Introduction to Deep Neural Network Verification

January 28, 2026 (latest version available on Github)

Contents

Part I

Basics of Neural Networks and Verification

Chapter 1

Neural Networks

1.1 Basics of Neural Networks

A *neural network* (**NN**) [27] consists of an input layer, multiple hidden layers, and an output layer. Each layer has a number of neurons, each connected to neurons in the next layer through a predefined set of weights (computed during training). A *Deep Neural Network* (**DNN**) is an NN with *two* or more hidden layers.

The output of an NN is obtained by iteratively computing the values of neurons in each layer. The value of a neuron in the input layer is the input data. The value of a neuron in the hidden layers is computed by applying an *affine transformation* (§1.2) to values of neurons in the previous layers, then followed by an *activation function* (§1.3) such as ReLU and Sigmoid. The value of a neuron in the output layer is computed similarly but may skip the activation function.

NN as a Function We can view an NN as a function that maps input vectors to output vectors:

$$f : \mathbb{R}^n \to \mathbb{R}^m \tag{1.1.1}$$

where n is the number of input neurons and m is the number of output neurons. The neurons in the input layer are the inputs to the function, and the neurons in the output layer are the outputs of the function. The neurons in the hidden layers are also functions that transform the inputs from the previous layer to produce outputs for the next layer.

Example 1.1.1. Fig. 1.1 shows an NN with two inputs $x_1, x_2 \in \mathbb{R}$, two hidden neurons x_3, x_4 in one hidden layer, and one output neuron x_5. The connections between the neurons are weighted edges, and the biases are shown below each neuron.

- The hidden neurons compute:

  $$x_3 = \mathrm{ReLU}\left(-0.5x_1 + 0.5x_2 + 1.0\right), \quad x_4 = \mathrm{ReLU}\left(0.5x_1 + -0.5x_2 + -1.0\right),$$

  where $\mathrm{ReLU}\left(x\right) = \max(x, 0)$ is the ReLU activation function.

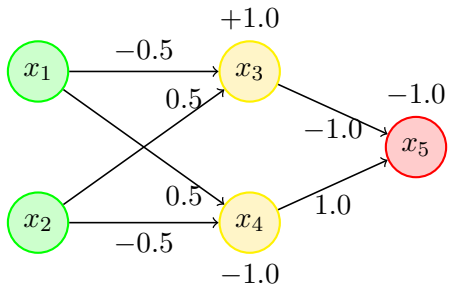


Figure 1.1: A simple DNN with two inputs x_1, x_2, two hidden neurons x_3, x_4, and one output neuron x_5.

- The output neuron computes:

$$x_5 = -1.0 \cdot x_3 + 1.0 \cdot x_4 - 1.0.$$

Thus, this NN computes a function $f : \mathbb{R}^2 \to \mathbb{R}$ where:

$$f(x_1, x_2) = -\text{ReLU}\left(-0.5x_1 + 0.5x_2 + 1.0\right) + \text{ReLU}\left(0.5x_1 - 0.5x_2 - 1.0\right) - 1.0.$$

1.2 Affine Transformation

The affine transformation (AF) of a neuron consists of a *linear combination*—a weighted sum[1]—of its inputs, followed by the addition of a bias term. More specifically, for a neuron with weights $w_1, \ldots, w_n$, bias b, and inputs $v_1, \ldots, v_n$ from the previous layer, the AF computes:

$$f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} w_i v_i + b. \tag{1.2.1}$$

Example 1.2.1. In Fig. 1.1, neuron x_3 receives inputs x_1 and x_2 with weights -0.5, 0.5, and bias 1.0, so its AF is $x_3 = -0.5x_1 + 0.5x_2 + 1.0$.

1.3 Activation Functions

Popular activation functions used in NNs include ReLU, Sigmoid, Tanh, and Softmax. All of these are non-linear[2] functions that introduce non-linearity to the network, allowing it to learn complex patterns in the data.

Tab. 1.1 summarizes the most common activation functions used in NNs, their equations, output ranges, and key uses.

[1] A weighted sum is a sum of the form $\sum_{i=1}^{n} w_i v_i$, where w_i are weights and v_i are variables or terms.

[2] Non-linear means that the output of the function is not a linear combination of its inputs.

Table 1.1: Summary of Common Neural Network Activation Functions

Name	Equation	Output Range	Key Use
ReLU	$\max(0, x)$	$[0, \infty)$	Hidden layers, fast train
Sigmoid	$\frac{1}{1+e^{-x}}$	$(0, 1)$	Binary classification
Tanh	$\tanh(x)$	$(-1, 1)$	Hidden layers, zero-centered
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	$(0, 1), \sum_i = 1$	Multi-class output

1.3.1 ReLU (Rectified Linear Unit)

ReLU is a widely used activation function in neural networks. It is defined as:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

ReLU is **piecewise linear** because it consists of two linear segments as shown as in Fig. 1.2: (i) a constant function (0) when $x \leq 0$, (ii) and (ii) a linear function (x) when $x > 0$. A ReLU activated neuron is said to be *active* if its input is greater than zero and *inactive* otherwise.

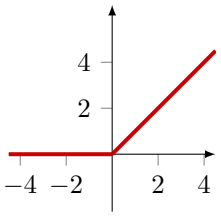


Figure 1.2: ReLU (Rectified Linear Unit) function.

Example 1.3.1. $\text{ReLU}(-1.2) = 0$ (inactive), $\text{ReLU}(0) = 0$ (inactive), and $\text{ReLU}(2.8) = 2.8$ (active).

Logical encoding ReLU can be encoded using the following logical formula:

$$y = \text{ReLU}(x) \iff (x \leq 0 \wedge y = 0) \vee (x > 0 \wedge y = x).$$

In other words, if $x \leq 0$, y must be zero; otherwise; otherwise, y must equal x.

Example 1.3.2. In Z3, we can declare ReLU using `If()`

```
import z3
def relu(x): return z3.If(x <= 0, 0, x)

relu(-1.2)  # returns 0
relu(0)     # returns 0
relu(2.8)   # returns 2.8
```

Nonlinear Property Despite being piecewise linear, ReLU is **nonlinear** because it does not satisfy the two core properties of a linear function

- *Additivity:* $\text{ReLU}(x+y) \neq \text{ReLU}(x) + \text{ReLU}(y)$ in general,
- *Homogeneity:* $\text{ReLU}(\alpha x) \neq \alpha \cdot \text{ReLU}(x)$ when $\alpha < 0$.

In simpler terms, ReLU is nonlinear because it does not form a straight line. It has a **kink** (a sharp bend) at $x = 0$, where the slope changes abruptly from 0 to 1. This discontinuity in the derivative prevents the function from being globally linear.

This non-linearity makes DNN verification difficult. In fact, verifying DNNs with ReLU has the NP-complete complexity as shown in §3.3. We will use ReLU throughout this book as the default activation function for hidden neurons in an NN.

1.3.2 Sigmoid

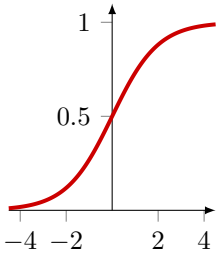


Figure 1.3: Sigmoid function.

Sigmoid, shown in Fig. 1.3, is a smooth—i.e., continuous and differentiable—non-linear activation function that maps any real value to the range (0,1). It is continuous, meaning that small changes in the input will result in small changes in the output, and differentiable, meaning that it has a well-defined derivative at every point. Sigmoid is often used in the output layer of a binary classification problem.

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{1.3.1}$$

Example 1.3.3. $\text{sigmoid}(-1.2) \approx 0.23$, $\text{sigmoid}(0) = 0.5$, and $\text{sigmoid}(2.8) \approx 0.94$. This means that the sigmoid function maps -1.2 to a value close to 0, 0 to 0.5, and 2.8 to a value close to 1.

1.3.3 Hyperbolic Tangent (Tanh)

Tanh, shown in Fig. 1.4, is similar to sigmoid (§1.3.2) but maps any real value to the range (-1,1). It is often used in the output layer of a multi-class classification problem.

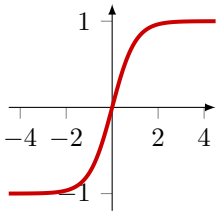


Figure 1.4: tanh (hyperbolic tangent) activation function.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.3.2}$$

Example 1.3.4. $\tanh(-1.2) \approx -0.83$, $\tanh(0) = 0$, and $\tanh(2.8) \approx 0.99$. This means that the tanh function maps -1.2 to a value close to -1, 0 to 0, and 2.8 to a value close to 1.

1.3.4 Softmax

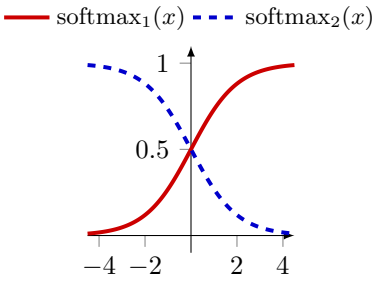


Figure 1.5: Softmax function for 2 classes.

Softmax, shown in Fig. 1.5, is a generalization of sigmoid (§1.3.2) that maps any real value to the range (0,1) and ensures that the sum of the output values is 1. It is often used in the output layer of a multi-class classification problem.

$$\text{softmax}\,(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{1.3.3}$$

Example 1.3.5. For a vector $x = [2, 1, 0]$, softmax computes:

$$\begin{aligned}
\text{softmax}\,(x) &= \left[\frac{e^2}{e^2 + e^1 + e^0}, \frac{e^1}{e^2 + e^1 + e^0}, \frac{e^0}{e^2 + e^1 + e^0}\right] \\
&= \left[\frac{7.389}{7.389 + 2.718 + 1}, \frac{2.718}{7.389 + 2.718 + 1}, \frac{1}{7.389 + 2.718 + 1}\right] \\
&\approx [0.71, 0.24, 0.05]
\end{aligned}$$

This means that softmax maps the input vector x to a probability distribution over the three classes, where the first class has a probability of 0.71, the second class has a probability of 0.24, and the third class has a probability of 0.05.

Example 1.3.6. Use Z3 to encode the DNN in Fig. 1.1 and compute its output x_5 for the input $(x_1, x_2) = (2.0, 0.5)$.

```
#setup the network
x1 = z3.Real('x1')
x2 = z3.Real('x2')
x3 = -0.5*x1 + 0.5*x2 + 1.0
x3_ = z3.If(x3 <= 0, 0, x3)  # ReLU
x4 = 0.5*x1 + -0.5*x2 + -1.0
x4_ = z3.If(x4 <= 0, 0, x4)  # ReLU
x5 = -1.0*x3_ + 1.0*x4_ - 1.0
```

```
#check output on given input
s = z3.Solver()
s.add(x1 == 2.0, x2 == 0.5)
if s.check() == z3.sat:
    m = s.model()
    print("Output x5 is ", m.evaluate(x5))
```

1.4 Neural Network Architectures and Layers

NNs vary in architecture depending on how information flows through them and how computations are structured. Most common models are variations of the *feed-forward network*, with additional structures or constraints layered on top. Tab. 1.2 summarizes several common NN architectures and their typical application domains.

Table 1.2: Popular NN Architectures and Applications

Name	Acronym	Typical Applications
Feedforward NN	FNN / MLP	General function approximation, tabular data
Convolutional NN	CNN	Image processing, video analysis
Residual NN	ResNet	Deep image recognition, medical imaging
Recurrent NN	RNN	Sequence modeling, NLP, time series
Transformer	–	NLP, summarization, code generation, vision
Graph NN	GNN	Graph-structured data, molecule modeling, recommendation

1.4.1 Feedforward Neural Networks (FNNs)

In an FNN, information flows in one direction: from the input layer, through one or more hidden layers, and finally to the output layer. There are no loops or cycles in the computation graph.

Widely used feedforward architectures include fully connected, convolutional, and residual networks. Each architecture has its own strengths and is suited for different types of tasks.

Fully Connected NNs (FCNs) In FCNs, each neuron in a layer is connected to every neuron in the next layer. Thus, every neuron in the input layer is connected to every neuron in the first hidden layer, every neuron in the first hidden layer is

connected to every neuron in the second hidden layer, and so on, until the output layer. Fully connected NNs, sometimes called *dense networks*, are the most basic type of FNNs and are commonly used for tasks like classification.

Example 1.4.1. Fig. 1.1 earlier is an FCN with two inputs and one hidden layer with two neurons, and one output neuron. Fig. 1.6 below shows an FCN with four inputs, two hidden layers with five neurons each, and three output neurons (weights and biases not shown for simplicity).

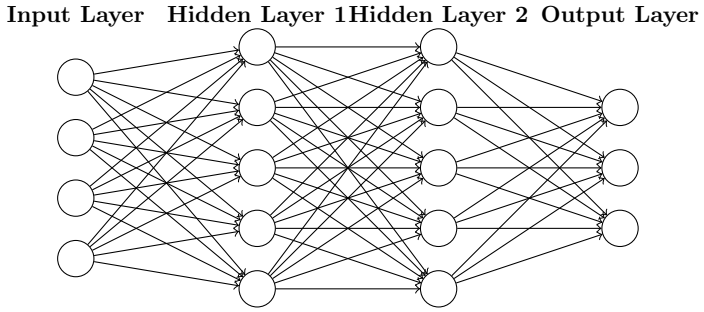


Figure 1.6: A fully connected NN with two hidden layers.

Example 1.4.2 (Classifier)**.** A common use case for FCNs is classification. They take an input, e.g., pixels of an image, and predict what the image is (e.g., cat, dog, car). The output layer represents the probabilities of each class (e.g., y_1 is the probability of cat, y_2 is the probability of dog). Moreover, the outputs are often passed through a softmax activation function (§1.3.4) to convert them into probabilities that sum to 1, and the class with the highest probability is chosen as the predicted label.

Convolutional NNs (CNNs) CNNs replace fully connected layers with *convolutional layers*, which apply local filters across the input space. In CNNs, each neuron receives several inputs, takes a weighted sum over them, passes it through an activation function, and responds with an output. CNNs are commonly used in computer vision and image processing. Despite their local structure, CNNs remain feedforward: data flows forward without cycles.

Example 1.4.3. Fig. 1.7 shows a simple CNN with four inputs, three hidden neurons, and three outputs. Given an input vector $\mathbf{x} = [x_1, x_2, x_3, x_4]$, the computation of the first output proceeds as follows. The first hidden unit forms a linear combination of its inputs as $h_1 = 2x_1 - x_2 + 0.5$. This value is then passed through the ReLU activation function, resulting in $\hat{h}_1 = \text{ReLU}(h_1) = \max(0, 2x_1 - x_2 + 0.5)$. Finally, the first output is simply $y_1 = \hat{h}_1 - 1$.

Residual Networks (ResNets) Resnets extend FNNs by adding *skip connections*—direct links that bypass one or more layers. Resnets are often used in image recognition and classification.

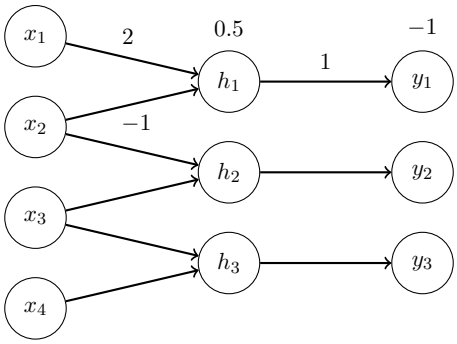


Figure 1.7: 1-dimensional CNN

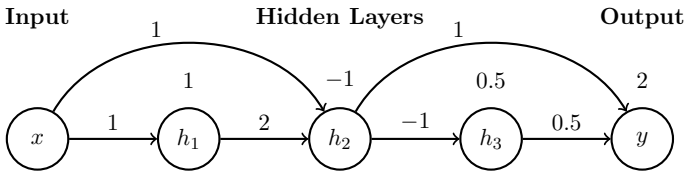


Figure 1.8: ResNet block with weights on all connections and biases above nodes. Each node is labeled and typical ReLU is applied after each hidden sum.

Example 1.4.4. Fig. 1.8 shows an example of a Resnet. Assume the input x and each node applies the ReLU activation. With the weights and biases shown in the diagram, the outputs are computed as follows:

$$h_1 = \text{ReLU}\,(x+1) \quad h_2 = \text{ReLU}\,(2h_1 + x - 1)$$
$$h_3 = \text{ReLU}\,(-h_2 + 0.5) \quad y = 0.5h_3 + h_2 + 2$$

1.4.2 Other NN Architectures

Not all NNs are feedforward. Some architectures introduce cycles, dynamic connections, or non-Euclidean[3] data structures (e.g., graphs).

1.4.2.1 Recurrent Neural Networks (RNNs)

RNNs, often used in natural language processing (NLP) and speech recognition, are designed to recognize patterns in sequences of data. RNNs have *loops* in them, allowing information to be sent forward and backward.

Example 1.4.5. Fig. 1.9 shows a simple RNN cell. Assume the input sequence is $\mathbf{x} = [x_1, x_2, x_3, x_4]$, and the initial hidden state is h_0. For the first time step, the hidden state is computed as $h_1 = \text{ReLU}\,(2x_1 - h_0 + 0.5)$, and the output is $y_1 =$

[3]Euclidean data refers to data that can be represented in a flat, two-dimensional space, such as images.

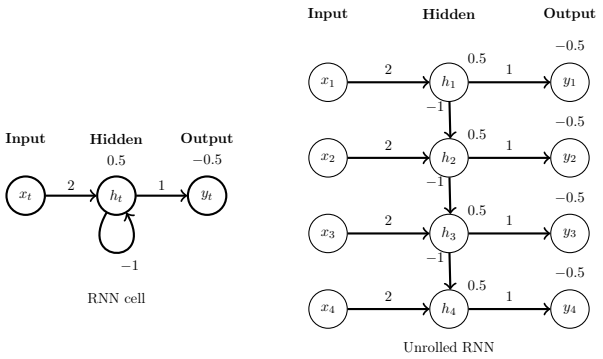


Figure 1.9: RNN cell (left) and unrolled RNN sequence structure (right). Weights on connections; biases above hidden and output nodes.

$h_1 - 0.5$. For the second time step, the hidden state is $h_2 = \mathrm{ReLU}\,(2x_2 - h_1 + 0.5)$, and the output is $y_2 = h_2 - 0.5$.

1.4.2.2 Transformers

Transformers are designed for long-range dependencies using *self-attention* rather than recurrence. They dominate applications in natural language processing and are increasingly used in vision and reinforcement learning.

1.4.2.3 Graph Neural Networks (GNNs)

operate on graphs, allowing each node to aggregate information from its neighbors. GNNs are used in applications involving structured data like molecules or social networks.

1.5 ONNX: Modelling Neural Networks

ONNX (Open Neural Network Exchange) [43] is an open source and widely adopted standard for representing neural networks. It provides a common format for representing the structure and parameters of neural networks, enabling interoperability between different ML frameworks and tools.

ONNX operators that cover most sequential feedforward networks include:

- `Add, Sub, Gemm, MatMul`: basic arithmetic and matrix operations.
- `ReLU, Sigmoid, SoftMax`: activation functions.
- `AveragePool, MaxPool, Flatten, Reshape`: tensor manipulation.
- `Conv, BatchNormalization, LRN`: CNN layers and normalizations.

