Introduction to Neural Network Verification

AWS ALBARGHOUTHI

IN PROGRESS; DO NOT CIRCULATE

LAST UPDATED: JUNE 4, 2021

About This Book

Why This Book?

Over the past decade, a number of hardware and software advances 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: Everyday we are seeing new applications, from healthcare to art, and the 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 book 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 Book For?

Given that the book'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 book can serve as an introduction to the field for first-year graduate students or senior undergraduates, even if have not been exposed to deep learning or verification.

What Does This Book Cover?

The book 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 book. 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 disjoint; the reader may go directly from Part 1 to Part 3 without losing context.

Table of Contents

III Abstraction-Based Verification

Part I

Neural Networks & Correctness

Chapter 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 book 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, 1989). 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 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

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

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 book, 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* (Turing, 1948). 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 only recently exploded in popularity, 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 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?

Remember 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 us 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 the 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 a 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 chapters, 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.

Chapter 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 chapter, 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 real numbers to 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

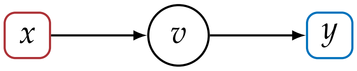


Figure 2.1 A very simple network

operation on x and spits out a value 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 (1). The function f_y is an *activation* function, because it turns on or off. When its input is negative, f_y outputs 0, otherwise it outputs its input. Specifically, f_y is called a *rectified linear unit* (ReLU), and it is a very popular activation function in modern deep neural networks. Activation functions are used to add non-linearity into a neural network.

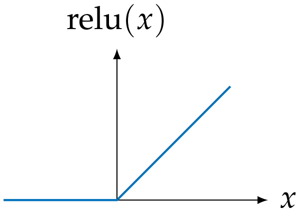


Figure 2.2 Rectified linear unit

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.

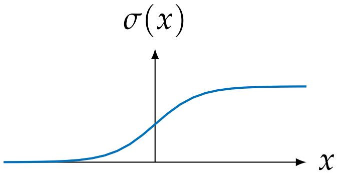


Figure 2.3 Sigmoid activation function

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 separate the two operations on to two different nodes of the graph, as it will simplify our life in later chapters 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*, 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! 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 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—

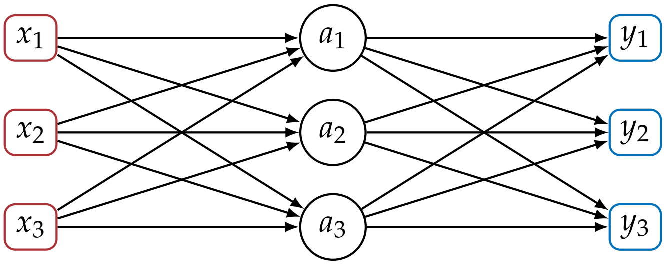


Figure 2.4 A multilayer perceptron

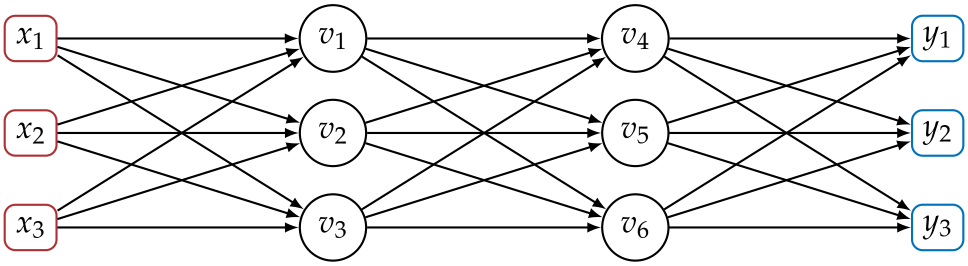


Figure 2.5 A multilayer perceptron with two hidden layers

or architecture—has the grandiose name of *multilayer perceptron* (MLP). Usually, we have a bunch of hidden layers in an MLP, like in Figure 2.5. Layers in an MLP are called *fully connected* 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 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:

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

To see why this actually works, please see Nielsen (2018, Chapter 3) for an interactive visualization.

Given an 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 us 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_i, 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.

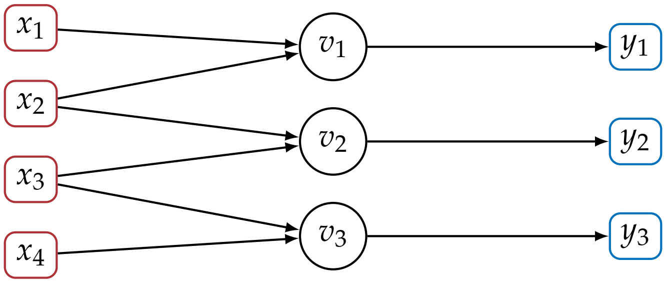


Figure 2.6 1-dimensional convolution

Typically, a *convolutional neural network* 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 chapters when we verify properties of such networks.

2.4 **Where are the Loops?**

All 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 graphs are really just directed acyclic graphs (DAGs). 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* (RNNs) 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.

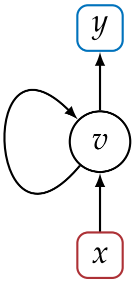


Figure 2.7 Recurrent neural network

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. 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.

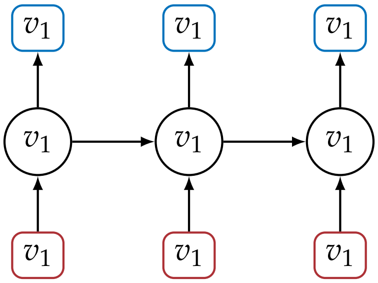


Figure 2.8 Unrolled recurrent neural network

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 Networks**

We are done with looking at pretty graphs. Let us now look at pretty symbols. We will now formally define graphs and discuss some of their properties.

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.
- $V^{\text{in}} \subseteq V$ is a non-empty set of input nodes.
- $V^{\text{o}} \subset V$ is a non-empty set of output nodes.

- Each non-input node v is associated with a function $f_v : \mathbb{R}^n \to \mathbb{R}$, where n is the number of edges whose target is node v. Notice that we assume, for simplicity but without loss of generality, that a node v only outputs a single real value. The vector of real values $\mathbb{R}^n$ that v takes as input is all the outputs of nodes v' such that $(v', v) \in E$.

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.

Semantics of DAGs

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

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

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 $(v_1, v), \ldots, (v_n, v)$ 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

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

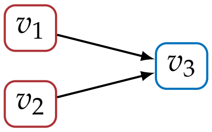
where $x_i = \mathsf{out}(v_i)$, for $i \in [1, n]$.

The base case of this definition is input nodes, since they have no edges incident on them. Suppose we are given an input $\boldsymbol{x} \in \mathbb{R}^n$, where we will use $\boldsymbol{x}$ to denote a vector and x_i to denote its ith element. 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 us look at an example graph G



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

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

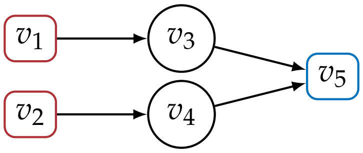
and that we are given the input vector $(11, 79)$ to the network, where node v_1 gets the value 11 and v_2 the value 79. Then, we have

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

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.[1] 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



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

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

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

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 operations

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

Many of the operations 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:

$$f(\boldsymbol{x}) = \sum_i c_i x_i + b$$

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 iff for any $x \geqslant y$, we have $f(x) \geqslant f(y)$. Both activation functions we saw earlier in the chapter, 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 are ready to pose questions about their behavior. In the next chapter, we will formally define a language for posing those questions. Then, in the chapters 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, 2019; Sherman et al., 2020). 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.

Chapter 3
Correctness Properties

In this chapter, 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 chapter 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 book, 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 networ, 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 discussion of 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 checking is *robustness* of such classifier. A classifier is robust if its prediction does not change with small variations in input. For example, changing the brightness slightly or damaging a few pixels should not change classification.

Let us 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 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 specified a set of images $\boldsymbol{x}$ that are brighter or darker than $\boldsymbol{c}$, and the postcondition specified that the classification of f remain unchanged.

That's a desirable property: you don't want classification to change with a small movement in the brightness slider. But there are many other things 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.)

Natural language

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. We are 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

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 to think that it's a benign program. One trick the attacker may use is adding a piece of code to that does not change its 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 us 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

We are now ready to fully formalize our specification language. Our specifications are going to look like this:

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

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 are 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—we will also say *does not hold*.

Example 3.A 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). Then, we can state the following specification, which informally says that the 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 us 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.

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 that does not hold is a valuation of the variables in the precondition (the $\boldsymbol{x}_i$s) that falsifies the postcondition. For example, in Example 3.A, a counterexample would be an image $\boldsymbol{x}$ whose classification by f is different than that of image $\boldsymbol{c}$.

Example 3.B Here's a simple, concrete example

$$\begin{gathered}\{\, x \leqslant 0.1 \,\} \\ r \leftarrow x + 1 \\ \{\, r \leqslant 1 \,\}\end{gathered}$$

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*. Specifications in Hoare logic are called *Hoare triples*, 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 book, we will not define the rules of Hoare logic, but many of them will crop up implicitly throughout the book.

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 are exactly the same function, 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 images, but keep in mind that most images (combinations of pixels) are 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 the full set of rules.

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 us 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.

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

Notice that the precondition specifies that the distance between the two aircrafts is more than 55K feet, that the aircraft's velocity is high and the intruder's velocity is low. In which case, 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.

Katz et al. (2017) explore a number of such properties, and also consider robustness properties in this setting. But how do we come up with such specific properties? It's not straigtforward. 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. It is a common sentiment in the verification community 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, like 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 the neural networks understanding of the pendulum's dynamics 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. As it goes up, its potential energy increases and kinetic energy decreases; as it goes down, the opposite happens. The sum of kinetic and potential energies should only decrease over time. We can state this property as follows:

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

We break the input and output vectors of the network into their three components for clarity. The expression $E(v, h, w)$ 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

Let us recall the natural language example from earlier in the chapter, 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.

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

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 $T(c_i)$. 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 1 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):

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

Looking Ahead

We are done with the first part of the book. 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'll survey those later. The robustness properties for natural-language processing we have defined follow those of Ebrahimi et al. (2017) and Huang et al. (2019).

Part II

Constraint-Based Verification

Chapter 4
Logics and Satisfiability

In this part of the book, we will look into constraint-based techniques for verification. The idea is to take a correctness property and encode it as a set of constraints. By solving the constraints, we can decide whether the correctness property holds or not.

The constraints we will use are formulas in *first-order logic* (FOL). FOL is a very big and beautiful place, but neural networks only live in a small and cozy corner of it—the corner that we will explore in this chapter.

4.1 **Propositional Logic**

We begin with the purest of all, *propositional logic*. A formula F in propositional logic is over Boolean variables that are traditionally given the names $p, q, r, \ldots$. A formula F is defined using the following grammar:

$$\begin{array}{rlr} F := & \mathsf{true} & \\ & \mathsf{false} & \\ & \mathit{var} & \text{Variable} \\ & \mid F \land F & \text{Conjunction (and)} \\ & \mid F \lor F & \text{Disjunction (or)} \\ & \mid \neg F & \text{Negation (not)} \end{array}$$

Essentially, a formula in propositional logic defines a circuit with Boolean variables, AND gates ($\land$), OR gates ($\lor$), and NOR gates ($\neg$). Negation has the highest operator precedence, followed by conjunction and then disjunction. At the end of the day, all programs can be defined as circuits, because everything is a bit on a computer and there is a finite amount of memory, and therefore a finite number of variables.

We will use $\mathit{fv}(F)$ to denote the set of *free* variables appearing in the formula. For now, it is the set of all variables that are syntactically present in the formula;

Example 4.A As an example, here is a formula

$$F \triangleq (p \land q) \lor \neg r$$

Observe the use of $\triangleq$; this is to denote that we are syntactically defining F to be the formula on the right of $\triangleq$, as opposed to saying that the two formulas are semantically equivalent (more on this in a bit). The set of free variables in F is $\mathit{fv}(F) = \{p, q, r\}$. ■

Interpretations

Let F be a formula over a set of variables $\mathit{fv}(F)$. An interpretation I of F is a map from variables $\mathit{fv}(F)$ to true or false. Given an interpretation I of a formula F, we will use $I(F)$ to denote the formula where we have replaced each variable $\mathit{fv}(F)$ with its interpretation in I.

Example 4.B Say we have the formula

$$F \triangleq (p \land q) \lor \neg r$$

and the interpretation

$$I = \{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}, r \mapsto \mathsf{false}\}$$

Applying I to F, we get

$$I(F) \triangleq (\mathsf{true} \land \mathsf{true}) \lor \neg\mathsf{false}$$

■

Evaluation rules

We will define the following evaluation, or simplification, rules for a formula. The formula on the right of $\equiv$ is an equivalent, but syntactically simpler, variant of the one on the left:

$$\begin{array}{rcll}
\mathsf{true} \land F & \equiv & F & \text{Conjunction} \\
F \land \mathsf{true} & \equiv & F & \\[1ex]
\mathsf{false} \land F & \equiv & \mathsf{false} & \\
F \land \mathsf{false} & \equiv & \mathsf{false} & \\[3ex]
\mathsf{false} \lor F & \equiv & F & \text{Disjunction} \\
F \lor \mathsf{false} & \equiv & F & \\
\mathsf{true} \lor F & \equiv & \mathsf{true} & \\
F \lor \mathsf{true} & \equiv & \mathsf{true} & \\[1ex]
\neg\mathsf{true} & \equiv & \mathsf{false} & \text{Negation} \\
\neg\mathsf{false} & \equiv & \mathsf{true} &
\end{array}$$

If a given formula has no free variables, then applying these rules repeatedly, you will get true or false. We will use $\mathsf{eval}(F)$ to denote the simplest form of F we can get by repeatedly applying the above rules.

Satisfiability

A formula F is *satisfiable* (SAT) if and only if there exists an interpretation I such that

$$\mathsf{eval}(I(F)) = \mathsf{true}$$

in which case we will say that I is a *model* of F and denote it

$$I \models F$$

We will also use $I \not\models F$ to denote that I is not a model of F. It follows from our definitions that $I \not\models F$ iff $I \models \neg F$.

