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.