- `Concat, Dropout, Unsqueeze`: tensor operations.

Example 1.5.1. For the network in Fig. 1.1, the ONNX representation would include:

- Input: x_1, x_2.
- Hidden layer: $x_3 = \text{ReLU}\,(-0.5x_1 + 0.5x_2 + 1.0)$, $x_4 = \text{ReLU}\,(0.5x_1 - 0.5x_2 + 1.0)$.
- Output: $x_5 = -x_3 + x_4 - 1$.

The ONNX representation would look like:

```
ir_version: 9
opset_import { version: 13 }
graph example_nn {
  input: "x"
  output: "x5"

  node { op_type: "Gemm" input: "x" input: "W1" input: "b1" output: "h1" }   # -0.5*x1+0.5*x2+1
  node { op_type: "Relu" input: "h1" output: "x3" }

  node { op_type: "Gemm" input: "x" input: "W2" input: "b2" output: "h2" }   # 0.5*x1-0.5*x2+1
  node { op_type: "Relu" input: "h2" output: "x4" }

  node { op_type: "Neg" input: "x3" output: "nx3" }
  node { op_type: "Add" input: "nx3" input: "x4" output: "s1" }
  node { op_type: "Add" input: "s1" input: "c_minus_1" output: "x5" }

  initializer { name: "W1" values: [-0.5, 0.5] }
  initializer { name: "b1" values: [1.0] }
  initializer { name: "W2" values: [0.5, -0.5] }
  initializer { name: "b2" values: [1.0] }
  initializer { name: "c_minus_1" values: [-1.0] }
}
```

Chapter 2

Properties

Similar to traditional software systems, neural networks (NNs) have desirable properties to ensure the network behaves as expected. These could be specific to the applications modeled by the network, e.g., safety properties for a network modelling a collision avoidance system, or general properties that are desired by all networks, e.g., robustness to small perturbations in the input data.

Below we will discuss properties that are relevant to the verification of NNs. Specifically, these properties can be expressed in a formal language supported by a DNN verifier. Additional properties can be found in the literature [46].

2.1 Definition

As described in §1 NNs define functions of the form $f : \mathbb{R}^n \to \mathbb{R}^m$, where n is the dimension of the input and m is the dimension of the output. Thus, the properties or specifications of an NN—similarly to properties of software program—are defined in terms of its input and output:

> *For any input $x \in \mathbb{R}^n$ satisfying a precondition P, the neural network should produce an output $f(x) \in \mathbb{R}^m$ that satisfies a postcondition Q.*

This says that if the input x satisfies the precondition P, then the output $f(x)$ should satisfy the postcondition Q.

2.2 Common Properties in Neural Networks

We now define some commonly studied properties in NNs verification.

2.2.1 Robustness

Robustness ensures that small changes in the input do not drastically change the output. This is a desirable property for all neural networks, especially classifiers,

where we want to ensure that similar inputs yield similar outputs. For example, a slightly blurred image of a red light should still be classified as a red light.

There are two types of robustness properties: *local* (robustness around a chosen point) and *global* (robustness everywhere).

Local Robustness A neural network f is ϵ-locally-robust at point x with respect to norm $\|\cdot\|$ if

$$\forall x', \qquad \|x - x'\| \leq \epsilon \implies f(x) = f(x'). \tag{2.2.1}$$

where $\|x - x'\| \leq \epsilon$ indicates that the difference between the two points is within a certain (small) threshold ϵ.

Thus local robustness says that a network is robust if all nearby inputs x' (within radius ϵ) are classified the same as x. In other words, no small perturbation around this specific point x will fool the classifier."

Local robustness is what most adversarial robustness papers mean, e.g., checking whether an image of a cat is still classified as a cat under small pixel noise.

Example 2.2.1 (Local Robustness: Image Classification)**.** Consider a neural network f that classifies images into different categories (e.g., dog, cat, etc.). A robustness property requires that if an input image c is classified as a dog, then any perturbed image x that is visually similar to c should also be classified as a dog. This can be expressed as:

$$\forall x, \qquad \|c - x\| \leq \epsilon \implies f(x) = f(c)$$

Global Robustness A neural network f is ϵ-globally-robust with respect to norm $\|\cdot\|$ if

$$\forall x_1, x_2, \qquad \|x_1 - x_2\| \leq \epsilon \implies f(x_1) = f(x_2). \tag{2.2.2}$$

This says the property must hold for all pairs of inputs in the domain: whenever two points are within ϵ of each other, they must share the same label.

Global robustness is a very strong definition, and in practice, almost impossible unless the network is trivial (e.g., outputs the same label everywhere).

Problem 2.2.1. Suppose that an network classifies an input image x as digit 7. We want to ensure that if the image is slightly perturbed (e.g., brightness changed by a small amount ϵ), the network still outputs 7.

2.2.2 Safety

Safety properties ensure that the network conforms to certain safety constraints. This is particularly important in safety-critical applications, such as autonomous vehicles or medical diagnosis systems, where incorrect outputs can lead to catastrophic consequences.

Problem 2.2.2 (Collision Avoidance System)**.** A safety property in a collision avoidance system such as an autonomous vehicle might be that if the intruder is distant and significantly slower than us, then we stay below a certain velocity threshold. Formally, this can be expressed as:

$$d_{intruder} > d_{threshold} \wedge v_{intruder} < v_{threshold} \implies v_{us} < v_{threshold},$$

where $d_{intruder}$ is the distance to the intruder, $d_{threshold}$ is a predefined safe distance, $v_{intruder}$ is the speed of the intruder, and v_{us} is our speed.

Unlike robustness properties, which are often desirable in all networks, safety properties are often *specific* to the application domain. For example, a safety property for an autonomous vehicle may not be relevant for a surgical robot.

Problem 2.2.3 (Safety: Self-Driving Car)**.** A network controlling a self-driving car on a highway might require that: If the car in front is at least 160 meters away and moving slower than us, then we should not accelerate.

2.3 Other properties

2.3.1 Consistency

Consistency requires that a NN behaves consistently when given semantically equivalent or related inputs.

Example 2.3.1 (Logical Consistency in LLMs)**.** Consider queries q_1, q_2, and q_3 about a person's age:

$$q_1 : \text{``How old is person X?''}$$

$$q_2 : \text{``What year was X born if they are currently Y years old?''}$$

$$q_3 : \text{``Will X be Z years old in 2025?''}$$

Thus, if the LLM outputs age Y for q_1, then q_2 should output $\texttt{current\_year} - Y$, and the answer to q_3 should be logically consistent with the stated age. This prevents scenarios where an LLM claims someone is 30 years old but was born in 1985 when the current year is 2024.

2.3.2 Monotonicity

Monotonicity ensures that the NN maintains a consistent ordering relationship between inputs and outputs: an increase in certain input features always leads to a non-decreasing output value. This property is important in applications where domain knowledge dictates logical ordering constraints, such as fairness-aware systems, medical diagnosis, and scientific applications where physical laws impose natural ordering relationships.

Example 2.3.2 (Fairness)**.** A network modelling the probability of admission to a university should be monotonically non-decreasing with respect to GPA and test scores, regardless of gender. Formally, for applicants with profiles (p, s, g) and (p', s', g') where p, p' are GPAs, s, s' are test scores, and g, g' are gender indicators:

$$p \leq p' \land s = s' \land g \neq g' \implies f(p, s, g) \leq f(p', s', g'),$$

$$s \leq s' \land p = p' \land g \neq g' \implies f(p, s, g) \leq f(p, s', g'),$$

where f is the neural network computing admission probability. Additionally, for fairness:

$$f(p, s, \text{male}) = f(p, s, \text{female}) \text{ for all } p, s,$$

ensuring that applicants with identical academic qualifications receive the same treatment regardless of gender.

2.4 Counterexamples

A *counterexample* (**cex**) is a witness that falsifies the correctness property. Given the property defined in §2.1, a counterexample is an input x that satisfies the precondition P but produces an output $f(x)$ that violates the postcondition Q.

Example 2.4.1 (Counterexample to Robustness Property)**.** For local robustness property (§2.2.1):

$$f(x) \neq f(x') \land \|x - x'\| \leq \epsilon \implies x' \text{ is a counterexample.}$$

The goal of DNN verification (§3.1) is to either prove that a property holds—no cex exist—or find a cex that violates the property.

Example 2.4.2 (Counterexample: Monotonicity in Admission)**.** A network predicts admission probability to a university. Inputs: GPA p and test score s.

We want: a higher GPA with same score should not decrease admission probability.

But we observe a case violating this:

- A: GPA = 3.0, score = 1500, prediction = 0.8
- B: GPA = 3.5, score = 1500, prediction = 0.6

Using the numbers in this violating case, write the monotonicity requirement and show how this is a counterexample.

Monotonicity requirement:

$$p \leq p' \land s = s' \implies f(p, s) \leq f(p', s')$$

Counterexample:

$$3.0 \leq 3.5 \land 1500 = 1500 \quad \text{but} \quad f(3.0, 1500) = 0.8 > f(3.5, 1500) = 0.6$$

2.5 The VNN-LIB Specification Language

The VNN-LIB standard [19, 49] defines a format to describe neural networks and properties. Such a standard format enables the sharing of benchmarks across different tools and platforms, facilitating evaluations and comparisons of their performance. VNN-COMP [9] uses VNN-LIB to evaluate different neural network verification tools.

Specifically, VNN-LIB defines a common format for the following components:

- **Neural Network (or model) representation** in the ONNX format [43].
- **Property specification** in SMT-LIB format [6].

2.5.1 VNN-LIB: Property Specification Language

Verification tasks involve proving that the output of a network remains within some desired post-condition Σ, given inputs within a bounded set Π.

Formal Specification Let $\nu : D^{n_1 \times \cdots \times n_h} \to D^{m_1 \times \cdots \times m_k}$ be a neural network, and x and y its input and output tensors. A property is expressed as:

$$\forall x \in \Pi \to \nu(x) \in \Sigma$$

This includes:

- **Precondition** Π: constraints on inputs.
- **Postcondition** Σ: required properties of outputs.

Properties are encoded in **SMT-LIB2**, referencing input/output variable names consistent with ONNX.

Example 2.5.1. A typical ACAS XU network (**??**) maps 5D input to 5D output via 6 layers of 50 ReLU neurons. A property is of the network is

$$-\varepsilon_i \leq x_i \leq \varepsilon_i \quad (0 \leq i < 5)$$

$$y_0 \leq y_1, \quad y_0 \leq y_2, \quad y_0 \leq y_3, \quad y_0 \leq y_4,$$

where x_i are the input variables, y_i are the output variables, and ε_i are small perturbation bounds for each input dimension. This property states that if the inputs are perturbed within the bounds ε_i, then the first output neuron y_0 is less than or equal to all other output neurons y_1, y_2, y_3, y_4.

The VNN-LIB code for this property is as follows:

```
; declaring the input variables
(declare-const X_0 Real)
(declare-const X_1 Real)
(declare-const X_2 Real)
(declare-const X_3 Real)
(declare-const X_4 Real)
(declare-const X_5 Real)
; declaring neuron outputs
(declare-const Y_0 Real)
(declare-const Y_1 Real)
(declare-const Y_2 Real)
(declare-const Y_3 Real)
(declare-const Y_4 Real)
; asserting the input relations
(assert (<= X_0 eps0))
(assert (>= X_0 -eps0))
(assert (<= X_1 eps1))
(assert (>= X_1 -eps1))
(assert (<= X_2 eps2))
(assert (>= X_2 -eps2))
(assert (<= X_3 eps3))
(assert (>= X_3 -eps3))
(assert (<= X_4 eps4))
(assert (>= X_4 -eps4))
; asserting the output relations
(assert (<= Y_0 Y_1))
(assert (<= Y_0 Y_2))
(assert (<= Y_0 Y_3))
(assert (<= Y_0 Y_4))
```

2.6 Problems

Problem 2.6.1 (Robustness + Safety: Drone Controller)**.** An autonomous drone computes its thrust level using a network controller $f(w, d)$, where w is winds speed and d the distance to nearest obstacle. We want two properties:

1. Robustness: If the wind speed reading changes by at most 1 unit ($|w - w'| \leq 1$) and distance remains the same, the thrust decision should remain the same.

2. Safety: If the obstacle distance is less than 5 meters, then thrust must not be greater than 2.

Problem 2.6.2 (Global Consistency: Celsius and Fahrenheit)**.** An NN answers two related questions: Q1: "What is the temperature in Celsius?" (input t_C) Q2: "What is the temperature in Fahrenheit?" (input t_F)

We want consistency: If Q1 outputs y, then Q2 must output $1.8y + 32$.

Problem 2.6.3 (Counterexample: Safety in Vehicles)**.** A NN computes car acceleration a. We want a safety property: if distance to obstacle $d < 10$, then $a \leq 0$. But we observe a scenario where $d = 5$ but $a = +2$.

Write the safety requirement and show how this is a counterexample.

Problem 2.6.4. Consider the NN in Fig. 2.1. Note that in this network the hidden neurons $(v_1 \dots v_4)$ use ReLU activation, but the output neurons (y_1, y_2) just use affine transformation.

- Encode this network using Z3.
  - Try to use Z3 to evaluate the network on various inputs.
- Come UB with **three** properties that this network *does not* have. Recall that a property often has the form in §2.1.

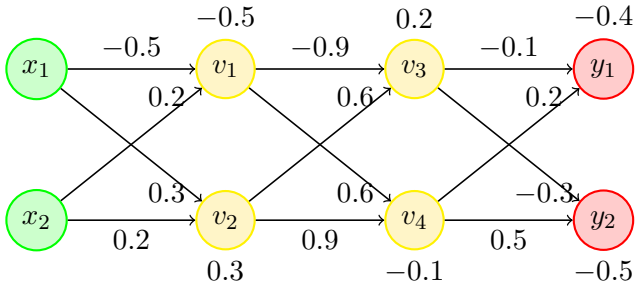


Figure 2.1: A simple NN with 2 inputs, 4 hidden ReLU neurons, and 2 outputs.

  - Argue why each property does not hold by providing a counterexample (§2.4).
  - (Optional, Easy) Try to use Z3 to show that the properties you came UB with do not hold by asking Z3 to find counterexamples (one for each property).

- Come UB with **three** properties that that this network has (i.e., as long as the precondition holds, the postcondition holds).
  - Argue why each property always holds (e.g., for any input satisfying the range X, the first layer outputs values in range Y, etc).
  - (Optional) Show that the properties you came UB will always hold by using Z3 to prove that no counterexample exists (i.e., `unsat`).

Chapter 3

Verification of Neural Networks

This chapter discusses the problem of verifying neural networks, i.e., checking if a given property holds for a neural network. We define the NN verification (NNV) problem in §3.1 and its satisfiability formulation, which is commonly used by DNN verifiers.

3.1 The Neural Network Verification (NNV) Problem

Definition 3.1.1 (NNV)**.** Given a DNN N and a property ϕ, the *NN verification* (NNV) problem asks if ϕ is a valid property[1] of N.

Typically, ϕ is a formula of the form

$$\phi_{in} \Rightarrow \phi_{out},$$

where ϕ_{in} is the *precondition*, i.e., a condition over the inputs of N, and ϕ_{out} is the *postcondition*, i.e., a condition over the outputs of N.

A DNN verifier attempts to find a *counterexample* input to N that satisfies ϕ_{in} but violates ϕ_{out}. If no such counterexample exists, ϕ is a valid property of N. Otherwise, ϕ is not valid and the counterexample can be used to retrain or debug the DNN [29].

Example 3.1.1 (Safety Property for Collision Avoidance System)**.** In Prob. 2.2.2, we defined a safety property (§2.2.2) for a collision avoidance system:

$$d_{intruder} > d_{threshold} \wedge v_{intruder} < v_{threshold} \implies v_{us} < v_{threshold},$$

where $d_{intruder}$ is the distance to the intruder, $d_{threshold}$ is a predefined safe distance, $v_{intruder}$ is the speed of the intruder, and v_{us} is our speed.

Here, the precondition is

$$\phi_{in} = d_{intruder} > d_{threshold} \wedge v_{intruder} < v_{threshold},$$

[1] §2 provides various examples of properties.

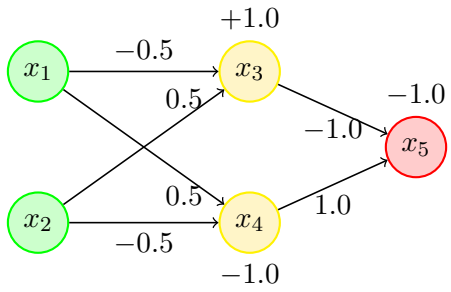


Figure 3.1: A simple DNN (similar to Fig. 1.1).

and the postcondition is

$$\phi_{out} = v_{us} < v_{threshold}.$$

Example 3.1.2. For the DNN in Fig. 3.1 a *valid* property is that the output is $x_5 \leq 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$.

An *invalid* property is that $x_5 > 0$ for those similar inputs. A counterexample showing this property violation is $\{x_1 = -1, x_2 = 2\}$, from which the network evaluates to $x_5 = -3.5$.

3.2 Satisfiability Formulation and Checking

As with traditional software verification, DNN verification is often represented as a satisfiability problem, which can be solved using an SMT solver (e.g., Z3 [17]) or a MILP solver (e.g., Gurobi [28]).

Formulation To formulate the problem, we first define a formula α to represent the network. Typically α is a conjunction ($\bigwedge$) of constraints representing the affine transformation (§1.2) and activation function (§1.3) of each neuron in the network. For example, for a fully-connected neural network (§1.4.1) with L layers, N ReLU neurons per layer, this formula is:

$$\alpha = \bigwedge_{\substack{i\in[1,L]\\ j\in[1,N]}} v_{i,j} \equiv \text{ReLU}\left(\sum_{k\in[1,N]} (w_{i-1,j,k} \cdot v_{i-1,j}) + b_{i,j}\right) \tag{3.2.1}$$

where $v_{i,j}$ is the output of the j-th neuron in layer i, $w_{i-1,j,k}$ is the weight connecting the k-th neuron in layer $i-1$ to the j-th neuron in layer i, and $b_{i,j}$ is the bias of the j-th neuron in layer i. The input layer is layer 0, i.e., $v_{0,j}$ are the input variables.

With this the NNV problem (Def. 3.1.1) can be formulated as checking the validity of the following formula:

$$\alpha \implies (\phi_{in} \implies \phi_{out}). \tag{3.2.2}$$

Formula Eq. 3.2.2 checks if network N satisfies (implies) the property $\phi_{in} \implies \phi_{out}$. This validity checking can be reduced to checking the satisfiability of the formula:

$$\alpha \wedge \phi_{in} \wedge \neg\phi_{out} \tag{3.2.3}$$

If Eq. 3.2.3 is unsatisfiable, then ϕ is a valid property of N. Otherwise, ϕ is not valid. Moreover, we can extract a counterexample for the original problem from the satisfying assignment of Eq. 3.2.3.