Equivalently, a formula F is *unsatisfiable* (UNSAT) if and only if for every interpretation I we have $\mathsf{eval}(I(F)) = \mathsf{false}$.

Example 4.C Consider the formula $F \triangleq (p \lor q) \land (\neg p \lor r)$. This formula is satisfiable; here is a model $I = \{p \mapsto \mathsf{true}, q \mapsto \mathsf{false}, r \mapsto \mathsf{true}\}$. ■

Example 4.D Consider the formula $F \triangleq (p \lor q) \land \neg p \land \neg q$. This formula is unsatisfiable. ■

Validity and equivalence

To prove properties of neural networks, we will be asking *validity* questions. A formula F is valid iff every possible interpretation I is a model of F. It follows that a formula F is valid if and only if $\neg F$ is unsatisfiable.

Example 4.E Here is a valid formula $F \triangleq (\neg p \lor q) \lor p$. Pick any interpretation I that you like, and you will find that $I \models F$. ■

We will say that two formulas, A and B, are *equivalent* if and only if every model I of A is a model of B, and vice versa. We will denote equivalence as $A \equiv B$. There are many equivalences that are helpful when working with formulas. For any formulas A, B, and C, we have commutativity of conjunction and disjunction:

$$\begin{aligned} A \land B &\equiv B \land A \\ A \lor B &\equiv B \lor A \end{aligned}$$

we can push negation inwards:

$$\begin{aligned} \neg(A \land B) &\equiv \neg A \lor \neg B \\ \neg(A \lor B) &\equiv \neg A \land \neg B \end{aligned}$$

and we also have distributivity of conjunction over disjunction, and vice versa (*DeMorgan's laws*):

$$\begin{aligned} A \lor (B \land C) &\equiv (A \lor B) \land (A \lor C) \\ A \land (B \lor C) &\equiv (A \land B) \lor (A \land C) \end{aligned}$$

Implication and bi-implication

We will often use an *implication* $A \Rightarrow B$ to denote the formula

$$\neg A \vee B$$

Similarly, we will use a *bi-implication* $A \Leftrightarrow B$ to denote the formula

$$(A \Rightarrow B) \wedge (B \Rightarrow A)$$

4.2 Arithmetic Theories

We can now extend propositional logic using *theories*. Each Boolean variable now becomes a more complex Boolean expression over variables of different types. For example, we can use the theory of *linear real arithmetic* (LRA), where a Boolean expression is, for instance:

$$x + 3y + z \leqslant 10$$

Alternatively, we can use the theory of *arrays*, and so an expression may look like:

$$a[10] = x$$

where a is an array indexed by integers. There are many other theories that people have studied, including *bitvectors* (to model machine arithmetic) and *strings* (to model string manipulation). The satisfiabilty problem is now called *satisfiability modulo theories* (SMT), as we check satisfiability with respect to interpretations of the theory.

In this section, we will focus on the theory of linear real arithmetic (LRA), as it is (1) decidable and (2) can represent a large class of neural-network operations, as we will see in the next chapter.

Linear Real Arithmetic

In linear real arithmetic, each propositional variable is replaced by a linear inequality of the form:

$$\sum_{i=1}^{n} c_i x_i + b \leqslant 0$$

or

$$\sum_{i=1}^{n} c_i x_i + b < 0$$

where $c_i, b \in \mathbb{R}$ and $\{x_i\}_i$ is a fixed set of variables. For example, we can have a formula of the form:

$$(x + y \leqslant 0 \ \land \ x - 2y < 10) \lor x > 100$$

Note that $>$ and $\geqslant$ can be rewritten into $<$ and $\leqslant$. Also note when a multiplier c_i is 0, we simply drop the term $c_i x_i$, as in the inequality $x > 0$ above, which does not include y. An equality $x = 0$ can be written as the conjunction $x \geqslant 0 \land x \leqslant 0$. Similarly, a disequality $x \neq 0$ can be written as $x < 0 \lor x > 0$.

Models in LRA

As with propositional logic, the free variables $fv(F)$ of a formula F in LRA is the set of variables appearing in the formula.

An interpretation I of a formula F is an assignment of every free variable to a real number. An interpretation I is a model of F, i.e., $I \models F$, iff $\mathsf{eval}(I(F)) = \mathsf{true}$. Here, the extension of the simplification rules to LRA formulas is straightforward: all we need is to add standard rules for evaluating arithmetic inequalities, e.g., $2 \leqslant 0 \equiv \mathsf{false}$.

Example 4.F As an example, consider the following formula:

$$F \triangleq x - y > 0 \ \land \ x \geqslant 0$$

A model I for F is

$$\{x \mapsto 1, y \mapsto 0\}$$

Applying I to F, i.e., $I(F)$, results in

$$1 - 0 > 0 \ \land \ 1 \geqslant 0$$

Applying the evaluation rules, we get true. ■

Real vs. Rational

In the literature, you might find LRA being referred to as *linear rational arithmetic*. There two interrelated reasons for that: First, whenever we write formulas in practice, the constants in those formulas are rational values—we can't really represent π, for instance, in computer memory. Second, let us say that F contains only rational coefficients. Then, it follows that, if F is satisfiable, there is a model of F that assigns all free variables to rational values.

Example 4.G Let us consider a simple formula like $x < 10$. While $\{x \mapsto \pi\}$ is a model of $x < 10$, it also has satisfying assignments that assign x to a rational constant, like $\{x \mapsto 1/2\}$. This will always be the case: we cannot construct formulas that only have irrational models, unless the formulas themselves contain irrational constants, e.g., $x = \pi$. ■

Non-Linear Arithmetic

Deciding satisfiability of formulas in LRA is an NP-complete problem. If we extend our theory to allow for polynomial inequalities, then the best known algorithms are worst case doubly exponential in the size of the formula. If we allow for transcedental functions—like exp, cos, log, etc.—then satisfiability becomes undecidable. Thus, for all practical purposes, we stick to LRA. Even though it is NP-complete (a term that sends shivers down the spines of theoreticians), we have very efficient algorithms that can scale to large formulas.

Connections to MILP

Formulas in LRA, and the SMT problem for LRA, is equivalent to the *mixed integer linear programming* (MILP) problem. Just as there are many SMT solvers, there are many MILP solvers out there, too. So the natural question to ask is why don't we use MILP solvers? In short, we can, and maybe sometimes they will actually be faster than SMT solvers. However, the SMT framework is quite general and flexible. So not only can we write formulas in LRA, but we can (1) write formulas in different theories, as well as (2) formulas *combining* theories.

First, in practice, neural networks do not operate over real or rational arithmetic. They run using floating point, fixed point, or machine-integer arithmetic. If we wish to be as precise as possible at analyzing neural networks, we can opt for a bit-level encoding of its operations and use bitvector theories employed by SMT solvers. (Machine arithmetic, surprisingly, is practically more expensive to solve than linear real arithmetic, so most of the time we opt for a real-arithmetic encoding of neural networks.)

Second, as we move forward and neural networks start showing up everywhere, we do not want to verify them in isolation, but in conjunction with other pieces of code that the neural network interacts with. For example, think of a piece of code that parses text and puts it in a form ready for the neural network to consume. Analyzing such piece of code might require using *string* theories, which allow us to use string concatenation and other string operations in formulas. SMT solvers employ theorem-proving techniques for *combining* theories, and so we can write formulas, for example, over strings and linear arithmetic.

These are the reasons why in this book we use SMT solvers as the target of our constraint-based verification: they give us many first-order theories and allow us to combine them. However, it is important to note that, at the time of writing this, most research on constraint-based verification focuses on linear real arithmetic encodings.

Looking Ahead

In the next chapter, we will look at how to encode neural-network semantics, and correctness properties, as formulas in LRA, thus enabling automated verification using SMT solvers. After that, we will spend some time studying the algorithms underlying SMT solvers.

In verification, we typically use fragments of first-order logic to encode programs. FOL has a long and storied history. FOL is a very general logic, and its satisfiability is undecidable, thanks to proofs by Church and Turing. SMT solvers, which have been heavily studied over the past twenty years or so aim at solving fragments of FOL, like LRA and other theories. I encourage the interested reader to consult the *Handbook of Satisfiability* for an in-depth exposition (Biere et al., 2009).

Chapter 5

Encodings of Neural Networks

Our goal in this chapter is to translate a neural network into a formula in linear real arithmetic (LRA). The idea is to have the formula precisely (or soundly) capture the input–output relation of the neural network. Once we have such a formula, we can use it to verify correctness properties using SMT solvers.

5.1 **Encoding Nodes**

We begin by characterizing a relational view of a neural network. This will help us establish the correctness of our encoding.

Input-output relations

Recall that a neural network is a graph G that defines a function $f_G : \mathbb{R}^n \to \mathbb{R}^m$. We define the *input–output relation* of f_G as the binary relation containing every possible input and its corresponding output after executing f_G. Formally, the input–output relation of f_G is:

$$R_G = \{(\boldsymbol{a}, \boldsymbol{b}) \mid \boldsymbol{a} \in \mathbb{R}^n,\ \boldsymbol{b} = f_G(\boldsymbol{a})\}$$

We will similarly use R_v to define the input–output relation of the function f_v of a single node in G.

Example 5.A Consider the simple function $f_G(x) = x + 1$. Then the relation $R_G = \{(a, a+1) \mid a \in \mathbb{R}\}$. ■

Encoding a single node, illustrated

We begin by considering the case of a single node v and the associated function $f_v : \mathbb{R} \to \mathbb{R}$. Note that the node has a single input; by definition, each node in our networks can only produce a single real-valued output. Say $f_v(x) = x + 1$.

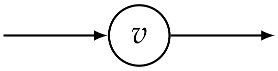


Figure 5.1 A single-input node

Then, we can construct the following formula in LRA to model the relation $R_v = \{(a, a+1) \mid a \in \mathbb{R}\}$:

$$F_v \triangleq v^{\mathsf{o}} = v^{\mathsf{in},1} + 1$$

where v^{o} and $v^{\mathsf{in},1}$ are real-valued variables. v^{o} denotes the output of node v and $v^{\mathsf{in},1}$ denotes its first input (it only has a single input).

Consider the models of F_v; they are all of the form:

$$\{v^{\mathsf{in},1} \mapsto a,\ v^{\mathsf{o}} \mapsto a+1\}$$

for any real number a. We can see a clear one-to-one correspondence between elements of R_v and models of F_v.

Let us now take a look at a node v with two inputs, and let us assume that $f_v(\boldsymbol{x}) = x_1 + 1.5x_2$.

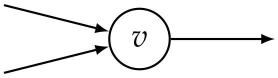


Figure 5.2 A two-input node

The encoding F_v is as follows:

$$F_v \triangleq v^{\mathsf{o}} = v^{\mathsf{in},1} + 1.5v^{\mathsf{in},2}$$

Observe how the elements of the input vector, x_1 and x_2, correspond to the two logical variables $v^{\mathsf{in},1}$ and $v^{\mathsf{in},2}$.

Encoding a single node, formalized

Now that we have seen a couple of examples, let us formalize the process of encoding the operation f_v of some node v. We will assume that f_v is piecewise-linear, i.e., of the form

$$f(\boldsymbol{x}) = \begin{cases} \sum_i c_i^1 \cdot x_i + b^1, & S_1 \\ \vdots \\ \sum_i c_i^l \cdot x_i + b^k, & S_l \end{cases}$$

We will additionally assume that each condition S_i is defined as a formula in LRA over the elements of the input $\boldsymbol{x}$. Now, the encoding is as follows:

$$F_v \triangleq \bigwedge_{i=1}^{l} \left[S_i \Rightarrow \left(v^{\mathsf{o}} = \sum_{j=1}^{n} c_j^i \cdot v^{\mathsf{in},j} + b^i \right) \right]$$

The way to think of this encoding is as a combination of *if* statements: if S_i is true, then v^{o} is equal to the ith inequality. The implication ($\Rightarrow$) gives us the ability to model a conditional, where the left side of the implication is the condition, and the right side is the condition.

Example 5.B The above encoding is way too general with too many superscripts and subscripts. Here's a simple and practical example, the ReLU function:

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

A node v such that f_v is ReLU would be encoded as follows:

$$F_v \triangleq (\underbrace{v^{\mathsf{in},1} > 0}_{x>0} \Longrightarrow v^{\mathsf{o}} = v^{\mathsf{in},1}) \land (\underbrace{v^{\mathsf{in},1} \leqslant 0}_{x \leqslant 0} \Longrightarrow v^{\mathsf{o}} = 0)$$

■

Soundness and completeness

The above encoding precisely captures the semantics of a piecewise-linear node. Let us formally capture this fact: Fix some node v with a piecewise-linear function f_v. Let F_v be its encoding, as defined above.

First, our encoding is *sound*: any execution of the node is captured by a model of the formula F_v. Informally, soundness means that our encoding does not miss any behavior of f_v. Formally, let $(\boldsymbol{a}, b) \in R_v$ and let

$$I = \{v^{\mathsf{in},1} \mapsto a_1, \ldots, v^{\mathsf{in},1} \mapsto a_n, v^{\mathsf{o}} \mapsto b\}$$

Then, $I \models F_v$.

Second, our encoding is *complete*: any model of F_v maps to a behavior of f_v. Informally, completeness means that our encoding is tight, or does not introduce new behaviors not exhibited by f_v. Formally, let the following be a model of F_v:

$$I = \{v^{\mathsf{in},1} \mapsto a_1, \ldots, v^{\mathsf{in},1} \mapsto a_n, v^{\mathsf{o}} \mapsto b\}$$

Then, $(\boldsymbol{a}, b) \in R_v$.

5.2 **Encoding a Neural Network**

We have shown how to encode a single node of a neural network. We are now ready to encode the full-blown graph. The encoding is broken up into two pieces: (1) a formula encoding semantics of all nodes, and (2) a formula encoding the connections between them, i.e., the edges.

Encoding the nodes

Recall that a neural network is a graph $G = (V, E)$, where the set of nodes V contains input nodes V^{in}, which do not perform any operations. The following formula combines the encodings of all non-input nodes in G:

$$F_V \triangleq \bigwedge_{v \in V \setminus V^{\mathsf{in}}} F_v$$

This formula is meaningless on its own: it simply encodes the input–output relation of every node, but not the connections between them!

