values of $\boldsymbol{r}$.
3. Check that all values of $\boldsymbol{r}$ satisfy the postcondition.

We've seen how to do (2), abstractly interpreting the neural network. We will now see how to do (1) and (3) for specific correctness properties from the literature.

11.1 Robustness in Image Recognition

In image recognition, we're often interested in ensuring that all images similar to some image $\boldsymbol{c}$ have the same prediction as the label of $\boldsymbol{c}$. Let's say that the label of $\boldsymbol{c}$ is y. Then we can define robustness using the following property:

$$\begin{gathered}\{\ \|\boldsymbol{x}-\boldsymbol{c}\|_p \leqslant \epsilon\ \} \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ \{\ \mathsf{class}(\boldsymbol{r}) = y\ \}\end{gathered}$$

where $\|\boldsymbol{x}\|_p$ is the ℓ_p norm of a vector and $\epsilon > 0$. Typically we use the ℓ_2 (Euclidean) or the ℓ_∞ norm as the distance metric between two images:

$$\|\boldsymbol{z}\|_2 = \sqrt{\sum_i |z_i|^2}$$

$$\|\boldsymbol{z}\|_\infty = \max_i |z_i|$$

Intuitively, the ℓ_2 norm is the length of the straight-line between two images in $\mathbb{R}^n$, while ℓ_∞ is the largest discrepancy between two images. For example, if each element of an image's vector represents one pixel, then the ℓ_∞ norm tells us the biggest difference between two corresponding pixels.

Example 11.1.

$$\begin{aligned}\|(1,2)-(2,4)\|_2 &= \|(-1,-2)\|_2 \\ &= \sqrt{3} \\ \|(1,2)-(2,4)\|_\infty &= \|(-1,-2)\|_\infty \\ &= 2\end{aligned}$$

Example 11.2. Consider an image $\boldsymbol{c}$ where every element of $\boldsymbol{c}$ represents the brightness of a grayscale pixel, from black to white, say from 0 to 1. If we want to represent the set of all images that are like $\boldsymbol{c}$ but where

each pixel differs by a brightness amount of 0.2, then we can use the ℓ_∞ norm in the precondition, i.e., the set of images $\boldsymbol{x}$ where

$$\|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leqslant 0.2$$

This is because the ℓ_∞ norm captures the *maximum* discrepancy a pixel in $\boldsymbol{c}$ can withstand. As an example, consider the handwritten 7 digit on the left and a version of it on the right where each pixel's brightness was changed by up to 0.2 randomly:

Now consider the case where we want to represent all images that are like $\boldsymbol{c}$ but where a small region has a very different brightness. For example, on the left we see the handwritten 7 and on the right we see the same handwritten digit but with a small bright dot:

To characterize a set of images that have such noise, like the dot above, we shouldn't use ℓ_∞ norm, because ℓ_∞ bounds the brightness difference for *all* pixels, but not *some* pixels, and here the brightness difference that results in the white dot is extreme—from 0 (black) to white (1). Instead, we can use the ℓ_2 norm. For the above pair of images, their ℓ_∞-norm distance is 1; their ℓ_2-norm distance is also 1, but the

precondition

$$\{\|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leqslant 1\}$$

includes the images that are all black or all white, which are clearly not the digit 7. The precondition

$$\{\|\boldsymbol{x} - \boldsymbol{c}\|_2 \leqslant 1\}$$

on the other hand, only allows a small number of pixels to significantly differ in brightness.

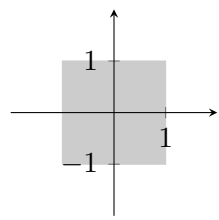
For verification, we will start by focusing on the ℓ_∞-norm case and the interval domain.

Abstracting the Precondition

Our first goal is to represent the precondition in the interval domain. The precondition is the set of the following images:

$$\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leqslant \epsilon\}$$

Example 11.3. Say $\boldsymbol{c} = (0, 0)$ and $\epsilon = 1$. Then the above set is the following region:



As the illustration above hints, it turns out that we can represent the set $\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{c}\|_\infty \leqslant \epsilon\}$ precisely in the interval domain as

$$I = ([c_1 - \epsilon, c_1 + \epsilon], \ldots, [c_n - \epsilon, c_n + \epsilon])$$

Informally, this is because the ℓ_∞ norm allows us to take any element of $\boldsymbol{c}$ and change it by ϵ independently of other dimensions.

Checking the Postcondition

Now that we have represented the set of values that $\boldsymbol{x}$ can take in the interval domain as I, we can go ahead and evaluate the abstract transformer $f^a(I)$, resulting in an output of the form

$$I' = ([l_1, u_1], \ldots, [l_m, u_m])$$

representing all possible values of $\boldsymbol{r}$, and potentially more.

The postcondition specifies that $\mathsf{class}(\boldsymbol{r}) = y$. Recall that $\mathsf{class}(\boldsymbol{r})$ is the index of the largest element of $\boldsymbol{r}$. To prove the property, we have to show that for all $\boldsymbol{r} \in I'$, $\mathsf{class}(\boldsymbol{r}) = y$. We make the observation that

$$\begin{aligned} &\text{if } l_y > u_i \text{ for all } i \neq y, \\ &\text{then for all } \boldsymbol{r} \in I', \mathsf{class}(\boldsymbol{r}) = y \end{aligned}$$

In other words, if the yth interval is larger than all others, then we know that the classification is always y. Notice that this is a one-sided check: if $l_y \leqslant u_i$ for some $i \neq y$, then we can't disprove the property. This is because the set I' overapproximates the set of possible predictions of the neural network on the precondition. So I' may include spurious predictions.

Example 11.4. Suppose that

$$f^a(I) = I' = ([0.1, 0.2], [0.3, 0.4])$$

Then, $\mathsf{class}(\boldsymbol{r}) = 2$ for all $\boldsymbol{r} \in I'$. This is because the second interval is strictly larger than the first interval.

Now suppose that

$$I' = ([0.1, 0.2], [0.15, 0.4])$$

These two intervals overlap in the region 0.15 to 0.2. This means that we cannot conclusively say that $\mathsf{class}(\boldsymbol{r}) = 2$ for all $\boldsymbol{r} \in I'$, and so verification fails.

Verifying Robustness with Zonotopes

Let's think of how to check the ℓ_∞-robustness property using the zonotope domain. Since the precondition is a hyperrectangular set,

we can precisely represent it as a zonotope Z. Then, we evaluate the abstract transformer $f^a(Z)$, resulting in a zonotope Z'.