Problem 3.2.1. Let α represent our network and the robustness property $|x - x| \leq \epsilon \implies f(x) = f(y)$. Form the satisfiability formula (Eq. 3.2.3) that we need to check. Check them using Z3.

Problem 3.2.2. Let α represent our network and the property $y > 0$ for any input $x_1 \in [r_1, r_2], x_2 \in [r_3, r_4]$. Form the satisfiability formula (Eq. 3.2.3) that we need to check. Check them using Z3.

Problem 3.2.3 (Validity Formulation)**.** Show that $\alpha \implies (\phi_{in} \implies \phi_{out})$ (Eq. 3.2.2) is valid if and only if $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$ (Eq. 3.2.3) is unsatisfiable. First, do this by hand by explicitly writing out the logical equivalences step by step. Then, verify your result using Z3, i.e., show that the negation of the first formula is equivalent to the second formula.

Example 3.2.1. We represent the network in Fig. 3.1 as a formula α:

$$\begin{aligned} x_3 &= \text{ReLU}\left(-0.5x_1 + 0.5x_2 + 1.0\right) \ \wedge \\ x_4 &= \text{ReLU}\left(0.5x_1 - 0.5x_2 - 1.0\right) \ \wedge \\ x_5 &= -x_3 + x_4 - 1.0, \end{aligned}$$

and the property $x_5 > 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$ as:

$$\phi_{in} = (-1 \leq x_1 \leq 1) \wedge (-2 \leq x_2 \leq 2); \qquad \phi_{out} = (x_5 > 0)$$

Satisfiability Solving We can check the satisfiability of $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$ using constraint solving. §4.1 shows how to perform symbolic execution and an SMT solver (e.g., Z3) to automatically generate this formula from a given DNN and property, and check its satisfiability. §4.2 shows how to encode the problem as a MILP constraints, solvable using a MILP solver (e.g., Gurobi).

In this case, the formula is satisfiable, i.e., the property is invalid, and the solver returns `sat`. Any satisfying assignment, e.g., $x_1 = -1$ and $x_2 = 2$, is a counterexample (§2.4) to the property, as it satisfies the precondition ϕ_{in} but violates the postcondition ϕ_{out}, i.e., $x_5 = -3.5$, which is < 0.

Problem 3.2.4. Use Z3 to do Ex. 3.2.1. You might find Ex. 1.3.6 useful. Make sure that you also ask Z3 to find a counterexample violating the property (does not have to be the same as above).

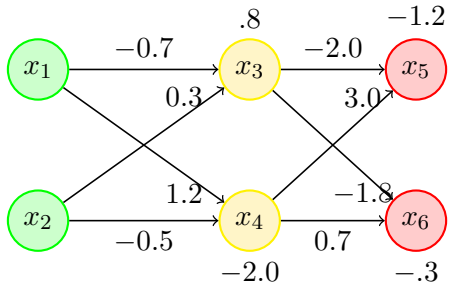


Figure 3.2: A simple DNN with 2 inputs, 2 hidden ReLU neurons, and 2 outputs.

Problem 3.2.5. Consider the DNN in Fig. 3.2. Do the following:

- Write the formula α representing the network.
- Create the property ϕ that the output $x_5 \geq x_6$ for any inputs $x_1 \in [-1, 1], x_2 \in [-1, 1]$.
- Generate the satisfiability formula (Eq. 3.2.3). You do not need to use Z3 to check it.

3.3 Complexity

ReLU-based NNV is NP-complete as shown in [31,45] by reducing the 3-SAT problem to it. This means that the problem of checking whether a given ReLu-based DNN satisfies a property is computationally hard, and no polynomial-time algorithm is known to solve it in the general case.

Part II

Constraint Solving and Abstraction

Chapter 4

Constraint Solving

4.1 Symbolic Execution and SMT Solving

As described in §3.2 the Neural Network Verification (NNV) problem can be formulated as a satisfiability problem. Specifically, we encode the network as a logical formula, and use a constraint solver to check that formula satisfies the property of interest.

A straightforward and automated way to do this encoding is using *symbolic execution* (SE) [4, 33], a well-known software testing technique for finding bugs. SE executes a program on symbolic inputs, i.e., inputs represented as symbols rather than concrete values, and tracks the reachability of program state as symbolic expressions, i.e., logical formulae over symbolic inputs. The satisfiability of these formulae is then checked using an SMT solver, and satisfying assignments represent inputs leading to the undesirable (buggy) program state.

4.1.1 Symbolic Execution

We can adapt traditional SE to our problem by treating the DNN as a program and neurons as variables and executing the DNN on symbolic inputs. Affine transformations can easily be represented as logical formulae because they are linear functions. Activation functions such as ReLUs are translated to disjunctions of linear functions or if-then-else statements, i.e., $\text{ReLU}\,(x) \;=\; \max(x, 0) \;=\; x \geq 0 \wedge x \vee 0 \wedge \neg x$.

Example 4.1.1. To create a logical formula representing the DNN in Fig. 4.1, we can symbolically execute the DNN on symbolic inputs x_1, x_2 and track the values of the neurons x_3, x_4, x_5 as a set (conjunction) of logical formulae. SE starts with the inputs x_1 and x_2 and computes the values of the neurons in the hidden layer, x_3 and x_4, using the affine transformations, followed by ReLUs. Finally, SE computes the output neuron x_5 as a linear combination of the hidden layer neurons.

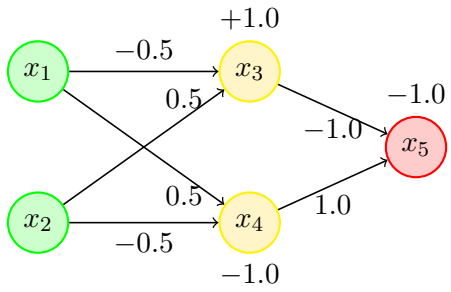


Figure 4.1: A simple DNN (similar to Fig. 1.1).

$$
\begin{aligned}
&x_5 = -x_3 + x_4 - 1.0 \wedge \\
&x_3 = \max(-0.5x_1 + 0.5x_2 + 1.0, 0) \wedge \\
&x_4 = \max(0.5x_1 - 0.5x_2 - 1.0, 0)
\end{aligned} \tag{4.1.1}
$$

4.1.2 SMT Solving

After obtaining the symbolic representation of the DNN, we can use an SMT solver [6] to check the satisfiability of the formula $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$, where α is the symbolic representation of the DNN, ϕ_{in} is the precondition on the inputs, and ϕ_{out} is the postcondition on the outputs. The solver checks if there exists an assignment to the symbolic inputs that satisfies the formula. If such an assignment exists, it means that the property is violated, and we can extract a counterexample from the satisfying assignment. Otherwise, if no such assignment exists, the property is valid

Example 4.1.2. To check that the DNN in Fig. 4.1 satisfies the property $x_5 > 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$ (Ex. 3.2.1), represented as:

$$\phi_{in} = (x_1 \geq -1) \wedge (x_1 \leq 1) \wedge (x_2 \geq -2) \wedge (x_2 \leq 2); \quad \phi_{out} = (x_5 > 0)$$

We check the satisfiability of $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$:

$$
\begin{aligned}
&x_5 = -x_3 + x_4 - 1.0 \wedge \\
&x_3 = \max(-0.5x_1 + 0.5x_2 + 1.0, 0) \wedge \\
&x_4 = \max(0.5x_1 - 0.5x_2 - 1.0, 0) \wedge \\
&\quad (x_1 \geq -1) \wedge (x_1 \leq 1) \wedge (x_2 \geq -2) \wedge (x_2 \leq 2) \quad \wedge (x_5 \leq 0)
\end{aligned}
$$

In this case, the SMT solver returns `sat` and a satisfying assignment, e.g., $x_1 = -1$ and $x_2 = 2$, which is a counterexample to the property. This means that for these inputs, the output $x_5 = -3.5$ violates the property $x_5 > 0$.

4.1.3 Limitations

While using symbolic execution and SMT solving is a straightforward way to verify DNNs, it has several practical limitations:

- **Path Explosion and Scalability**: the number of paths that the solver has to analyze can grow exponentially with the number of ReLU-based neurons and layers as each ReLU, represented as a disjunction of linear functions, has two possible outputs for any input (i.e., the input value itself or 0). This leads to the notorious *path explosion* problem and becoming intractable for large DNNs. Programming assignment 1 (§A.1) demonstrates this issue.
- **Non-linearity:** Other non-linear activation functions (§1.3), such as Sigmoid or Tanh, might not be easily representable as disjunctions of linear functions as ReLU. This can lead to complex formulae that are hard to reason about and/or result in a large search space for the SMT solver.
- **Precision Issues:** SMT solvers may struggle with precision issues when dealing with floating-point arithmetic, which is common in DNNs. This can lead to incorrect results or false positives/negatives in the verification process.

For these reasons, SMT solving is mostly used on toy examples, e.g., in a classroom setting. Modern-based DNN verification tools do not use SMT solving, and instead, rely on more efficient techniques such as abstraction (§5).

4.2 MILP

Instead of using SMT solving, we can encode the NNV problem as a Mixed Integer Linear Programming (MILP) constraints (§D.1), and then invoke an MILP solver such as Gurobi [28] to check their *feasibility* or satisfiability (§D.3.1).

4.2.1 ReLU Encoding

We first encode non-linear activation functions like ReLU using *binary indicator* variables and linear constraints. For each neuron, we introduce a binary variable (§D.2.1) that indicates whether the neuron is "active" (output equals input) or "inactive" (output is zero). This transforms the non-linear $\max(z, 0)$ operation into a set of linear inequalities controlled by the binary variable. We define

- z: the pre-activation value, i.e., the value that goes into ReLU
- $\hat{z}$: the post-activation value, i.e., the output of ReLU
- $a \in \{0, 1\}$: a binary indicator variable encoding whether the neuron is active ($z \geq 0$) or inactive ($z < 0$)

- l, u: lower and upper bounds on z ($l \leq z \leq u$).

The MILP encoding of $\hat{z} = \max(z, 0)$ is then (Ex. D.2.3 shows a simpler example that does not involve bounds):

$$\begin{aligned} &\hat{z} \geq z \\ &\hat{z} \geq 0 \\ &\hat{z} \leq a \cdot u \\ &\hat{z} \leq z - l(1 - a) \\ &a \in \{0, 1\} \end{aligned}$$

These constraints enforce $\hat{z} = z$ when $a = 1$ (active) and $\hat{z} = 0$ when $a = 0$ (inactive), which captures the two possible ReLU outputs (0 or z). Note that constraints are linear and involve both continuous variable $\hat{z}$ and binary variable a.

4.2.2 Computing Concrete Bounds

The mentioned lower and upper bounds (§4.2.1)—*concrete* values l and u such that $l \leq z \leq u$ over the input z to ReLU— are critical for ensuring that the binary indicator a can only take on values that are *valid* given the possible range of z. For example, if $u < 0$, then z is always negative, so the active phase ($a = 1$) is infeasible, and thus can be eliminated. Similarly, if $l \geq 0$, then z is always active.

Either of these cases indicates the neuron is *stable*—always active or always inactive—and thus significantly simplifies the MILP problem because ReLU can be replaced with a linear function (identity or constant 0). Of course, if $l < 0 < u$, then both active and inactive phases are possible, and the MILP solver has to consider both cases. §5 discusses abstraction techniques to approximate ReLU outputs.

Computing Bounds $l \leq e \leq u$ Given a linear expression e

$$e = a_1 x_1 + a_2 x_2 + \ldots + b,$$

where each variable x_i ranges over $[l_i, u_i]$, a simple way to compute the bounds $l \leq e \leq u$ is using *interval* arithmetic (§5.2.1):

- **For the lower bound** (l_3): For each x_i, use its upper bound u_i if $a_i < 0$, and use its lower bound l_i if $a_i \geq 0$.
- **For the upper bound** (u_3): For each x_i, use l_i if $a_i < 0$, and use u_i if $a_i \geq 0$.

This guarantees that the extreme values of e are achieved at some corner of the input box.

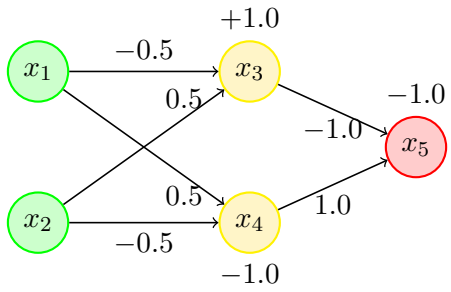


Figure 4.2: A simple DNN (similar to Fig. 1.1).

Example 4.2.1. Consider neuron x_3 in the DNN in Fig. 4.2. We have $x_3 = -0.5x_1 + 0.5x_2 + 1.0$ as the pre-activation value of x_3. The upper u_3 and lower l_3 bounds on x_3 over the input region $x_1 \in [-1, 1]$ and $x_2 \in [-2, 2]$ are:

$$\begin{aligned} z_3 &= -0.5x_1 + 0.5x_2 + 1.0 \\ l_3 &= -0.5(1) + 0.5(-2) + 1.0 = -0.5 \\ u_3 &= -0.5(-1) + 0.5(2) + 1.0 = 2.5 \end{aligned}$$

Notice that we use different pairs of values to compute the lower (1,-2) and upper (-1,2) bounds.

With the computed bounds, we encode the ReLU output of x_3:

$$\begin{aligned} & \hat{x}_3 \geq x_3 \\ & \hat{x}_3 \geq 0 \\ & \hat{x}_3 \leq a_3 \cdot u_3 \\ \Longrightarrow\ & \hat{x}_3 \leq a_3 \cdot 2.5 \\ & \hat{x}_3 \leq x_3 - l_3(1 - a_3) \\ \Longrightarrow\ & \hat{x}_3 \leq x_3 - (-0.5)(1 - a_3) \\ \Longrightarrow\ & \hat{x}_3 \leq x_3 + 0.5(1 - a_3) \\ & a_3 \in \{0, 1\} \end{aligned}$$

Note that because $l_3 = -0.5$ and $u_3 = 2.5$, a_3 can be either 0 or 1, depending on the value of x_3. If we had different bounds, e.g., $l_3 = 0.1$, then a_3 would be forced to be 1, as x_3 can never be less than 0.1 (i.e., x_3 is a stable neuron because it is always active). Stable neurons replaces ReLU with either 0 or identity function, which significantly simplifies the problem.

4.2.3 DNN encoding

More generally, we can encode the DNN as a set of MILP linear constraints as follows:

$$
\begin{aligned}
&\text{(a)} \quad z^{(i)} = W^{(i)}\hat{z}^{(i-i)} + b^{(i)};\\
&\text{(b)} \quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&\text{(c)} \quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0;\\
&\text{(d)} \quad a_j^{(i)} \in \{0,1\};\\
&\text{(e)} \quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned}
\tag{4.2.1}
$$

where x is input, y is output, and $z^{(i)}$, $\hat{z}^{(i)}$, $W^{(i)}$, and $b^{(i)}$ are the pre-activation, post-activation, weight, and bias vectors for layer i.

(a) defines the affine transformation computing the pre-activation value for a neuron in terms of outputs in the preceding layer;

(b) defines the inputs and outputs in terms of the adjacent hidden layers;

(c) asserts that post-activation values are non-negative and no less than pre-activation values;

(d) defines that the neuron activation status indicator variables that are either 0 or 1; and

(e) defines constraints on the upper, $u_j^{(i)}$, and lower, $l_j^{(i)}$, bounds of the pre-activation value of the jth neuron in the ith layer.

Deactivating a neuron, $a_j^{(i)} = 0$, simplifies the first of the (e) constraints to $\hat{z}_j^{(i)} \leq 0$, and activating a neuron simplifies the second to $\hat{z}_j^{(i)} \leq z_j^{(i)}$, which is consistent with the semantics of $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

Example 4.2.2 (Full example)**.** We use MILP to formulate and check if the DNN in Fig. 4.2 satisfies the property $x_5 > 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$. We do this by checking the feasibility of the MILP constraints encoding $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$, where α is the MILP encoding of the DNN, ϕ_{in} is the precondition on the inputs, and ϕ_{out} is the postcondition on the outputs. We do this in several steps:

1. Encoding precondition ϕ_{in} representing input bounds and the postcondition $\neg\phi_{out}$ representing the negation of the postcondition:

$$
\begin{aligned}
\phi_{in} &: -1 \leq x_1 \leq 1; \quad -2 \leq x_2 \leq 2\\
\neg\phi_{out} &: x_5 \leq 0
\end{aligned}
$$

2. Encoding the DNN Hidden layer (pre- and post-activation):

$$
\begin{aligned}
z_3 &= -0.5x_1 + 0.5x_2 + 1.0 \\
z_4 &= 0.5x_1 - 0.5x_2 - 1.0 \\
\hat{z}_3 &\geq z_3, \quad \hat{z}_3 \geq 0 \\
\hat{z}_4 &\geq z_4, \quad \hat{z}_4 \geq 0 \\
a_3, a_4 &\in \{0, 1\} \\
\hat{z}_3 &\leq a_3 \cdot u_3, \quad \hat{z}_3 \leq z_3 - l_3(1 - a_3) \\
\hat{z}_4 &\leq a_4 \cdot u_4, \quad \hat{z}_4 \leq z_4 - l_4(1 - a_4)
\end{aligned}
$$

Output layer:

$$x_5 = -\hat{z}_3 + \hat{z}_4 - 1.0$$

3. Computing upper and lower bounds over given input ranges. Here, with $x_1 \in [-1, 1]$ and $x_2 \in [-2, 2]$, we have:

$$
\begin{aligned}
z_3 &\in [-0.5 \cdot 1 + 0.5 \cdot (-2) + 1, -0.5 \cdot (-1) + 0.5 \cdot 2 + 1] = [-0.5, 2.5] \\
z_4 &\in [0.5 \cdot (-1) - 0.5 \cdot 2 - 1, 0.5 \cdot 1 - 0.5 \cdot (-2) - 1] = [-2.5, 0.5]
\end{aligned}
$$

So we set $l_3 = -0.5, u_3 = 2.5$, and $l_4 = -2.5, u_4 = 0.5$.