Encoding the edges

Let us now encode the edges. We will do this for every node individually, encoding all of its incoming edges. Fix some node $v \in V \setminus V^{\text{in}}$. Let $(v_1, v), \ldots (v_n, v)$ be an ordered sequence of all edges whose target is v. Recall that in Section 2.5, we assumed that there is a total ordering on edges. The reason for this ordering is to be able to know that the first edge feeds in the first input, the second edge feeds in the second input, and so on.

We can now define a formula for edges of v:

$$F_{\circ \to v} \triangleq \bigwedge_{i=1}^{n} v^{\text{in},i} = v_i^{\text{o}}$$

Intuitively, for each edge (v_i, v), we connect the output of node v_i with the ith input of v. We can now define F_E as the conjunction of all incoming edges of all non-input nodes:

$$F_E \triangleq \bigwedge_{v \in V \setminus V^{\text{in}}} F_{\circ \to v}$$

Putting it all together

Now that we have shown how to encode nodes and edges, there is nothing left to encode! So let's put things together. Given a graph $G = (V, E)$, we will define its encoding as follows:

$$F_G \triangleq F_V \wedge F_E$$

Just as for the single-node encoding, we get soundness and completeness. Let R_G be the input–output relation of G. Soundness means that F_G does not miss any of the behaviors in R_G. Completeness means that every model of F_G maps to an input–output behavior of G.

Correctness of the encoding

Assume we have the following ordered input nodes in V^{in}

$$v_1, \ldots, v_n$$

and the following output nodes in V^{o}

$$v_{n+1}, \ldots, v_{n+m}$$

Our encoding is sound and complete. First, let's state soundness: Let $(\boldsymbol{a}, \boldsymbol{b}) \in R_G$ and let

$$I = \{v_1{}^{\mathsf{o}} \mapsto a_1, \ldots, v_n{}^{\mathsf{o}} \mapsto a_n\} \cup \{v_{n+1}{}^{\mathsf{o}} \mapsto b_1, \ldots, v_{n+m}{}^{\mathsf{o}} \mapsto b_m\}$$

Then, there exists I' such that $I \cup I' \models F_G$.

Notice that, unlike the single-node setting, the model of F_G not only contains assignments to inputs and outputs of the network, but also the intermediate nodes. This is taken care of using I, which assigns values to the outputs of input and output nodes, and I', which assigns the inputs and outputs of all nodes and therefore its domain does not overlap with I.

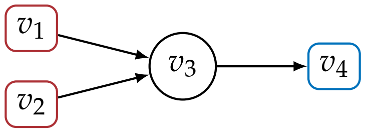
Similarly, completeness is stated as follows: Let the following be a model of F_G:

$$I = \{v_1{}^{\mathsf{o}} \mapsto a_1, \ldots, v_n{}^{\mathsf{o}} \mapsto a_n\} \cup \{v_{n+1}{}^{\mathsf{o}} \mapsto b_1, \ldots, v_{n+m}{}^{\mathsf{o}} \mapsto b_m\} \cup I'$$

Then, $(\boldsymbol{a}, \boldsymbol{b}) \in R_G$.

An example network and its encoding

Enough abstract mathematics. Let us look at a concrete example neural network G.



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

We begin by constructing formulas for non-input nodes:

$$\begin{aligned} F_{v_3} &\triangleq v_3{}^{\mathsf{o}} = 2v_3{}^{\mathsf{in},1} + v_3{}^{\mathsf{in},2} \\ F_{v_4} &\triangleq (v_4{}^{\mathsf{in},1} > 0 \Longrightarrow v_4{}^{\mathsf{o}} = v_4{}^{\mathsf{in},1}) \ \land \ (v_4{}^{\mathsf{in},1} \leqslant 0 \Longrightarrow v_4{}^{\mathsf{o}} = 0) \end{aligned}$$

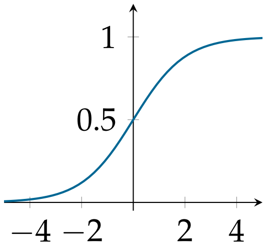


Figure 5.3 Sigmoid function

Next, we construct edge formulas:

$$\begin{aligned} F_{\circ \to v_3} &\triangleq (v_3{}^{\mathsf{in},1} = v_1{}^{\mathsf{o}}) \wedge (v_3{}^{\mathsf{in},2} = v_2{}^{\mathsf{o}}) \\ F_{\circ \to v_4} &\triangleq v_4{}^{\mathsf{in},1} = v_3{}^{\mathsf{o}} \end{aligned}$$

Finally, we conjoin all of the above formulas to arrive at the complete encoding of G:

$$F_G \triangleq \underbrace{F_{v_3} \wedge F_{v_4}}_{F_V} \wedge \underbrace{F_{\circ \to v_3} \wedge F_{\circ \to v_4}}_{F_E}$$

5.3 Handling Non-linear Activations

In the above, we have assumed that all of our nodes are associated with piecewise-linear functions, allowing us to precisely capture their semantics in linear real arithmetic. How can we handle non-piecewise-linear activations, like sigmoid and tanh? One way to encode them is by *overapproximating* their behavior, which gives us soundness but *not* completeness. As we will see, soundness means that our encoding can find proofs of correctness properties, and completeness means that our encoding can find counterexamples to correctness properties. So, by overapproximating an activation function, we give up on counterexamples.

Handling sigmoid

Let us begin with the concrete example of the sigmoid activation:

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

which is shown in Figure 5.3. The sigmoid function is (strictly) monotonically increasing, so if we have two points $a_1 < a_2$, we know that $\sigma(a_1) < \sigma(a_2)$. We can as a result overapproximate the behavior of σ by saying: for any input between a_1 and a_2, the output of the function can be *any* value between $\sigma(a_1)$ and $\sigma(a_2)$.

Consider Figure 5.4. Here we picked three points on the sigmoid curve, shown in red, with x coordinates -1, 0, and 1. The red rectangles define the lower and upper bound on the output of the sigmoid function between two values of x. For example, for inputs between 0 and 1, the output of the function is any value between 0.5 and 0.73. For inputs more than 1, we know that the output must be between 0.73 to 1 (the range of σ is upper bounded by 1).

Say some node v implements a sigmoid activation. Then, one possible encoding, following the approximation in Figure 5.4, is as follows:

$$\begin{aligned} F_v \triangleq\; & (v^{\mathsf{in},1} \leqslant -1 \implies 0 < v^{\mathsf{o}} \leqslant 0.26) \\ & \land (-1 < v^{\mathsf{in},1} \leqslant 0 \implies 0.26 < v^{\mathsf{o}} \leqslant 0.5) \\ & \land (0 < v^{\mathsf{in},1} \leqslant 1 \implies 0.5 < v^{\mathsf{o}} \leqslant 0.73) \\ & \land (v^{\mathsf{in},1} > 1 \implies 0.73 < v^{\mathsf{o}} < 1) \end{aligned}$$

Each conjunct specifies a range of inputs (left of implication) and the possible outputs in that range (right of implication). For example, the first conjunct specifies that, for inputs $\leqslant -1$, the output can be any value between 0 and 0.26.

Handling any monotonic function

We can generalize the above process to any monotonically (increasing or decreasing) function f_v.

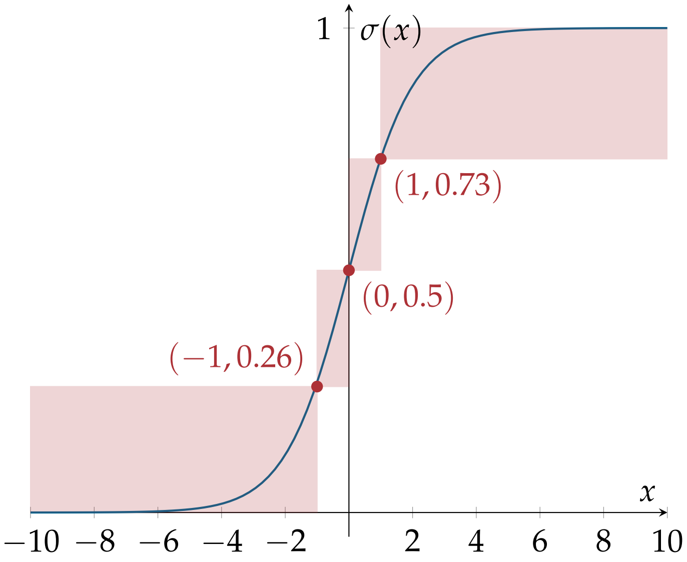


Figure 5.4 Sigmoid function with overapproximation

Let us assume that f_v is monotonically increasing. We can pick a sequence of real values $c_1 < \cdots < c_n$. Then, we can construct the following encoding:

$$\begin{aligned} F_v \triangleq\ & (v^{\mathsf{in},1} \leqslant c_1 \implies lb \leqslant v^{\mathsf{o}} \leqslant f_v(c_1)) \\ & \land (c_1 < v^{\mathsf{in},1} \leqslant c_2 \implies f_v(c_1) \leqslant v^{\mathsf{o}} \leqslant f_v(c_2)) \\ & \vdots \\ & \land (c_n < v^{\mathsf{in},1} \implies f_v(c_n) \leqslant v^{\mathsf{o}} \leqslant ub) \end{aligned}$$

where lb and ub are the lower and upper bounds of the range of f_v; for example, for sigmoid, they are 0 and 1, respectively. If a function is unbounded, then we can drop the constraints $lb \leqslant v^{\mathsf{o}}$ and $v^{\mathsf{o}} \leqslant ub$.

The more points c_i we choose and the closer they are to each other, the better our approximation is. This encoding is sound but *incomplete*.

5.4 Encoding Correctness Properties

Now that we have shown how to encode the semantics of neural networks as logical constraints, we are ready for the main dish: encoding entire correctness properties.

Checking robustness example

We begin with a concrete example before seeing the general form. Say we have a neural network G defining a binary classifier $f_G : \mathbb{R}^n \to \mathbb{R}^2$. f_G takes a grayscale image as a vector of reals, between 0 and 1, describing the intensity of each pixel (black to white), and predicts whether the image is of a *cat* or a *dog*. Say we have an image $\boldsymbol{c}$ that is correctly classified as *cat*. We want to prove that a small perturbation to the brightness of $\boldsymbol{c}$ does not change the prediction. We formalize this as follows:

$$\begin{gathered}\{\ |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\ \} \\ \boldsymbol{r} \leftarrow f_G(\boldsymbol{x}) \\ \{\ r_1 > r_2\ \}\end{gathered}$$

where the first output, r_1, is the probability of *cat*, while r_2 is the probability of *dog*.

The high-level intuition for the encoding of this correctness property follows how the property is written. The formula that we generate to check this statement, called the *verification condition* (VC), looks roughly like this:

$$(\text{precondition} \land \text{neural network}) \implies \text{postcondition}$$

If this formula is *valid*, then the correctness property holds.

Let us assume for our example that the input nodes of the neural network are $\{v_1, \ldots, v_2\}$ and the output nodes are $\{v_{n+1}, v_{n+2}\}$. Assume also that the formula F_G encodes the network, as described earlier in this chapter. We encode the correctness property as follows:

$$\underbrace{\left(\bigwedge_{i=1}^{n} |x_i - c_i| \leqslant 0.1\right)}_{\text{precondition}} \land \underbrace{F_G}_{\text{network}} \land \underbrace{\left(\bigwedge_{i=1}^{n} x_i = v_i{}^{\mathrm{o}}\right)}_{\text{network input}} \land \underbrace{(r_1 = v_{n+1}{}^{\mathrm{o}} \land r_2 = v_{n+2}{}^{\mathrm{o}})}_{\text{network output}}$$

$$\implies \underbrace{r_1 > r_2}_{\text{postcondition}}$$

Here's the breakdown:

- The precondition is directly translated to an LRA formula. Since LRA formulas don't support vector operations, we decompose the vector into its constituent scalars. Note that the absolute-value operation $|\cdot|$ is not present natively in LRA, but, fear not, it is actually encodable: A linear inequality with absolute value, like $|x| \leqslant 0$, can be written in LRA as $x \leqslant 0 \lor -x \leqslant 0$.

- The network is encoded as a formula F_G, just as we saw earlier in the chapter. The trick is that we now also need to connect the variables of F_G with the inputs $\boldsymbol{x}$ and output $\boldsymbol{r}$. This is captured by the two subformulas labeled "network input" and "network output".

- The postcondition is encoded as is.

Encoding correctness, formalized

A correctness property is of the form

$$\begin{gathered}\{\ P\ \}\\ \boldsymbol{r_1} \leftarrow f_{G_1}(\boldsymbol{x_1})\\ \boldsymbol{r_2} \leftarrow f_{G_2}(\boldsymbol{x_2})\\ \vdots\\ \boldsymbol{r_l} \leftarrow f_{G_l}(\boldsymbol{x_l})\\ \{\ Q\ \}\end{gathered}$$

Assume that the precondition and postcondition are encodable in LRA. We then encode the verification condition as follows:

$$\left(P \land \bigwedge_{i=1}^{l} F_i\right) \implies Q$$

where F_i is the encoding of the ith assignment $\boldsymbol{r_i} \leftarrow f_{G_i}(\boldsymbol{x_i})$. The assignment encoding F_i combines the encoding of the neural network F_{G_i} along with *connections* with inputs and outputs, $\boldsymbol{x_i}$ and $\boldsymbol{r_i}$, respectively:

$$F_i \triangleq F_{G_i} \wedge \left(\bigwedge_{j=1}^{n} x_{i,j} = v_i{}^{\circ} \right) \wedge \left(\bigwedge_{j=1}^{m} r_{i,j} = v_{n+j}{}^{\circ} \right)$$

Here the assumption is that the input and output variables of the encoding of G_i are $v_1, \ldots, v_n$ and $v_{n+1}, \ldots, v_{n+m}$, respectively.

