VERIFIED DEEP LEARNING

AWS ALBARGHOUTHI

IN PROGRESS; DO NOT CIRCULATE

LAST UPDATED: MARCH 9, 2020

About This Book

Why This Book

I believe that deep learning is here to stay and that we have only scratched the surface of what neural networks can actually do. 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 is 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 even if they have not been exposed to deep learning or verification.

What Does This Book Cover

The book is divided into four 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.

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.

Part 4 Finally, we will discuss verification technology as applied to deep reinforcement learning tasks, where neural networks are used as controllers in a dynamical system.

Parts 2 and 3 are disjoint; the reader can go directly from Part 1 to Part 3.

Table of Contents

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 no bridges between them, until very recently. Interestingly, 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*. 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 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

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

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*. 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 developments, 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. And, 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.

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 factorial? 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.

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 these tasks. The whole point of using deep learning to do these tasks 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. 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. 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. 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 to the on top of object B, B is on top of C, and C is on top of A. But this cannot be!

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 (hint: 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, 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, $\mathbb{R}^n \to \mathbb{R}^m$. Consider the following very simple graph. The red node is an

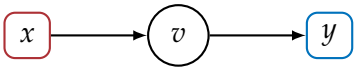


Figure 2.1 A very simple network

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

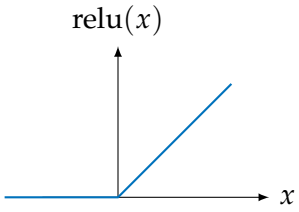


Figure 2.2 Rectified linear unit

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.

Often, in the literature and practice, the affine transformation and the activation function are combined into a single operation. Our graph model of neural networks can capture that, but we usually prefer to distribute the two operations on two different nodes of the graph as it will simplify our life in later chapters when we start analyzing those graphs.

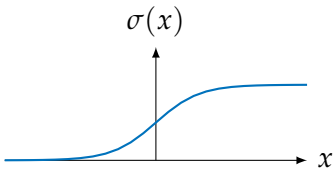


Figure 2.3 Sigmoid activation function

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 transformations to approximate any 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!

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, $\mathbb{R}^3 \to \mathbb{R}^3$.

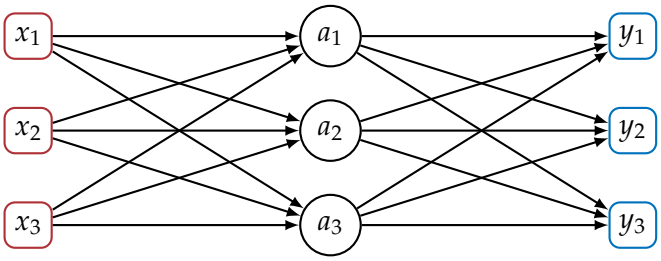


Figure 2.4 A multilayer perceptron

Notice that the nodes of the graph form layers, the input layer, the output

layer, and the layer in the middle which is called the *hidden* layer. This form of graph—or architecture—has the grandiose name of *multilayer perceptron* (MLP). Usually, we have a bunch of hidden layers in an MLP, like in Figure 2.5. Layers in an MLP are called *fully connected* layers, since each node

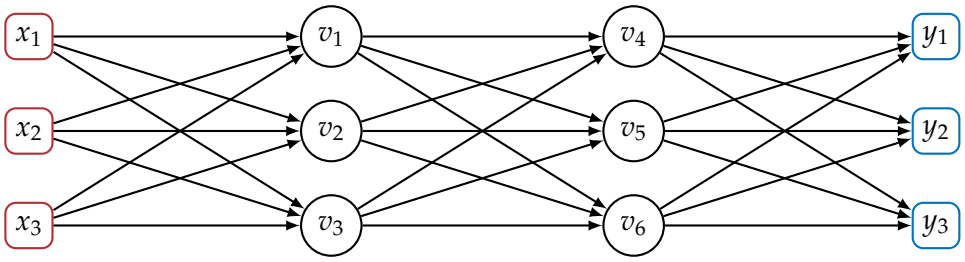


Figure 2.5 A multilayer perceptron with two hidden layers

receives all outputs from the preceding layer.

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 visualize why this actually works, please see Nielsen (2018, Chapter 3).

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 same thing is true of sentences in natural language. 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.

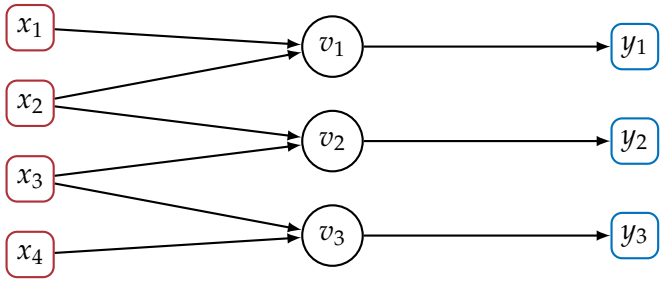


Figure 2.6 1-dimensional convolution

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 formally define 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.