The fun bit is checking the postcondition. We want to make sure that dimension y is greater than all others. The problem boils down to checking if a 1-dimensional zonotope is always > 0. Consider the zonotope

$$Z' = (\langle c_{1i} \rangle, \ldots \langle c_{mi} \rangle)$$

To check that dimension y is greater than dimension j, we check if the lower bound of the 1-dimensional zonotope

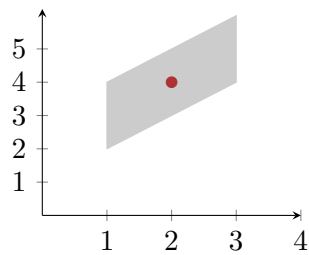
$$\langle c_{yi} \rangle - \langle c_{ji} \rangle$$

is > 0.

Example 11.5. Suppose that

$$Z' = (2 + \epsilon_1,\ 4 + \epsilon_1 + \epsilon_2)$$

which is visualized as follows, with the center point (2,4) in red:



Clearly, for any point (x, y) in this region, we have $y > x$. To check that $y > x$ mechanically, we subtract the x dimension from the y dimension:

$$(4 + \epsilon_1 + \epsilon_2) - (2 + \epsilon_1) = 2 + \epsilon_2$$

The resulting 1-dimensional zonotope $(2 + \epsilon_2)$ denotes the interval $[1, 3]$, which is greater than zero.

Verifying Robustness with Polyhedra

With the polyhedron domain, the story is analogous to zonotopes but requires invoking a linear-program solver. We represent the precondition

as a hyperrectangular polyhedron Y. Then, we evaluate the abstract transformer, $f^a(Y)$, resulting in the polyhedron

$$Y' = (\langle c_{1i} \rangle, \ldots \langle c_{mi} \rangle, F)$$

To check if dimension y is greater than dimension j, we ask a linear-program solver if the following constraints are satisfiable

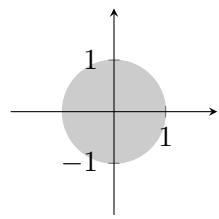
$$F \wedge \langle c_{yi} \rangle > \langle c_{ji} \rangle$$

Robustness in ℓ_2 Norm

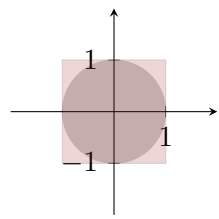
Let's now consider the precondition with the set of images within an ℓ_2 norm of $\boldsymbol{c}$:

$$\{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{c}\|_2 \leqslant \epsilon\}$$

Example 11.6. Say $\boldsymbol{c} = (0, 0)$ and $\epsilon = 1$. Then the above set is the following circular region:

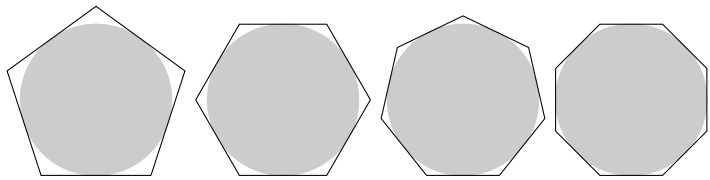


This set cannot be represented precisely in the interval domain. To ensure that we can verify the property, we need to overapproximate the circle with a box. The best we can do is using the tightest box around the circle, i.e., $([-1, 1], [-1, 1])$, shown below in red:



The zonotope and polyhedron domains also cannot represent the circular set precisely. However, there isn't a tightest zonotope or polyhedron that overapproximates the circle. For example, with polyhedra,

one can keep adding more and more faces, getting a better and better approximation, as illustrated below:



In practice, there is, of course, a precision–scalability tradeoff: more faces mean more complex constraints and therefore slower verification.

11.2 Robustness in Natural-Language Processing

We will now take a look at another robustness property from natural-language processing. The goal is to show that replacing words with synonyms does not change the prediction of the neural network. For instance, a common task is sentiment analysis, where the neural network predicts whether, say, a movie review is positive or negative. Replacing "amazing" with "outstanding" should not fool the neural network into thinking a positive review is a negative one.

We assume that the input to the neural network is a vector where element i is a numerical representation of the ith word in the sentence, and that each word w has a finite set of possible synonyms S_w, where we assume $w \in S_w$. Just as with images, we assume a fixed sentence $\boldsymbol{c}$ with label y for which we want to show robustness. We therefore define the correctness property as follows:

$$\begin{gathered}\{\ x_i \in S_{c_i} \text{ for all } i\ \} \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ \{\ \mathsf{class}(\boldsymbol{r}) = y\ \}\end{gathered}$$

Intuitively, the precondition defines all vectors $\boldsymbol{x}$ that are like $\boldsymbol{c}$ but where some words are replaced by synonyms.

The set of possible vectors $\boldsymbol{x}$ is finite, but it is exponentially large in the length of the input sentence. So it is not wise to verify the property by evaluating the neural network on every possible $\boldsymbol{x}$. We can, however, represent an overapproximation of the set of possible sentences in the

interval domain. The idea is to take interval between the largest and smallest possible numerical representations of the synonyms of every word, as follows:

$$([\min S_{c_1}, \max S_{c_1}], \ldots, [\min S_{c_n}, \max S_{c_n}])$$

This set contains all the values of $\boldsymbol{x}$, and more, but it is easy to construct, since we need only go through every set of synonyms S_{c_i} individually, avoiding an exponential explosion.

The rest of the verification process follows that of image robustness. In practice, similar words tend to have close numerical representations, thanks to the power of *word embeddings* (Mikolov *et al.*, 2013). This ensures that the interval is pretty tight. If words received arbitrary numerical representations, then our abstraction can be arbitrarily bad.

Looking Ahead

We saw examples of how to verify properties via abstract interpretation. The annoying thing is that for every abstract domain and every property of interest, we may need custom operations. Most works that use abstract interpretation so far have focused on the properties I covered in this section. Other properties from earlier in the monograph can also be verified via the numerical domains we've seen. For example, the aircraft controller from Section 3 has properties of the form:

$$\begin{gathered}\{\ d \geqslant 55947,\ v_{own} \geqslant 1145,\ v_{int} \leqslant 60\ \} \\ \boldsymbol{r} \leftarrow f(d, v_{own}, v_{int}, \ldots) \\ \{\ \text{score of nothing in } \boldsymbol{r} \text{ is below } 1500\ \}\end{gathered}$$

Note that the precondition can be captured precisely in the interval domain.

At the time of writing, abstraction-based techniques have been applied successfully to relatively large neural networks, with up to a million neurons along more than thirty layers (Müller *et al.*, 2020; Tran *et al.*, 2020b). Achieving such results requires performant implementations, particularly for more complicated domains like the zonotope and polyhedron domain. For instance, Müller *et al.* (2020) come up with data-parallel implementations of polyhedron abstract transformers that

run on a GPU. Further, there are heuristics that can be employed to minimize the number of generators in the zonotope domain—limiting the number of generators reduces precision while improving efficiency. It is also important to note that thus far most of the action in the abstraction-based verification space, and verification of neural networks at large, has been focused on ℓ_p-robustness properties for images. (We're also starting to see evidence that the ideas can apply to natural-language robustness (Zhang *et al.*, 2021).) So it's unclear whether verification will work for more complex perceptual notions of robustness—e.g., rotating an image or changing the virtual background on a video—or other more complex properties and domains, e.g., malware detection.