Soundness and completeness

Say we have a correctness property that we have encoded as a formula F. Then, we have the following soundness guarantee:

Theorem 5.A If F is valid, then the correctness property is true. ■

Completeness depends on whether everything is encodable in LRA. Assuming it is, then, if F is invalid, we know that there is a model $I \models \neg F$. This model is a counterexample to the correctness property. From this model, we can read values for the input variables that result in outputs that do not satisfy the postcondition. This is best seen through an example:

Example 5.C Take the following simple correctness property, where $f(x) = x$:

$$\{ \ |x - 1| \leqslant 0.1 \ \}$$
$$r \leftarrow f(\boldsymbol{x})$$
$$\{ \ r \geqslant 1 \ \}$$

This property is not true. Let $x = 0.99$; this satisfies the precondition. But, $f(0.99) = 0.99$, which is less than 1. If we encode a formula F for this property, then we will have a model $I \models \neg F$ such that x is assigned 0.99. ■

Looking Ahead

Ahh, this chapter was tiring! Thanks for sticking around. We have taken neural networks, with all their glory, and translated them into formulas. In the coming chapters, we will study algorithms for checking satisfiability of these formulas.

Our encoding of sigmoid follow that of Ehlers (2017). A number of papers have considered MILP encodings that are similar to the ones we give (Tjeng et al., 2018). In MILP, we don't have disjunction, so we simulate disjunction with an integer that can take the values $\{0, 1\}$. The main issue with LRA and MILP encodings is disjunction; with no disjunction, the problem is polynomial-time solvable. Disjunctions mostly arise due to ReLUs. We say that a ReLU is active if it's output is > 0 and inactive otherwise. If a ReLU is active or inactive for all possible inputs to the neural network, as prescribed by the precondition, then we can get rid of the disjunction, and treat it as the function $f(x) = 0$ (inactive) or $f(x) = x$ (active). With this idea in mind, there are two tricks to simplify verification: (1) We can come up with lightweight techniques to discover which ReLUs are active or inactive. The abstraction-based verification techniques discussed in Part III of the book can be used. (2) Typically, when we train neural networks, we aim to maximize accuracy on some training data; we can additionally bias the training towards neural networks where most of the ReLUs are always active/inactive (Xiao et al., 2018).

In practice, neural networks are evaluate using finite precision, where real numbers are approximated as floating-point numbers, fixed-point numbers, or even machine integers. Some papers carefully ensure that verification results hold for a floating-point interpretation of the network (Katz et al., 2017). A recent paper has shown that verified neural networks in LRA may not really be robust when one considers the bit-level behavior (Jia and Rinard, 2020b). A number of papers have also considered bit-level verification of neural networks, using propositional logic instead of LRA (Jia and Rinard, 2020a; Narodytska et al., 2018).

Chapter 6

DPLL Modulo Theories

In the previous chapter, we saw how to reduce the verification problem to that of checking satisfiability of a logical formula. But how do we actually check satisfiability? In this chapter, we will meet the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, which was developed decades ago for checking satisfiability of Boolean formulas. Then we will see an extension of DPLL that can handle first-order formulas over theories.

These algorithms underlie modern SAT and SMT solvers. I will give a complete description of DPLL, in the sense that you can follow the chapter and implement a working algorithm. But note that there are numerous data structures, implementation tricks, and heuristics that make DPLL *really* work in practice, and we will not cover those here. (At the end of the chapter, I point you to additional resources.)

6.1 The DPLL Algorithm

The goal of DPLL is to take a Boolean formula F and decide whether it is SAT or UNSAT. In case F is SAT, DPLL should also return a model of F. We begin by talking about the *shape* of the formula DPLL expects as input.

Conjunctive Normal Form (CNF)

DPLL expects formulas to be in *conjunctive normal form* (CNF). Luckily, all Boolean formulas can be rewritten into equivalent formulas in CNF. (We will see how later in this chapter.) A formula F in CNF is of the following form:

$$C_1 \wedge \cdots \wedge C_n$$

where each subformula C_i is called a *clause* and is of the form:

$$\ell_1 \vee \cdots \vee \ell_{m_i}$$

where each ℓ_i is called a *literal* and is either a variable, e.g., p, or its negation, e.g., $\neg p$.

Example 6.A The following is a CNF formula with two clauses, each of which contains two literals:

$$(p \vee \neg r) \wedge (\neg p \vee q)$$

The following formula is *not* in CNF:

$$(p \wedge q) \vee (\neg r)$$

■

DPLL

The completely naïve way to decide satisfiability is by trying every possible interpretation and checking if it is a model of the formula. Of course, there are exponentially many interpretations in the number of variables. DPLL tries to avoid doing a completely blind search, but, on a bad day, it will just devolve into an exponential enumeration of all possible interpretations—after all, satisfiability is the canonical NP-complete problem.

DPLL alternates between two phases: *deduction* and *search*. Deduction tries to simplify the formula using the laws of logic. Search simply searches for an interpretation.

Deduction The deduction part of DPLL is called Boolean constant propagation (BCP). Imagine that you have the following formula in CNF:

$$(\ell) \land C_2 \land \cdots C_n$$

Notice that the first clause consists of a single literal—we call it a *unit clause*. Clearly, any model of this formula must assign ℓ the value true: Specifically, if ℓ is a variable p, then p must be assigned true; if ℓ is a negation $\neg p$, then p must be assigned false.

The BCP phase of DPLL will simply look for all unit clauses and replace their literals with true. BCP is enabled by the fact that formulas are in CNF, and it can be quite effective at proving SAT or UNSAT.

Example 6.B Consider the following formula

$$F \triangleq (p) \land (\neg p \lor r) \land (\neg r \lor q)$$

BCP first finds the unit clause (p) and assigns p the value true. This results in the following formula:

$$\begin{aligned} & (\mathsf{true}) \land (\neg \mathsf{true} \lor r) \land (\neg r \lor q) \\ \equiv\ & (r) \land (\neg r \lor q) \end{aligned}$$

Clearly BCP's job is not done: the simplification has produced another unit clause, (r). BCP sets r to true, resulting in the following formula:

$$\begin{aligned} & (\mathsf{true}) \land (\neg \mathsf{true} \lor q) \\ \equiv\ & (q) \end{aligned}$$

Finally, we are left with a single clause, the unit clause (q). Therefore BCP assigns q the value true, resulting in the final formula true. This means that F is SAT, and $\{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}, r \mapsto \mathsf{true}\}$ is a model. ■

In the above example, BCP managed to show satisfiability of F. Note that BCP can also prove unsatisfiability, by simplifying the formula to false. But BCP might get stuck if it cannot find unit clauses. This is where search is employed by DPLL.

Algorithm 1: DPLL

```
Data: A formula $F$ in CNF form
Result: $I \models F$ or UNSAT

▷ Boolean constant propagation (BCP)
while there is a unit clause $(\ell)$ in $F$ do
    Let $F$ be $F[\ell \mapsto \text{true}]$

if $F$ is true then return SAT

▷ Search
for every variable $p$ in $F$ do
    if dpll($F[p \mapsto \text{true}]$) is sat then return SAT
    if dpll($F[p \mapsto \text{false}]$) is sat then return SAT
return UNSAT
```

Deduction + search Algorithm 1 shows the entire DPLL algorithm. The algorithm takes a formula F in CNF.

The first part performs BCP: the algorithm keeps simplifying the formula until no more unit clauses exist. We use the notation $F[\ell \mapsto \text{true}]$ to mean replace all occurrences of ℓ in F with true and simplify the resulting formula. Specifically, if ℓ is a variable p, then all occurrences of p are replaced with true; if ℓ is a negation $\neg p$, then all occurrences of p are replaced with false.

After BCP is done, the algorithm checks if the formula is true, which means that BCP has proven that the formula is SAT.

If BCP is unsuccessful in proving SAT, then DPLL moves to the search phase: it iteratively chooses variables and tries to replace them with true or false, calling DPLL recursively on the resulting formula. The order in which variables are chosen in search is critical to DPLL's performance. There is a lot of research on variable selection. One of the popular heuristics maintains a continuously updated score sheet, where variables with higher scores are chosen first.

Algorithm 1, as presented, returns SAT when the input formula is satisfiable, but does not return a model. The model is implicit in the sequence of assignments to variables (of the form $[l \mapsto \cdot]$ and $[p \mapsto \cdot]$) made by BCP and search that led to SAT being returned. The algorithm returns UNSAT when it has exhausted all possible satisfying assignments.

Example 6.C Consider the following formula given to DPLL

$$F \triangleq (p \vee r) \wedge (\neg p \vee q) \wedge (\neg q \vee \neg r)$$

First level of recursion DPLL begins by attempting BCP, which cannot find any unit clauses. Then, it proceeds to search. Suppose search chooses variable p, setting it to true by invoking DPLL on

$$F_1 = F[p \mapsto \mathsf{true}] = q \wedge (\neg q \vee \neg r)$$

Second level of recursion Next, DPLL attempts BCP on F_1. First, it sets q to true, resulting in the formula

$$F_2 = F_1[q \mapsto true] = (\neg r)$$

Then, it sets r to false, resulting in

$$F_3 = F_2[r \mapsto \mathsf{false}] = \mathsf{true}$$

Since F_3 is true, DPLL returns SAT. Implicitly, DPLL has built up a model of F:

$$\{q \mapsto \mathsf{true}, q \mapsto \mathsf{true}, r \mapsto \mathsf{false}\}$$

■

Partial Models Note that DPLL may terminate with SAT but without assigning every variable in the formula. We call the resulting model a *partial models*. You can take a partial model and extend by assigning the remaining variables in any way you like, and you'll still have a model of the formula.

Example 6.D Consider this simple formula:

$$F \triangleq p \wedge (q \vee p \vee \neg r) \wedge (p \vee \neg q)$$

The first DPLL will do is apply BCP, which will set the unit clause p to true. The rest of the formula then simplifies to true. This means that q and r are useless variables—give them any interpretation and you'll end up with a model, as long as p is assigned true. Therefore, we call $I = \{p \to \text{true}\}$ a partial model of F. Formally, $\text{eval}(I(F)) = \text{true}$. ■

6.2 DPLL Modulo Theories

We have seen how DPLL can decide satisfiability of Boolean formulas. We now present dpll *modulo theories*, or DPLL^T, an extension of DPLL that can handle formulas over, for example, arithmetic theories like LRA. The key idea of DPLL^T is to start by treating a formula as if it is completely Boolean, and then incrementally add more and more *theory* information until we can conclusively say that a formula is SAT or UNSAT. We begin by defining the notion of Boolean abstraction of a formula.

Boolean Abstraction

For illustration, we assume we are dealing with formulas in LRA, as with the previous chapters. Say we have the following formula in LRA:

$$F \triangleq (x \leqslant 0 \lor x \leqslant 10) \land (\neg x \leqslant 0)$$

The Boolean abstraction of F, denoted F_B, is the formula where every unique linear inequality in the formula is replaced with a special Boolean variable, as follows:

$$F^B \triangleq (p \lor q) \land (\neg p)$$

The inequality $x \leqslant 0$ is *abstracted* as p and $x \leqslant 10$ is abstracted as q. Note that both occurrences of $x \leqslant 0$ are replaced by the same Boolean variable, though this need not be the case for the correctness of our exposition. We will also use the superscript T to map Boolean formulas back to theory formulas, e.g., $(F^B)^T$ is F.

We call this process *abstraction* because constraints are lost in the process; namely, the relation between different inequalities is obliterated. Formally speaking, if F^B

Algorithm 2: DPLL^T

```
Data: A formula $F$ in CNF form over theory $T$
Result: $I \models F$ or UNSAT

Let $F^B$ be the abstraction of $F$
while true do
    if dpll($F^B$) is unsat then
        return UNSAT
    else
        Let $I$ be the model returned by DPLL
        Assume $I$ is represented as a formula
        if $T(I^T)$ is sat then
            return SAT and the model returned by $T$
        else
            Let $F^B$ be $F^B \wedge \neg I$
```

is UNSAT, then F is UNSAT. But the converse does not hold: if F^B is SAT, it does not mean that F is SAT.

Example 6.E Consider the formula $F \triangleq x \leqslant 0 \wedge x \geqslant 10$. This formula is clearly UNSAT. However, its abstraction, $p \wedge q$, is SAT. ■

Lazy DPLL Modulo Theories

The DPLL^T algorithm takes a formula F, over some theory like LRA, and decides satisfiability. DPLL^T assumes access to a *theory solver*. The theory solver takes a conjunction of, for example, linear inequalities, and checks their satisfiability. In a sense, the theory solver takes care of conjunctions and the DPLL algorithm takes care of disjunctions. In the case of LRA, the theory solver can be the Simplex algorithm, which will see in the next chapter.

DPLL^T, shown in Algorithm 2, works as follows: First, using vanilla DPLL, it checks if the abstraction F^B is UNSAT, in which case it can declare that F is UNSAT, following the properties of abstraction discussed above. The tricky part comes when dealing with the case where F^B is SAT, because that does not mean that F is

SAT. This is where the theory solver comes into play. We take the model I returned by DPLL and map it to a formula I^T in the theory, a conjunction of atoms. If the theory solver deems I^L satisfiable, then we know that F is satisfiable and we're done. Otherwise, DPLL^T *learns* the fact that I is not a model. So it negates I and conjoins it to F^B. In a sense, the DPLL^T *lazily* learns more and more facts, refining the abstraction, until it can decide SAT or UNSAT.

Example 6.F Consider the following LRA formula F:

$$x \geqslant 10 \wedge (x < 0 \vee y \geqslant 0)$$

and its abstraction F^B:

$$p \wedge (q \vee r)$$

where p denotes $x \geqslant 10$, q denotes $x < 0$, and r denotes $y \geqslant 0$.

First iteration DPLL^T begins by invoking DPLL on F^B. Suppose DPLL returns the partial model

$$I_1 = \{p \mapsto \mathsf{true}, q \mapsto \mathsf{true}\}$$

We will represent I_1 as a formula

$$p \wedge q$$

Next, we check if I_1 is indeed a model of F. We do so by invoking the theory solver on I_1^T, which is

$$\underbrace{x \geqslant 10}_{p} \wedge \underbrace{x < 0}_{q}$$

The theory solver will say that I_1^T is UNSAT, because x cannot be $\geqslant 10$ and < 0. Therefore, DPLL^T blocks this model by conjoining $\neg I_1$ to F^B. This makes F^B the following formula, which is still in CNF, because $\neg I_1$ is a clause:

$$p \wedge (q \vee r) \wedge \underbrace{(\neg p \vee \neg q)}_{\neg I_1}$$

In other words, we're saying that we cannot have a model that sets both $x \geqslant 10$ and $x < 0$ to true.