4. **Substituting bounds into the constraints:**

$$
\begin{aligned}
\hat{z}_3 &\leq a_3 \cdot 2.5, \quad \hat{z}_3 \leq z_3 - (-0.5)(1 - a_3) = z_3 + 0.5(1 - a_3) \\
\hat{z}_4 &\leq a_4 \cdot 0.5, \quad \hat{z}_4 \leq z_4 - (-2.5)(1 - a_4) = z_4 + 2.5(1 - a_4)
\end{aligned}
$$

5. **The final MILP encoding:**

$$
\begin{aligned}
& -1 \leq x_1 \leq 1; \quad -2 \leq x_2 \leq 2; \\
& z_3 = -0.5x_1 + 0.5x_2 + 1.0; \\
& z_4 = 0.5x_1 - 0.5x_2 - 1.0; \\
& \hat{z}_3 \geq z_3, \quad \hat{z}_3 \geq 0; \\
& \hat{z}_4 \geq z_4, \quad \hat{z}_4 \geq 0; \\
& a_3, a_4 \in \{0, 1\}; \\
& \hat{z}_3 \leq a_3 \cdot 2.5, \quad \hat{z}_3 \leq z_3 + 0.5(1 - a_3); \\
& \hat{z}_4 \leq a_4 \cdot 0.5, \quad \hat{z}_4 \leq z_4 + 2.5(1 - a_4); \\
& x_5 = -\hat{z}_3 + \hat{z}_4 - 1.0; \\
& x_5 \leq 0; \\
& \text{where } z_3 \in [-0.5, 2.5], z_4 \in [-2.5, 0.5], \hat{z}_3 \in [0, 2.5], \hat{z}_4 \in [0, 0.5].
\end{aligned}
$$

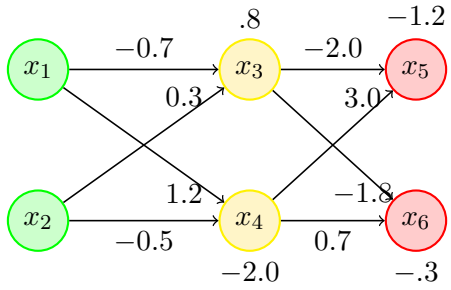


Figure 4.3: A simple DNN (similar to Fig. 3.2).

6. **Solving** From the MILP formulation, the MILP solver attempts to find a feasible (§D.3.1) or satisfying assignment representing a counterexample violating the property. In this example, it might find $x_1 = -1, x_2 = 2$, which leads to:

$$
\begin{aligned}
z_3 &= -0.5(-1) + 0.5(2) + 1.0 = 2.5 \\
z_4 &= 0.5(-1) - 0.5(2) - 1.0 = -2.5 \\
a_3 &= 1, \hat{z}_3 = 2.5 \quad \text{(neuron active)} \\
a_4 &= 0, \hat{z}_4 = 0 \quad \text{(neuron inactive)} \\
x_5 &= -2.5 + 0 - 1.0 = -3.5
\end{aligned}
$$

Since $x_5 = -3.5 \leq 0$, this assignment satisfies our search for $x_5 \leq 0$ and thus is a counterexample.

Problem 4.2.1. Use MILP encoding and solving to check if the network in Fig. 4.3 satisfies the property ϕ that the output $x_5 \geq x_6$ for any inputs $x_1 \in [-1, 1], x_2 \in [-1, 1]$. Specificaly, do the following steps:

1. Write down the precondition ϕ_{in} and postcondition ϕ_{out} representing the property ϕ.
2. Write down the MILP encoding of the DNN.
3. Compute the upper and lower bounds of the pre-activation values of the neurons in the hidden layer.
4. Substitute the bounds into the MILP encoding.
5. Write down the final MILP encoding of $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$.
6. Check the feasibility of the MILP encoding (e.g., using Z3) and report the result.

Problem 4.2.2. As shown in Ex. 4.2.1, the upper and lower bounds of neuron x_3 of the DNN in Fig. 4.2 are $u_3 = 2.5$ and $l_3 = -0.5$.

- Compute the upper and lower bounds of x_4 in the same example.
- Change the weights and/or biases of the network in Fig. 4.2, but keep the input ranges $-1 \leq x_1 \leq 1$ and $-2 \leq x_2 \leq 2$, such that x_3 becomes a stable neuron (always active or always inactive). Show the new weights and/or biases, and compute the new upper and lower bounds of x_3.

4.2.4 Limitations

- **Scalability:** While MILP is efficient in dealing with linear constraints, it still cannot be applied directly to real-world DNNs due to the large search space. Each ReLU application to an unstable neuron introduces a binary variable (§4.2.1), leading to exponential number of possible branches, and real-world DNNs can have millions of such ReLUs, making the search space intractable.
- **Limited exploitation of network structure and modern hardware** Off-the-shelf MILP solvers are designed for arbitrary MILP problems and do not exploit DNN-specific structures and concepts such as activation patterns (§6.1) to prune the search space. Moreover, MILP solvers, even industrial-strength ones such as Gurobi [28], are primarily CPU-based and does not leverage the massive parallelism provided by modern GPUs to speed UB computation.
- **Advanced Abstraction and Heuristics** Interval analysis is efficient and commonly used for computing neuron bounds, but cannot capture dependencies between neurons, leading to precision loss in deeper networks. SOTA DNN verification tools use more advanced abstract domains such as zonotopes and polytopes to improve precision (§5).

  Modern DNN verification tools also employ heuristics to decide which neurons to branch on and to determine stable neurons to avoid unnecessary branching. These heuristics are not available in general MILP solvers.

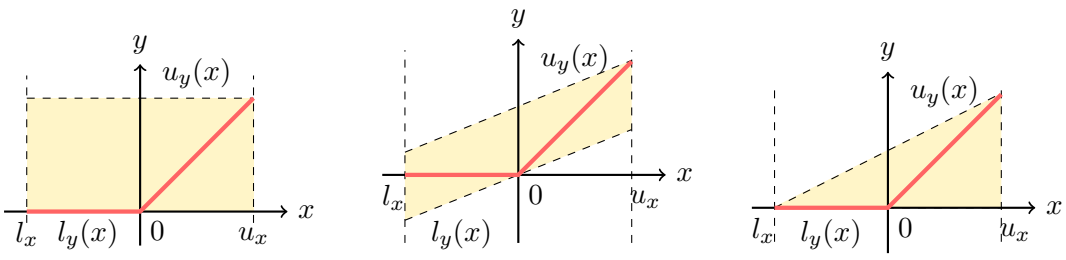
Chapter 5

Abstractions

As mentioned in §4.2.1 the computation of upper and lower bounds of the neuron values help determine neuron stability and therefore can significantly reduce the complexity of verification. Modern DNN verifiers explore different *abstraction* techniques to compute these bounds more precisely while balancing computational efficiency.

This chapter discusses the interval, zonotope, and polytope abstract domains commonly used in verification. Each domain provides a different way to represent and compute the bounds of neurons, with its own trade-offs in terms of precision and computational complexity.

5.1 Overview and Background



(a) Interval (b) Zonotope (c) Polytope (lower flat)

Figure 5.1: Abstractions of $\mathrm{ReLU}(x) = \max(0, x)$ over $x \in [l_x, u_x]$ using (a) interval, (b) zonotope, (c) polytope abstractions.

Common Abstractions for ReLU We use ReLU to illustrate different abstractions. Our goal is to approximate the bounds of the post-ReLU value of a neuron given the bounds of its pre-ReLU value. For example, for a ReLU neuron $y = \max(0, x)$, we want to compute the bounds $[l_y(x), u_y(x)]$ of y given the bounds $[l_x, u_x]$ on x.

Fig. 5.1 illustrates common abstractions (or *over-approximations*) for the ReLU function $y = \max(0, x)$, where $x \in [l_x, u_x]$ and $y \in [l_y(x), u_y(x)]$. The values of ReLU are shown as points on the red line, which is non-convex and consists of two linear segments: one for $x < 0$ (where $y = 0$) and another for $x \geq 0$ (where $y = x$). To compute the bounds $l_y(x)$ and $u_y(x)$, we can use different abstractions:

- **Interval Abstraction:** Interval represents the post-ReLU value as an interval $[l_y(x), u_y(x)]$ where $l_y(x) = 0$ and $u_y(x) = u_x$. Interval is a simple and efficient abstraction, but it can be *imprecise*, e.g., it does not capture the relationship between the input and output of ReLU.

  In Fig. 5.1a, the yellow rectangle (box) represents the interval abstraction in the 2D space of (x, y). As can be seen, this box is too large of an over-approximation (too coarse or loose), as it includes many points that are not achievable by ReLU, e.g., the point $(0.5, 0.0)$ is not on the red line.

- **Zonotope Abstraction:** Zonotopes, commonly represented using center and generators, can capture linear relationships between variables more effectively. In Fig. 5.1b, the a parallelogram —a zonotope in 2D—is arguably tighter than the interval in Fig. 5.1a. It captures the linear relationship between x and y and excludes points that are not achievable by ReLU, e.g., the point $(0.5, 0.0)$ that was included in the interval abstraction is not included in the zonotope. However, it still includes non-ReLU points and is also not strictly better than interval, e.g., the point $(0.5, -0.5)$ is included in the zonotope but not in the interval abstraction (or ReLU).

- **Polytope Abstraction:** Polytopes can represent arbitrary linear constraints and provide very precise bounds. We can construct a polytope that tightly encloses the non-convex shape of the ReLU function.

  In Fig. 5.1c, the polytope is shown as a trapezoid that captures the linear segments of ReLU. The lower bound is $l_y(x) = 0$ and the upper bound is $u_y(x) = u_x$.

5.1.1 Geometric Representations

Because our abstract domains are interval, zonotope, and polytopes, it is useful to understand how these shapes can be represented. There are two common ways to describe a geometric shape:

- **Corner-based representation:** Define the shape by listing all of its corner points (vertices). For example, a rectangle in 2D is defined by its four corners, and a box in 3D is defined by its eight corners. This representation is intuitive and directly shows the boundaries of the shape.

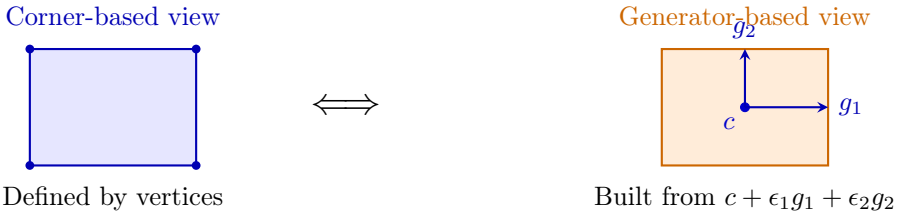


Figure 5.2: Two equivalent descriptions of a 2D interval abstraction (a rectangle). Left: defined by its four corners (independent variable bounds). Right: represented as a zonotope with center c and orthogonal generators g_1, g_2.

- **Generator-based representation:** Define the shape using a *center point* and several direction vectors (*generators*) that define how the shape can stretch or tilt. Instead of listing all corners, it specifies how to reach any point in the shape by starting from the center and moving along the generators within certain limits. This representation is often more compact, especially for higher-dimensional shapes.

Fig. 5.2 uses corner-based and generator-based representations to represent a 2D rectangle, which is an interval abstraction over two variables. The left side shows the corner-based view, where the rectangle is defined by its 4 corners. The right side shows the generator-based view, where the rectangle is represented as a zonotope with a center point c and two orthogonal generators g_1 and g_2.

Example 5.1.1. An interval for a single variable is simply a line segment defined by its *two* endpoints (corners)—the lower and upper bounds. Equivalently, it can be represented using a zonotope with a center point and a single generator vector that points in both directions (left and right) from the center:

$$[l, u] = \{c + \epsilon g \mid \epsilon \in [-1, 1]\}$$

where $c = \frac{l+u}{2}$ is the center, $g = \frac{u-l}{2}$ is the generator, and ϵ controls how far we move along the generator.

Example 5.1.2. Consider the interval:

$$x \in [-2, 3].$$

We can express this interval using a single generator as shown in Fig. 5.3.

$$\begin{aligned} c &= \frac{-2+3}{2} = 0.5, && \text{the midpoint of the interval} \\ g &= \frac{3-(-2)}{2} = 2.5, && \text{the half-width (distance from the center to each end)} \\ \epsilon &\in [-1, 1], && \text{acts as a knob to move along the generator} \end{aligned} \tag{5.1.1}$$

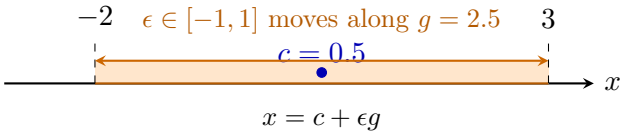


Figure 5.3: An interval $[-2, 3]$ is a 1D zonotope with center $c = 0.5$, generator $g = 2.5$, and parameter $\epsilon \in [-1, 1]$. Varying ϵ covers the entire interval.

Thus, any point x in this interval can be written as:

$$x = c + \epsilon g \text{ or } x = 0.5 + \epsilon \cdot 2.5, \quad \text{where} \quad \epsilon \in [-1, 1].$$

For example:

$$\epsilon = -1 \Rightarrow x = -2, \quad \epsilon = 0 \Rightarrow x = 0.5, \quad \epsilon = +1 \Rightarrow x = 3.$$

As ϵ varies between -1 and $+1$, we cover the entire interval $[-2, 3]$.

5.1.2 Transformer Functions

The concept of computing abstractions is central to program analysis, e.g., through *abstract interpretation* techniques. It allows us reason about the behavior of a program without evaluating it on all concrete inputs—which may be infinite. Instead, we use an *abstract domain*, such as interval, to summarize sets of concrete values, enabling sound and scalable approximation.

5.1.2.1 Abstraction Functions

In abstraction interpretation, we have the *abstraction function*

$$\alpha : D \to D^a,$$

which maps a concrete value from the domain D to an element in a finite or simpler abstract domain D^a.

Example 5.1.3 (Odd/Even)**.** The `odd/even` or parity abstraction is defined as:

$$\alpha_{\texttt{parity}}(x \in \mathbb{Z}) = \begin{cases} \texttt{even} & \text{if } x \bmod 2 = 0 \\ \texttt{odd} & \text{if } x \bmod 2 = 1 \end{cases}$$

Even though $\mathbb{Z}$ is infinite, this abstraction maps all integers to a finite set $\{\texttt{odd}, \texttt{even}\}$.

5.1.2.2 Transformer Functions

Once we have values in the abstract domains, we often define an *abstract transformer function*

$$f^a : D^a \to D^a,$$

for each operation f to reason about its behavior on abstract values.

Example 5.1.4. Consider the function $f(x) = x + 1$. We define the abstract transformers f^a for different abstract domains D^a:

- **Odd/Even abstraction**: $D^a = \{\texttt{odd}, \texttt{even}\}$. Then:

$$f^a(\texttt{odd}) = \texttt{even}, \quad f^a(\texttt{even}) = \texttt{odd}$$

- **Sign abstraction**: $D^a = \{\texttt{neg}, \texttt{zero}, \texttt{pos}\}$. Then:

$$f^a(\texttt{neg}) = \{\texttt{neg}, \texttt{zero}\}, \quad f^a(\texttt{zero}) = \texttt{pos}, \quad f^a(\texttt{pos}) = \texttt{pos}$$

- **Interval abstraction**: $D^a = \{[a, b] \mid a \leq b \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$. Then:

$$f^a([a, b]) = [a + 1, b + 1]$$

Notice that the input and output of the abstract transformer f^a are both in the abstract domain D^a (e.g., odd/even, sign, or interval).

5.2 Abstract Domains

We now introduce several abstract domains that are commonly used in DNN verification. Each domain has its own abstract transformer functions to compute the bounds of neurons.

Note that the input to the transformer functions (§5.1.2) are the abstract values and the output is also an abstract value. For example, for interval transformers, the input is an interval $[l, u]$ and the output is also an interval $[l', u']$. Similarly for zonotope transformers, the input is a zonotope defined by center and generators, and the output is also a zonotope.

5.2.1 Interval

Interval is a very simple abstraction which represents the possible values of a variable as an interval $[l, u]$, where l is the lower bound and u is the upper bound. For example, the set of values $\{-2.5, -8.2, -10.7, 2, 4.7, 5.1\}$ can be represented as $[-10.7, 5.1]$.

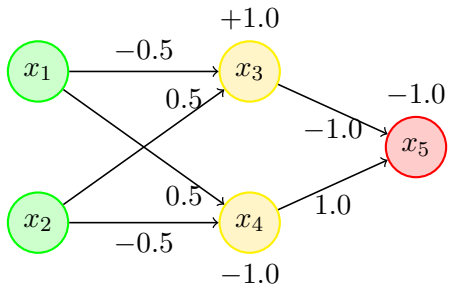


Figure 5.4: A simple DNN (similar to Fig. 1.1).

Definition The interval for one variable v is defined as:

$$v \in [l, u] = \{v \in \mathbb{R} \mid l \leq v \leq u\}$$

For n variables, the interval becomes a box (like a rectangle in 2D, a cupoid in 3D, or a hyperrectangle in nD):

$$[v_1, v_2, \ldots, v_n] \in [l_1, u_1] \times [l_2, u_2] \times \cdots \times [l_n, u_n]$$

5.2.1.1 Affine Transformer

For the linear or affine function f in §1.2

$$f(v_1, v_2, ..., v_n) = \sum_{i=1}^{n} w_i v_i + b$$

where w_i is the weight for the input v_i, n is the number of output nodes from the previous layer and b is the bias term, the abstract transformer f^a is:

$$f^a([l_1, u_1], ..., [l_n, u_n]) = [f^a_L, f^a_U] = \left[b + \sum_{i=1}^{n} \big(\min(w_i l_i, w_i u_i), \; b + \sum_{i=1}^{n} \big(\max(w_i l_i + w_i u_i) \right].$$

Example 5.2.1. Consider the DNN in Fig. 5.4 with inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$. The affine function for the neuron x_3 is:

$$f(x_1, x_2) = -0.5\, x_1 + 0.5\, x_2 + 1.0$$

Then the interval for x_3 can be computed as:

$$\begin{aligned} f^a([1,2],[-1,3]) = &\Big[1 + \min(-0.5 \cdot 1,\ -0.5 \cdot 2) + \min(0.5 \cdot -1,\ 0.5 \cdot 3), \\ &\quad 1 + \max(-0.5 \cdot 1,\ -0.5 \cdot 2) + \max(0.5 \cdot -1,\ 0.5 \cdot 3) \Big] \\ =&[1 - 1.0 - 0.5,\ 1 - 0.5 + 1.5] \\ =&[-0.5,\ 2.0] \end{aligned}$$

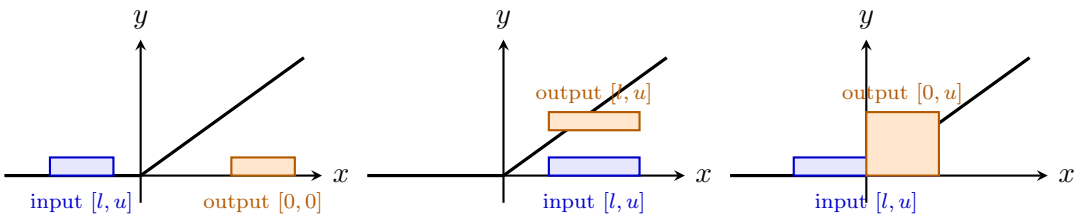


Figure 5.5: ReLU transformer for intervals. Left: all inputs negative $\Rightarrow [0, 0]$. Middle: all inputs positive $\Rightarrow$ unchanged. Right: interval crosses zero $\Rightarrow [0, u]$. The ReLU function clamps negative parts to zero while preserving positive regions.