The robustness properties we discussed check if a fixed region of inputs surrounding a point lead to the same prediction. Alternatively, we can ask, *how big is the region around a point that leads to the same prediction?* Naïvely, we can do this by repeatedly performing verification with larger and larger ℓ_2 or ℓ_∞ bounds until verification fails. Some techniques exploit the geometric structure of a neural network—induced by ReLUs—to grow a robust region around a point (Zhang *et al.*, 2018b; Fromherz *et al.*, 2021).

As we discussed throughout this part of the monograph, abstract-interpretation techniques can make stupid mistakes due to severe over-approximations. However, abstract interpretation works well in practice for verification. Why? Two recent papers shed light on this question from a theoretical perspective (Baader *et al.*, 2020; Wang *et al.*, 2020). The papers generalize the *universal approximation* property of neural networks (Section 2.1) to verification with the interval domain or any domain that is more precise. Specifically, imagine that we have a neural network that is robust as per the ℓ_∞ norm; i.e., the following property is true for a bunch of inputs of interest:

$$\begin{gathered}\{\ \|\boldsymbol{x}-\boldsymbol{c}\|_\infty \leqslant \epsilon\ \} \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ \{\ \mathsf{class}(\boldsymbol{r}) = y\ \}\end{gathered}$$

But suppose that abstract interpretation using the interval domain fails to prove robustness for most (or all) inputs of interest. It turns out that we can always construct a neural network f', using any realistic

activation function (ReLU, sigmoid, etc.), that is very similar to f—as similar as we like—and for which we can prove robustness using abstract interpretation. The bad news, as per Wang *et al.*, 2020, is that the construction of f' is likely exponential in the size of the domain.

12

Abstract Training of Neural Networks

You have reached the final section of this glorious journey. So far on our journey, we have assumed that we're given a neural network that we want to verify. These neural networks are, almost always, constructed by learning from data. In this section, we will see how to train a neural network that is more amenable to verification via abstract interpretation for a property of interest.

12.1 Training Neural Networks

We begin by describing neural network training from a data set. Specifically, we will focus throughout this section on a classification setting.

Optimization Objective

A dataset is of the form

$$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$$

where each $\boldsymbol{x}_i \in \mathbb{R}^n$ is an *input* to the neural network, e.g., an image or a sentence, and $y_i \in \{0, 1\}$ is a binary *label*, e.g., indicating if a given image is that of a cat or if a sentence has a positive or negative

sentiment. Each item in the dataset is typically assumed to be sampled independently from a probability distribution, e.g., the distribution of all images of animals.

Given a dataset, we would like to construct a function in $\mathbb{R}^n \to \mathbb{R}$ that makes the right prediction on most of the points in the dataset. Specifically, we assume that we have a family of functions represented as a parameterized function f_θ, where θ is a vector of *weights*. We would like to find the best function by searching the space of θ values. For example, we can have the family of affine functions

$$f_\theta(\boldsymbol{x}) = \theta_1 + \theta_2 x_1 + \theta_3 x_2$$

To find the best function in the function family, we effectively need to solve an optimization problem like this one:

$$\operatorname*{argmin}_\theta \frac{1}{m} \sum_{i=1}^{m} \mathbb{1}[f_\theta(\boldsymbol{x}_i) = y_i]$$

where $\mathbb{1}[b]$ is 1 if b is *true* and 0 otherwise. Intuitively, we want the function that makes the smallest number of prediction mistakes on our dataset $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\}$.

Practically, this optimization objective is quite challenging to solve, since the objective is non-differentiable—because of the Boolean $\mathbb{1}[\cdot]$ operation, which isn't smooth. Instead, we often solve a relaxed optimization objective like *mean squared error* (MSE), which minimizes *how far* f_θ's prediction is from each y_i. MSE looks like this:

$$\operatorname*{argmin}_\theta \frac{1}{m} \sum_{i=1}^{m} (f_\theta(\boldsymbol{x}_i) - y_i)^2$$

Once we've figured out the best values of θ, we can predict the label of an input $\boldsymbol{x}$ by computing $f_\theta(\boldsymbol{x})$ and declaring label 1 iff $f_\theta(\boldsymbol{x}) \geqslant 0.5$.

We typically use a general form to describe the optimization objective. We assume that we're given a *loss* function $L(\theta, \boldsymbol{x}, y)$ which measures how *bad* is the prediction $f_\theta(\boldsymbol{x})$ is compared to the label y. Formally, we solve

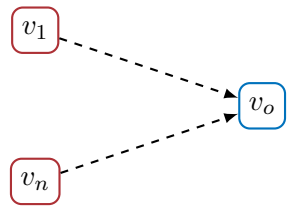
$$\operatorname*{argmin}_\theta \frac{1}{m} \sum_{i=1}^{m} L(\theta, \boldsymbol{x}_i, y_i) \tag{12.1}$$

Squared error is one example loss function, but there are others, like *cross-entropy loss.* For our purposes here, we're not interested in what loss function is used.

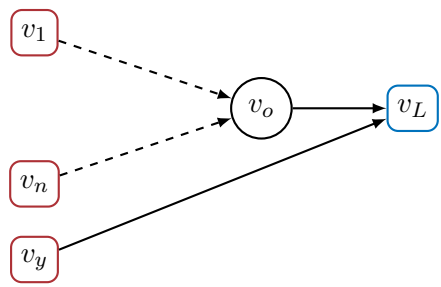
Loss Function as a Neural Network

The family of functions f_θ is represented as a neural network graph G_θ, where every node v's function f_v may be parameterized by θ. It turns out that we can we represent the loss function L also as a neural network; specifically, we represent L as an extension of the graph G_θ by adding a node at the very end that computes, for example, the squared difference between $f_\theta(\boldsymbol{x})$ and y. By viewing the loss function L as a neural network, we can abstractly interpret it, as we shall see later in the section.

Suppose that $f_\theta : \mathbb{R}^n \to \mathbb{R}$ has a graph of the form



where the dotted arrows indicate potentially intermediate nodes. We can construct the graph of a loss function $L(\theta, \boldsymbol{x}, y)$ by adding an input node v_y for the label y and creating a new output node v_L that compares the output of f_θ (the node v_o) with y.



Here, input node v_y takes in the label y and f_{v_L} encodes the loss function, e.g., mean squared error $(f(\boldsymbol{x}) - y)^2$.

Gradient Descent

How do we find values of θ that minimize the loss? Generally, this is a hard problem, so we just settle for a good enough set of values. The simplest thing to do is to randomly sample different values of θ and return the best one after some number of samples. But this is a severely inefficient approach.