Second iteration In the second iteration, DPLL^T invokes DPLL on the updated F^B. DPLL cannot give us the same model I_1. So it gives us another one, say $I_2 = p \land \neg q \land r$. The theory solver checks I_2^T, which is satisfiable, and returns its own theory-specific model, e.g., $\{x \mapsto 10, y \mapsto y \mapsto 0\}$. We are now done, and we return the model.

■

6.3 Tseitin's Transformation

We've so far assumed that formulas are in CNF. Indeed, we can take any formula and turn it into an equivalent formula in CNF. We can do this by applying DeMorgan's laws (see Chapter 4), by distributing disjunction over conjunction. For example, $r \lor (p \land q)$ can be rewritten into the equivalent $(r \lor p) \land (r \lor q)$. This transformation, unfortunately, can lead to an exponential explosion in the size of the formula. It turns out that there's a simple technique, known as *Tseitin's transformation*, that produces a formula of size linear in the size of the non-CNF formula.

Tseitin's transformation takes a formula F produces a CNF formula F'. The set of variables of F is a subset of the variables in F'; i.e., Tseitin's transformation creates new variables. Tseitin's transformation guarantees the following properties:

1. Any model of F' is also a model of F, if we disregard the interpretations of newly added variables.

2. If F' is UNSAT, then F is UNSAT.

Therefore, given a non-CNF formula F, to check its satisfiability, we can simply invoke DPLL on F'.

Intuition

Tseitin's transformation is pretty much the same as rewriting a complex arithmetic expression in a program into a sequence of instructions where every instruction is an application of a single unary or binary operator. (This is pretty much what a compiler does when compiling a high-level program to assembly or an intermediate representation.) For example, consider the following function (in Python syntax):

```
def f(x,y,z):
    return x + (2*y + 3)
```

The returned expression can be rewritten into a sequence of operations, each operating on 1 or 2 variables, as follows, where `t1`, `t2`, and `t3` are temporary variables:

```
def f(x,y,z):
    t1 = 2 * y
    t2 = t1 + 3
    t3 = x + t2
    return t3
```

Intuitively, each subexpression is computed and stored in a temporary variable: `t1` contains the expression `2*y`, `t2` contains the expression `2*y + 3`, and `t3` contains the entire expression `x + (2*y + 3)`.

Transformation steps

We define a subformula of F to be any subformula that contains a disjunction or conjunction—i.e., we don't consider subformulas at the level of literals.

Example 6.G The following formula F is decomposed into 4 subformulas:

$$F \triangleq \underbrace{\underbrace{(p \land q)}_{F_1} \lor \underbrace{(\underbrace{q \land \neg r}_{F_2} \land s)}_{F_3}}_{F_4}$$

F_1 and F_2 are in the deepest *level of nesting*, while F_4 is in the shallowest level. Notice that F_2 is a subformula of F_3, and all of F_i are subformulas of F_4. ■

Now for the transformation steps, we assume that F has n subformulas:

1. For every subformula F_i of F, create a fresh variable t_i. *These variables are analogous to the temporary variables* `t`$_i$ *we introduced to our Python program above.*

2. Next, for every subformula F_i, starting with those most-deeply nested, create the following formula: Let F_i be of the form $\ell_i \circ \ell'_i$, where $\circ$ is $\wedge$ or $\vee$. Note that one or both literals ℓ_i and ℓ'_i may be the new variable t_j denoting a subformula F_j of F_i. Create formula

   $$F'_i \triangleq t_i \Leftrightarrow (\ell_i \circ \ell'_i)$$

   These formulas are analogous to the assignments to the temporary variables in our Python program above, where $\Leftrightarrow$ is the logical analogue of variable assignment (`=`*).*

Example 6.H Continuing our running example, for subformula F_1, we create the formula

$$F'_1 \triangleq t_1 \Leftrightarrow (p \wedge q)$$

For subformula F_2, we create

$$F'_2 \triangleq t_2 \Leftrightarrow (q \wedge \neg r)$$

For subformula F_3, we create

$$F'_3 \triangleq t_2 \Leftrightarrow (t_2 \wedge s)$$

(Notice that $q \wedge \neg r$ are replaced by the variable t_2.) Finally, for subformula F_4, we create

$$F'_4 \triangleq t_4 \Leftrightarrow (t_1 \vee t_3)$$

■

Notice that each F'_i can be written in CNF. This is because, following the definition of $\Leftrightarrow$ and DeMorgan's laws, we have:

$$\ell_1 \Leftrightarrow (\ell_2 \vee \ell_3) \equiv (\neg\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_1 \vee \neg\ell_2) \wedge (\ell_1 \neg \vee \ell_3)$$

and

$$\ell_1 \Leftrightarrow (\ell_2 \wedge \ell_3) \equiv (\neg\ell_1 \vee \ell_2) \wedge (\neg\ell_1 \vee \ell_3) \wedge (\ell_1 \vee \neg\ell_3 \vee \neg\ell_3)$$

Finally, we construct the following CNF formula:

$$F' \triangleq t_n \wedge \bigwedge_i F'_i$$

By construction, in any model of F', each t_i is assigned true iff subformula F_i evaluates to true. Therefore, the t_n constraint says F must be true. *You can think of t_n as the return statement in our transformed Python program.*

Looking Ahead

I gave a trimmed down version of DPLL^T. A key idea in modern SAT and SMT solvers is *conflict-driven clause learning*, a graph data structure that helps us cut the search space by identifying sets of interpretations that do not make satisfying assignments. I encourage the interested reader to consult Biere et al. (2009) for a detailed exposition of clause learning and other ideas.

I also encourage to play with popular SAT and SMT solvers. For example, MiniSAT Sorensson and Een (2005), as the name suggests, has a small and readable codebase. For SMT solvers, I recommend checking out Z3 (De Moura and Bjørner, 2008) and CVC4 (Barrett et al., 2011). One of the interesting ideas underlying SMT solvers is *theory combination* Nelson and Oppen (1979), where you can solve formulas using different theories. This is useful when doing general program verification, where programs may manipulate strings, arrays, integers, etc. In the future, I strongly suspect theory combination will be needed for neural network verification, because we will start looking at neural networks as components of bigger pieces of software.

Chapter 7

Neural Theory Solvers

In the previous chapter, we discussed DPLL^T for solving formulas in FOL. In this chapter, we study the *Simplex algorithm* for solving conjunctions of literals in linear real arithmetic. Then, we extend the solver to natively handle rectified linear units (ReLUs), which would normally be encoded as disjunctions and thus dealt with using the SAT solver.

7.1 Theory Solving and Normal Forms

The Problem

The theory solver for LRA receives a formula F as a conjunction of linear inequalities:

$$\bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i \right)$$

where $c_{ij}, b_i \in \mathbb{R}$. The goal is to check satisfiability of F, and, if satisfiable, discover a model $I \models F$.

Notice that our formulas do not have strict inequalities ($<$). The approach we present here can be easily generalized to handle strict inequalities, but for simplicity, we stick with inequalities.[1]

[1] See Dutertre and De Moura (2006) for how to handle strict inequalities. In most instances of verifying properties of neural networks, we do not need strict inequalities to encode network semantics or properties.

Simplex Form

The Simplex algorithm, discussed in the next section, expects formulas to be in a certain form (just like how DPLL expected propositional formulas to be in CNF). Specifically, Simplex expects formulas to be conjunctions of *equalities* of the form

$$\sum_{i=1} c_i \cdot x_i = 0$$

and *bounds* of the form

$$l_i \leqslant x_i \leqslant u_i$$

where $u_i, l_i \in \mathbb{R} \cup \{\infty, -\infty\}$.

Therefore, given a formula F, we need to translate it into an equivalent formula of the above form, which we will call the *Simplex form*. It turns out that translating a formula into Simplex form is pretty simple. Say

$$F \triangleq \bigwedge_{i=1}^{n} \left(\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i \right)$$

Then, we take every inequality and translate it into two conjuncts, an equality and a bound. From the ith inequality,

$$\sum_{j=1}^{m} c_{ij} \cdot x_j \leqslant b_i$$

we generate the equality

$$s_i = \sum_{j=1}^{m} c_{ij} \cdot x_j$$

and the bound

$$s_i \leqslant b_i$$

where s_i is a new variable, called a *slack variable*. Slack variables are analogous to the temporary variables in Tseitin's transformation (Chapter 6).

Example 7.A Consider the formula F, which we will use as our running example:

$$\begin{aligned} x + y &\geqslant 0 \\ -2x + y &\geqslant 2 \\ -10x + y &\geqslant -5 \end{aligned}$$

For clarity, we will drop the conjunction operator and simply list the inequalities. We convert F into a formula F_s in Simplex form:

$$\begin{aligned}
s_1 &= x + y \\
s_2 &= -2x + y \\
s_3 &= -10x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant -5
\end{aligned}$$

■

This transformation is a simple rewriting of the original formula that maintains satisfiability. Let F_s be the Simplex form of some formula F. Then, we have the following guarantees (again, the analogue of Tseitin's transformation for non-CNF formulas):

1. Any model of F_s is a model of F, disregarding assignments to slack variables.
2. If F_s is UNSAT, then F is UNSAT.

7.2 The Simplex Algorithm

We are now ready to present the Simplex algorithm. This is a very old idea, due to George Dantizg, who developed it in 1947 (Dantzig, 1990). The goal of the algorithm is to find a satisfying assignment that maximizes some objective function. Our interest in verification is typically to find *any* satisfying assignment, and so the algorithm we will present is a subset of Simplex.

Intuition

One can think of the Simplex algorithm as a procedure that simultaneously looks for a model and a proof of unsatisfiability. It starts with some interpretation and continues to update it in every iteration, until it finds a model or discovers a proof of unsatisfiability. We start from the interpretation I that sets all variables to 0. This assignment satisfies all the equalities, but may not satisfy the bounds. In every iteration of Simplex, we pick a bound that is not satisfied, and we modify I to satisfy it, or we discover that the formula is unsatisfiable. Let us see this process pictorially on a satisfiable example before we dig into the math.

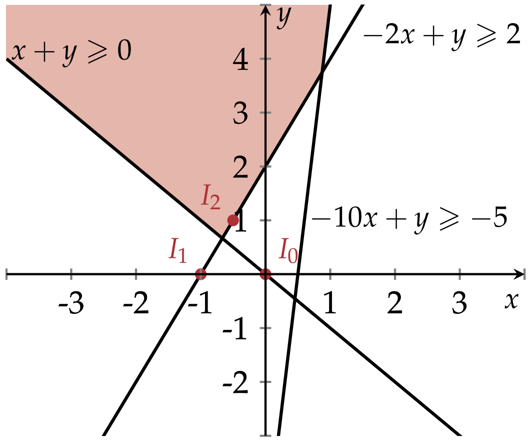


Figure 7.1 Simplex example

Example 7.B Recall the formula F from our running example, illustrated in Figure 7.1, where the satisfying assignments are shaded.

$$\begin{aligned} x + y &\geqslant 0 \\ -2x + y &\geqslant 2 \\ -10x + y &\geqslant -5 \end{aligned}$$

Simplex begins with the initial interpretation

$$I_0 = \{x \mapsto 0, y \mapsto 0\}$$

shown in Figure 7.1, which is not a model of the formula.

Simplex notices that $I_0 \not\models -2x + y \geqslant 2$, and so it decreases the interpretation of x to -1, resulting in I_1. Then, it notices that $I_1 \not\models x + y \geqslant 0$, and so it increases the interpretation of y to 1, resulting in the satisfying assignment I_2. (Notice that

x also changes in I_2; we will see why shortly.) In a sense, Simplex plays Whac-A-Mole, trying to satisfy one inequality only to break another, until it arrives at an assignment that satisfies all inequalities. Luckily, the algorithm actually terminates. ■

Basic and non-basic variables

Recall that Simplex expects an input formula to be in Simplex form. The set of variables in the formula are broken into two subsets:

Basic variables are those that appear on the left hand side of an equality; initially, these are the slack variables.

Non-basic variables are all other variables.

As Simplex progresses, it will rewrite the formula, thus some basic variables will become non-basic and vice versa.

Example 7.C In our running example, initially the set of basic variables is $\{s_1, s_2, s_3\}$ and non-basic variables is $\{x, y\}$. ■

To ensure termination of Simplex, we will fix a total ordering on the set of all (basic and non-basic) variables. So, when we say *"the first variable that..."*, we are referring to the first variable per our ordering. To easily refer to variables, we will assume they are of the form $x_1, \ldots, x_n$. Given a basic variable x_i and a non-basic variable x_j, we will use c_{ij} to denote the coefficient of x_j in the equality

$$x_i = \ldots + c_{ij} \cdot x_j + \ldots$$

For a variable x_i, we will use l_i and u_i to denote its lower bound and upper bound, respectively. If a variable does not have an upper bound (resp. lower bound), its upper bound is be ∞ (resp. $-\infty$). Note that non-slack variables have no bounds.

Simplex in detail

We are now equipped to present the Simplex algorithm, shown in Algorithm 3. The algorithm maintains the following two invariants:

1. The interpretation I always satisfies the equalities, so only the bounds may be violated. This is initially true, as I assigns all variables to 0.