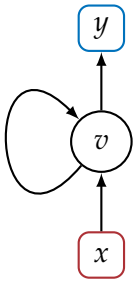


Figure 2.7 Recurrent neural network

Effectively, this graph represents an infinite family of acyclic graphs that unroll this loop a finite number of times. For example, the following 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.

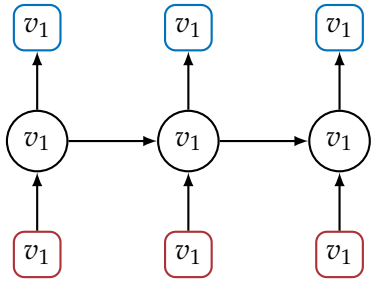


Figure 2.8 Unrolled recurrent neural network

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. With this in mind, 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 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

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

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 v. Then,

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

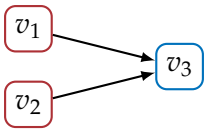
where $x_i = \mathsf{out}(v_i)$, for $i \in [1, n]$.

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

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

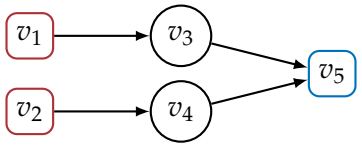
and that we are given the input vector $(11, 79)$ to the network, where node v_1 gets the value 11 and v_2 the value 79. Then, we have

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

Data flow and control flow

The graphs we have defined are known in the field of 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



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.

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

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 monotone if 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 monotone. 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.

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.

Images

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 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 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 $\boldsymbol{x}$ to sneak it past the neural network by fooling it to think that it's a good program. One trick the attacker may use is adding a piece of code to $\boldsymbol{x}$ 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:

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

The *precondition* is a Boolean predicate that 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.:

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

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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ satisfying the precondition, 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*—then there must *exist* a *counterexample*. A counterexample to the correctness property is an assignment to the variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ that result in violating the postcondition.

Example

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:

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

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.

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 faster and more lightweight, 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:

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

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 system 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 what to do: 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. They 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.

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

Notice that the precondition specifies that the distance between the two aircrafts is more than 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 the literature. 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 that 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.

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

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):

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

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.

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:

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

Essentially, a formula in propositional logic defines a circuit with Boolean variables, AND gates ($\wedge$), OR gates ($\vee$), 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 $fv(F)$ 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

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

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 $fv(F) = \{p, q, r\}$. ■

Interpretations

Let F be a formula over a set of variables $fv(F)$. An interpretation I of F is a map from variables $fv(F)$ 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 $fv(F)$ with its interpretation in I.

Example 4.B Say we have the formula

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

and the interpretation

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

Applying I to F, we get

$$I(F) \triangleq (\mathsf{true} \wedge \mathsf{true}) \vee \neg \mathsf{false}$$

■

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:

$$\begin{array}{rcll} \mathsf{true} \wedge F & \equiv & F & \text{Conjunction} \\ F \wedge \mathsf{true} & \equiv & F & \\ & & & \\ \mathsf{false} \wedge F & \equiv & \mathsf{false} & \\ F \wedge \mathsf{false} & \equiv & \mathsf{false} & \\ & & & \\ \mathsf{false} \vee F & \equiv & F & \text{Disjunction} \\ F \vee \mathsf{false} & \equiv & F & \\ \mathsf{true} \vee F & \equiv & \mathsf{true} & \\ F \vee \mathsf{true} & \equiv & \mathsf{true} & \\ & & & \\ \neg\mathsf{true} & \equiv & \mathsf{false} & \text{Negation} \\ \neg\mathsf{false} & \equiv & \mathsf{true} & \end{array}$$

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

$$\begin{aligned} A \wedge B &\equiv B \wedge A \\ A \vee B &\equiv B \vee A \end{aligned}$$

we can push negation inwards:

$$\begin{aligned} \neg(A \wedge B) &\equiv \neg A \vee \neg B \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B \end{aligned}$$

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

$$\begin{aligned} A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \\ A \wedge (B \vee C) &\equiv (A \wedge B) \vee (A \wedge C) \end{aligned}$$

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:

$$(x + y \leqslant 0 \ \wedge \ x - 2y < 10) \vee x > 100$$

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 $x \geqslant 0 \wedge x \leqslant 0$. Similarly, a disequality $x \neq 0$ can be written as $x < 0 \vee x > 0$.

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:

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

A model I for F is

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

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

$$1 - 0 > 0 \ \wedge \ 1 \geqslant 0$$

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.

