Chapter 8

Abstraction for Neural Networks

In the previous part of the book, we described how to precisely capture the semantics of a neural network by encoding it, along with a correctness property, as a formula in first-order logic. Typically, this means that we are solving an NP-complete problem, like satisfiability modular linear real arithmetic (equivalently, mixed integer linear programming). While we have fantastic algorithms and tools that surprisingly work well for such hard problems, scaling to large neural networks remains an issue.

In this part of the book, we will look at approximate techniques for neural network verification. By approximate, we mean that they overapproximate—or abstract—the semantics of neural-network operations, and therefore can produce proofs of correctness, but when they fail, we do not know whether a correctness property holds or not. The approach we use is based on *abstract interpretation*, a well-studied framework for defining program analyses. Abstract interpretation is a very rich theory, and the math can easily make you want to quite computer science and live a monastic life in the woods, away from anything that can be considered technology. But, fear not, it is a very simple idea, and we will take a pragmatic approach here in defining it and using it for neural-network verification.

8.1 Set Semantics and Verification

Let us focus on the following correctness property, defining robustness of a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$ on an input image $\boldsymbol{c}$ whose classification label

is 1.

$$\begin{gathered}(\!| \, |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1} \, |\!) \\ \boldsymbol{r} \leftarrow f(\boldsymbol{x}) \\ (\!| \, \mathsf{class}(\boldsymbol{r}) = 1 \, |\!)\end{gathered}$$

Concretely, this property makes the following statement: Pick any image $\boldsymbol{x}$ that is like $\boldsymbol{c}$ but is slightly brighter or darker by at most 0.1 per pixel, assuming each pixel is some real-value encoding its brightness. Now, execute the network on $\boldsymbol{x}$. The network must predict that $\boldsymbol{x}$ is of class 1.

The issue in checking such statement is that there are infinitely many possible images $\boldsymbol{x}$. Even if there are finitely many images—because, at the end of the day, we're using bits—the number is still enormous that we cannot conceivably run all those images through the network and ensure that each and every one of them is assigned class 1. But let's just, for the sake of argument, imagine that we can lift the function f to work over sets of images. That is, we will define a version of f of the form:

$$f^s : \mathcal{P}(\mathbb{R}^n) \to \mathcal{P}(\mathbb{R}^m)$$

where $\mathcal{P}(S)$ is the powerset of set S. Specifically,

$$f^s(X) = \{\boldsymbol{y} \mid \boldsymbol{x} \in X,\ \boldsymbol{y} = f(\boldsymbol{x})\}$$

Armed with f^s, we can run it with the following input

$$X = \{\boldsymbol{x} \mid |\boldsymbol{x} - \boldsymbol{c}| \leqslant \mathbf{0.1}\}$$

which is the set of all images $\boldsymbol{x}$ defined above in the precondition of our correctness property. By computing $f^s(X)$, we get the outputs for all images in X. To check our property, we simply check that

$$f^s(X) \subseteq \{\boldsymbol{y} \mid \mathsf{class}(\boldsymbol{y}) = 1\}$$

In other words, all runs of f on every image $\boldsymbol{x} \in X$ result in the network predicting class 1.

The above discussion may sound like crazy talk: we cannot simply take a neural network f and generate a version f^s that takes an infinite set of images. In this chapter, we will see that we actually *can*, but we will often have to lose precision: we will define an abstract version of our theoretical f^s

that returns more answers. The trick is to define infinite sets of inputs using data structures that we can manipulate, called *abstract domains*.

In what follows, we will first fully define the *set semantics* of a neural network, and then we discuss how to abstract those semantics, making them amenable to computation and therefore amenable to automated verification.

8.2 Formalizing Set Semantics of a Neural Network

Recall that a neural network is a defined by a graph $G = (V, E)$, where the set of nodes V has input nodes, V^{in}, output nodes, V^{o}, and all other nodes. The network G defines a function $f_G : \mathbb{R}^{|V^{\mathsf{in}}|} \to \mathbb{R}^{|V^{\mathsf{o}}|}$.