2. The bounds of non-basic variables are all satisfied. This is initially true, as non-basic variables have no bounds.

In every iteration of the while loop, the algorithm looks for a basic variable whose bounds are not satisfied, and attempts to fix it. There two symmetric cases, encoded as two branches of the *if* statement, $x_i < l_i$ or $x_i > u_i$. Let us consider the first case. Since x_i is below l_i, we need to increase its assignment in I. We do this indirectly, by modifying the assignment of a non-basic variable x_j. But which x_j should we pick? In principle, we can pick any x_j such that the coefficient $c_{ij} \neq 0$, and adjust the interpretation of x_j accordingly. If you look at the algorithm, there are a few extra conditions. If we cannot find an x_j that satisfies these conditions, then the problem is UNSAT. We will discuss the unsatisfiability conditions shortly. For now, assume we have found an x_j. We can increase its current interpretation by $\frac{l_i - I(x_i)}{c_{ij}}$; this makes the interpretation of x_i increase by $l_i - I(x_i)$, thus barely satisfying the lower bound $I(x_i) = l_i$. Note that the assignments of basic variables are assumed to change automatically when we change the interpretation of non-basic variables. This maintains the first invariant of the algorithm.[2]

After we have updated the interpretation of x_j, there is a chance that we have violated one of the bounds of x_j. Therefore, we rewrite the formulas such that x_j becomes a basic variable and x_i a non-basic variable. This is known as the *pivot* operation, and it is mechanically done as follows: Take the following equality, where N is the set of indices of non-basic variables:

$$x_i = \sum_{k \in N} c_{ik} x_k$$

[2] Basic variables are sometimes called *dependent* variables and non-basic variables *independent* variables, indicating that the assignments of basic variables depend on those of non-basic variables.

Algorithm 3: Simplex

```
Data: A formula $F$ in Simplex form
Result: $I \models F$ or UNSAT

Let $I$ be the interpretation that sets all variables $fv(F)$ to 0
while true do
    if $I \models F$ then return $I$
    Let $x_i$ be the first basic variable s.t. $I(x_i) < l_i$ or $I(x_i) > u_i$
    if $I(x_i) < l_i$ then
        Let $x_j$ be the first non-basic variable s.t.
            $(I(x_j) < u_j$ and $c_{ij} > 0)$ or $(I(x_j) > l_j$ and $c_{ij} < 0)$
        if If no such $x_j$ exists then return UNSAT
        $I(x_j) \leftarrow I(x_j) + \frac{l_i - I(x_i)}{c_{ij}}$
    else
        Let $x_j$ be the first non-basic variable s.t.
            $(I(x_j) > l_j$ and $c_{ij} > 0)$ or $(I(x_j) < u_j$ and $c_{ij} < 0)$
        if If no such $x_j$ exists then return UNSAT
        $I(x_j) \leftarrow I(x_j) + \frac{u_i - I(x_i)}{c_{ij}}$
    Pivot $x_i$ and $x_j$
```

and rewrite it by moving x_j to the left-hand side:

$$x_j = \underbrace{-\frac{x_i}{c_{ij}} + \sum_{k \in N \setminus \{j\}} \frac{c_{ik}}{c_{ij}} x_k}_{\text{replace } x_j \text{ with this}}$$

Now, replace x_j in all other equalities with the expression above. This operation results in a set of equalities where x_j only appears once, on the left-hand side. And so after pivoting it x_j becomes a basic variable and x_i a non-basic one.

Example 7.D Let us now work through our running example in detail. Recall, our formula is:

$$\begin{aligned}
s_1 &= x + y \\
s_2 &= -2x + y \\
s_3 &= -10x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant -5
\end{aligned}$$

Say the variables are ordered as follows:

$$x, y, s_1, s_2, s_3$$

Initially, the bounds of s_1 and s_3 are satisfied, but s_2 is violated, because $s_2 \geqslant 2$ but $I_0(s_2) = 0$, as all variables are assigned 0.

First iteration In the first iteration, we pick the variable x to fix the bounds of s_2, as it is the first one in our ordering. Note that x is unbounded (i.e., its bounds are $-\infty$ and ∞); so it easily satisfies the conditions. To increase the interpretation of s_2 to 2, and satisfy its lower bound, we can decrease $I_0(x)$ to -1, resulting in the following satisfying assignment:

$$I_1 = \{x \mapsto -1, y \mapsto 0, s_1 \mapsto -1, s_2 \mapsto 2, s_3 \mapsto 10\}$$

We now pivot s_2 and x, producing the following set of equalities (the bounds always remain the same):

$$\begin{aligned}
x &= 0.5y - 0.5s_2 \\
s_1 &= 1.5y - 0.5s_2 \\
s_3 &= -4y + 5s_2
\end{aligned}$$

Second iteration The only basic variable not satisfying its bounds is now s_1, since $I_1(s_1) = -1 < 0$. The first non-basic variable that we can tweak is y. We can increase the value of $I(y)$ by 1, resulting in the following interpretation:

$$I_2 = \{x \mapsto -0.5, y \mapsto 1, s_1 \mapsto 0.5, s_2 \mapsto 2, s_3 \mapsto 6\}$$

At this point, we pivot y with s_1.

Third iteration Simplex terminates since $I_2 \models F$.

■

Why is Simplex Correct?

First, you may wonder, why does Simplex terminate? The answer is due to the fact that we order variables and always look for the *first* variable violating bounds. This is known as *Bland's rule* (Bland, 1977). It ensures that we never revisit the same set of basic and non-basic variables.

Second, you may wonder, is Simplex actually correct? If Simplex returns an interpretation I, it is easy to see that $I \models F$, since Simplex checks that condition before it terminates. But what about the case when it says UNSAT? To illustrate correctness in this setting, we will look at an example.

Example 7.E Consider the following formula in Simplex form:

$$\begin{aligned}
s_1 &= x + y \\
s_2 &= -x - 2y \\
s_3 &= -x + y \\
s_1 &\geqslant 0 \\
s_2 &\geqslant 2 \\
s_3 &\geqslant 1
\end{aligned}$$

This formula is UNSAT—use your favorite SMT solver to check this. Imagine an execution of Simplex that performs the following two pivot operations: (1) s_1 with x, and (2) s_2 with y.

The first pivot results in the following formula:

$$\begin{aligned}
x &= s_1 - y \\
s_2 &= -s_1 - y \\
s_3 &= -s_1 + 2y
\end{aligned}$$

The second pivot results in the following formula:

$$\begin{aligned} x &= 2s_1 + s_2 \\ y &= -s_2 - s_1 \\ s_3 &= -3s_1 - 2s_s \end{aligned}$$

The algorithm maintains the invariant that all non-basic variables satisfy their bounds. So we have $s_1, s_2 \geqslant 0$. Say s_3 violates its bound, i.e.,

$$-3s_1 - 2s_2 < 1$$

The only way to fix this is by decreasing the interpretations of s_2 and s_3. But even if we assign s_2 and s_3 the value 0, we cannot make $s_3 \geqslant 1$. So Simplex figures out the formula is UNSAT. The conditions for choosing variable x_j in Algorithm 3 encode this argument. ■

7.3 The Reluplex Algorithm

Using the Simplex algorithm as the theory solver within DPLL^T allows us to solve formulas in LRA. So, at this point in our development, we know how to algorithmically reason about neural networks with piecewise-linear activations, like ReLUs. Unfortunately, this approach has been shown to not scale to large networks. One of the reasons is that ReLUs are encoded as disjunctions, as we saw in Chapter 5. This means that the SAT-solving part of DPLL^T will handle the disjunctions, and may end up considering every possible case of the disjunction—ReLU being active (output = input) or inactive (output = 0)—leading to many calls to Simplex, exponential in the number of ReLUs.

To fix those issues, the work of Katz et al. (2017) developed an extension of Simplex, called *Reluplex*, that natively handles ReLU constraints in addition to linear inequalities. The key idea is to try to *delay* case splitting on ReLUs. In the worst case, Reluplex may end up with an exponential explosion, just like DPLL^T with Simplex, but empirically it has been shown to be a promising approach for scaling SMT solving to larger neural networks. In what follows, we present the Reluplex algorithm.

Reluplex form

Just like with Simplex, Reluplex expects formulas to be in a certain form. We will call this form *Reluplex form*, where formulas contain (1) equalities (same as Simplex), (2) bounds (same as Simplex), and (3) *ReLU constraints* of the form

$$x_i = \text{relu}(x_j)$$

Given a conjunction of inequalities and ReLU constraints, we can translate them into Reluplex form by translating the inequalities into Simplex form. Additionally, for each ReLU constraint $x_i = \text{relu}(x_j)$, we can add the bound $x_i \geqslant 0$, which is implied by the definition of a ReLU.

Example 7.F Consider the following formula:

$$\begin{aligned} x + y &\geqslant 2 \\ y &= \text{relu}(x) \end{aligned}$$

We translate it into the following Reluplex form:

$$\begin{aligned} s_1 &= x + y \\ y &= \text{relu}(x) \\ s_1 &\geqslant 2 \\ y &\geqslant 0 \end{aligned}$$

■

Reluplex in Detail

We now present the Reluplex algorithm. The original presentation by Katz et al. (2017) is a set of rules that can be applied non-deterministically to arrive at an answer. Here, we present a specific schedule of the Reluplex algorithm.

The key idea of Reluplex is to call Simplex on equalities and bounds, and then try to massage the interpretation returned by Simplex to satisfy all ReLU constraints. Reluplex is shown in Algorithm 4.

Algorithm 4: Reluplex

```
Data: A formula $F$ in Reluplex form
Result: $I \models F$ or UNSAT

Let $I$ be the interpretation that sets all variables $fv(F)$ to 0
Let $F'$ be the non-ReLU constraints of $F$
while true do
    ▷ Calling Simplex (note that we supply Simplex with a reference to the initial interpretation and it can modify it)
    $r \leftarrow \text{Simplex}(F', I)$
    if $r$ is unsat then
        return UNSAT
    else
        if $r$ is an interpretation and $r \models F$ then return $r$

    ▷ Handle violated ReLU constraint
    Let ReLU constraint $x_i = \text{relu}(x_j)$ be s.t. $I(x_i) \neq \text{relu}(I(x_j))$
    if $x_i$ is basic then
        pivot $x_i$ with non-basic variable $x_k$, where $k \neq j$ and $c_{ik} \neq 0$
    if $x_j$ is basic then
        pivot $x_j$ with non-basic variable $x_k$, where $k \neq i$ and $c_{jk} \neq 0$
    Perform one of the following operations:
        $I(x_i) \leftarrow \text{relu}(I(x_j))$   or   $I(x_j) \leftarrow I(x_i)$

    ▷ Case splitting (ensures termination)
    if $u_j > 0$, $l_j < 0$, and $x_i = relu(x_j)$ considered more than $\tau$ times then
        $r_1 \leftarrow \text{Reluplex}(F \wedge x_j \geqslant 0 \wedge x_i = x_j)$
        $r_2 \leftarrow \text{Reluplex}(F \wedge x_j \leqslant 0 \wedge x_i = 0)$
        if $r_1 = r_2 =$ unsat then return UNSAT
        if $r_1 \neq$ unsat then return $r_1$
        return $r_2$
```

Initially, Simplex is invoked on the formula F', which is the original formula F but without the ReLU constraints. If Simplex returns UNSAT, then we know that F is UNSAT, this is because $F \Rightarrow F'$ is valid. Otherwise, if Simplex returns a model $I \models F'$, it may not be the case that $I \models F$, since F' is a weaker (less constrained) formula.

If $I \not\models F$, then we know that one of the ReLU constraints is not satisfied. We pick one of the violated ReLU constraints $x_i = \text{relu}(x_j)$ and modify I to make sure it is not violated. Note that if any of x_i and x_j is a basic variable, we pivot it with a non-basic variables. This is because we want to modify the interpretation of one of x_i or x_j, which may affect the interpretation of the other variable if it is a basic variable and $c_{ij} \neq 0$.[3] Finally, we modify the interpretation of x_i or x_j, ensuring that $I \models x_i = \text{relu}(x_j)$. Note that the choice of x_i or x_j is up to the implementation.

The problem is that fixing a ReLU constraint may end up violating a bound, and so Simplex need be invoked again. We assume that the interpretation I in Reluplex is the same one that is modified by invocations of Simplex.

Case splitting

Note that if we simply apply Reluplex without the last piece of the algorithm—case splitting—it may not terminate. Specifically, it may get into a loop where Simplex satisfies all bounds but violates a ReLU, and then satisfying the ReLU causes a bound to be violated, and on and on.

The last piece of Reluplex checks if we are getting into an infinite loop, by ensuring we do not attempt to fix a ReLU constraint more than τ times, some fixed threshold. If this threshold is exceeded, then the ReLU constraint $x_i = \text{relu}(x_j)$ is split into its two cases:

$$F_1 \triangleq x_j \geqslant 0 \wedge x_i = x_j$$

and

$$F_2 \triangleq x_j \leqslant 0 \wedge x_i = 0$$

and Reluplex is invoked recursively on two instances of the problem, $F \wedge F_1$ and $F \wedge F_2$. If both instances are UNSAT, then the F is UNSAT. If any of the instances is SAT, then F is SAT. This is due to the fact that

$$F \equiv (F \wedge F_1) \vee (F \wedge F_2)$$

[3] These conditions are not explicit in Katz et al. (2017), but their absence may lead to wasted iterations (or Update rules in Katz et al. (2017)) that do not fix violations of ReLU constraints.

Looking Ahead

We are done with constraint-based verification. In the next part of the book, we will look at different approaches that are more efficient at the expense of failing to provide proofs in some cases.

There are a number of interesting problems that we didn't cover. A critical one is soundness with respect to machine arithmetic. Our discussion has assumed real-valued variables, but, of course, that's not the case in the real world—we use machine arithmetic. As mentioned earlier, a recent paper has shown that verified neural networks in LRA may not really be robust when one considers the bit-level behavior (Jia and Rinard, 2020b).

Another issue is scalability of the analysis. Using arbitrary-precision rational numbers can be very expensive, as the size of the numerators and denominators can blow up due to pivot operations. Reluplex (Katz et al., 2017) ends up using floating-point approximations, and carefully ensure the results are sound by keeping track of round-off errors.

In Chapter 13, we also discuss *theory combination*, an idea that is fundamental in theorem proving, but has thus far seen almost no adoption in neural-network verification.

Part III

Abstraction-Based Verification

Chapter 8
Neural Interval Abstraction

In the previous part of the book, 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 are 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 book, 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 us focus on the following correctness property, defining robustness of a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ on an input monochrome 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-value 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 that 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

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

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 check 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 chapter, 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 returns more answers. The trick is to define infinite sets of inputs using data structures that we can manipulate, called *abstract domains*.

In this chapter, 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 Chapter 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; that is, I would like to somehow evaluate

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

on a given set 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 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 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

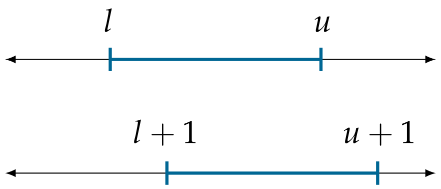


Figure 8.1 Illustration of an abstract transformer of $f(x) = x + 1$.