Typically, neural-network training employs a form of *gradient descent*. Gradient descent is a very old algorithm, due to Cauchy in the mid 1800s. It works by starting with a random value of θ and iteratively nudging it towards better values by following the gradient of the optimization objective. The idea is that starting from some point x_0, if we want to minimize $g(x_0)$, then our best bet is to move in the direction of the negative gradient at x_0.

The gradient of a function $g(\theta)$ with respect to inputs θ, denoted ∇g, is the vector of partial derivatives[1]

$$\left(\frac{\partial g}{\partial \theta_1}, \ldots, \frac{\partial g}{\partial \theta_n}\right)$$

The gradient at a specific value θ^0, denoted $(\nabla g)(\theta^0)$, is

$$\left(\frac{\partial g}{\partial \theta_1}(\theta^0), \ldots, \frac{\partial g}{\partial \theta_n}(\theta^0)\right)$$

If you haven't played with partial derivatives in a while, I recommend Deisenroth *et al.* (2020) for a machine-learning-specific refresher.

Gradient descent can be stated as follows:

1. Start with $j = 0$ and a random value of θ, called θ^0.
2. Set θ^{j+1} to $\theta^j - \eta((\nabla g)(\theta^i))$.
3. Set j to $j + 1$ and repeat.

Here $\eta > 0$ is the *learning rate*, which constrains the size of the change of θ: too small a value and you'll make baby steps towards a good solution; too large a value and you'll bounce wildly around unable to catch a

[1] The gradient is typically a column vector, but for simplicity of presentation we treat it as a row vector here.

good region of solutions for θ, potentially even diverging. The choice of η is typically determined empirically by monitoring the progress of the algorithm for a few iterations. The algorithm is usually terminated when the loss has been sufficiently minimized or when it starts making tiny steps, asymptotically converging to a solution.

In our setting, our optimization objective is

$$\frac{1}{m}\sum_{i=1}^{m} L(\theta, \boldsymbol{x}_i, y_i)$$

Following the beautiful properties of derivatives, the gradient of this function is

$$\frac{1}{m}\sum_{i=1}^{m} \nabla L(\theta, \boldsymbol{x}_i, y_i)$$

It follows that the second step of gradient descent can be rewritten as

$$\text{Set } \theta^{j+1} \text{ to } \theta^j - \tfrac{\eta}{m}\textstyle\sum_{i=1}^{m} \nabla L(\theta^j, \boldsymbol{x}_i, y_i).$$

In other words, we compute the gradient for every point in the dataset independently and take the average.

Stochastic Gradient Descent

In practice, gradient descent is incredibly slow. So people typically use *stochastic gradient descent* (SGD). The idea is that, instead of computing the average gradient in every iteration for the entire dataset, we use a random subset of the dataset to *approximate* the gradient. SGD is also known as *mini-batch gradient descent*. Specifically, here's how SGD looks:

1. Start with $j = 0$ and a random value of θ, called θ^0.
2. Divide the dataset into a random set of k *batches*, $B_1, \ldots, B_k$.
3. For i from 1 to k,

   $$\text{Set } \theta^{j+1} \text{ to } \theta^j - \frac{\eta}{m}\sum_{(\boldsymbol{x},y)\in B_i} \nabla L(\theta^j, \boldsymbol{x}, y)$$

   $$\text{Set } j \text{ to } j+1$$

4. Go to step 2.

In practice, the number of batches k (equivalently size of the batch) is typically a function of how much data you can cram into the GPU at any one point.[2]

12.2 Adversarial Training with Abstraction

The standard optimization objective for minimizing loss (Equation (12.1)) is only interested in, well, minimizing the average loss for the dataset, i.e., getting as many predictions right. So there is no explicit goal of generating robust neural networks, for any definition of robustness. As expected, this translates to neural networks that are generally not very robust to perturbations in the input. Furthermore, even if the trained network is robust on some inputs, verification with abstract interpretation often fails to produce a proof. This is due to the overapproximate nature of abstract interpretation. One can always rewrite a neural network—or any program for that matter—into one that fools abstract interpretation, causing it to loose a lot of precision and therefore fail to verify properties of interest. Therefore, we'd like to train neural networks that are *friendly* for abstract interpretation.

We will now see how to change the optimization objective to produce robust networks and how to use abstract interpretation within SGD to solve this optimization objective.

Robust Optimization Objective

Let's consider the image-recognition-robustness property from the previous section: For every $(\boldsymbol{x}, y)$ in our dataset, we want the neural network to predict y on all images $\boldsymbol{z}$ such that $\|\boldsymbol{x} - \boldsymbol{z}\|_\infty \leqslant \epsilon$. We can characterize this set as

$$R(\boldsymbol{x}) = \{\boldsymbol{z} \mid \|\boldsymbol{x} - \boldsymbol{z}\|_\infty \leqslant \epsilon\}$$

Using this set, we will rewrite our optimization objective as follows:

$$\operatorname*{argmin}_\theta \frac{1}{m} \sum_{i=1}^{m} \max_{\boldsymbol{z} \in R(\boldsymbol{x}_i)} L(\theta, \boldsymbol{z}, y_i) \tag{12.2}$$

[2] To readers from the future: In the year 2021, graphics cards and some specialized accelerators used to be the best thing around for matrix multiplication. What have you folks settled on, quantum or DNA computers?

Intuitively, instead of minimizing the loss for $(\boldsymbol{x}_i, y_i)$, we minimize the loss for the worst-case perturbation of $\boldsymbol{x}_i$ from the set $R(\boldsymbol{x}_i)$. This is known as a *robust-optimization* problem (Ben-Tal *et al.*, 2009). Training the neural network using such objective is known as adversarial training—think of an adversary (represented using the *max*) that's always trying to mess with your dataset to maximize the loss as you are performing the training (Madry *et al.*, 2018).

Solving Robust Optimization via Abstract Interpretation

We will now see how to solve the robust-optimization problem using SGD and abstract interpretation!

Let's use the interval domain. The set $R(\boldsymbol{x})$ can be defined in the interval domain precisely, as we saw in the last section, since it defines a set of images within an ℓ_∞-norm bound. Therefore, we can overapproximate the inner maximization by abstractly interpreting L on the entire set $R(\boldsymbol{x}_i)$. (Remember that L, as far as we're concerned, is just a neural network.) Specifically, by virtue of soundness of the abstract transformer L^a, we know that

$$\left(\max_{\boldsymbol{z} \in R(\boldsymbol{x}_i)} L(\theta, \boldsymbol{z}, y_i)\right) \leqslant u$$

where

$$L^a(\theta, R(\boldsymbol{x}_i), y_i) = [l, u]$$

In other words, we can overapproximate the inner maximization by abstractly interpreting the loss function on the set $R(\boldsymbol{x}_i)$ and taking the upper bound.