Remember that we've assumed a fixed ordering on nodes and edges (Chapter 2). Henceforth assume that the ordered set of nodes is $v_1, \ldots, v_n$. We will use node indices to refer to them.

Standard semantics, reviewed

We will now review the standard semantics we defined in Chapter 2. We defined the function out that gives us the output of a given node for some input to the network.

Let $\boldsymbol{x}$ be an input to the network. Let $v_{i_1}, \ldots, v_{i_{|V^{\mathsf{in}}|}}$ be the ordered list of inputs nodes. Then, for each v_{i_j}, we have

$$\mathsf{out}(v_{i_j}) = x_j$$

Let v be a non-input node, and let $(v_{i_1}, v), \ldots, (v_{i_k}, v)$ be the ordered sequence of edges whose target is node v. Then,

$$\mathsf{out}(v) = f_v(\mathsf{out}(v_{i_1}), \ldots, \mathsf{out}(v_{i_k}))$$

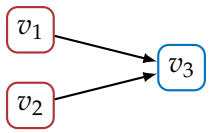
We can read off the output of the network by looking at the result of out on output nodes.

The naunces of set semantics

We can now extend the definition of out to be over sets, i.e., a node takes in a set of inputs and returns a set of outputs. To ensure that we precisely cap-

ture the set semantics, we need to remember the relations between different inputs. Let us look at a simple example to understand the problem:

Example 8.A Take this tiny network, with two input nodes and a single output node.



Say $f_{v_3}(x_1, x_2) = x_1 + 2x_2$. Given the input $(0, 1)$ to the network, we have

$$\begin{aligned}\mathsf{out}(v_1) &= 0\\ \mathsf{out}(v_2) &= 1\\ \mathsf{out}(v_3) &= 0 + 2 * 1 = 2.\end{aligned}$$

Say now that we want to execute this network on a set of two inputs: $\{(0, 1), (0, 2)\}$, and let's imagine a version of out that operates over sets. We'll have the following:

$$\begin{aligned}\mathsf{out}(v_1) &= \{0\}\\ \mathsf{out}(v_2) &= \{1, 2\}\\ \mathsf{out}(v_3) &= \{0 + 2 * 1,\ 0 + 2 * 2\} = \{1, 4\}\end{aligned}$$

Notice the node v_2 produces two outputs, corresponding to the set of possible right (second) values in an input pair. The issue with this view is that we've lost correspondence between the part of the input that v_1 receives and that v_2 receives. In this example it does not show, becaues v_1 receives a singleton set, $\{0\}$. But if we change the input slightly, we notice the problem. Take the input set

$$\{(0, 1), (2, 3)\}$$

The network should produce the set $\{2, 8\}$. But because we break up the input, we will get the following:

$$\begin{aligned}\mathsf{out}(v_1) &= \{0, 2\}\\ \mathsf{out}(v_2) &= \{1, 3\}\\ \mathsf{out}(v_3) &= \{0 + 2 * 1, 0 + 2 * 3, 1 + 2 * 1, 2 + 2 * 3\} = \{2, 3, 6, 8\}\end{aligned}$$

Notice the extraneous elements 3 and 6 in the output, which results from the fact that we additionally consider the inputs $(0, 3)$ and $(1, 3)$, which are not part of the input set.

The moral of the story is: as we propagate sets through a network, we need to make sure we do not mix up different inputs, as we did here. To do so, we will need additional bookkeeping. ■

Set semantics, formalized

To capture set semantics precisely, each node will need to know the values of other nodes in the network. So, we will make a node in our network produce a set of vectors of real numbers, where a vector captures the output of every node in the network. Henceforth, we will use X to denote a set in $\mathcal{P}(\mathbb{R}^n)$, where $n = |V|$. The following example shows how X looks like for an input node.

Example 8.B Take our neural network from the previous example, with the two input nodes and one output node. Say we want to execute it on the set of two inputs

$$\{(0, 1), (2, 3)\}$$

We define the input to the network as the set

$$X = \{(0, 1, x_3), (2, 3, x_3) \mid x_3 \in \mathbb{R}\}$$

Here x_3 is the output of node v_3, the output node. Notice that x_3 it is unconstrained in the set X. This is because initially we do not know what the output of v_3 will be—it can be anything! So we only define the values of the input nodes. ■

We are now ready to define a version of out over sets, which we will call out^s. Let X be the set of inputs, where all non-input nodes are unconstrained. For every input node v,

$$\mathsf{out}^s(v) = X$$

In other words, an input node simply propagates the set X.