5.2.1.2 ReLU Transformer

For $\mathrm{ReLU}(x) = \max(0, x)$, the abstract transformer is defined as:

$$\mathrm{ReLU}^a([l, u]) = [\mathrm{ReLU}(l), \mathrm{ReLU}(u)] = [\max(0, l), \max(0, u)]$$

This is equivalent to three cases shown in Fig. 5.5:

1. If $u < 0$, then $\mathrm{ReLU}^a([l, u]) = [0, 0]$. If inputs are negative, the output is also negative.
2. If $l \geq 0$, then $\mathrm{ReLU}^a([l, u]) = [l, u]$. If inputs are positive, the output is exactly the same.
3. If $l < 0 < u$, then $\mathrm{ReLU}^a([l, u]) = [0, u]$. If inputs are mixed, the output is approximated to $[0, u]$

Example 5.2.2. For Ex. 5.2.1, applying ReLU^a to neuron x_3 gives:

$$\mathrm{ReLU}^a([-0.5, 2.0]) = [\mathrm{ReLU}(-0.5), \mathrm{ReLU}(2.0)] = [0, 2.0].$$

Problem 5.2.1. Apply interval abstraction to the DNN in Fig. 5.4 with inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$. Compute the bounds for all neurons. Clearly indicate the steps (e.g., results after affine and ReLU transformers).

5.2.1.3 Efficiency and Precision

Intervals is very efficient and scales well to large networks. The affine transformer only requires a linear number of multiplications and min/max operations, and ReLU reduces to only three matching cases. However, the cost of efficiency is precision.

Example 5.2.3. Suppose we have $v_1 \in [0, 1]$, $v_2 = -v_1$, and $z = v_1 + v_2$. The concrete value of z would be always 0, but interval abstraction gives $z \in [-1, 1]$, which is a very loose over-approximation. Moreover, if we apply ReLU, then the output would be $\mathrm{ReLU}(z) = \mathrm{ReLU}(0) = 0$, but the interval gives $\mathrm{ReLU}^a([-1, 1]) = [0, 1]$, which is again a loose over-approximation.

Interval overapproximation grows quickly as we propagate through many layers of a large network, i.e., it keeps "inflating" the bounds, leading to a loose approximation of the output and becoming unable to verify valid properties. For example, $z \leq 0$ is a valid property for Ex. 5.2.3, but the interval abstractions gives $z \in [0, 1]$, which is not tight (precise) enough to show this property.

Nonetheless, interval domain remains popular due to its simplicity and efficiency. In some cases, despite being imprecise, it can still successfully verify properties of neural networks, e.g., for Ex. 5.2.3 if we want to verify that $z \leq 2$, then the interval $z \in [0, 1]$ would suffice.

Problem 5.2.2. Suppose y can take values from 0 to 3. Student A computes an interval abstraction $y \in [-2, 4]$, while student B computes $y \in [1, 2]$.

1. Which student's abstraction is correct? If both are correct, which is more precise? Explain your answer.

2. Using the correct abstraction (if both are correct, use the more precise one), can we use it verify the property $y \leq 3$? How about $y \leq 5$? Explain your answers.

5.2.2 Zonotope

Interval abstraction (§5.2.1) treats each variable independently. For example, if $x_1 \in [1, 2]$ and $x_2 \in [3, 4]$, interval assumes any combination of x_1 and x_2 is possible. But variables can correlate: e.g., when x_1 increases, x_2 also increases. Zonotopes can capture such correlations and therefore provide a tighter abstraction.

Definition A zonotope is often represented using a generator-based representation as illustrated in §5.1.1. It is built by starting from a center point and then stretching along several directions defined by generator vectors. Each generator can move forward or backward within a certain range, and together these movements sweep out the entire shape. Zonotope generalizes the idea of an interval, which itself is a zonotope (e.g., see Ex. 5.1.1).

Formally, a **zonotope** $\mathcal{Z}$ in $\mathbb{R}^n$ is defined as:

$$\mathcal{Z} = \left\{ c + \sum_{i=1}^{n} \epsilon_i g_i \mid \epsilon_i \in [-1, 1] \right\}$$

where:

- $c \in \mathbb{R}^n$ is the *center* (analogous to the midpoint of an interval),
- $g_i \in \mathbb{R}^n$ are *generator vectors* (directions of variability), and

Figure 5.6: Left: independent generators produce an axis-aligned rectangle (interval abstraction). Right: correlated generators tilt the shape into a parallelogram, capturing dependency between x_1 and x_2.

- $\epsilon_i \in [-1, 1]$ are independent coefficients that determine how much we move along each generator. For example, $\epsilon_i = -1$ moves in the negative direction, $\epsilon_i = 0$ stays at the center, and $\epsilon_i = +1$ moves in the positive direction.

Example 5.2.4. A common zonotope in 2D is a parallelogram. It can be defined by its center point and two generator vectors that define its sides. For instance, parallelogram on the left side of Fig. 5.6 has the center $c = (1, 1)$ and the generators $g_1 = (1, 0.25)$ and $g_2 = (0.5, 1)$. This zonotope represents all points that can be reached by moving along these generators within the range $[-1, 1]$:

$$\mathcal{Z} = \{(1, 1) + \epsilon_1(1, 0.25) + \epsilon_2(0.5, 1) \mid \epsilon_1, \epsilon_2 \in [-1, 1]\}$$

This zonotope includes points like:

$$(1, 1) + (-1)(1, 0.25) + (1)(0.5, 1) = (-1, 2.5)$$

$$(1, 1) + (1)(1, 0.25) + (-1)(0.5, 1) = (3, -0.5)$$

Example 5.2.5. In Fig. 5.6 the parallelogram zonotope on the right of captures points with correlated x_1 and x_2 values, e.g., increasing x_1 by moving along g_1 also increases x_2 due to the non-zero second component of g_1. In contrast, the rectangle zonotope on the left, which represents an interval abstraction, has independent generators, i.e., changing x_1 does not affect x_2, and therefore cannot capture the correlation.

5.2.3 Computing Bounds of a Zonotope

In DNN verification, we often need to compute the bounds of each variable represented by a zonotope. This allows us to determine stable neurons, which in turn helps optimize the verification process.

For each coordinate (dimension) of the zonotope, we compute its lower and upper bounds as follows:

$$l_j = \min_{\epsilon_i \in [-1,1]} \left(c_j + \sum_{i=1}^{n} \epsilon_i g_{i,j} \right), \quad u_j = \max_{\epsilon_i \in [-1,1]} \left(c_j + \sum_{i=1}^{n} \epsilon_i g_{i,j} \right),$$

where c_j is the j-th component of the center and $g_{i,j}$ is the j-th component of the i-th generator.

A simpler way. Since each $\epsilon_i \in [-1, 1]$, each term $\epsilon_i g_{i,j}$ can vary between $-|g_{i,j}|$ and $+|g_{i,j}|$. Hence:

$$\boxed{l_j = c_j - \sum_{i=1}^{n} |g_{i,j}|, \qquad u_j = c_j + \sum_{i=1}^{n} |g_{i,j}|.}$$

Example (1D). Given the interval zonotope:

$$x = 3 + \epsilon_1(2), \quad \epsilon_1 \in [-1, 1],$$

the bounds are computed as:

$$[l, u] = [3 - |2|, 3 + |2|] = [1, 5].$$

Example (2D).

$$\mathcal{Z} = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \epsilon_1 \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} + \epsilon_2 \begin{bmatrix} -0.3 \\ 0.2 \end{bmatrix}, \quad \epsilon_1, \epsilon_2 \in [-1, 1] \right\}.$$

Then for each coordinate:

$$\begin{aligned} l_1 &= 1 - (|0.5| + |-0.3|) = 0.2, \quad & u_1 &= 1 + (|0.5| + |-0.3|) = 1.8, \\ l_2 &= 2 - (|1.0| + |0.2|) = 0.8, & u_2 &= 2 + (|1.0| + |0.2|) = 3.2. \end{aligned}$$

5.2.3.1 Affine Transformer

For the affine function f in §1.2

$$f(v_1, v_2, \ldots, v_n) = \sum_{i=1}^{n} w_i v_i + b$$

where w_i is the weight associated with input v_i, n is the number of inputs, and b is the bias term.

Applying an affine transformation to a zonotope transforms its center by the same affine rule (weights and bias) and transforms each generator by the linear part

(weights only). In other words, affine operations *preserve* the structure of zonotopes exactly: no approximation needed. Imagine a

More formally, given a zonotope

$$\mathcal{Z} = \left\{ c + \sum_{j=1}^{n} \epsilon_j g_j \mid \epsilon_j \in [-1, 1] \right\},$$

the abstract transformer f^a for the affine function computes the new zonotope as:

$$f^a(\mathcal{Z}) = \left\{ f(c) + \sum_{j=1}^{n} \epsilon_j f(g_j) \mid \epsilon_j \in [-1, 1] \right\}$$

That is, the new zonotope has:

- **center:** $f(c) = \sum_{i=1}^{n} w_i c_i + b$
- **generators:** each $f(g_j) = \sum_{i=1}^{n} w_i (g_j)_i$

Example 5.2.6. Consider the DNN in Fig. 5.4 with inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$. The affine function for neuron x_3 is:

$$f(x_1, x_2) = -0.5\, x_1 + 0.5\, x_2 + 1.0$$

We want to compute the bounds for x_3 using zonotope abstraction.

Represent the input as a zonotope We represent the input intervals $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$ as a zonotope $\{c + \sum_{i=1}^{n} \epsilon_i g_i \mid \epsilon_i \in [-1, 1]\}$.

The center c_i represents the midpoint of each input interval:

$$c_1 = \frac{1+2}{2} = 1.5, \quad c_2 = \frac{-1+3}{2} = 1.0,$$

and the generator g_i captures the half-width of each interval:

$$g_1 = \frac{2-1}{2} = 0.5, \quad g_2 = \frac{3-(-1)}{2} = 2.0$$

Thus, the input zonotope is

$$\mathcal{Z} = \{(1.5, 1.0) + \epsilon_1(0.5, 0) + \epsilon_2(0, 2.0), \quad \epsilon_1, \epsilon_2 \in [-1, 1]\}.$$

Apply the affine transformation We apply the affine function $f(x_1, x_2) = -0.5\,x_1 + 0.5\,x_2 + 1.0$, to the input zonotope $\mathcal{Z}$:

$$\begin{aligned} f^a(\mathcal{Z}) &= f(c_1, c_2) + \epsilon_1\, f(g_1) + \epsilon_2\, f(g_2) \\ &= \big(-0.5\,c_1 + 0.5\,c_2 + 1.0\big) + \epsilon_1\big(-0.5\,g_{1x} + 0.5\,g_{1y}\big) + \epsilon_2\big(-0.5\,g_{2x} + 0.5\,g_{2y}\big) \\ &= \big(-0.5(1.5) + 0.5(1.0) + 1.0\big) + \epsilon_1\big(-0.5(0.5) + 0.5(0)\big) + \epsilon_2\big(-0.5(0) + 0.5(2.0)\big) \\ &= 0.75 - 0.25\,\epsilon_1 + 1.0\,\epsilon_2. \end{aligned} \tag{5.2.1}$$

Hence, the output zonotope for x_3 is

$$\{0.75 + \epsilon_1(-0.25) + \epsilon_2(1.0), \quad \epsilon_1, \epsilon_2 \in [-1, 1]\}.$$

Compute output bounds From the output zonotope, we can compute the lower and upperbounds for x_3 by subtracting and adding the absolute values of the generator coefficients:

$$\begin{aligned} f^a_L &= c' - (|g'_1| + |g'_2|) = 0.75 - (0.25 + 1.0) = -0.5, \\ f^a_U &= c' + (|g'_1| + |g'_2|) = 0.75 + (0.25 + 1.0) = 2.0. \end{aligned} \tag{5.2.2}$$

Thus, the output range is $x_3 \in [-0.5, 2.0]$.

The reason for computing the bounds from the zonotope representation is to determine stable neurons as described in §4.2.2. If the lower bound is non-negative, ReLU will be active; if the upper bound is non-positive, ReLU will be inactive. In these cases, ReLU can be simplified to either the identity function or zero, respectively, which helps reduce overapproximation in subsequent layers.

Note that for this simple example, both interval (§5.2.1.1) and zonotope abstractions yield identical bounds $[-0.5, 2.0]$. The benefit of zonotopes becomes clear in deeper layers, where they preserve linear correlations that interval abstractions lose.

5.2.3.2 ReLU Transformer

Activation functions like ReLU are non-affine and do not preserve zonotope structure. For $\text{ReLU}(x) = \max(0, x)$, the shape of the input region changes at $x = 0$: $x < 0$ maps to 0 (a flat line), while $x \geq 0$ maps to x (a slanted line). Thus, we will overapproximate these shapes with a new zonotope—a parallelogram—that contains the entire ReLU output region.

We can distinguish three possible cases of ReLU depending on the bounds $[l, r]$ of the input zonotope.

- **Active case** ($l \geq 0$): All values are non-negative, so $\text{ReLU}^a(x) = x$ (identity function), i.e., the output zonotope is the same as the input zonotope.
- **Inactive case** ($u \leq 0$): All values are negative, so $\text{ReLU}^a(x) = \{0\}$, i.e., the output zonotope is a single point at zero.

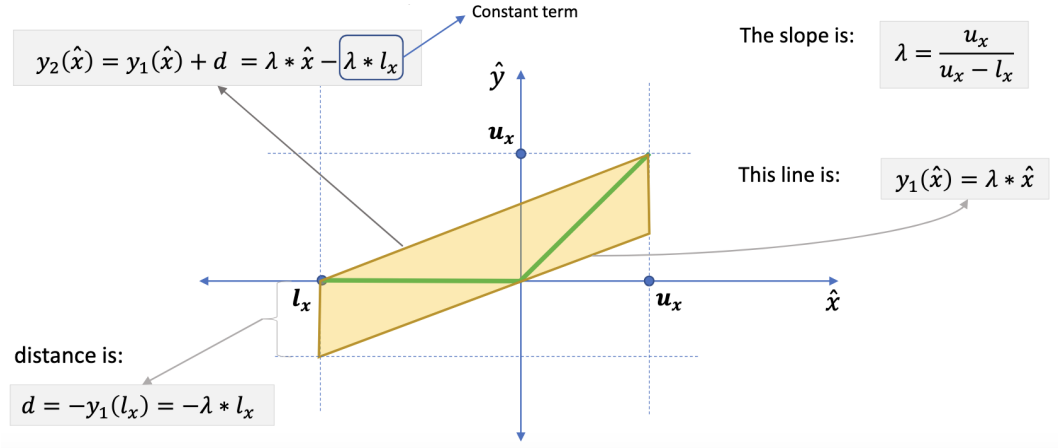


Figure 5.7: Lower and upper bounds computation for a zonotope input.

- **Unstable case** ($l < 0 < u$): The range crosses zero, requiring over-approximation using a parallelogram that bounds ReLU over $[l, u]$ as described below.

Building parallelogram Given the input zonotope $x = \{(c + \sum_{i=1}^{n} \epsilon_i g_i) \mid \epsilon_i \in [-1, 1]\}$ with x ranging over $[l, u]$ where $l < 0 < u$ (i.e., the unstable case), we construct a parallelogram that bounds the ReLU function over this interval. We capture the parallelogram using two linear constraints representing its upper and lower edges as shown in Fig. 5.7.

- **Upper bound constraint:** $y \leq \lambda x - \lambda l$, where $\lambda = \frac{u}{u-l}$ is the slope of the line connecting points $(l, 0)$ and (u, u)
- **Lower bound constraint:** $y \geq \lambda x$ is the line with the same slope λ passing through $(0, 0)$.

Mergining these two constraints, we get the following equation describing all points (x, y) within the parallelogram:

$$\begin{aligned} \lambda x \leq \quad & y \quad \leq \lambda x - \lambda l \\ \Leftrightarrow \quad & y = \lambda x - \alpha \lambda l, \quad \alpha \in [0, 1] \end{aligned}$$

The parameter α interpolates between the lower and upper bounds of the parallelogram. When $\alpha = 0$, we are on the lower bound $y = \lambda x$; when $\alpha = 1$, we are on the upper bound $y = \lambda x - \lambda l$.

Our zonotope requires $\epsilon_i \in [-1, 1]$, thus we substitute:

$$\alpha = \frac{1 + \epsilon_{new}}{2}, \quad \epsilon_{new} \in [-1, 1]$$

For example, when $\epsilon_{new} = -1$, we have $\alpha = 0$ (lower bound); when $\epsilon_{new} = 1$, we have $\alpha = 1$ (upper bound).

The output zonotope after ReLU transformation is then:

$$\begin{aligned} y &= \lambda x - \alpha\lambda l \\ &= \lambda x - \frac{1+\epsilon_{new}}{2}\lambda l \\ &= \lambda \underbrace{\left(c + \sum_{i=1}^{m} \epsilon_i g_i\right)}_{x} - \epsilon_{new}\frac{\lambda l}{2} - \frac{\lambda l}{2} \\ &= \underbrace{\left(\lambda c - \frac{\lambda l}{2}\right)}_{\text{new center}} + \underbrace{\sum_{i=1}^{m} \epsilon_i(\lambda g_i) - \epsilon_{new}\frac{\lambda l}{2}}_{\text{new generator}} \end{aligned}$$

Example 5.2.7 (ReLU on zonotope)**.** Continuing from Ex. 5.2.6, applying ReLU to the zonotope output

$$x_3 = 0.75 + \epsilon_1(-0.25) + \epsilon_2(1.0), \quad \epsilon_1, \epsilon_2 \in [-1, 1].$$

Determine the bounds of the input zonotope As shown in Ex. 5.2.6, the output zonotope (before ReLU) has bounds $[-0.5, 2.0]$. So the input zonotope to ReLU has bounds $[-0.5, 2.0]$. Since $l = -0.5 < 0 < 2.0 = u$, this is an unstable neuron requiring over-approximation.

Approximate Slope We compute the slope of the upper bound line of the parallelogram (which connects points $(-0.5, 0)$ and $(2.0, 2.0)$):

$$\lambda = \frac{u-0}{u-l} = \frac{2.0}{2.0-(-0.5)} = \frac{2.0}{2.5} = 0.8.$$

Transform zonotope The new center becomes:

$$c' = \lambda c - \frac{\lambda l}{2} = 0.8 \cdot 0.75 - \frac{0.8 \cdot (-0.5)}{2} = 0.6 + 0.2 = 0.8$$

Existing generators are scaled by λ:

$$\begin{aligned} g_1' &= \lambda g_1 = 0.8 \cdot (-0.25) = -0.2, \\ g_2' &= \lambda g_2 = 0.8 \cdot 1.0 = 0.8 \end{aligned}$$

Create new generator for approximation error A new generator is introduced to capture the ReLU approximation error:

$$g_3' = -\frac{\lambda l}{2} = -\frac{0.8 \cdot (-0.5)}{2} = 0.2$$

Result: The output zonotope is

$$0.8 + \epsilon_1(-0.2) + \epsilon_2(0.8) + \epsilon_3(0.2), \quad \epsilon_i \in [-1, 1].$$

with concrete bounds:

$$l' = 0.8 - 0.2 - 0.8 - 0.2 = -0.4 \to 0 \text{ (clamp negative values)}$$
$$u' = 0.8 + 0.2 + 0.8 + 0.2 = 2.0$$

Thus, the final bounds after ReLU are $[0, 2.0]$.

Example 5.2.8 (End-to-end Example)**.** Consider the DNN in Fig. 5.4 with inputs $x_1 \in [-2, 2]$ and $x_2 \in [-1, 3]$.