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.

Example 8.A 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'll be using this 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.

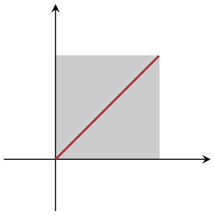
The Interval Domain is Non-relational

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.B 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 rectangle between $(0, 0)$ and $(1, 1)$, as illustrated by 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 boxes whose faces are parallel to the axes. This means that we can't capture the fact that 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 Interval Abstract Transformers

Abstract transformers of n-ary functions

Addition Let's now look at addition, a binary function: $f(\boldsymbol{x}) = x_1 + x_2$. The concrete transformer $f^s : \mathcal{P}(\mathbb{R}^2) \to \mathcal{P}(\mathbb{R})$ is defined as follows:

$$f^s(X) = \{x_1 + x_2 \mid (x_1, x_2) \in X\}$$

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.C Consider

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

The lower bound, 101, results from adding the lower bound of x and y (1 + 100); the upper bound, 205, results from adding the upper bound 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 $l+l' \leqslant x+y \leqslant u+u'$ and, by definition of intervals, $x+y \in [l+l', u+u']$.

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

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

We're in trouble: $[1,-1]$ 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/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.D $f^a([-1,1],[-1,-1]) = [\min(B), \max(B)] = [-1,1]$, where $B = \{1,-1\}$. ■

General abstract transformers

We will now define generic abstract transformers for operations that commonly appear in neural networks.

Affine functions For an affine function

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

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

$$f([5,10],[20,30]) = [3 * 5 + 2 * 20,\ 3 * 10 + 2 * 30] = [55,90]$$

■

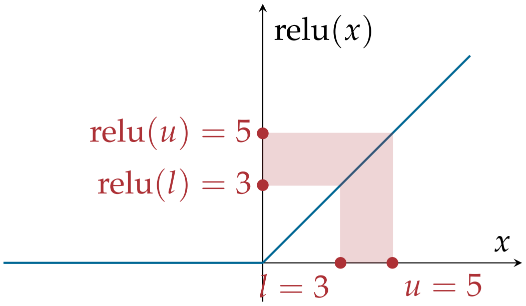


Figure 8.2 ReLU function over an interval of inputs $[l, u]$

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

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.G Let $g(x) = 3x$ and $f(x) = \text{relu}(x)$. Let $h(x) = f(g(x))$. 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

$$h^a([2,3]) = f^a(g^a([2,3])) = f^a([6,9]) = [6,9]$$

■

8.4 **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 $m = |V^{\mathsf{o}}|$. We would like to construct an abstract transformer f_G^a that takes n intervals and outputs m intervals.

We define $f_G^a([l_1, u_1], \ldots, [l_n, u_n])$ 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.

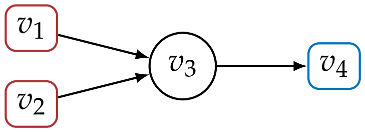
- Second, for every non-input node v, we define

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

  where f_v^a 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 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.H Consider the following simple neural network G:

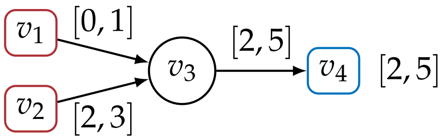


Assume that $f_{v_3}(\boldsymbol{x}) = 2x_1 + x_2$ and $f_{v_4}(x) = \text{relu}(x)$.

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.

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

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

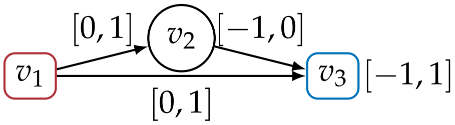


■

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.

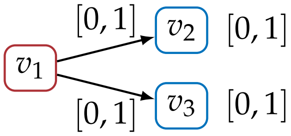
Example 8.1 Consider the following, admittedly bizarre, neural network:



where $f_{v_2}(x) = -x$ and $f_{v_3}(\boldsymbol{x}) = x_1 + x_2$. 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 the two intervals. ■

Example 8.J 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, perhaps 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.

Chapter 9

Neural Zonotope Abstraction

In the previous chapter, 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 set in the interval domain captures constraints of the form:

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

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

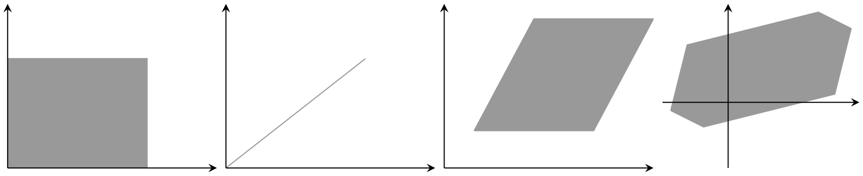


Figure 9.1 Examples of zonotopes in $\mathbb{R}^2$

9.1 **What the Heck is a Zonotope?**

Let's begin with defining a 1-dimensional *zonotope*. We assume a set of *generator* variables, denoted $\epsilon_1, \ldots, \epsilon_m \in \mathbb{R}$. A 1D 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 1D zonotope is just a convoluted way of defining an interval. For example, if we have one generator, 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 is the set of all points

$$\left\{ \left(c_{1,0} + \sum_{i=1}^{m} c_{1,i} \cdot \epsilon_i,\ \ldots,\ c_{n,0} + \sum_{i=1}^{m} c_{n,i} \cdot \epsilon_i \right) \,\middle|\, \epsilon_i \in [-1, 1] \right\}$$

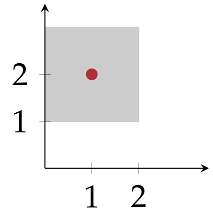
This is best illustrated through a series of examples in $\mathbb{R}^2$.[1]

Example 9.A Consider the following zonotope with 2 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 book, 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:

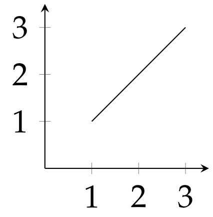
$$(\underbrace{1} + \epsilon_1, \underbrace{2} + \epsilon_2)$$

■

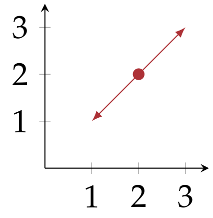
Example 9.B Now consider the following zonotope with 1 generator:

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

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:

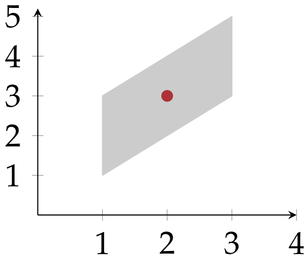


■

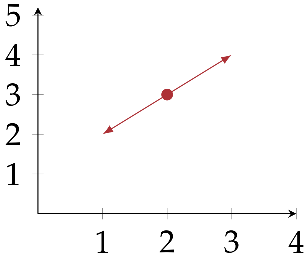
Example 9.C Now consider the following zonotope with 2 generators,

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

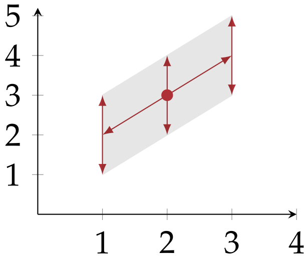
which is visualized as follows, with the center point (2,3) in red.



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) as follows, 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 zonotope on the right in Figure 9.1 uses three generators to produce the three pairs of parallel faces.

A Compact Notation

Going forward, we will use a compact notation to describe an n-dimensional zonotope

$$\left\{ \left(c_{1,0} + \sum_{i=1}^{m} c_{1,i} \cdot \epsilon_i,\ \ldots,\ c_{n,0} + \sum_{i=1}^{m} c_{n,i} \cdot \epsilon_i \right) \middle| \ \epsilon_i \in [-1,1] \right\}$$

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

$$(\langle c_{1,0}, \ldots, c_{1,m} \rangle, \ldots, \langle c_{n,0}, \ldots, c_{n,m} \rangle)$$

For an even more compact presentation, will also use

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

where i ranges from 0 to m, and drop the 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:

$$\begin{aligned} \max\ & c_{j,0} + \sum_{i=1}^{m} c_{j,i} \epsilon_i \\ \text{s.t.}\ & \epsilon_i \in [-1,1] \end{aligned}$$

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

Example 9.D Recall our parallelogram from Example 9.C:

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

In our compact notation, we write this as

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

The upper bound in the 2nd dimension is

$$3 + 1 + 1 = 5$$

computed by setting ϵ_1 and ϵ_2 to 1. ■

9.2 **Basic Abstract Transformers**

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

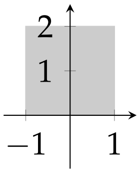
Addition

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 intervals 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.E Consider the simple zonotope $(0 + \epsilon_1, 1 + \epsilon_2)$. This represents a box (two intervals).



The set of possible values we can get by adding points in this box is the interval between -1 and 3. Following the definition of 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, coefficients of ϵ_1, ϵ_2, etc.

Example 9.F Consider $f(x,y) = 3x + 2y$. Then,

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

■

9.3 **Abstract Transformers of Activation Functions**

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

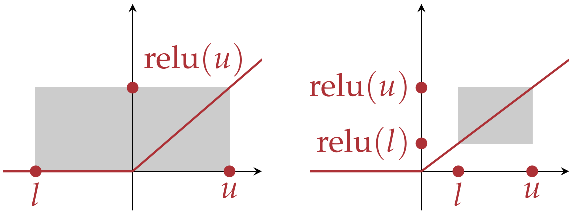
Limitations of the Interval Domain

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 to which outputs.

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



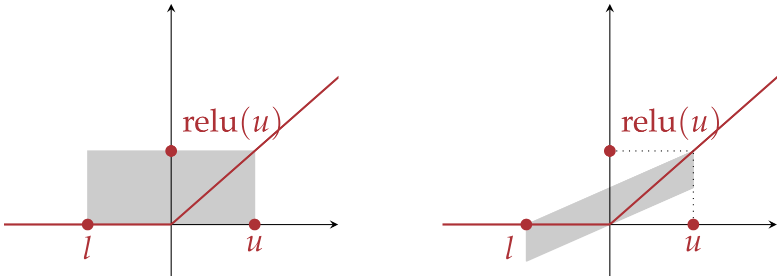
The figure on the left shows the case where the lower bound is negative and the upper bound is positive; the right shows the case where the lower bound is positive.

A Zonotope Transformer for ReLU

Let's slowly build the ReLU abstract transformer for zonotopes. We're given a 1D 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. So, we will set

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

It follows that the space 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 we capture the relation between the input and output. Let's look at an example for clarity:

Example 9.G Say we invoke relu^a with the interval between $l = -1$ and $u = 1$, i.e.,

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

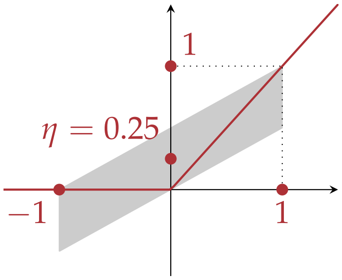
Here, $\lambda = 1/2$ and $\eta = 1/4$. So the result of relu^a is the following zonotope:

$$\langle 0.25, 0.5, 0.25 \rangle$$

We can now plot the 2D zonotope composed of the input and output zonotopes of relu^a:

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

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. 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_v^a is the abstract transformer of f_v, and v has the incoming edges $(v_1, v), \ldots, (v_k, v)$,

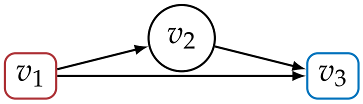
- Finally, the output of f_G^a is the m-dimensional zonotope

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

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.H Consider the following neural network, which we saw in the last chapter.



where $f_{v_2}(x) = -x$ and $f_{v_3}(\boldsymbol{x}) = x_1 + x_2$. Clearly, for any input x, $f_G(x) = 0$. Consider any input zonotope $\langle c_i \rangle$. The output node, v_3, receives the 2D 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'll look at full-blown polyhedra.

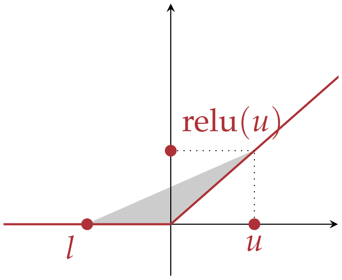
To my knowledge, the zonotope domain was first introduced by Girard (2005) in the context of hyrbid-system model checking. In the context of neural-network verification, Gehr et al. (2018) were the first to use zonotopes, and introduced precise abstract transformers (Singh et al., 2018), one of which 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 are 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.

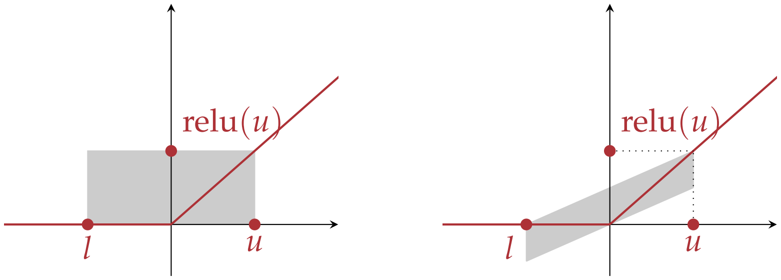
Chapter 10
Neural Polyhedron Abstraction

In the previous chapter, 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 generators $\{\epsilon_i\}_i$. 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:

$$\left\{ \left(c_{1,0} + \sum_{i=1}^{m} c_{1,i} \cdot \epsilon_i, \ \ldots, \ c_{n,0} + \sum_{i=1}^{m} c_{n,i} \cdot \epsilon_i \right) \middle| \ F(\epsilon_1, \ldots, \epsilon_m) \right\}$$

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 ϵ_i variables, e.g.,

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

(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 1D case, a polyhedron is simply an interval. Let's look at higher dimensional examples:

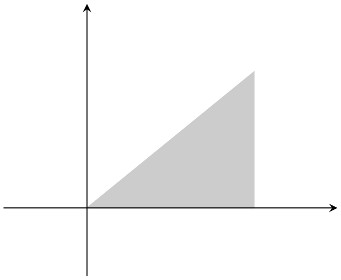
Example 10.A Consider the following 2D polyhedron:

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

where

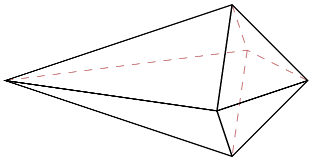
$$F \equiv 0 \leqslant \epsilon_1 \leqslant 1 \ \land \ \epsilon_2 \leqslant \epsilon_1 \ \land \ \epsilon_2 \geqslant 0$$

This polyhedron is illustrated as follows:



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

Example 10.B In 3 dimensions, a polyhedron may look something like this[1]



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

From now on, given a polyhedron

$$\left\{ \left(c_{1,0} + \sum_{i=1}^{m} c_{1,i} \cdot \epsilon_i,\ \ldots,\ c_{n,0} + \sum_{i=1}^{m} c_{n,i} \cdot \epsilon_i \right) \middle| F(\epsilon_1, \ldots, \epsilon_m) \right\}$$

we will abbreviate it as the tuple:

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

10.2 Computing Upper and Lower Bounds

Given a polyhedron $(\langle c_{1,i} \rangle_i, \ldots \langle c_{n,i} \rangle_i, F)$, 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

[1] Adapted from Westburg (2017).