Let v_i be a non-input node, and let $(v_{i_1}, v_i), \ldots, (v_{i_k}, v_i)$ be the ordered sequence of edges whose target is node v_i. Then,

$$\mathsf{out}^s(v_i) = f^s_{v_i}\left(\bigcap_{j=1}^{k} \mathsf{out}^s(v_{i_j})\right)$$

where $f^s_{v_i}$ is the set version of f_{v_i}, which we define as follows:

$$f^s_{v_i}(X') = \{\boldsymbol{x}[i \mapsto f_{v_i}(x_{i_1}, \ldots, x_{i_k})] \mid \boldsymbol{x} \in X'\}$$

This looks insane. Let's describe it through a series of examples.

Example 8.C Say we have a node v_i such that

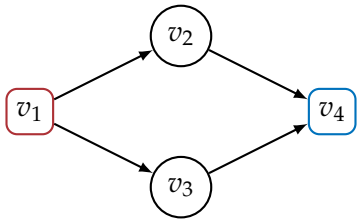
$$f_{v_i}(\boldsymbol{z}) = 2z_1 + 3z_2$$

In the normal world, this node receives two real-valued inputs from two nodes; call them v_j and v_k. But, in our set semantics world, $f^s_{v_i}$ receives a set of vectors of real numbers; call it X'. For each vector $\boldsymbol{x} \in X'$, $f^s_{v_i}$ can read the outputs of v_j and v_k, namely, x_j and x_k, then returns

$$\boldsymbol{x}[i \mapsto 2x_j + 3x_k]$$

which is the vector $\boldsymbol{x}$ but with the ith component (the output of node v_i) replaced with $2x_j + 3x_k$. ■

Example 8.D We will now demonstrate out^s on the following network of 4 nodes, with a single input and output nodes



where

$$f_{v_2}(z) = f_{v_3}(z) = z + 1$$

and

$$f_{v_4}(z_1, z_2) = z_1 + z_2$$

We would like to evaluate this network on the set of inputs $\{1, 2\}$. As such, the input node produces

$$\mathsf{out}^s(v_1) = \{(x_1, x_2, x_3, x_4) \mid x_1 \in \{1, 2\}\}$$

where x_2, x_3, x_4 are the results of nodes 2–4, which are initially unconstrained as they have not been "executed" yet. Now, for nodes v_2 and v_3, we have:

$$\mathsf{out}^s(v_2) = f^s_{v_2}(\mathsf{out}^s(v_1)) = \{(x_1, x_1 + 1, x_3, x_4) \mid x_1 \in \{1, 2\}\}$$

$$\mathsf{out}^s(v_3) = f^s_{v_3}(\mathsf{out}^s(v_1)) = \{(x_1, x_2, x_1 + 1, x_4) \mid x_1 \in \{1, 2\}\}$$

Notice that the result of node v_2 does not constrain x_3, and vice versa, since the executions of v_2 and v_3 are independent of each other—i.e., neither node receives input from the other.

Finally, we arrive at node v_4, which receives inputs from nodes v_2 and v_3. We take the intersection of the two sets, since they provide different pieces of information: $\mathsf{out}^s(v_2)$ constrains the result of node v_2 and $\mathsf{out}^s(v_3)$ constrains the result of node v_3. Therefore, by taking the intersection $\mathsf{out}^s(v_2) \cap \mathsf{out}^s(v_3)$, we effectively combine the results of nodes v_2 and v_3. Specifically:

$$\mathsf{out}^s(v_2) \cap \mathsf{out}^s(v_3) = \{(x_1, x_2, x_3, x_4) \mid x_1 \in \{1, 2\}, x_2 = x_3 = x_1 + 1\}$$

and

$$\begin{aligned}\mathsf{out}^s(v_4) &= f^s_{v_4}\left(\mathsf{out}^s(v_2) \cap \mathsf{out}^s(v_3)\right) \\ &= \{(x_1, x_2, x_3, x_2 + x_3) \mid x_1 \in \{1, 2\}, x_2 = x_3 = x_1 + 1\} \\ &= \{(1, 2, 2, 4), (2, 3, 3, 6)\}\end{aligned}$$