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

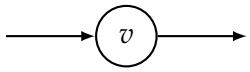


Figure 5.1 A single-input node

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

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

where v° and $v^{\mathsf{in},1}$ are real-valued variables. v° 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:

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

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 $f_v(\boldsymbol{x}) = x_1 + 1.5x_2$.

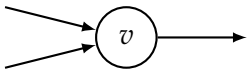


Figure 5.2 A two-input node

The encoding F_v is as follows:

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

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:

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

The way to think of this encoding is as a combination of *if* statements: if S_i is true, then v° 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:

$$F_v \triangleq (\underbrace{v^{\mathsf{in},1} > 0}_{x>0} \implies v^{\circ} = v^{\mathsf{in},1}) \wedge (\underbrace{v^{\mathsf{in},1} \leqslant 0}_{x \leqslant 0} \implies v^{\circ} = 0)$$

■

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.

The first theorem, below, states that 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.

Theorem 5.A Let $(\boldsymbol{a}, b) \in R_v$ and let

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

Then, $I \models F_v$. ∎

The second theorem, below, states that our encoding is *complete*: any model of F_v maps to 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.

Theorem 5.B Let the following be a model of F_v:

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

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 $v \in V \setminus V^{\mathsf{in}}$. 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:

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

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:

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

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.

In the following theorems, assume we have the following ordered input nodes in V^{in}

$$v_1, \ldots, v_n$$

and the following output nodes in V^{o}

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

Theorem 5.C Let $(\boldsymbol{a}, \boldsymbol{b}) \in R_G$ and let

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

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 in the theorem 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:

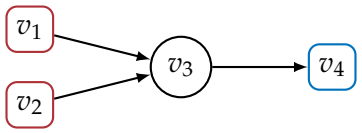
Theorem 5.D Let the following be a model of F_G:

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

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:

$$F_{v_3} \triangleq v_3{}^\circ = 2v_3{}^{\text{in},1} + v_3{}^{\text{in},2}$$
$$F_{v_4} \triangleq (v_4{}^{\text{in},1} > 0 \implies v_4{}^\circ = v_4{}^{\text{in},1}) \wedge (v_4{}^{\text{in},1} \leqslant 0 \implies v_4{}^\circ = 0)$$

Next, we construct edge formulas:

$$F_{\circ \to v_3} \triangleq (v_3{}^{\text{in},1} = v_1{}^\circ) \wedge (v_3{}^{\text{in},2} = v_2{}^\circ)$$
$$F_{\circ \to v_4} \triangleq v_4{}^{\text{in},1} = v_3{}^\circ$$

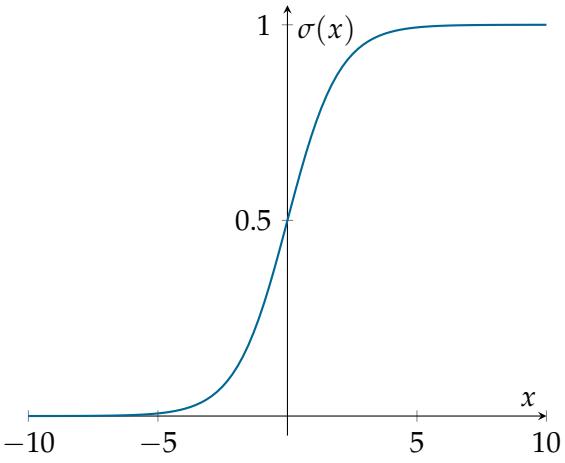


Figure 5.3 Sigmoid function

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:

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

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.

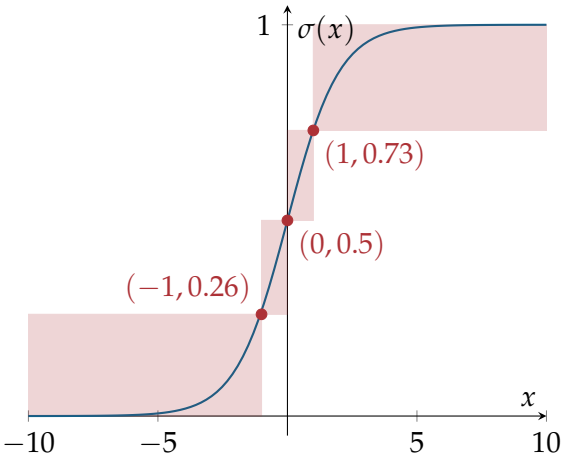


Figure 5.4 Sigmoid function with overapproximation

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:

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

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, and so only Theorem 5.A holds, but not Theorem 5.B.

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:

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

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:

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

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

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 $x \leqslant 0 \vee -x \leqslant 0$.

- The network is encoded as a formula F_G, just as we say 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:

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

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.E 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 $f(x) = x$:

$$\begin{gathered} \langle\!| \, |x - 1| \leqslant 0.1 \, |\!\rangle \\ r \leftarrow f(\boldsymbol{x}) \\ \langle\!| \, r \geqslant 1 \, |\!\rangle \end{gathered}$$

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.

