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

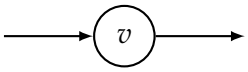


Figure 5.1 A single-input node

$f_v(x) = x + 1$. 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 $f_v(\boldsymbol{x}) = x_1 + 1.5x_2$.

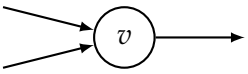


Figure 5.2 A two-input node

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:

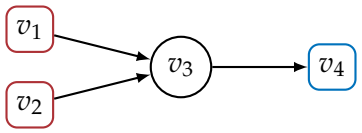
Theorem 5.D Let the following be a model of F_G:

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

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:

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

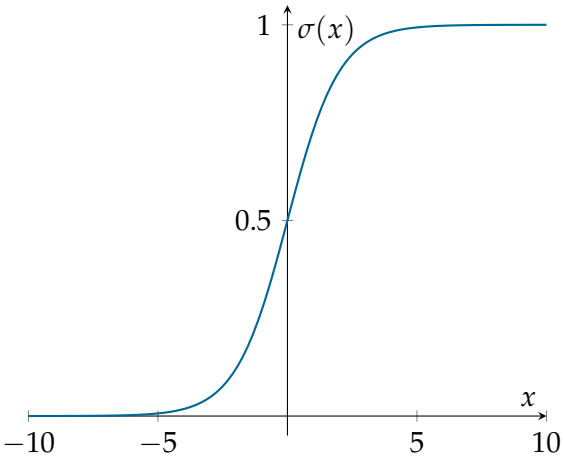


Figure 5.3 Sigmoid function

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.

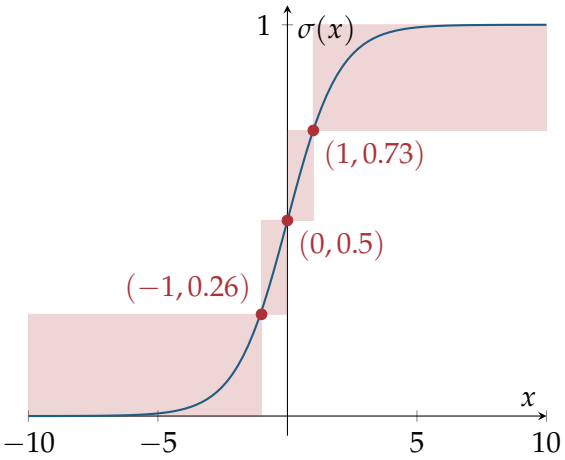


Figure 5.4 Sigmoid function with overapproximation

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 $f(x) = x$:

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

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.