At the end, we can throw out the elements pertaining to the non-output nodes of the network, to arrive at the set of outputs $\{4, 6\}$. ■

8.3 **Abstract Interpretation of Neural Networks**

The trouble with set semantics is that they are merely a theoretical tool: we cannot compute them on infinite sets of inputs, or very large ones. We will

now employ an *abstract interpretation* of set semantics that is amenable to computation, but loses precision. We will begin with a lightweight, semi-formal overview before getting into the mathematical details.

Abstract interpretation by example

Given a function f^s over sets, the idea of abstract interpretation is to define an approximate version, called the *abstract transformer* f^a, for sets that have a certain form that is amenable to efficient computation. The restriction of what kinds of sets we use is called the *abstract domain*. The simplest and most useful abstract domain for our purposes here is the *interval* domain, which restricts us to talking about intervals of real numbers—lower bounds and upper bounds. Consider the following set, defined as an interval $[0, 5]$, which is an infinite set of all real numbers between 0 and 5, and the function $f(x) = x + 1$. Evaluating the abstract transformer f^a on this set, we get

$$f^a([0, 5]) = [1, 6]$$

Simply, whenever we have a linear transformation like f, it suffices to apply it to the lower and upper bounds of the interval to arrive at the new set, instead of applying f to every single element in the infinite set $[0, 5]$!

The problem is that not every set can be represented as an interval. For example, take the set

$$\{x \mid x \leqslant 0 \text{ or } x \geqslant 100\}$$

We cannot represent this set as an interval, because it is comprised of two non-overlapping intervals, $(-\infty, 0]$ and $[100, \infty)$. The best we can do is the interval $(-\infty, \infty)$. This is what we mean by abstraction: sometimes we cannot represent things in our abstract domain, and we lose precision. In this example, we have a worst-case loss in precision—all we can say is that all real numbers are in our abstraction!

Abstract domains

In our set semantics, we have sets $X \subseteq \mathcal{P}(\mathbb{R}^k)$. These are standard mathematical sets: we can take their union $(\cup)$, intersection $(\cap)$, and check whether

one set is contained in another ($\subseteq$). Additionally, as we briefly saw, we can lift operations in our neural network to operate over sets.

Our goal is to define a domain of sets, the *abstract domain*, along with similar operations. But these operations may lose precision. We will use A to denote a set in our abstract domain, and use $\sqcup$, $\sqcap$, and $\sqsubseteq$ to denote the abstract analogues of set union, intersection, and containment.

Henceforth, we will call A an *abstract set* and X a *concrete set*.

Abstraction and concretization

To go back and forth between concrete sets (X) and abstract sets (A), we will define an abstraction function and a concretization function:

$$\alpha : \text{concrete sets} \to \text{abstract sets}$$
$$\gamma : \text{abstract sets} \to \text{concrete sets}$$

The idea is to design an α that gives us a good abstraction of a concrete set, one that loses little precision. The concretization function usually just gives the equivalent representation of an abstract set as a concrete set.

Example 8.E Say our abstract domain is the interval domain. Given a concrete set $X = \{1, 2.5, 5\}$, we can define

$$\alpha(X) = [1, 5]$$

Intuitively, $\alpha(X)$ is the smallest interval that contains the numbers $1, 2.5, 5$. Notice how we have lost precision: we went from a finite set of three elements to one that has all the real numbers from 1 to 5. We can now apply the concretization function to the interval $[1, 5]$:

$$\gamma([1, 5]) = \{x \mid 1 \leqslant x \leqslant 5\}$$

which simply gives us the set of real numbers between 1 and 5. ■

It is customary to use a different notation for concrete sets and abstract sets. For example, for the intervals domain, we use $[\cdot, \cdot]$ to define a set, while for the concrete domain we use $\{\cdot \mid \cdot\}$. This ensures we're always clear what kind of set we're talking about.