We can now rewrite our robust-optimization objective as follows:

$$\operatorname*{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^{m} \text{upper bound of } L^a(\theta, R(\boldsymbol{x}_i), y_i) \tag{12.3}$$

Instead of thinking of L^a as an abstract transformer in the interval domain, we can think of it as a function that takes a vector of inputs, denoting lower and upper bounds of $R(\boldsymbol{x})$, and returns the pair of lower and upper bounds. We call this idea *flattening* the abstract transformer; we illustrate flattening with a simple example:

Example 12.1. Consider the ReLU function $\text{relu}(x) = \max(0, x)$. The interval abstract transformer is

$$\text{relu}^a([l, u]) = [\max(0, l), \max(0, u)]$$

We can flatten it into a function $\text{relu}^{af} : \mathbb{R}^2 \to \mathbb{R}^2$ as follows:

$$\text{relu}^{af}(l, u) = (\max(0, l), \max(0, u))$$

Notice that relu^{af} returns a pair in $\mathbb{R}^2$ as opposed to an interval.

With this insight, we can flatten the abstract loss function L^a into L^{af}. Then, we just invoke SGD on the following optimization problem,

$$\underset{\theta}{\text{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_u^{af}(\theta, l_{i1}, u_{i1}, \ldots, l_{in}, u_{in}, y_i) \tag{12.4}$$

where L_u^{af} is only the upper bound of the output of L^{af}, i.e., we throw away the lower bound (remember Equation (12.3)), and $R(\boldsymbol{x}_i) = ([l_{i1}, u_{i1}], \ldots, [l_{in}, u_{in}])$.

SGD can optimize such objective because all of the abstract transformers of the interval domain that are of interest for neural networks are differentiable (almost everywhere). The same idea can be adapted to the zonotope domain, but it's a tad bit uglier.

Example 12.2. Given a function $f : \mathbb{R} \to \mathbb{R}$, its zonotope abstract transformer f^a is one that takes as input a 1-dimensional input zonotope with m generator variables, $\langle c_0, \ldots, c_m \rangle$, and outputs a 1-dimensional zonotope also with m generators, $\langle c'_0, \ldots, c'_m \rangle$. We can flatten f^a by treating it as a function in

$$f^{af} : \mathbb{R}^{m+1} \to \mathbb{R}^{m+1}$$

where the $m + 1$ arguments and outputs are the coefficients of the m generator variables and the center point.

Flattening does not work for the polyhedron domain, because it invokes a black-box linear-programming solver for activation functions, which is not differentiable.

Looking Ahead

We saw how to use abstract interpretation to train (empirically) more robust neural networks. It has been shown that neural networks trained with abstract interpretation tend to be (1) more robust to perturbation attacks and (2) are verifiably robust using abstract interpretation. The second point is subtle: You could have a neural network that satisfies a correctness property of interest, but that does not mean that an abstract domain will succeed at verifying that the neural network satisfies the property. By incorporating abstract interpretation into training, we guide SGD towards neural networks that are amenable to verification.

The first use of abstract interpretation within the training loop came in 2018 (Mirman *et al.*, 2018; Gowal *et al.*, 2018). Since then, many approaches have used abstract interpretation to train robust image-recognition as well as natural-language-processing models (Zhang *et al.*, 2020; Zhang *et al.*, 2021; Jia *et al.*, 2019; Xu *et al.*, 2020; Huang *et al.*, 2019). Robust optimization is a rich field (Ben-Tal *et al.*, 2009); to my knowledge, Madry *et al.* (2018) were the first to pose training of ℓ_p-robust neural networks as a robust-optimization problem.

There are numerous techniques for producing neural networks that are amenable to verification. For instance, Sivaraman *et al.*, 2020 use constraint-based verification to verify that a neural network is monotone. Since constraint-based techniques can be complete, they can produce counterexamples, which are then used to retrain the neural network, steering it towards monotonicity. Another interesting direction in the constraint-based world is to train neural networks towards ReLUs whose inputs are always positive or always negative (Xiao *et al.*, 2019). This ensures that the generated constraints have as few disjunctions as possible, because the encoding of the ReLU will be linear (i.e., no disjunction).

ℓ_p-robustness properties are closely related to the notion of *Lipschitz continuity*. For instance, a network $f : \mathbb{R}^n \to \mathbb{R}^m$ is K-Lipschitz under the ℓ_2 norm if

$$\|f(\boldsymbol{x}) - f(\boldsymbol{y})\|_2 \leqslant K \|\boldsymbol{x} - \boldsymbol{y}\|_2$$

The smallest K satisfying the above is called the Lipschitz constant of f. If we can bound K, then we can prove ℓ_2-robustness of f. A number of

works aim to construct networks with constrained Lipschitz constants, e.g., by adding special layers to the network architecture or modifying the training procedure (Trockman and Kolter, 2021; Leino *et al.*, 2021; Li *et al.*, 2019)

13

The Challenges Ahead

My goal with this monograph is to give an introduction to two salient neural-network verification approaches. But, as you may expect, there are many interesting ideas, issues, and prospects that we did not discuss.

Correctness Properties

In Part I of the monograph, we saw a general language of correctness properties, and saw a number of interesting examples across many domains. One of the hardest problems in the field verification—and the one that is least discussed—is how to actually come up with such properties (also known as *specifications*). For instance, we saw forms of the robustness property many times throughout the monograph. Robustness, at a high level, is very desirable. You expect an *intelligent* system to be robust in the face of silly transformations to its input. But how exactly do we define robustness? Much of the literature focuses on ℓ_p norms, which we saw in Section 11. But one can easily perform transformations that lie outside ℓ_p norms, e.g., rotations to an image, or work in domains where ℓ_p norms don't make much sense, e.g., natural language, source code, or other structured data.

Therefore, coming up with the right properties to verify and enforce is a challenging, domain-dependent problem requiring a lot of careful thought.

Verification Scalability

Every year, state-of-the-art neural networks blow up in size, gaining more and more parameters. We're talking about billions of parameters. There is no clear end in sight. This poses incredible challenges for verification. Constraint-based approaches are already not very scalable, and abstraction-based approaches tend to lose precision with more and more operations. So we need creative ways to make sure that verification technology keeps up with the parameter arms race.

Verification Across the Stack

Verification research has focused on checking properties of neural networks in isolation. But neural networks are, almost always, a part of a bigger more complex system. For instance, a neural network in a self-driving car receives a video stream from multiple cameras and makes decisions on how to steer, speed up, or brake. These video streams run through layers of encoding, and the decisions made by the neural network go through actuators with their own control software and sensors. So, if one wants to claim any serious correctness property of a neural-network-driven car, one needs to look at all of the software components together as a system. This makes the verification problem challenging for two reasons: (1) The size of the entire stack is clearly bigger than just the neural network, so scalability can be an issue. (2) Different components may require different verification techniques, e.g., abstract domains.