Table 5.1: Interval vs Zonotope Analysis Comparison

Layer	**Interval**	**Zonotope**
Input	$\boxed{x_1 \in [-2, 2]}$ $\boxed{x_2 \in [-1, 3]}$	$c_1 = \frac{l+u}{2} = \frac{-2+2}{2} = 0$ $c_2 = \frac{l+u}{2} = \frac{-1+3}{2} = 1$ $\boxed{c = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}}$ $g_1 = \frac{u-l}{2} = \frac{2-(-2)}{2} = 2$ $g_2 = \frac{u-l}{2} = \frac{3-(-1)}{2} = 2$ $\boxed{G = \text{diag}([g_1, g_2]) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}}$
Affine	$\boxed{f(x_1, x_2) = -0.5 \cdot x_1 + 0.5 \cdot x_2 + 1}$ $l_3 = f_L([-2, 2], [-1, 3])$ $= 1 + \min(-0.5 \cdot (-2), -0.5 \cdot 2)$ $+ \min(0.5 \cdot (-1), 0.5 \cdot 3)$ $= 1 + (-1.0) + (-0.5) = -0.5$ $u_3 = f_U([-2, 2], [-1, 3])$ $= 1 + \max(-0.5 \cdot (-2), -0.5 \cdot 2)$ $+ \max(0.5 \cdot (-1), 0.5 \cdot 3)$ $= 1 + 1.0 + 1.5 = 3.5$ $\boxed{x_3 \in [-0.5, 3.5]}$	$\boxed{f(x_1, x_2) = -0.5 \cdot x_1 + 0.5 \cdot x_2 + 1}$ $c_3 = Wc + b$ $= \begin{bmatrix} -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + 1 = 1.5$ $G_3 = WG$ $= \begin{bmatrix} -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} -1 & 1 \end{bmatrix}$ $e_3 = \lvert G_3 \rvert = \lvert -1 \rvert + \lvert 1 \rvert = 2$ $l_3 = c_3 - e_3 = 1.5 - 2 = -0.5$ $u_3 = c_3 + e_3 = 1.5 + 2 = 3.5$ $\boxed{x_3 \in [-0.5, 3.5]}$

	$\boxed{f(x_1, x_2) = 0.5 \cdot x_1 - 0.5 \cdot x_2 - 1}$ $l_4 = f_L([-2, 2], [-1, 3])$ $= -1 + \min(0.5 \cdot (-2), 0.5 \cdot 2)$ $+ \min(-0.5 \cdot (-1), -0.5 \cdot 3)$ $= -1 + (-1.0) + (-1.5) = -3.5$ $u_4 = f_U([-2, 2], [-1, 3])$ $= -1 + \max(0.5 \cdot (-2), 0.5 \cdot 2)$ $+ \max(-0.5 \cdot (-1), -0.5 \cdot 3)$ $= -1 + 1.0 + 0.5 = 0.5$ $\boxed{x_4 \in [-3.5, 0.5]}$	$\boxed{f(x_1, x_2) = 0.5 \cdot x_1 - 0.5 \cdot x_2 - 1}$ $c_4 = Wc + b$ $= \begin{bmatrix} 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} + (-1) = -1.5$ $G_4 = WG$ $= \begin{bmatrix} 0.5 & -0.5 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix}$ $e_4 = \lvert G_4 \rvert = \lvert 1 \rvert + \lvert -1 \rvert = 2$ $l_4 = c_4 - e_4 = -1.5 - 2 = -3.5$ $u_4 = c_4 + e_4 = -1.5 + 2 = 0.5$ $\boxed{x_4 \in [-3.5, 0.5]}$
ReLU	$x'_3 = \text{ReLU}(x_3)$ $= \text{ReLU}([-0.5, 3.5])$ $= [\max(0, -0.5), \max(0, 3.5)]$ $\boxed{x'_3 \in [0, 3.5]}$	$\lambda_3 = \dfrac{u_3}{u_3 - l_3} = \dfrac{3.5}{3.5 - (-0.5)} = 0.875$ $g_3 = \dfrac{(1 - \lambda_3)}{2} \cdot u_3 = \dfrac{(1 - 0.875)}{2} \cdot 3.5 = 0.21875$ $c'_3 = \lambda_3 \cdot c_3 + g_3$ $= 0.875 \cdot 1.5 + 0.21875 = 1.53125$ $G'_3 = \begin{bmatrix} \lambda_3 \cdot G_3 & g_3 \end{bmatrix}$ $= \begin{bmatrix} -0.875 & 0.875 & 0.21875 \end{bmatrix}$ $e'_3 = \lvert G'_3 \rvert = \lvert -0.875 \rvert + \lvert 0.875 \rvert + \lvert 0.21875 \rvert$ $= 1.96875$ $l'_3 = c'_3 - e'_3 = 1.53125 - 1.96875 = -0.4375$ $u'_3 = c'_3 + e'_3 = 1.53125 + 1.96875 = 3.5$ $\boxed{x'_3 \in [-0.4375, 3.5]}$
	$x'_4 = \text{ReLU}(x_4)$ $= \text{ReLU}([-3.5, 0.5])$ $= [\max(0, -3.5), \max(0, 0.5)]$ $\boxed{x'_4 \in [0, 0.5]}$	$\lambda_4 = \dfrac{u_4}{u_4 - l_4} = \dfrac{0.5}{0.5 - (-3.5)} = 0.125$ $g_4 = \dfrac{(1 - \lambda_4)}{2} \cdot u_4 = \dfrac{(1 - 0.125)}{2} \cdot 0.5 = 0.21875$ $c'_4 = \lambda_4 \cdot c_4 + g_4$ $= 0.125 \cdot (-1.5) + 0.21875 = 0.03125$ $G'_4 = \begin{bmatrix} \lambda_4 \cdot G_4 & g_4 \end{bmatrix}$ $= \begin{bmatrix} 0.125 & -0.125 & 0.21875 \end{bmatrix}$ $e'_4 = \lvert G'_4 \rvert = \lvert 0.125 \rvert + \lvert -0.125 \rvert + \lvert 0.21875 \rvert$ $= 0.46875$ $l'_4 = c'_4 - e'_4 = 0.03125 - 0.46875 = -0.4375$ $u'_4 = c'_4 + e'_4 = 0.03125 + 0.46875 = 0.5$ $\boxed{x'_4 \in [-0.4375, 0.5]}$

Affine	$\boxed{f(x'_3, x'_4) = -1 \cdot x'_3 + 1 \cdot x'_4 + (-1)}$ $l_5 = f_L([0, 3.5], [0, 0.5])$ $= -1 + \min(-1 \cdot 0, -1 \cdot 3.5)$ $+ \min(1 \cdot 0, 1 \cdot 0.5)$ $= -1 + (-3.5) + 0 = -4.5$ $u_5 = f_U([0, 3.5], [0, 0.5])$ $= -1 + \max(-1 \cdot 0, -1 \cdot 3.5)$ $+ \max(1 \cdot 0, 1 \cdot 0.5)$ $= -1 + 0 + 0.5 = -0.5$ $\boxed{x_5 \in [-4.5, -0.5]}$	$\boxed{f(x'_3, x'_4) = -1 \cdot x'_3 + 1 \cdot x'_4 + (-1)}$ $c'_5 = \begin{bmatrix} c'_3 \\ c'_4 \end{bmatrix} = \begin{bmatrix} 1.53125 \\ 0.03125 \end{bmatrix}$ $G'_5 = \begin{bmatrix} G'_3 \\ G'_4 \end{bmatrix} = \begin{bmatrix} -0.875 & 0.875 & 0.21875 \\ 0.125 & -0.125 & 0.21875 \end{bmatrix}$ $c_5 = W c'_5 + b$ $= \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} 1.53125 \\ 0.03125 \end{bmatrix} + (-1) = -2.5$ $G_5 = W G'_5$ $= \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} -0.875 & 0.875 & 0.21875 \\ 0.125 & -0.125 & 0.21875 \end{bmatrix}$ $= \begin{bmatrix} 1.0 & -1.0 & 0.0 \end{bmatrix}$ $e_5 = \lvert G_5 \rvert = \lvert 1.0 \rvert + \lvert -1.0 \rvert + \lvert 0.0 \rvert = 2.0$ $l_5 = c_5 - e_5 = -2.5 - 2 = -4.5$ $u_5 = c_5 + e_5 = -2.5 + 2 = -0.5$ $\boxed{x_5 \in [-4.5, -0.5]}$

5.3 Polytope

Part III

The Branch and Bound Approach

Chapter 6

The Branch and Bound Search Algorithm

As shown in §4, the NNV verification problem can be formulated as a satisfiability problem, solvable using a constraint solver, e.g., SMT and MILP solvers. However, such solvers not scale to large DNNs due to the complexity of the underlying formulae. Thus, modern DNN verification techniques reframe the problem to search for *activation patterns* that satisfy the constraints, and use the *Branch-and-Bound* algorithm to explore the space of possible activation patterns.

6.1 Activation Pattern Search

DNNs with piecewise linear activation functions have a special structure that can be exploited for verification. For example, each ReLU (§1.3.1) function has two *activation statuses*: active (identity function) and inactive (zero function). Each status partitions the input space into two regions—active and inactive. Within each region, the DNN behavior can be encoded as *a linear constraint*, which can be efficiently analyzed.

Thus, verifying a DNN reduces to checking that none of these linear regions contains a counterexample. More specifically, we can rewrite the satisfiability formulation in Eq. 3.2.3 as a disjunction:

$$\bigvee_{p \in P} (\alpha_p \wedge \phi_{in} \wedge \neg \phi_{out}) \tag{6.1.1}$$

where P is the set of all possible *activation patterns*—boolean assignments of activation statuses of all neurons– of the DNN, and α_p is the formula α restricted to the linear region defined by the activation pattern p. This means that α_p includes additional constraints that fix the activation status of each neuron according to p. For example, if p specifies that neuron n_i is active, then α_p includes the constraint $z_i \geq 0$,

indicating that the pre-activation value z_i of neuron n_i is non-negative. Similarly, if n_i is inactive, then α_p includes $z_i < 0$.

As long as one of the disjuncts in Eq. 6.1.1 is satisfiable, i.e., we could find a counterexample in one of the linear regions, the entire formula is satisfiable, indicating that the property is invalid. Conversely, if all disjuncts are unsatisfiable, the property is valid.

This allows us to break down the verification problem into smaller subproblems, each searches for an activation pattern that satisfies the formula in Eq. 3.2.3. Modern DNN verification techniques [3, 11, 20, 23, 25, 32, 44, 52] all adopt this idea and search for a satisfying activation pattern to find a counterexample.

6.1.1 Activation Patterns

Definition 6.1.1 (Activation Pattern)**.** Let N be a DNN with ReLU neurons $n_1, \ldots, n_m$. An *activation pattern* is a Boolean assignment of the activation status of a subset (or all) of the neurons. If a neuron n_i is active, we set $b_i = \texttt{true}$ (meaning $z_i \geq 0$); if it is inactive, $b_i = \texttt{false}$ (meaning $z_i < 0$).

Definition 6.1.2 (Complete Activation Pattern)**.** A *complete activation pattern* assigns an activation status to *every* neuron in the DNN:

$$p = \langle b_1, b_2, \ldots, b_m \rangle \in \{\texttt{true}, \texttt{false}\}^m.$$

Definition 6.1.3 (Partial Activation Pattern)**.** A *partial activation pattern* assigns activation statuses to only a *subset* of the neurons.

$$q = \langle b_1, b_2, *, b_4, *, \ldots, b_m \rangle \in \{\texttt{true}, \texttt{false}, *\}^m,$$

where $*$ means "undetermined" or "don't care". Each partial activation pattern represents multiple complete activation patterns.

Example 6.1.1. Consider a DNN with three ReLU neurons n_1, n_2, n_3. A complete activation pattern might be

$$p = \langle \texttt{true}, \texttt{false}, \texttt{true} \rangle,$$

which means n_1 and n_3 are active while n_2 is inactive.

In contrast, a partial activation pattern such as

$$q = \langle \texttt{true}, \texttt{false}, * \rangle$$

represents *both* complete patterns $\langle \texttt{true}, \texttt{false}, \texttt{true} \rangle$ and $\langle \texttt{true}, \texttt{false}, \texttt{false} \rangle$.

Definition 6.1.4 (Empty Activation Pattern)**.** An *empty* activation pattern is a partial pattern that assigns *no* activation statuses to any neurons:

$$p = \langle *, *, \ldots, * \rangle \in \{\texttt{true}, \texttt{false}, *\}^m.$$

It thus represents all 2^m complete activation patterns.

Problem 6.1.1 (Pattern Enumeration)**.** Consider an NN with 3 ReLU neurons:

1. How many complete activation patterns are there?
2. How many partial patterns of size 2 (i.e., fixing 2 neurons but leaving 1 unspecified) are there?
3. Give an explicit example of one partial pattern and list all the complete patterns it represents.

Problem 6.1.2 (Counting Complete Patterns Compatible with a Partial Pattern)**.** Suppose we have a DNN with $m = 10$ ReLU neurons. Consider a partial pattern p' that fixes 4 neurons.

1. How many complete patterns extend p'?
2. Suppose we restrict p' further by adding 2 more neuron assignments. How many extensions now?
3. Generalize: If p' fixes k neurons out of m, how many complete patterns are there?

6.1.2 Set Notation of Activation Patterns

We can use set notation to represent activation patterns more concisely by listing only the neurons whose activation statuses are fixed. For example, the activation pattern

$$p = \langle \texttt{true}, \texttt{false}, *, \texttt{true}, \texttt{false} \rangle$$

can be represented as

$$p = \{n_1 = \texttt{true}, n_2 = \texttt{false}, n_3 = *, n_4 = \texttt{true}, n_5 = \texttt{false}\},$$

or more compactly

$$p = \{n_1, \overline{n_2}, n_4, \overline{n_5}\},$$

where n_i indicates that neuron n_i is active, $\overline{n_i}$ indicates that it is inactive, and the absence of n_3 indicates that its activation status is unspecified.

Thus, given a set of neurons, we can construct an activation pattern by specifying which neurons are active and which are inactive. If the set includes all neurons, it is a complete pattern; otherwise, it is a partial pattern (and if it is an empty set $\emptyset$, it is the empty pattern as shown in Def. 6.1.4).

6.1.3 State Space Reduction

Once having an activation pattern p, we can simplify the formula α representing the DNN by replacing each ReLU function with a linear constraint according to the activation status specified in p. For example, if p specifies that neuron n_i is active, we replace $\text{ReLU}(z_i)$ with the constraint $z_i \geq 0$, i.e., the pre-activation value z_i must be non-negative. Otherwise, if n_i is inactive, we replace $\text{ReLU}(z_i)$ with $z_i < 0$. This gives us the formula α_p corresponding to the linear region defined by p.

A complete activation pattern p defines a unique linear region of the DNN. This is because with a complete pattern p, α_p becomes a purely linear formula. In contrast, a partial activation pattern q defines multiple linear regions, one for each complete pattern it represents. With q, α_q may still contain some ReLU functions that are not fixed by q. However, since q fixes the status of some neurons, we can simplify α by replacing the ReLU functions of those neurons with linear constraints.

In any case, given an activation pattern, we can reduce the complexity of the satisfiability check by fixing the activation status of some neurons. Thus, checking the satisfiability of $\alpha_p \wedge \phi_{in} \wedge \neg\phi_{out}$ is easier than checking that of $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$.

Example 6.1.2. Recall the DNN from Fig. 3.1 can be represented as the formula α

$$\begin{aligned}
\hat{x}_3 &= -0.5x_1 + 0.5x_2 + 1.0 \wedge \\
\hat{x}_4 &= 0.5x_1 - 0.5x_2 + 1.0 \wedge \\
x_3 &= \text{ReLU}(\hat{x}_3) \wedge \\
x_4 &= \text{ReLU}(\hat{x}_4) \wedge \\
x_5 &= -x_3 + x_4 - 1.0,
\end{aligned}$$

Here $\hat{x}_3$ and $\hat{x}_4$ are the pre-activation values of the ReLU neurons x_3 and x_4, respectively. Thus, if we fix the activation status of x_3 and x_4, we can simplify α by replacing the ReLUs with linear constraints.

- **Complete activation pattern.** A complete activation pattern specifies the status of both ReLU neurons. For instance,

  $$p = \{x_3, \overline{x_4}\}$$

  means that x_3 is active while x_4 is inactive; that is, $\hat{x}_3 \geq 0$ (so $x_3 = \hat{x}_3$) while $\hat{x}_4 < 0$ (so $x_4 = 0$). In this case, α reduces to a single linear constraint (region):

  $$x_5 = -(-0.5x_1 + 0.5x_2 + 1.0) + 0 - 1.0 = 0.5x_1 - 0.5x_2 - 2.0.$$

- **Partial activation pattern.** A partial pattern specifies only some activation statuses. For example,

  $$q = \{x_3\}$$

means $\hat{x}_3 \geq 0$ but places no restriction on $\hat{x}_4$. Here, α reduces to

$$\begin{aligned}
\hat{x}_4 &= 0.5x_1 - 0.5x_2 + 1.0 \wedge \\
x_4 &= \text{ReLU}\,(\hat{x}_4) \ \wedge \\
x_5 &= -(-0.5x_1 + 0.5x_2 + 1.0) + x_4 - 1.0,
\end{aligned}$$

Because q does not fix the status of x_4, we cannot simplify the ReLU for x_4. Thus, the formula still contains a ReLU function which splits x_4 into two linear regions corresponding to the two possible activation statuses of x_4.

Problem 6.1.3. Consider the DNN in Fig. 3.1. Suppose we fix the activation pattern $p = \{x_3, x_4\}$.

1. Provide the constraints α induced by p.
2. Consider an invalid property $x_5 > 0$ for inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$. Find an activation pattern that satisfies the formula $\alpha_p \wedge \phi_{in} \wedge \neg\phi_{out}$. In other words, find a pattern that contains a counterexample to the property.

6.2 The Branch and Bound Algorithm

Many modern DNN verifiers adopt the Branch-and-Bound (BaB) approach to explore the space of possible neuron activation patterns (§6.1). BaB splits (*branch*) the verification problem into smaller subproblems by fixing activation statuses of neurons, and uses abstraction (*bound*) techniques to compute upper and lower bounds on the output values for these subproblems. If the bounds, computed using abstraction (§5), indicate infeasible (cannot contain a counterexample), the subproblem is pruned from the search space. This process continues until either a counterexample is found or all subproblems are exhausted, proving the property holds.

Note that because BaB splits ReLU neurons into active/inactive cases, it is also called *"neuron-splitting"*, which contrasts with *"input-splitting"* techniques (§9.1) that partition the input space.

Reference Alg Alg. 1 shows $\texttt{BaB}_{\texttt{NNV}}$, a reference [40] BaB architecture for DNN verification. $\texttt{BaB}_{\texttt{NNV}}$ takes as input a ReLU-based DNN $\mathcal{N}$ and a formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property of interest. $\texttt{BaB}_{\texttt{NNV}}$ maintains a set of activation patterns (ActPatterns) that represent the current activation pattern of the network. Initially, ActPatterns is initialized with an empty activation pattern (line 1).