Soundness of the abstraction

Of course, we cannot give arbitrary definitions to $\alpha, \gamma, \sqcup$, etc. Our whole goal is to capture set semantics. We will therefore give a set of conditions that ensure *soundness* of our analysis. Essentially, these conditions ensure that abstract operations always produce more answers—i.e., overapproximate set semantics.

The soundness conditions are as follows: for all X, A_1, A_2, we have

$$\gamma(\alpha(X)) \supseteq X \tag{8.1}$$

$$\gamma(A_1 \sqcup A_2) \supseteq \gamma(A_1) \cup \gamma(A_2) \tag{8.2}$$

$$\gamma(A_1 \sqcap A_2) \supseteq \gamma(A_1) \cap \gamma(A_2) \tag{8.3}$$

$$A_1 \sqsubseteq A_2 \Leftrightarrow \gamma(A_1) \subseteq \gamma(A_2) \tag{8.4}$$

The first condition says that abstraction need capture the set X, while potentially losing precision (see Example 8.E and check that the property holds for the given set X). The second and third conditions ensure that abstract set union and intersection capture all results of normal set union and intersection, while, again, potentially losing precision. Finally, the fourth condition ensures that our notion of containment is the same in the abstract domain as the concrete domain.

In addition to the above conditions on basic set operations, whenever we have a function f^s and its abstract transformer f^a, we need to ensure that f^a produces all results of f^s; that is, for any abstract set A,

$$\gamma(f^a(A)) \supseteq f^s(\gamma(A)) \tag{8.5}$$

8.4 Abstractly interpreting the neural network

Now that we have defined what an abstract domain is, we can adapt the definition of our set semantics to be over abstract sets.

We will define a version of out^s over abstract sets, which we will call out^a. The definition is very similar to that of out^s, but uses operations over abstract sets. Let X be the set of inputs, where all non-input nodes are unconstrained. For every input node v,

$$\mathsf{out}^a(v) = \alpha(X)$$

In other words, an input node simply propagates the absraction of the set X.

Let v_i be a non-input node, and let $(v_{i_1}, v_i), \ldots, (v_{i_k}, v_i)$ be the ordered sequence of edges whose target is node v_i. Then,

$$\mathsf{out}^a(v_i) = f^a_{v_i}\left(\prod_{j=1}^{k} \mathsf{out}^a(v_{i_j})\right)$$

where $f^a_{v_i}$ is the abstract version of $f^s_{v_i}$.

8.5 The Interval Domain

We will now look in detail at the interval abstract domain. In general, our goal is to abstract sets in $\mathcal{P}(\mathbb{R}^k)$. For simplicity, we will begin with the 1-dimensional case, i.e., $\mathcal{P}(\mathbb{R})$. Therefore, our intervals are of the form $[l, u]$, where $l \in \mathbb{R} \cup \{-\infty\}$ and $u \in \mathbb{R} \cup \{\infty\}$.[1]

Interval abstraction and concretization

Given a set X, let $\min(X)$ and $\max(X)$ be the smallest and largest real number in X, or $(-)\infty$ if they don't exist. Then, we can define the abstraction function as follows:

$$\alpha(X) = [\min(X), \max(X)]$$

Example 8.F Let $X = \{1, 2, \pi, 8, 9\}$. Then, $\alpha(X) = [1, 9]$. ■

The concretization function is simply

$$\gamma([l, u]) = \{x \mid l \leqslant x \leqslant u\}$$

Theorem 8.A α and γ satisfy Equation (8.1). ■

[1] If we want to be rigorous, whenever u is ∞, we should use) instead of], and analogously for l and $-\infty$. But we avoid doing that for simplicity.

Interval union and intersection

The abstract set union operation for intervals can be defined as follows:

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

Example 8.G $[1, 5] \sqcup [20, 30] = [1, 30]$. Notice the loss in precision incurred by this operation: the union contains the set of numbers $(5, 20)$, which appear in neither interval. Unfortunately, this the best we can do with an interval. ■

Similarly, abstract set intersection is defined as follows:

$$[l_1, u_1] \sqcap [l_2, u_2] = [\max(l_1, l_2), \min(u_1, u_2)]$$

Note that, in principle, this may result in an empty interval, if the two intervals do not overlap. For our purposes of neural-network verification, we will not get into that problem, so we won't bother adding all the corner cases for empty intervals.