Another issue with verification approaches is the lack of emphasis on the training algorithms that produce neural networks. For example, training algorithms may themselves not be robust: a small corruption to the data may create vastly different neural networks. For instance, a number of papers have shown that *poisoning* the dataset through minimal manipulation can cause a neural network to pick up on spurious correlations that can be exploited by an attacker. Imagine a neural

network that detects whether a piece of code is malware. This network can be trained using a dataset of malware and non-malware. By adding silly lines of code to some of the non-malware code in the dataset, like `print("LOL")`, we can force the neural network to learn a correlation between the existence of this print statement and the fact that a piece of code is not malware (Ramakrishnan and Albarghouthi, 2020). This can then be exploited by an attacker. This idea is known as installing a *backdoor* in the neural network.

So it's important to prove that our training algorithm is not susceptible to small perturbations in the input data. This is a challenging problem, but researchers have started to look at it for simple models (Drews *et al.*, 2020; Rosenfeld *et al.*, 2020).

Verification in Dynamic Environments

Often, neural networks are deployed in a dynamic setting, where the neural network interacts with the environment, e.g., a self-driving car. Proving correctness in this setting is rather challenging. First, one has to understand the interaction between the neural network and the environment—the *dynamics*. This is typically hard to pin down precisely, as real-world physics may not be as clean as textbook formulas. Further, the world can be uncertain, e.g., we have to somehow reason about other crazy drivers on the road. Second, in such settings, one needs to verify that a neural-network-based controller maintains the system in a safe state (e.g., on the road, no crash, etc.). This requires an inductive proof, as one has to reason about arbitrarily many time steps of control. Third, sometimes the neural network is learning on-the-go, using *reinforcement learning*, where the neural network tries things to see how the environment responds, like a toddler stumbling around. So we have to ensure that the neural network does not do stupid things as it is learning.

Recently, there have been a number of approaches attempting to verify properties of neural networks in dynamic and reinforcement-learning settings (Bastani *et al.*, 2018; Zhu *et al.*, 2019; Ivanov *et al.*, 2019; Anderson *et al.*, 2020).

Probabilistic Approaches

The verification problems we covered are hard, yes-or-no problems. A recent approach, called *randomized smoothing* (Cohen *et al.*, 2019; Lécuyer *et al.*, 2019), has shown that one can get probabilistic guarantees, at least for some robustness properties (Ye *et al.*, 2020; Bojchevski *et al.*, 2020). Instead of saying a neural network is robust or not around some input, we say it is robust with a high probability.

Acknowledgements

Thanks to the best focus group ever: the CS 839 students and TA, Swati Anand, at the University of Wisconsin–Madison. A number of insightful people sent me comments that radically improved the presentation: Frantisek Plasil, Georg Weissenbacher, Sayan Mitra, Benedikt Böing, Vivek Garg, Guy Van den Broeck, Matt Fredrikson, in addition to some anonymous folks.

References

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by K. Keeton and T. Roscoe. USENIX Association. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Abadi, M. and G. D. Plotkin. (2020). "A simple differentiable programming language". *Proc. ACM Program. Lang.* 4(POPL): 38:1–38:28. DOI: 10.1145/3371106.

Anderson, G., A. Verma, I. Dillig, and S. Chaudhuri. (2020). "Neurosymbolic Reinforcement Learning with Formally Verified Exploration". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html.

Baader, M., M. Mirman, and M. T. Vechev. (2020). "Universal Approximation with Certified Networks". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. URL: https://openreview.net/forum?id=B1gX8kBtPr.

Barrett, C. W., C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. (2011). "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. *Lecture Notes in Computer Science*. Springer. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

Bastani, O., Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi. (2016). "Measuring Neural Net Robustness with Constraints". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett. 2613–2621. URL: https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html.

Bastani, O., Y. Pu, and A. Solar-Lezama. (2018). "Verifiable Reinforcement Learning via Policy Extraction". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 2499–2509. URL: https://proceedings.neurips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html.

Ben-Tal, A., L. E. Ghaoui, and A. Nemirovski. (2009). *Robust Optimization*. Vol. 28. *Princeton Series in Applied Mathematics*. Princeton University Press. DOI: 10.1515/9781400831050.

Biere, A., M. Heule, H. van Maaren, and T. Walsh, eds. (2009). *Handbook of Satisfiability*. Vol. 185. *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bland, R. G. (1977). "New Finite Pivoting Rules for the Simplex Method". *Math. Oper. Res.* 2(2): 103–107. DOI: 10.1287/moor.2.2.103.

Bojchevski, A., J. Klicpera, and S. Günnemann. (2020). "Efficient Robustness Certificates for Discrete Data: Sparsity-Aware Randomized Smoothing for Graphs, Images and More". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. *Proceedings of Machine Learning Research*. PMLR. 1003–1013. URL: http://proceedings.mlr.press/v119/bojchevski20a.html.

Caviness, B. F. and J. R. Johnson. (2012). *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media.

Church, A. (1936). "A Note on the Entscheidungsproblem". *J. Symb. Log.* 1(1): 40–41. DOI: 10.2307/2269326.

Cohen, J. M., E. Rosenfeld, and J. Z. Kolter. (2019). "Certified Adversarial Robustness via Randomized Smoothing". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. *Proceedings of Machine Learning Research*. PMLR. 1310–1320. URL: http://proceedings.mlr.press/v97/cohen19c.html.

Cousot, P. and R. Cousot. (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by R. M. Graham, M. A. Harrison, and R. Sethi. ACM. 238–252. DOI: 10.1145/512950.512973.

Cousot, P. and N. Halbwachs. (1978). "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by A. V. Aho, S. N. Zilles, and T. G. Szymanski. ACM Press. 84–96. DOI: 10.1145/512760.512770.

Dantzig, G. B. (1990). "Origins of the simplex method". In: *A history of scientific computing*. 141–151.

Deisenroth, M. P., A. A. Faisal, and C. S. Ong. (2020). *Mathematics for machine learning*. Cambridge University Press.

Drews, S., A. Albarghouthi, and L. D'Antoni. (2020). "Proving data-poisoning robustness in decision trees". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by A. F. Donaldson and E. Torlak. ACM. 1083–1097. DOI: 10.1145/3385412.3385975.

Dutertre, B. and L. De Moura. (2006). "Integrating simplex with DPLL (T)". *Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01*.

Ebrahimi, J., A. Rao, D. Lowd, and D. Dou. (2018). "HotFlip: White-Box Adversarial Examples for Text Classification". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*. Ed. by I. Gurevych and Y. Miyao. Association for Computational Linguistics. 31–36. DOI: 10.18653/v1/P18-2006.

Een, N. (2005). "MiniSat: A SAT solver with conflict-clause minimization". In: *Proc. SAT-05: 8th Int. Conf. on Theory and Applications of Satisfiability Testing*. 502–518.

Ehlers, R. (2017). "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*. Ed. by D. D'Souza and K. N. Kumar. Vol. 10482. *Lecture Notes in Computer Science*. Springer. 269–286. DOI: 10.1007/978-3-319-68167-2_19.