In each BaB iteration i (line 2), the algorithm selects and removes an activation pattern σ_i from ActPatterns (line 3). It then calls `Deduce` to *quickly* determine if the current problem, i.e., the original satisfiability problem with activation pattern σ_i, is feasible. For example, it can use interval abstraction (§5) to quickly computes the

Algorithm 1. The $\texttt{BaB}_{\texttt{NNV}}$ algorithm.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$
output : unsat if property is valid, otherwise (sat, cex)

```
1  ActPatterns ← {∅} // initialize verification problems
2  while ActPatterns do // main loop
3      σ_i ← Select(ActPatterns) // process problem i-th
4      if Deduce(N, φ_in, φ_out, σ_i) then
5          cex ← LP(N, φ_in, φ_out, σ_i) // check satisfiability
6          if cex then // found a valid cex
7              return (sat, cex)
8          v_i ← Decide(N, σ_i)
           // create new activation patterns
9          ActPatterns ← ActPatterns ∪ {σ_i ∪ {v_i} ; σ_i ∧ {¬v_i}}
10 return unsat
```

bounds of the output values with respect to σ_i, e.g., by replacing ReLU functions according to the activation statuses specified in σ_i (§6.1.3).

If the computed bounds indicate that no counterexample can exist, e.g., the lower bound of the output is greater than 0 when checking $y < 0$, then the problem is *infeasible*. Otherwise, the problem is *potentially feasible* (because the bounds are over-approximations).

(Note that we do this even on the empty activation pattern, which is the initial state of the search, because the problem might be trivially infeasible, in which case we can exit and return `unsat` immediately (line 10).)

- If `Deduce` determines that the problem is infeasible, $\texttt{BaB}_{\texttt{NNV}}$ discards the current processing activation pattern and loops back (to line 2) to process the next activation pattern. In other words, it prunes the current search branch with activation pattern σ_i. When there are no more activation patterns to process, $\texttt{BaB}_{\texttt{NNV}}$ returns `unsat` (line 10), indicating that it cannot find counterexamples and the property is valid.

- If `Deduce` determines that the problem is feasible, $\texttt{BaB}_{\texttt{NNV}}$ checks the satisfiability of the problem using an LP solver (line 5)[1]. If the solver finds a satisfying assignment, it returns `sat` and the counterexample represented by the satisfying assignment (line 7), indicating that the property is invalid. Otherwise, `LP` returns unsatisfiable, and $\texttt{BaB}_{\texttt{NNV}}$ calls `Decide` to select a neuron v_i to split, i.e., to fix the activation status of v_i as either active or inactive (line 8). This means

[1]The problem is formulated as a MILP problem as described in §4.2 and checked using an LP solver. Note that we use LP and MILP interchangeably here for simplicity. In practice, most LP solvers such as Gurobi handles MILP problems as well. Just like SMT solvers handle both SAT and SMT problems.

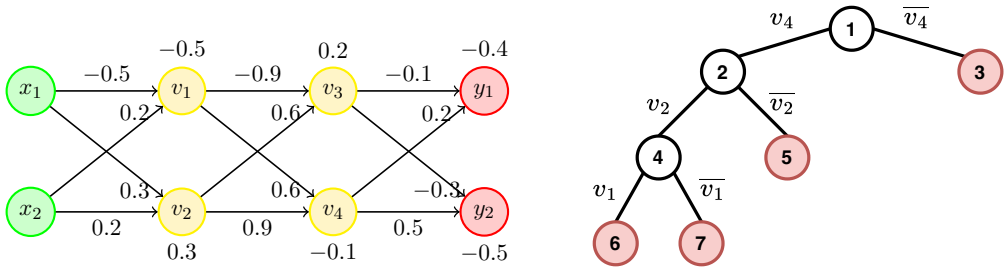


Figure 6.1: (a) A simple DNN (similar to Fig. 2.1), and (b) BaB search tree. White nodes correspond to $\texttt{BaB}_{\texttt{NNV}}$ neuron splitting. Red nodes correspond to leaf nodes where $\texttt{BaB}_{\texttt{NNV}}$determines infeasibility and prunes the branch.

the problem is split into two independent subproblems: one with v_i (active) and the other with $\overline{v_i}$ (inactive). By splitting (or fixing the neuron status), we create two *simpler* subproblems (less neurons to abstract) that are easier to solve. Thus, this step *refines* the precision of abstract interpretation. $\texttt{BaB}_{\texttt{NNV}}$ then adds the two new activation patterns $\sigma_i \wedge v_i$ and $\sigma_i \wedge \overline{v_i}$ to ActPatterns. $\texttt{BaB}_{\texttt{NNV}}$ then loops back to line 2 to process the next activation pattern.

Deduce vs. LP The difference between `Deduce` and `LP` is that the former uses abstraction to compute over-approximated bounds to determine feasibility, while the latter uses an LP solver to check exact satisfiability. Abstraction is (very) quick but may yield false positives (declaring feasible when it is not), while LP is precise but more computationally expensive. Their combination allows $\texttt{BaB}_{\texttt{NNV}}$ to efficiently prune infeasible branches while accurately checking for counterexamples.

Example 6.2.1. Fig. 6.1a illustrates a DNN and how $\texttt{BaB}_{\texttt{NNV}}$ verifies that this DNN has the property

$$(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2),$$

by showing that counterexample exists, i.e., no inputs (x_1, x_2) in the given input range such that $y_1 \leq y_2$.

First, $\texttt{BaB}_{\texttt{NNV}}$ initializes ActPatterns()[2] with an empty activation pattern $\emptyset$ (i.e., no neurons fixed). Then $\texttt{BaB}_{\texttt{NNV}}$ enters its main loop.

1. **First iteration** $\texttt{BaB}_{\texttt{NNV}}$ selects the only available activation pattern $\emptyset$, i.e., $\sigma_i = \emptyset$, and calls `Deduce` to check the feasibility of the problem based σ_i.

   Here, `Deduce` determines that the problem is feasible, e.g., the bounds of y_1 and y_2 indicate that $y_1 \leq y_2$ can be satisfied. Thus, $\texttt{BaB}_{\texttt{NNV}}$ calls `LP` to check the

[2] `Queue` is often used to implement the problems set ActPatterns.

satisfiability of the problem using an LP solver. The LP solver returns unsatisfiable (or unknown)–meaning no counterexample is found in this activation pattern. $\texttt{BaB}_{\texttt{NNV}}$ then randomly selects a neuron to split (`Decide`). Assume that it chooses v_4 to split, i.e., the problem spawns two independent subproblems: one with v_4 active and the other with v_4 inactive. It does so by unioning $\{v_4\}$ and $\{\overline{v_4}\}$ with σ_i and adds both new activation patterns to ActPatterns. Next, $\texttt{BaB}_{\texttt{NNV}}$ loops back to process the next activation pattern.

2. **Second iteration** $\texttt{BaB}_{\texttt{NNV}}$ has two subproblems. For the subproblem with activation pattern $\{v_4\}$, `Deduce` decides feasibility but `LP` returns unsatisfiable (or unknown), so it selects v_2 to split by adding $\{v_4, v_2\}$ and $\{v_4, \overline{v_2}\}$ to `ActPatterns`. For the other subproblem with $\overline{v_4}$ inactive, `Deduce` determines infeasibility and thus discards this subproblem.

3. **Third iteration** $\texttt{BaB}_{\texttt{NNV}}$ has two subproblems corresponding to activation patterns $\{v_4, v_2\}$ and $\{v_4, \overline{v_2}\}$. For the first subproblem, $\texttt{BaB}_{\texttt{NNV}}$ selects v_1 to split by unioning v_1 and then $\overline{v_1}$ to the current activation pattern $v_4 \wedge v_2$ and adds them to ActPatterns. For the second subproblem, `Deduce` determines infeasibility and $\texttt{BaB}_{\texttt{NNV}}$ discards this subproblem.

4. **Fourth iteration** $\texttt{BaB}_{\texttt{NNV}}$ has two subproblems corresponding to activation patterns $\{v_4, v_2, v_1\}$ and $\{v_4, v_2, \overline{v_1}\}$, both which are then determined infeasible and discarded.

5. **Fith iteration** Finally, $\texttt{BaB}_{\texttt{NNV}}$ has no more activation patterns in ActPatterns, stops the search, and returns `unsat`, indicating that the property is valid.

Fig. 6.1b illustrates the BaB search tree corresponding to the above process. Each white node represents a branching decision where $\texttt{BaB}_{\texttt{NNV}}$ splits a neuron, while each red node represents a leaf node where `Deduce` determines infeasibility and prunes the branch. Notice that not all activation patterns are explored because some branches are pruned early due to infeasibility. In other words, while there are $2^4 = 16$ possible complete activation patterns for the three ReLU neurons, $\texttt{BaB}_{\texttt{NNV}}$ only explores 6 of them in this example.

Problem 6.2.1. Do Ex. 6.2.1 but come UB with concrete values for each step, e.g., what are the bounds computed by `Deduce`, what does `LP` return, etc to illustrate the process in more detail.

Problem 6.2.2. Do Ex. 6.2.1 again but make the property is invalid and find a counterexample. It is also OK to change the problem, e.g., the input bounds or the property itself, to make it invalid.

Problem 6.2.3. Apply BaB on the DNN in Fig. 6.2 to verify the property $x_5 > 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$. Use interval abstraction to compute bounds.

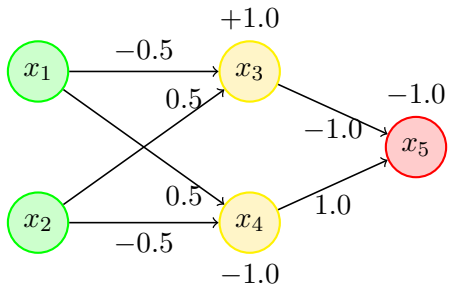


Figure 6.2: A simple DNN (similar to Fig. 1.1).

For LP solving, you can use any LP solver or solve it manually. Show all steps, e.g., iterations, activation patterns, bounds computed, LP results, etc.

Problem 6.2.4. Test your understanding of BaB and activation patterns by answering the following questions:

1. In Alg. 1, what happens if `Deduce` always returns infeasible?
2. What happens if `Deduce` always returns feasible but `LP` always returns satisfiable?
3. What is the maximum number of activation patterns that $\texttt{BaB}_{\texttt{NNV}}$ may need to explore in the worst case for a DNN with m ReLU neurons?
4. What is the minimum number of activation patterns that $\texttt{BaB}_{\texttt{NNV}}$ may need to explore in the best case for a DNN with m ReLU neurons? What are these patterns?

6.2.1 Beyond the Basic

What described above is the basic and minimal BaB algorithm. Modern DNN verifiers implement many optimizations and strategies to improve the performance of the search. For example, they apply various engineering tricks to eliminate easy cases or find easy counterexamples (§7) before running the full BaB search.

Even the BaB search itself is optimized to avoid exploring the entire search space. For example, if `Deduce` step determines infeasibility, a smarter $\texttt{BaB}_{\texttt{NNV}}$ variant (e.g., the `NeuralSAT` tool) can analyze the conflict and add a new clause to the set of clauses (`clauses`) to prevent the same activation pattern from being selected again. This is similar to conflict-driven clause learning (CDCL) in modern SAT solvers and is implemented in the `NeuralSAT` DNN verification tool [22]. In addition, heuristics are also used to select, e.g., which neuron to split next (`Decide`). We explore these optimizations and strategies in later chapters (e.g., §10).

Chapter 7

Adversarial Attacks

A full branch and bound (BaB) search (§6) is typically expensive and slow on large networks. Thus most DNN verification tools implement optimizations to improve performance. A common optimization is to use *adversarial attack* techniques to find counterexamples before running BaB. Examples of such techniques include randomized attacks [15] and gradient-based attacks [36], described in §7.

Thus, modern verification tools have two phases: (i) attempt to *falsify* the property with adversarial attacks, and (ii) if no counterexample is found, attempt to *verify* the property using search algorithms such as $\texttt{BaB}_{\texttt{NNV}}$ (§6). Of course, during the verification phase, the verifier may also discover counterexamples that the falsify phase misses.

7.1 Random Search Attack

Random search is a simple adversarial attack method. It randomly samples points in the allowed input ranges and checks if any samples violate the property; if so, a counterexample is found.

Example 7.1.1. Suppose the DNN input ranges are:

$$-1 \leq x_1 \leq 1, \quad -2 \leq x_2 \leq 2$$

and the network output is:

$$y = 2x_1 - 1.5x_2 + 1.$$

We wish to use random search to find a counterexample to the property $y > 0$; i.e., trying random inputs (x_1, x_2) satisfying the given ranges and producing $y \leq 0$.

- Try 1: $x_1 = 0.2,\ x_2 = -0.5$

  $$y = 2 \times 0.2 - 1.5 \times (-0.5) + 1 = 0.4 + 0.75 + 1 = 2.15 > 0$$

  Not a violation.

- Try 2: $x_1 = -1,\ x_2 = 2$

$$y = 2 \times (-1) - 1.5 \times 2 + 1 = -2 - 3 + 1 = -4 < 0$$

Counterexample found: $(x_1 = -1,\ x_2 = 2)$.

Problem 7.1.1. Consider the input ranges

$$-1 \leq x_1 \leq 1, \quad -1 \leq x_2 \leq 1,$$

and network output

$$y = 3x_1 + 2x_2 - 2.$$

Use random search to find a counterexample to the property $y > 0$:

1. Randomly sample three valid points and compute y at each point.
2. Check if any sample violates the property. Note, if none violates, that's OK; just report the results.
3. Based on your samples, discuss whether random search is likely to be efficient for this problem.

7.2 Projected Gradient Descent (PGD)

PGD is a powerful and widely used adversarial attack that builds on the idea of *gradient descent*. Gradient descent is the procedure of repeatedly moving the input a small amount in the direction that most rapidly *decreases* (or, for an attack, increases the likelihood of) some objective by following the negative (or positive) gradient.

Specifically, at each step PGD computes the gradient of the network output with respect to the input and takes a small gradient-descent step that pushes the input toward a stronger property violation. Next, PGD *projects* (clips) the updated input back into the allowed domain if the step goes outside the input bounds.

We will reuse Ex. 7.1.1 to illustrate PGD. Here, the DNN input ranges are $-1 \leq x_1 \leq 1,\ -2 \leq x_2 \leq 2$, and the output is $y = 2x_1 - 1.5x_2 + 1$, and the goal is to find a counterexample that satisfies the input ranges and produces $y \leq 0$.

1. **Initialize** Choose a valid starting input, for example:

$$(x_1^{(0)}, x_2^{(0)}) = (0, 0)$$

This point is within the allowed input ranges, but does not violate the property $y > 0$ because $y = 2 * 0 - 1.5 * 0 + 1 = 1 > 0$. So we need to continue.

2. **Iterative update** For each step $t = 0, 1, 2, \ldots$:

(a) **Compute the gradient.** The gradient $\nabla_x y$ tells us how sensitive the output y is to each input variable:

$$\nabla_x y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \ldots, \frac{\partial y}{\partial x_n} \right)$$

It represents the direction of steepest *increase* in y. To move toward a smaller y (to make violation $y \leq 0$ more likely), we go in the *opposite* direction, i.e., $-\nabla_x y$.

For example, if:

$$y = 2x_1 - 1.5x_2 + 1,$$

then the partial derivatives are:

$$\frac{\partial y}{\partial x_1} = 2, \qquad \frac{\partial y}{\partial x_2} = -1.5,$$

resulting in the gradient:

$$\nabla_x y = (2, -1.5).$$

This gradient means that increasing x_1 by a small amount increases y by about twice that amount, while increasing x_2 by a small amount decreases y by about 1.5 times that amount.

(b) **Gradient update** Move a small distance η (called the *step size* or *learning rate*) in the negative gradient direction:

$$x_1^{(t+1)} = x_1^{(t)} - \eta \cdot 2, \qquad x_2^{(t+1)} = x_2^{(t)} - \eta \cdot (-1.5)$$

This decreases y as quickly as possible.

For example, if $\eta = 0.5$ and starting from $(x_1^{(0)}, x_2^{(0)}) = (0, 0)$:

$$x_1^{(1)} = 0 - 0.5 \times 2 = -1.0, \qquad x_2^{(1)} = 0 - 0.5 \times (-1.5) = +0.75$$

Thus, the candidate input after one step is $(-1.0, +0.75)$.

(c) **Project (Clip) to valid input range** If the new values go outside the valid range, bring them back to the closest valid boundary:

$$x_1^{(t+1)} = \max(-1, \min(x_1^{(t+1)}, 1)), \qquad x_2^{(t+1)} = \max(-2, \min(x_2^{(t+1)}, 2))$$

This ensures every step stays within $[-1, 1] \times [-2, 2]$.

Our candidate input $(-1.0, +0.75)$ is already within the valid ranges, so no clipping is needed.

(d) **Check for violation** Compute the output at the new input. If $y(x^{(t+1)}) \leq 0$, then a counterexample has been found, and the attack succeeds.
In our example, compute:

$$y = 2(-1) - 1.5(0.75) + 1 = -2 - 1.125 + 1 = -2.125 < 0$$

Thus, PGD finds a counterexample at $(x_1, x_2) = (-1, 0.75)$.

Example 7.2.1 (PGD with Clipping)**.** We continue the example above but with a larger step size $\eta = 1.0$. Starting again from $(x_1^{(0)}, x_2^{(0)}) = (0, 0)$:

$$x_1^{(1)} = 0 - 1.0 \times 2 = -2.0$$
$$x_2^{(1)} = 0 - 1.0 \times (-1.5) = +1.5$$

Here, $x_1^{(1)} = -2.0$ is outside the allowed range $[-1, 1]$, so we *clip* it:

$$x_1^{(1)} = -1 \quad \text{(projected to boundary)}$$

Meanwhile, $x_2^{(1)} = 1.5$ is within $[-2, 2]$, so it remains unchanged.

After projection, we have:

$$(x_1^{(1)}, x_2^{(1)}) = (-1, 1.5)$$

Compute the output:

$$y = 2(-1) - 1.5(1.5) + 1 = -2 - 2.25 + 1 = -3.25 < 0,$$

which is a counterexample.

Example 7.2.2. Consider

$$y = x_1^2 + 3x_2.$$

The gradient of y with respect to (x_1, x_2) is

$$\nabla_x y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2} \right).$$

Compute each partial derivative:

$$\frac{\partial y}{\partial x_1} = 2x_1, \qquad \frac{\partial y}{\partial x_2} = 3.$$

Therefore the gradient is

$$\nabla_x y = (2x_1,\ 3).$$

For instance, at point $(x_1, x_2) = (1, 2)$,

$$\nabla_x y = (2 \times 1,\ 3) = (2, 3).$$

This means that near $(1, 2)$, increasing x_1 by a small amount increases y twice as fast as that same increase in x_2 would.

Problem 7.2.1. For each of the following output functions, compute the gradient

$$\nabla_x y = \left(\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2} \right).$$

1. $y = 2x_1 - x_2 + 1$
2. $y = -x_1^2 + 4x_2$
3. $y = 3x_1 x_2 - x_2^2$

Explain what the gradient tells you about how y changes with x_1 and x_2.