Example 8.H $[1, 10] \sqcap [5, \infty] = [5, 10]$. ■

Theorem 8.B $\sqcup$ and $\sqcap$ satisfy Equations (8.2) and (8.3), respectively. ■

Interval containment

Finally, for interval containment, we have

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \text{ iff } (l_1 \geqslant l_2 \text{ and } u_1 \leqslant u_2)$$

Theorem 8.C $\sqsubseteq$ satisfies Equation (8.4) ■

Multi-dimensional intervals

All our above definitions are over intervals of the form $[l, u]$, which correspond to concrete sets in $\mathcal{P}(\mathbb{R})$. All these operations can be easily extended

to intervals over higher dimensions, in order to represent sets in $\mathcal{P}(\mathbb{R}^k)$. We will define such abstract sets as a vector of k intervals, each representing the lower and upper bounds for one of the dimensions:

$$([l_1, u_1], \ldots, [l_k, u_k])$$

We will sometimes abbreviate this vector as $([l_i, u_i])$ and assume its length is know from context. Now, all of the above operations can be extended to such intervals in an element-wise fashion. We give the definition of the abstraction and concretization functions and ask the reader to derive the rest (not because I'm lazy, but because I have better things to do).

Let $X \subseteq \mathcal{P}(\mathbb{R}^k)$. So each element of X is a k-ary vector. Let

$$X_i = \{x_i \mid \boldsymbol{x} \in X\}$$

meaning the projection onto the ith element of each vector in X. Now, we can define α as follows:

$$\alpha(X) = ([\min(X_1), \max(X_1)], \ldots, [\min(X_k), \max(X_k)])$$

Example 8.I Let $X = \{(1,2), (1,3), (0,1)\}$. Then, $\alpha(X) = ([0,1], [1,3])$. ■

Now let $A = ([l_1, u_1], \ldots, [l_k, u_k])$. We define γ as follows:

$$\gamma(A) = \{(x_1, \ldots, x_k) \mid l_i \leqslant x_i \leqslant u_i,\ i \in [1,k]\}$$

Example 8.J . Let $A = ([0,1], [1,3])$. Then,

$$\gamma(A) = \{(x_1, x_2) \mid 0 \leqslant x_1 \leqslant 1,\ 1 \leqslant x_2 \leqslant 3\}$$

■

8.6 Interval abstraction of neural operations

Now that we have defined the basics of the interval domain, let's discuss how to construct abstract transformers of operations that appear in neural networks: linear transformations, activation functions, and (arg)max.

Linear transformations

Before being very precise and notationally careful, let us look at a simple linear transformation and think of how to lift it to intervals.

Example 8.K Take the function $f(z_1, z_2) = 2z_1 + 3z_2$. To lift it to intervals, we simply apply the function twice: if we apply it to the lower bounds of inputs, we get the lower bound of the output, and the same for upper bounds. For example,

$$f^a([0,10],[5,20]) = [2*0+3*5,\ 2*10+3*20] = [15,80]$$

The only complication in general is when dealing with negative coefficients, which may flip lower and upper bounds (more on this below). ■

Suppose node v_i in a neural network performs the following linear transformation:

$$f_{v_i}(\boldsymbol{z}) = \sum_j c_j * z_j + b$$

and that $(v_{i_1}, v_i), \ldots, (v_{i_k}, v_i)$ is an ordered list of edges whose target is v_i. First, we define an interval version of this function:

$$g_{v_i}(([l_j, u_j])_j) = \left[\sum_j \min(c_j * l_j, c_j * u_j) + b,\ \sum_j \max(c_j * l_j, c_j * u_j) + b\right]$$

Notice the min/max operations which are added to deal with negative coefficients.