Eykholt, K., I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. (2018). "Robust Physical-World Attacks on Deep Learning Visual Classification". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society. 1625–1634. DOI: 10.1109/CVPR.2018.00175.

Fromherz, A., K. Leino, M. Fredrikson, B. Parno, and C. S. Pasareanu. (2021). "Fast Geometric Projections for Local Robustness Certification". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. URL: https://openreview.net/forum?id=zWy1uxjDdZJ.

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

Girard, A. (2005). "Reachability of Uncertain Linear Systems Using Zonotopes". In: *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings*. Ed. by M. Morari and L. Thiele. Vol. 3414. *Lecture Notes in Computer Science*. Springer. 291–305. DOI: 10.1007/978-3-540-31954-2_19.

Goodfellow, I. J., Y. Bengio, and A. C. Courville. (2016). *Deep Learning. Adaptive computation and machine learning*. MIT Press. URL: http://www.deeplearningbook.org/.

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

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming". *Commun. ACM*. 12(10): 576–580. DOI: 10.1145/363235.363259.

Hornik, K., M. B. Stinchcombe, and H. White. (1989). "Multilayer feedforward networks are universal approximators". *Neural Networks*. 2(5): 359–366. DOI: 10.1016/0893-6080(89)90020-8.

Huang, P., R. Stanforth, J. Welbl, C. Dyer, D. Yogatama, S. Gowal, K. Dvijotham, and P. Kohli. (2019). "Achieving Verified Robustness to Symbol Substitutions via Interval Bound Propagation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Ed. by K. Inui, J. Jiang, V. Ng, and X. Wan. Association for Computational Linguistics. 4081–4091. DOI: 10.18653/v1/D19-1419.

Ivanov, R., J. Weimer, R. Alur, G. J. Pappas, and I. Lee. (2019). "Verisig: verifying safety properties of hybrid systems with neural network controllers". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*. Ed. by N. Ozay and P. Prabhakar. ACM. 169–178. DOI: 10.1145/3302504.3311806.

Jia, K. and M. Rinard. (2020a). "Efficient Exact Verification of Binarized Neural Networks". Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/1385974ed5904a438616ff7bdb3f7439-Abstract.html.

Jia, K. and M. Rinard. (2020b). "Exploiting Verified Neural Networks via Floating Point Numerical Error". *CoRR*. abs/2003.03021. arXiv: 2003.03021. URL: https://arxiv.org/abs/2003.03021.

Jia, R., A. Raghunathan, K. Göksel, and P. Liang. (2019). "Certified Robustness to Adversarial Word Substitutions". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Ed. by K. Inui, J. Jiang, V. Ng, and X. Wan. Association for Computational Linguistics. 4127–4140. DOI: 10.18653/v1/D19-1423.

Katz, G., C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. (2017). "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by R. Majumdar and V. Kuncak. Vol. 10426. *Lecture Notes in Computer Science*. Springer. 97–117. DOI: 10.1007/978-3-319-63387-9_5.

LeCun, Y., C. Cortes, and C. Burges. (2010). "MNIST handwritten digit database". *ATT Labs*. 2. URL:http://yann.lecun.com/exdb/mnist.

Lécuyer, M., V. Atlidakis, R. Geambasu, D. Hsu, and S. Jana. (2019). "Certified Robustness to Adversarial Examples with Differential Privacy". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE. 656–672. DOI: 10.1109/SP.2019.00044.

Leino, K., Z. Wang, and M. Fredrikson. (2021). "Globally-Robust Neural Networks". In: *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*. Ed. by M. Meila and T. Zhang. Vol. 139. *Proceedings of Machine Learning Research*. PMLR. 6212–6222. URL: http://proceedings.mlr.press/v139/leino21a.html.

Leshno, M., V. Y. Lin, A. Pinkus, and S. Schocken. (1993). "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". *Neural Networks*. 6(6): 861–867. DOI: 10.1016/S0893-6080(05)80131-5.

Li, Q., S. Haque, C. Anil, J. Lucas, R. B. Grosse, and J. Jacobsen. (2019). "Preventing Gradient Attenuation in Lipschitz Constrained Convolutional Networks". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 15364–15376. URL: https://proceedings.neurips.cc/paper/2019/hash/1ce3e6e3f452828e23a0c94572bef9d9-Abstract.html.

Liu, C., T. Arnon, C. Lazarus, C. A. Strong, C. W. Barrett, and M. J. Kochenderfer. (2021). "Algorithms for Verifying Deep Neural Networks". *Found. Trends Optim.* 4(3-4): 244–404. DOI: 10.1561/2400000035.

Madry, A., A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. (2018). "Towards Deep Learning Models Resistant to Adversarial Attacks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=rJzIBfZAb.

Mikolov, T., K. Chen, G. Corrado, and J. Dean. (2013). "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. URL: http://arxiv.org/abs/1301.3781.

Mirman, M., T. Gehr, and M. T. Vechev. (2018). "Differentiable Abstract Interpretation for Provably Robust Neural Networks". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. *Proceedings of Machine Learning Research*. PMLR. 3575–3583. URL: http://proceedings.mlr.press/v80/mirman18b.html.

Moura, L. M. de and N. Bjørner. (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. *Lecture Notes in Computer Science*. Springer. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

Müller, C., G. Singh, M. Püschel, and M. T. Vechev. (2020). "Neural Network Robustness Verification on GPUs". *CoRR*. abs/2007.10868. arXiv: 2007.10868. URL: https://arxiv.org/abs/2007.10868.

Nair, V. and G. E. Hinton. (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. Ed. by J. Fürnkranz and T. Joachims. Omnipress. 807–814. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

Narodytska, N., S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. (2018). "Verifying Properties of Binarized Deep Neural Networks". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press. 6615–6624. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16898.

Nelson, G. and D. C. Oppen. (1979). "Simplification by Cooperating Decision Procedures". *ACM Trans. Program. Lang. Syst.* 1(2): 245–257. DOI: 10.1145/357073.357079.

Nielsen, M. A. (2018). *Neural Networks and Deep Learning*. Determination Press. URL: http://neuralnetworksanddeeplearning.com/.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 8024–8035. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

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

Qin, C., K. (Dvijotham, B. O'Donoghue, R. Bunel, R. Stanforth, S. Gowal, J. Uesato, G. Swirszcz, and P. Kohli. (2019). "Verification of Non-Linear Specifications for Neural Networks". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=HyeFAsRctQ.

Ramakrishnan, G. and A. Albarghouthi. (2020). "Backdoors in Neural Models of Source Code". *CoRR*. abs/2006.06841. arXiv: 2006.06841. URL: https://arxiv.org/abs/2006.06841.