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:

$$\begin{aligned}&\min c_{j,0} + \sum_{i=1}^{m} c_{j,i}\epsilon_i \\ &\text{s.t. } F\end{aligned}$$

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

Example 10.C Take our triangle shape from Example 10.A, 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

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

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

10.3 **Abstract Transformers for Polyhedra**

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:

$$f^a(\langle c_{1,i}\rangle, \ldots \langle c_{n,i}\rangle, F) = \left(\left\langle \sum_j a_j c_{j,0}, \ldots, \sum_j a_j c_{j,m} \right\rangle, F\right)$$

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.

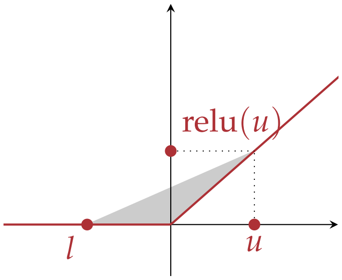
Example 10.D Consider $f(x, y) = 3x + 2y$. Then,

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

■

ReLUs

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



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.

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

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

Note that (1) l and u are the lower and upper bounds of the input polyhedron, which can be computed using linear programming, and (2) $\langle c_i \rangle_i$ is used to denote the full term $c_0 + \sum_{i=1}^{m} c_i \epsilon_i$. Observe also 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.E Consider the 1D polyhedron

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

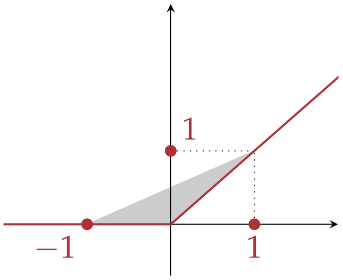
which is the interval between -1 and 1. Invoking

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

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 $n = |V^{\text{in}}|$ and $m = |V^{\text{o}}|$. We would like to construct an abstract transformer f_G^a that takes an n-dimensional polyhedron and outputs an m-dimensional polyhedron.

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

- First, for every input node v_i, we define

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

Recall that we assume a fixed ordering of nodes.

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

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

  where f_v^a 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 are combining the constraints from the incoming edges. This ensures that we capture the relations between incoming values.

- Finally, the output of f_G^a 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., 2020). For a good description of efficient polyhedron domain operations and representations, for general programs, please consult Singh et al. (2017).

Chapter 11
Verifying with Abstract Interpretation

We have seen a number of abstract domains, which 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:

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

where the precondition defines a set of possible values for $\boldsymbol{x}$ and the postcondition defines a set of possible correct values of $\boldsymbol{r}$. To verify such properties with abstract interpretation, we have 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'll 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 are 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 ℓ_2 (Euclidean) and 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, ℓ_2 norm is the length of the straight-line distance 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 ℓ_∞ tells us the biggest difference between two corresponding pixels.

Example 11.A $||(1,2)-(2,4)||_\infty = ||(-1,-2)||_\infty = 2$ ■

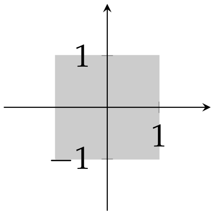
We will start by focusing on the ℓ_∞ norm case and the interval domain.

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



■

As the example above hints, it turns out that we can represent this set precisely in the interval domain as

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

This is because the ℓ_∞ norm allows us to take any element of $\boldsymbol{c}$ and change it by ϵ.

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

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

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.

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 1D zonotope is always > 0. Consider the zonotope

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

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

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

is > 0.

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

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

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

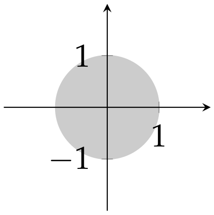
$$F \wedge \langle c_{yi} \rangle > \langle c_{ji} \rangle$$

Robustness in ℓ_2 Norm

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

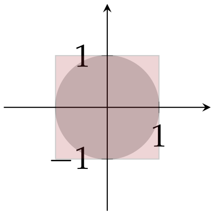
$$\{\boldsymbol{x} \mid ||\boldsymbol{x} - \boldsymbol{c}||_2 \leqslant \epsilon\}$$

Example 11.C Say $\boldsymbol{c} = (0, 0)$ and $\epsilon = 1$. Then the above set is the following circular region:

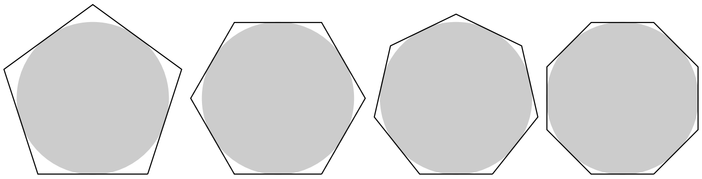


■

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 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:

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

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 chapter. Other properties from earlier in the book can also be verified via the numerical domains we've seen. For example, the aircraft controller from Chapter 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.

Chapter 12
Abstract Training of Neural Networks

You have reached the final chapter of this glorious journey. So far on our journey, we have assumed that we are given a neural network that we want to verify. These neural networks are, almost always, constructed by learning from data. In this chapter, 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 chapter 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 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:

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

where $\mathbb{1}[b]$ is 1 if b is *true* and 0 otherwise. Intuitively, we want the function that makes the smallest number of errors 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. Instead, we often solve an optimization objective like *mean squared error* (MSE) which minimizes *how far f_θ's* prediction is from each y_i. MSE looks like this:

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

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 we are 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

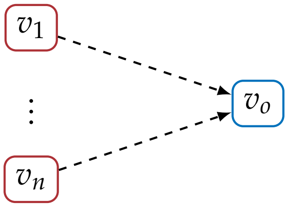
$$\underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L(\theta, \boldsymbol{x}_i, y_i) \tag{12.1}$$

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

Loss Function as a Neural Network

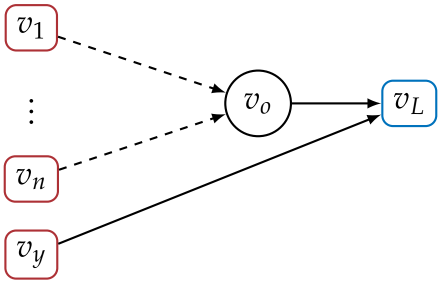
The family of functions f_θ is represented as a neural network graph G_θ, where every node function f_v may be parameterized by θ. We can represent the loss function L as an extension of G_θ, by adding a node at the very end that computes, for example, the squared difference between $f_\theta(\boldsymbol{x})$ and y.

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 define the graph of a loss function $L(\theta, \boldsymbol{x}, y)$ by adding an input node for the label y and comparing the output node 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 a good solution; too large a value, and you'll bounce wildly around unable to catch a 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

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

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

In other words, we compute the gradient for every point in the dataset 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 mini-batches $B_1, \ldots, B_k$.

3. For i from 1 to k,

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

4. Set j to $j + 1$ and repeat.

The number of batches k (equivalently size of the mini-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. As expected, this translates to (1) neural networks that are generally not very robust to perturbations in the input, and (2) even if they are robust on some inputs, verification with abstract interpretation fails to produce a proof.

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

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 chapter: 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:

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

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 $R(\boldsymbol{x})$. 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. (2017).

Solving Robust Optimization via Abstract Interpretation

We will now see how to solve the robust-optimization problem using SGD, with the help of 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 chapter. Therefore, we can overapproximate the inner maximization by abstractly interepreting L on the entire set $R(\boldsymbol{x}_i)$. 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 $L^a(\theta, R(\boldsymbol{x}), y_i) = [l, u]$. In other words, we can overapproximate the inner maximization by abstractly interpreting the loss function on the set $R(\boldsymbol{x})$ 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 list of inputs denoting lower and upper bounds of $R(\boldsymbol{x})$ and returns an upper bound (we don't need the lower bound). We call this idea *flattening* the abstract transformer; we illustrate flattening with a simple example:

Example 12.A 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 $\text{relu}^{af} : \mathbb{R}^2 \to \mathbb{R}$ as follows:

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

■

With this insight, we just invoke SGD on

$$\operatorname*{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^{m} L^{af}(\theta, l_{i,1}, u_{i,1}, \ldots, l_{i,n}, u_{i,n}, y_i) \tag{12.4}$$

where $R(\boldsymbol{x}_i) = ([l_{i,1}, u_{i,1}], \ldots, [l_{i,n}, u_{i,n}])$.

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. I encourage you to try flattening abstract transformers of the zonotope domain.

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, 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. (2017) were the first to pose training of robust neural networks as a robust-optimization problem.

Chapter 13
Looking Ahead

My goal with this book 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 book, 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 book. 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 Chapter 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. 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; Lecuyer 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.

Bibliography

Martín Abadi and Gordon D Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In *NeurIPS*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/448d5eda79895153938a8431919f4c9f-Abstract.html.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 2499–2509, 2018.

Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust optimization*. Princeton university press, 2009.

Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

Robert G Bland. New finite pivoting rules for the simplex method. *Mathematics of operations Research*, 2(2):103–107, 1977.

Aleksandar Bojchevski, Johannes Klicpera, and Stephan Günnemann. Efficient robustness certificates for discrete data: Sparsity-aware randomized smoothing for graphs, images and more. In *International Conference on Machine Learning*, pages 1003–1013. PMLR, 2020.

Jeremy Cohen, Elan Rosenfeld, and Zico Kolter. Certified adversarial robustness via randomized smoothing. In *International Conference on Machine Learning*, pages 1310–1320. PMLR, 2019.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

George B Dantzig. Origins of the simplex method. In *A history of scientific computing*, pages 141–151. 1990.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. Proving data-poisoning robustness in decision trees. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1083–1097, 2020.

Bruno Dutertre and Leonardo De Moura. Integrating simplex with dpll (t). *Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01*, 2006.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751*, 2017.

Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 269–286. Springer, 2017.

Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1625–1634, 2018.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *International Workshop on Hybrid Systems: Computation and Control*, pages 291–305. Springer, 2005.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. 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)*, pages 4074–4084, 2019.

Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 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*, pages 169–178, 2019.

Kai Jia and Martin Rinard. Efficient exact verification of binarized neural networks, 2020a.

Kai Jia and Martin Rinard. Exploiting verified neural networks via floating point numerical error. *arXiv preprint arXiv:2003.03021*, 2020b.

Robin Jia, Aditi Raghunathan, Kerem Göksel, and Percy Liang. Certified robustness to adversarial word substitutions. *arXiv preprint arXiv:1909.00986*, 2019.

Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672. IEEE, 2019.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

Matthew Mirman, Timon Gehr, and Martin Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, pages 3578–3586. PMLR, 2018.

Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2): 245–257, 1979.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018. URL `http://neuralnetworksanddeeplearning.com/`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

Chongli Qin, Krishnamurthy "DJ"' Dvijotham, Brendan O'Donoghue, Rudy Bunel, Robert Stanforth, Sven Gowal, Jonathan Uesato, Grzegorz Swirszcz, and Pushmeet Kohli. 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, 2019. URL `https://openreview.net/forum?id=HyeFAsRctQ`.

Elan Rosenfeld, Ezra Winston, Pradeep Ravikumar, and Zico Kolter. Certified robustness to label-flipping attacks via randomized smoothing. In *International Conference on Machine Learning*, pages 8230–8241. PMLR, 2020.

Benjamin Sherman, Jesse Michel, and Michael Carbin. $lambda_s$: Computable semantics for differentiable programming with higher-order functions and datatypes. *arXiv preprint arXiv:2007.08017*, 2020.

Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 46–59, 2017.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and effective robustness certification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10825–10836, 2018.

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems*, 32:15098–15109, 2019a.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019b.

Niklas Sorensson and Niklas Een. Minisat v1.13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL http://arxiv.org/abs/1312.6199.

Vincent Tjeng, Kai Y Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2018.

Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T Johnson. Verification of deep convolutional neural networks using imagestars. In *International Conference on Computer Aided Verification*, pages 18–42. Springer, 2020.

Alan Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72, 1989.

Alan Mathison Turing. Intelligent machinery, 1948.

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1599–1614, 2018.

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

Kai Y Xiao, Vincent Tjeng, Nur Muhammad Mahi Shafiullah, and Aleksander Madry. Training for faster adversarial robustness verification via inducing relu stability. In *International Conference on Learning Representations*, 2018.

Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1129–1141. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/0cbc5671ae26f67871cb914d81ef8fc1-Paper.pdf.

Mao Ye, Chengyue Gong, and Qiang Liu. 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*, pages 3465–3475, 2020.

Yuhao Zhang, Aws Albarghouthi, and Loris D'Antoni. 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*, volume 119 of *Proceedings of Machine Learning Research*, pages 11023–11032. PMLR, 2020. URL http://proceedings.mlr.press/v119/zhang20b.html.

Yuhao Zhang, Aws Albarghouthi, and Loris D'Antoni. Certified robustness to programmable transformations in lstms. *CoRR*, abs/2102.07818, 2021. URL https://arxiv.org/abs/2102.07818.

He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 686–701, 2019.