Recall that the set semantics version of $f_{v_i}(\boldsymbol{z})$ is

$$f^s_{v_i}(X') = \{\boldsymbol{x}[i \mapsto f_{v_i}(x_{i_1}, \ldots, x_{i_k})] \mid \boldsymbol{x} \in X'\}$$

Finally, we define the abstract transformer analogously

$$f^a_{v_i}(A) = A[i \mapsto g_{v_i}((A_{i_1}, \ldots, A_{i_k}))]$$

where A_i is the ith interval of the abstract set A.

Theorem 8.D $f^a_{v_i}$ satisfies Equation (8.5). ■

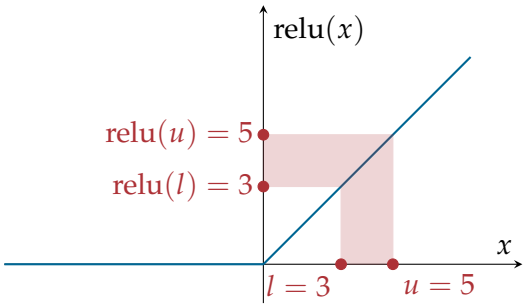


Figure 8.1 ReLU function over an interval of inputs $[l, u]$

Activation functions

Consider an activation function $f : \mathbb{R} \to \mathbb{R}$ that is monotonically increasing. This is true for ReLU and sigmoid, for instance. Monotonicity makes it really easy to lift such function to an interval of inputs. Simply, we can apply f to the lower and upper bounds:

$$f^a([l, u]) = [f(l), f(u)]$$

The reason this is sound is because, by monotonicity, we know that $f(l) \leqslant f(u)$ and for any number $l \leqslant y \leqslant u$, we have $f(l) \leqslant f(y) \leqslant f(u)$.

Example 8.L Figure 8.1 illustrates how to apply ReLU to an interval $[3, 5]$. The shaded region shows that any value y in the interval $[3, 5]$ results in a value $\text{relu}(3) \leqslant \text{relu}(y) \leqslant \text{relu}(5)$. ■

We can now define the abstract transformer for activation functions in the context of a neural network. Suppose that f_{v_i} is a mononontically increasing function. Let g_{v_i} be the interval version of f_{v_i}, as defined above. Then,

$$f^a_{v_i}(A) = A[i \mapsto g_{v_i}(A_j)]$$

where (v_j, v_i) is the only edge whose target is v_i.

Theorem 8.E $f^a_{v_i}$ satisfies Equation (8.5) for any monotonically increasing function f_{v_i}. ■

Maximum value

Another operation that appears in neural networks is for a node to output the value of its larges input. For example, this appears in convolutional neural networks and is known as a *max pooling*. Can you define the max function for the interval domain?

Classification with interval output

When we are dealing with a classification problem, we usually perform an argmax operation over results of output nodes, taking the index of the node with the largest value. Recall that we called this operation class. In the context of the interval domain, each output node will produce an interval of results. How do we define class for intervals?

Say we have a vector of intervals $([l_i, u_i])_i$. We can define class over intervals as follows:

$$\mathsf{class}^a(([l_i, u_i])_i) = \begin{cases} i & \forall j \neq i,\ l_i > u_j \\ \text{inconclusive} & \text{otherwise} \end{cases}$$

Intuitively, if the ith interval is larger than all others, that is, its lower bound is larger than all upper bounds, then classification will always return label i. Otherwise, if some intervals overlap, we cannot tell what is the result of classification, so we say the result is inconclusive.

Example 8.M Say there are two output nodes with the following intervals as output: $[0, 0.5]$ and $[0.75, 1]$. Then class^a returns 2, since all values in the second interval are larger than the first. But, if the two intervals are $[0, 0.6]$ and $[0.45, 0.55]$, then we really cannot tell, as they overlap. It may very well be that the actual values are 0.45 and 0.55, but we've lost too much precision to tell, or they could be 0.6 and 0.4. So we have an inconclusive result in this case. ■

Looking Ahead

In this chapter, we looked at abstract interpretation of neural networks. We saw one example of an abstract domain, the interval domain. The interval domain is quite simple: it can only talk about lower and upper bounds. In the next chapter, we will look at more sophisticated domains that can give better precision.