VERIFIED DEEP LEARNING

AWS ALBARGHOUTHI

IN PROGRESS; DO NOT CIRCULATE

LAST UPDATED: JANUARY 21, 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 Forward

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

Chapter 3

Correctness Properties

Part II

Constraint-Based Verification

Chapter 4

Decidable Theories of First-Order Logic

Chapter 5

Encodings of Neural Networks

Chapter 6

Decision Procedures

Chapter 7

Specialized Decision Procedures

Part III

Abstraction-Based Verification

Chapter 8

Just Enough Abstract Interpretation

Chapter 9

Numerical Abstract Domains

Chapter 10

Abstract Execution of Neural Networks

Chapter 11

Abstract Deep Learning

Part IV

Verified Reinforcement Learning

Chapter 12

Neural Networks as Policies

Chapter 13

Verifying RL Policies

Chapter 14

Efficient Policy Verification

Chapter 15

Enforcing Properties in RL