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.

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:

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

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:

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

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.