Chapter 6

DPLL Modulo Theories

6.1 Conjunctive Normal Form

6.2 Basic DPLL

6.3 DPLL Calculus

The state of the algorithm

Rule 1: Decide

The first rule simply picks a literal ℓ and decides to set it to true or false. We call such a literal a *decision* literal, and designate it ℓ^d

State is of the form I

If $\ell \notin I$ and $\ell \in F$

Then transform state into $I\ell^d$

Example 6.A ... ■

Rule 2: Propagate

State is of the form I

If $I \models \neg C$, where $C \vee \ell \in F$ and $\ell \notin I$

Then transform state into $I\ell$

Example 6.B ... ■

Rule 3: UNSAT

State is of the form I

If $I \models \neg C$, where $C \vee F$ and $\ell^d \notin I$, for any literal ℓ

Then transform state into F is unsatisfiable

Example 6.C ... ■

Rule 4: Backjump

State is of the form $I\ell^d I'$

If there is a clause $C \vee \ell'$, where $F \Rightarrow (C \vee \ell')$ is valid, $I \models \neg C$, and $\ell' \notin I$

Then transform state into $I\ell'$

Example 6.D ... ■

Soundness and Completeness

Clause learning

6.4 DPLL Modulo Theories

Chapter 7

Specialized Theory Solvers

In the previous chapter, we discussed how to extend SAT solvers to first-order theories by invoking a black-box theory solver. 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*.

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, as formalized in the following theorem:

Theorem 7.A Let F_s be the Simplex form of some formula F.

1. If $I_s \models F_s$, then $I_s \models F$.
2. If $I \models F$, then $I' \models F_s$, where

$$I' = I \cup \left\{ s_i \mapsto \sum_{j=1}^{n} c_{ij} \cdot I(x_j) \right\}_{i=1}^{m}$$

■

The second case is a bit more involved, since F_s has more variables that F: m slack variables, one for each inequality in F. So any model of F need be extended with assignments to slack variables.

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 interepretation, 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 the the formula is unsatisfiable. Let us see this process pictorially on a satisfiable example before we dig into the math.

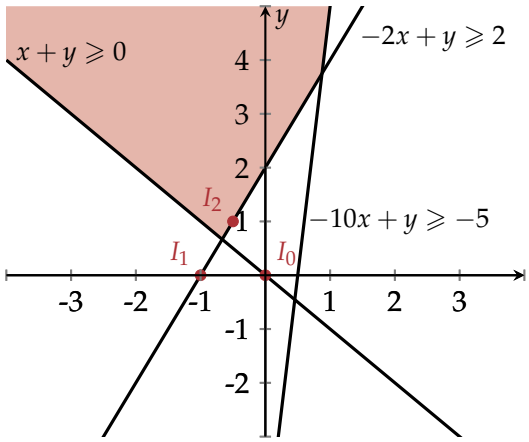


Figure 7.1 Simplex example

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 formulas, 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 1. 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. But, 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 1: 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. $x_i < l_i$ or $x_i > u_i$
  if $x_i < l_i$ **then**
    Let x_j be the first non-basic variable s.t.

$$(I(x_j) < u_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) > l_j \text{ and } c_{ij} < 0)$$

    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.

$$(I(x_j) > l_j \text{ and } c_{ij} > 0) \text{ or } (I(x_j) < u_j \text{ and } c_{ij} < 0)$$

    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, and refer the reader to Dutertre and De Moura (2006) for a full proof.

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:

$$x = 2s_1 + s_2$$
$$y = -s_2 - s_1$$
$$s_3 = -3s_1 - 2s_s$$

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 the formula is UNSAT. The conditions for choosing variable x_j in Algorithm 1 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 begin active (output is 0) or active (output = input)—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 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

Algorithm 2: 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:

$$I(x_i) \leftarrow \text{relu}(I(x_j)) \quad \textbf{or} \quad I(x_j) \leftarrow I(x_i)$$

  ▷ Case splitting (ensures termination)
  if $u_j > 0$, $l_j < 0$, *and* $x_i = relu(x_j)$ *considered more than* τ *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

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 that τ times. 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

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

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

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

failing to provide proofs in some cases.

Part III

Abstraction-Based Verification

Chapter 8

Abstract Interpretation of Neural Networks

Chapter 9

Relational Abstractions for Neural Networks

Chapter 10

Abstract Training of Neural Networks

Part IV

Verified Reinforcement Learning

Chapter 11

Neural Networks as Policies

Chapter 12

Verifying RL Policies

Chapter 13

Efficient Policy Verification

Chapter 14

Enforcing Properties in RL

Bibliography

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

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

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

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.

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