Rosenfeld, E., E. Winston, P. Ravikumar, and J. Z. Kolter. (2020). "Certified Robustness to Label-Flipping Attacks via Randomized Smoothing". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. *Proceedings of Machine Learning Research*. PMLR. 8230–8241. URL: http://proceedings.mlr.press/v119/rosenfeld20b.html.

Sharma, R., A. V. Nori, and A. Aiken. (2014). "Bias-variance tradeoffs in program analysis". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by S. Jagannathan and P. Sewell. ACM. 127–138. DOI: 10.1145/2535838.2535853.

Sherman, B., J. Michel, and M. Carbin. (2021). "λ_s: computable semantics for differentiable programming with higher-order functions and datatypes". *Proc. ACM Program. Lang.* 5(POPL): 1–31. DOI: 10.1145/3434284.

Singh, G., R. Ganvir, M. Püschel, and M. T. Vechev. (2019a). "Beyond the Single Neuron Convex Barrier for Neural Network Certification". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 15072–15083. URL: https://proceedings.neurips.cc/paper/2019/hash/0a9fdbb17feb6ccb7ec405cfb85222c4-Abstract.html.

Singh, G., T. Gehr, M. Mirman, M. Püschel, and M. T. Vechev. (2018). "Fast and Effective Robustness Certification". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 10825–10836. URL: https://proceedings.neurips.cc/paper/2018/hash/f2f446980d8e971ef3da97af089481c3-Abstract.html.

Singh, G., T. Gehr, M. Püschel, and M. T. Vechev. (2019b). "An abstract domain for certifying neural networks". *Proc. ACM Program. Lang.* 3(POPL): 41:1–41:30. DOI: 10.1145/3290354.

Singh, G., M. Püschel, and M. T. Vechev. (2017). "Fast polyhedra abstract domain". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM. 46–59. DOI: 10.1145/3009837.3009885.

Sivaraman, A., G. Farnadi, T. D. Millstein, and G. V. den Broeck. (2020). "Counterexample-Guided Learning of Monotonic Neural Networks". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. URL: https://proceedings.neurips.cc/paper/2020/hash/8ab70731b1553f17c11a3bbc87e0b605-Abstract.html.

Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. (2014). "Intriguing properties of neural networks". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. URL: http://arxiv.org/abs/1312.6199.

Tarski, A. (1998). "A decision method for elementary algebra and geometry". In: *Quantifier elimination and cylindrical algebraic decomposition*. Springer. 24–84.

Tjeng, V., K. Y. Xiao, and R. Tedrake. (2019a). "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=HyGIdiRqtm.

Tjeng, V., K. Y. Xiao, and R. Tedrake. (2019b). "Evaluating Robustness of Neural Networks with Mixed Integer Programming". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=HyGIdiRqtm.

Tran, H., S. Bak, W. Xiang, and T. T. Johnson. (2020a). "Verification of Deep Convolutional Neural Networks Using ImageStars". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by S. K. Lahiri and C. Wang. Vol. 12224. *Lecture Notes in Computer Science*. Springer. 18–42. DOI: 10.1007/978-3-030-53288-8_2.

Tran, H., S. Bak, W. Xiang, and T. T. Johnson. (2020b). "Verification of Deep Convolutional Neural Networks Using ImageStars". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by S. K. Lahiri and C. Wang. Vol. 12224. *Lecture Notes in Computer Science*. Springer. 18–42. DOI: 10.1007/978-3-030-53288-8_2.

Trockman, A. and J. Z. Kolter. (2021). "Orthogonalizing Convolutional Layers with the Cayley Transform". In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. URL: https://openreview.net/forum?id=Pbj8H%5C_jEHYv.

Turing, A. (1949). "On checking a large routine". In: *Report of a Conference on 11i9h Speed Automatic Calculating Machines*. 67–69.

Turing, A. (1969). "Intelligent machinery. 1948". *The Essential Turing*: 395.

Wang, S., K. Pei, J. Whitehouse, J. Yang, and S. Jana. (2018). "Formal Security Analysis of Neural Networks using Symbolic Intervals". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by W. Enck and A. P. Felt. USENIX Association. 1599–1614. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi.

Wang, Z., A. Albarghouthi, G. Prakriya, and S. Jha. (2020). "Interval Universal Approximation for Neural Networks". *CoRR*. abs/2007.06093. arXiv: 2007.06093. URL: https://arxiv.org/abs/2007.06093.

Weng, T., H. Zhang, H. Chen, Z. Song, C. Hsieh, L. Daniel, D. S. Boning, and I. S. Dhillon. (2018). "Towards Fast Computation of Certified Robustness for ReLU Networks". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. *Proceedings of Machine Learning Research*. PMLR. 5273–5282. URL: http://proceedings.mlr.press/v80/weng18a.html.

Westburg, J. (2017). https://tex.stackexchange.com/questions/356121/how-to-draw-these-polyhedrons.

Xiao, K. Y., V. Tjeng, N. M. (Shafiullah, and A. Madry. (2019). "Training for Faster Adversarial Robustness Verification via Inducing ReLU Stability". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=BJfIVjAcKm.

Xu, K., Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. (2020). "Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc. 1129–1141. URL: https://proceedings.neurips.cc/paper/2020/file/0cbc5671ae26f67871cb914d81ef8fc1-Paper.pdf.

Ye, M., C. Gong, and Q. Liu. (2020). "SAFER: A Structure-free Approach for Certified Robustness to Adversarial Word Substitutions". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Ed. by D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault. Association for Computational Linguistics. 3465–3475. DOI: 10.18653/v1/2020.acl-main.317.

Zhang, H., T. Weng, P. Chen, C. Hsieh, and L. Daniel. (2018a). "Efficient Neural Network Robustness Certification with General Activation Functions". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 4944–4953. URL: https://proceedings.neurips.cc/paper/2018/hash/d04863f100d59b3eb688a11f95b0ae60-Abstract.html.

Zhang, X., A. Solar-Lezama, and R. Singh. (2018b). "Interpreting Neural Network Judgments via Minimal, Stable, and Symbolic Corrections". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 4879–4890. URL: https://proceedings.neurips.cc/paper/2018/hash/300891a62162b960cf02ce3827bb363c-Abstract.html.

Zhang, Y., A. Albarghouthi, and L. D'Antoni. (2020). "Robustness to Programmable String Transformations via Augmented Abstract Training". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. *Proceedings of Machine Learning Research*. PMLR. 11023–11032. URL: http://proceedings.mlr.press/v119/zhang20b.html.

Zhang, Y., A. Albarghouthi, and L. D'Antoni. (2021). "Certified Robustness to Programmable Transformations in LSTMs". *CoRR*. abs/2102.07818. arXiv: 2102.07818. URL: https://arxiv.org/abs/2102.07818.

Zhu, H., Z. Xiong, S. Magill, and S. Jagannathan. (2019). "An inductive synthesis framework for verifiable reinforcement learning". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM. 686–701. DOI: 10.1145/3314221.3314638.