Problem 7.2.2. Let

$$y = 2x_1 - 3x_2 + 1, \quad -1 \leq x_1, x_2 \leq 1.$$

Starting from $(x_1^{(0)}, x_2^{(0)}) = (0, 0)$ with some step size $\eta > 0$, use PGD to minimize y so that it violates the property $y > 0$. Show all steps (e.g., computing gradient, updating point, clipping, etc) until a counterexample is found.

Depend on your choice of step size η, PGD might have one or more iterations before finding a counterexample. Thus, for this problem, identify two different step sizes η_1 and η_2 such that:

1. With step size η_1, PGD finds a counterexample in *one* iteration.
2. With step size η_2, PGD finds a counterexample in *two or more* iterations.

Chapter 8

Proof Generation and Checking

As DNN tools become more complex (e.g., state-of-the-art tools have 20K+ LoCs), they are more prone to bugs. VNN-COMP'23 [10] showed that 3 of the top 7 participants produced unsound results by claiming unsafe DNNs are safe, i.e., they produce `unsat` on problems that are actually `sat`.

While checking counterexamples is relatively straightforward (we can just evaluate the DNN on the input), checking `unsat` results—proving no counterexample exists—is more challenging as it requires tracing the reasoning steps of the verifier. In this chapter, we discuss work [21] on generating and checking proofs of `unsat` results generated by BaB-based DNN verifiers.

8.1 Proof Generation for Branch and Bound Algorithms

Recall that the BaB algorithm, shown in Alg. 1, splits the problem into smaller subproblems and use abstraction to compute bounds to prune the search space. This structure allows us to bring proof generation capabilities with minimal overhead to existing DNN verification tools that use BaB.

Alg. 2 extends Alg. 1 to show $\texttt{BaB}_{\texttt{ProofGen}}$, a BaB-based DNN verification algorithm with proof generation capability. The key idea is to introduce a **proof tree** (line 2) and recording the branching decisions to it (line 12). Thus, the proof tree has a binary structure, where each node represents a neuron and its left and right edges represent its activation decision (active or inactive). At the end of the verification process, the proof tree is returned as the proof of `unsat` result.

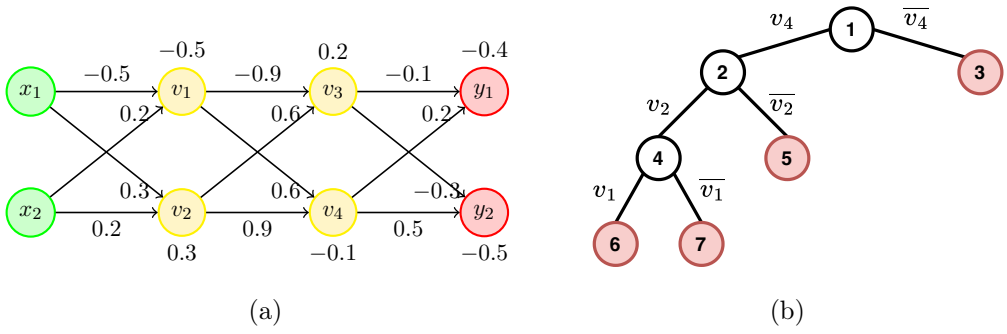
Example 8.1.1. We reuse the example in Ex. 6.2.1 to illustrate $\texttt{BaB}_{\texttt{ProofGen}}$. Recall the goal is to verify that the DNN in Fig. 8.1(a) has the property $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2)$.

For this problem $\texttt{BaB}_{\texttt{ProofGen}}$ generates the proof tree in Fig. 8.1(b) to show unsatisfiability, i.e., the property is valid. $\texttt{BaB}_{\texttt{ProofGen}}$ first splits neuron v_4, creating two subproblems: one with v_4 active and the other with v_4 inactive (split node

Algorithm 2. The $\texttt{BaB}_{\texttt{ProofGen}}$ DNN verification with proof generation.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$
output : (unsat, proof) if property is valid, otherwise (sat, cex)

```
ActPatterns ← {∅} // initialize verification problems
proof ← { } // initialize proof tree
while ActPatterns do // main loop
    σ_i ← Select(ActPatterns) // process problem i-th
    if Deduce(N, φ_in, φ_out, σ_i) then
        cex ← LP(N, φ_in, φ_out, σ_i) // check satisfiability
        if cex then // found a valid cex
            return (sat, cex)
        v_i ← Decide(N, σ_i)
        // create new activation patterns
        ActPatterns ← ActPatterns ∪ {σ_i ∪ {v_i} ; σ_i ∧ {¬v_i}}
    else // detect infeasibility
        proof ← proof ∪ {σ_i} // build proof tree
return (unsat, proof)
```



(a) (b)

Figure 8.1: (a) A simple DNN (similar to Fig. 2.1), and (b) A proof tree produced by $\texttt{BaB}_{\texttt{ProofGen}}$. White nodes correspond to $\texttt{BaB}_{\texttt{NNV}}$ neuron splitting. Red nodes correspond to leaf nodes where $\texttt{BaB}_{\texttt{NNV}}$ determines infeasibility and prunes the branch.

1). The inactive v_4 subproblem is determined to be unsatisfiable (leaf node 3). The active v_4 subproblem is further split on v_2 (split node 2). The inactive v_2 subproblem is determined to be unsatisfiable (leaf node 5). The active v_2 subproblem is further split on v_1 (split node 4). Both the active and inactive v_1 subproblems are determined to be unsatisfiable (leaf nodes 6 and 7). Since all branches lead to unsatisfiability, the property is valid.

Problem 8.1.1. Consider the following toy network:

$$\begin{aligned}
\hat{x}_3 &= x_1 - 2x_2 + 1, \\
x_3 &= \mathrm{ReLU}\left(\hat{x}_3\right), \\
\hat{x}_4 &= -x_1 + x_3 + 0.5, \\
x_4 &= \mathrm{ReLU}\left(\hat{x}_4\right), \\
y &= -x_3 + 2x_4.
\end{aligned}$$

The input region is

$$(x_1, x_2) \in [-1, 1] \times [-1, 1],$$

and the property to verify is:

$$y \leq 5.$$

A verifier uses a BaB-based proof generation method and performs *neuron splitting* on the ReLU nodes x_3 and x_4. Assume the verifier splits neurons in the order:

1. First split on x_3.
2. For the *active*-x_3 branch, split on x_4.

The solver determines that:

- When x_3 is inactive, the subproblem is `unsat`.
- When x_3 is active and x_4 is inactive, the subproblem is `unsat`.
- When both x_3 and x_4 are active, the subproblem is also `unsat`.

Do the following:

1. Draw the *proof tree* for this verification process, labeling each split node with the neuron being split (e.g., x_3 or x_4), etc like in Fig. 8.1(b).
2. For each split, explain:
   - which constraints are added (e.g., $\hat{x}_3 \geq 0$, $\hat{x}_4 \leq 0$),
   - how the equations simplify under these constraints,
   - why the resulting subproblem is `unsat`.
3. Explain why the proof tree is a correctness proof. In other words, justify why showing that all leaf nodes are `unsat` means that the property $y \leq 5$ is valid.

⟨*proof*⟩ ::= ⟨*declarations*⟩ ⟨*assertions*⟩
⟨*declarations*⟩ ::= ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*declarations*⟩
⟨*declaration*⟩ ::= (**declare-const** ⟨*input-vars*⟩ **Real**)
| (**declare-const** ⟨*output-vars*⟩ **Real**)
| (**declare-pwl** ⟨*hidden-vars*⟩ ⟨*activation*⟩)
⟨*input-vars*⟩ ::= ⟨*input-var*⟩ | ⟨*input-var*⟩ ⟨*input-vars*⟩
⟨*output-vars*⟩ ::= ⟨*output-var*⟩ | ⟨*output-var*⟩ ⟨*output-vars*⟩
⟨*hidden-vars*⟩ ::= ⟨*hidden-var*⟩ | ⟨*hidden-var*⟩ ⟨*hidden-vars*⟩
⟨*activation*⟩ ::= ReLU | Leaky ReLU | ...
⟨*assertions*⟩ ::= ⟨*assertion*⟩ | ⟨*assertion*⟩ ⟨*assertions*⟩
⟨*assertion*⟩ ::= (**assert** ⟨*formula*⟩)
⟨*formula*⟩ ::= (⟨*operator*⟩ ⟨*term*⟩ ⟨*term*⟩)
| (**and** ⟨*formula*⟩+) | (**or** ⟨*formula*⟩+)
⟨*term*⟩ ::= ⟨*input-var*⟩ | ⟨*output-var*⟩
| ⟨*hidden-var*⟩ | ⟨*constant*⟩
⟨*operator*⟩ ::= < | ≤ | > | ≥
⟨*input-var*⟩ ::= X_⟨*constant*⟩
⟨*output-var*⟩ ::= Y_⟨*constant*⟩
⟨*hidden-var*⟩ ::= N_⟨*constant*⟩
⟨*constant*⟩ ::= **Int** | **Real**

Figure 8.2: The $\texttt{BaB}_{\texttt{ProofLang}}$ proof language.

```
; Declare variables
(declare-const X_0 X_1 Real)
(declare-const Y_0 Y_1 Real)
(declare-pwl N_1 N_2 N_3 N_4 ReLU)

; Input constraints
(assert (>= X_0 -2.0))
(assert (<= X_0  2.0))
(assert (>= X_1 -1.0))
(assert (<= X_1  1.0))

; Output constraints
(assert (<= Y_0 Y_1))

; Hidden constraints
(assert (or
    (and (<  N_4 0))
    (and (<  N_2 0)
         (>= N_4 0))
    (and (>= N_2 0)
         (>= N_1 0)
         (>= N_4 0))
    (and (>= N_2 0)
         (<  N_1 0)
         (>= N_4 0))
))
```

Figure 8.3: $\texttt{BaB}_{\texttt{ProofLang}}$ example format of the proof tree in Fig. 8.1b.

8.2 Proof Language

Rather than recording the generated proof tree in an verifier-specific format, it is more desirable to have a standard format that is human-readable, is compact, can be efficiently generated by verification tools, and can be efficiently and independently processed by proof checkers.

The proof language $\texttt{BaB}_{\texttt{ProofLang}}$ is designed to meet these requirements. $\texttt{BaB}_{\texttt{ProofLang}}$ is inspired by the SMTLIB format [6] used for SMT solving, which has also been adopted by the VNNLIB language [49] to specify DNNs and their properties for verification (see §2.5 for examples).

Fig. 8.2 outlines the $\texttt{BaB}_{\texttt{ProofLang}}$ syntax and grammar, represented as production rules. A proof is composed of *declarations* and *assertions*. Declarations define the variables and their types within the proof. Specifically, *input variables* (prefixed with `X`) and *output variables* (prefixed with `Y`) are declared as real numbers, representing the inputs and outputs of the neural network, respectively. Additionally, *hidden variables*, which correspond to internal nodes of the neural network, are declared with specific piece-wise linear (PWL) activation functions, such as ReLU or Leaky ReLU. Assertions are logical statements that specify the conditions or properties that must hold within the proof. Assertions over input variables are *preconditions*

and those over output variables are *post-conditions*.

The `declare-*` statements declare input, output, and hidden variables, while the `assert` statements specify the constraints on these variables (i.e., the pre and postcondition of the desired property). The hidden constraints represent the activation patterns of the hidden neurons in the network (i.e., the proof tree). Each `and` statement represents a tree path that represents an activation pattern.

Example 8.2.1. The proof in Fig. 8.3 corresponds to the proof tree in Fig. 8.1b. The statement `(and (< N_4 0))` corresponds to the rightmost path of the tree with $\overline{v_4}$ decision (leaf 3). The statement `(and (< N_2 0) (>= N_4 0))` corresponds to the path with $v_4 \wedge \overline{v_2}$ (leaf 5).

The $\texttt{BaB}_{\texttt{ProofLang}}$ language is intentionally designed to (a) not explicitly include weights/bias terms to minimize size of the proof structure, and (b) explicitly reflect a DNF structure to enable easy parallelization. The DNN weight and bias terms are readily available in the standard ONNX [43] format, which is typically used to represent the DNN input to a $\texttt{BaB}_{\texttt{ProofGen}}$-based DNN verification tool and can be accessed by any $\texttt{BaB}_{\texttt{ProofLang}}$ checker like the one described next in §8.3.

8.3 Proof Checker

We present $\texttt{BaB}_{\texttt{ProofCheck}}$, a proof checker for $\texttt{BaB}_{\texttt{ProofLang}}$ proofs generated by BaB-based DNN verifiers. $\texttt{BaB}_{\texttt{ProofCheck}}$ is verifier-independent and support $\texttt{BaB}_{\texttt{ProofLang}}$ proofs generated by different verification tools. $\texttt{BaB}_{\texttt{ProofCheck}}$ also has several optimizations to handle large proofs efficiently.

8.3.1 The Core $\texttt{BaB}_{\texttt{ProofCheck}}$ Algorithm

The goal of $\texttt{BaB}_{\texttt{ProofCheck}}$ is to verify that the $\texttt{BaB}_{\texttt{ProofLang}}$ tree generated by a DNN verification tool is correct (i.e., the proof tree is a proof of unsatisfiability of the DNN verification problem). $\texttt{BaB}_{\texttt{ProofCheck}}$ thus must verify that the constraint represented by each *leaf* node in the proof tree is unsatisfiable. To check each node, $\texttt{BaB}_{\texttt{ProofCheck}}$ forms an MILP problem (§4.2) consisting of the constraint in Eq. 3.2.3 (the DNN, the input condition, and the negation of the output) with the constraints representing the activation pattern encoded by the tree path to the leaf node. $\texttt{BaB}_{\texttt{ProofCheck}}$ then invokes an LP solver to check that the MILP problem is infeasible, which indicates unsatisfiability of the leaf node.

Alg. 3 shows a minimal (core) $\texttt{BaB}_{\texttt{ProofCheck}}$ algorithm, which takes as input a DNN $\mathcal{N}$, a property $\phi_{in} \Rightarrow \phi_{out}$, a proof tree proof, and returns certified if the proof tree is valid and uncertified otherwise. $\texttt{BaB}_{\texttt{ProofCheck}}$ first checks the validity of the proof tree (line 3), i.e., the input must represent a proper $\texttt{BaB}_{\texttt{ProofLang}}$ proof tree (§8.2). If the proof tree is invalid, $\texttt{BaB}_{\texttt{ProofCheck}}$ raises an error. $\texttt{BaB}_{\texttt{ProofCheck}}$ next creates a MILP model (line 3) representing the input. $\texttt{BaB}_{\texttt{ProofCheck}}$ then enters a

Algorithm 3. $\texttt{BaB}_{\texttt{ProofCheck}}$ algorithm.

input : DNN $\mathcal{N}$, property $\phi_{in} \Rightarrow \phi_{out}$, proof
output : certified if proof is valid, otherwise uncertified

```
1  if ¬ RepOK (proof) then
2      RaiseError(Invalid proof tree)
   // initialize MILP model with inputs
3  model ← CreateStabilizedMILP(N, φ_in, φ_out)
4  node ← null // initialize current processing node
5  while proof do
6      node ← Select(proof, node) // get next node to check
7      model ← AddConstrs(model, node) // add constraints
8      if CheckFeasibility(model) then
9          return uncertified // cannot certify
10 return certified
```

loop (line 5) that selects a (random) leaf node from the proof tree (line 6) and adds its MILP constraint to the model (line 7). It then checks the model using an LP solver to determine whether the leaf node is unsatisfiable. If the LP solver returns feasibility, $\texttt{BaB}_{\texttt{ProofCheck}}$ returns uncertified, i.e., it cannot verify the input proof tree. $\texttt{BaB}_{\texttt{ProofCheck}}$ continues until all leaf nodes are checked and returns certified, indicating the proof tree is valid.

Example 8.3.1. For the $\texttt{BaB}_{\texttt{ProofLang}}$ proof in Fig. 8.3, we check that the four leaf nodes 3, 5, 6, and 7 of the proof tree in Fig. 8.1b are unsatisfiability. Assume $\texttt{BaB}_{\texttt{ProofCheck}}$ first selects node 3, it forms the MILP problem for leaf node 3 by conjoining the constraint representing $0.6v_1 + 0.9v_2 - 0.1 \leq 0$ (i.e., $\overline{v_4}$) with the constraints in Eq. 3.2.3 representing the input ranges and the DNN with the objective of optimizing the output. $\texttt{BaB}_{\texttt{ProofCheck}}$ then invokes an LP solver, which determines that this MILP is infeasible, i.e., leaf node 3 indeed leads to unsatisfiability.

$\texttt{BaB}_{\texttt{ProofCheck}}$ continues this process for the other three leaf nodes and returns certified as all leaf nodes are unsatisfiable.

8.3.2 Optimizations

While the core $\texttt{BaB}_{\texttt{ProofCheck}}$ algorithm in Alg. 3 is minimal, it can be inefficicient for large proofs. $\texttt{BaB}_{\texttt{ProofCheck}}$ employs several optimizations to improve its efficiency. These are crucial for checking large proof trees generated by DNN verification tools for challenging problems.

Neuron Stabilization A primary challenge in DNN analysis is the presence of large numbers of piecewise-linear constraints (e.g., ReLU) which generate a large number of branches and yield large proof trees. In the MILP formulation, this creates

many disjunctions which are hard to solve. To reduce the number of disjunctions, $\texttt{BaB}_{\texttt{ProofCheck}}$ uses *neuron stabilization* [23] to determine neurons that are *stable*, either active or inactive, for all inputs defined by the property pre-condition. For all stable neurons, the disjunctive ReLU constraint is replaced with a linear constraint that represents the neuron's value. This simplifies the MILP problem.

Pruning Leaf Nodes Another optimization used is that $\texttt{BaB}_{\texttt{ProofGen}}$ does not check child nodes if the parent node is unsatisfiable. In a $\texttt{BaB}_{\texttt{ProofLang}}$ proof tree, a child node adds constraints to the parent (e.g., node 6 adds the constraint of v_1 to node 4, which adds the constraint of v_2 to node 2 in Fig. 8.1b). Thus, if we determine that the constraint of the parent is unsatisfiable, we can skip the child nodes, which must also be unsatisfiable.

$\texttt{BaB}_{\texttt{ProofCheck}}$ uses a backtracking mechanism to check the parent node only when the child nodes are infeasible. Specifically, it starts checking a leaf node l. If it determines unsatisfiability it will check the parent p of l. If p is unsatisfiable it immediately removes the children of p (more specifically the sibling of l). Next it backtracks to the parent of p and repeats until meeting a stopping criteria. This optimization reduces the number of LP problems that need to be solved, making the proof checking process more efficient.

$\texttt{BaB}_{\texttt{ProofGen}}$ implement a backjumping strategy that allows for backtracking multiple levels, N, rather than a single level at a time. A large value of N offers the chance for greater pruning if an unsatisfiable node is found by backjumping, but such nodes also represent less constrained, and therefore, more complex MILP problems and are less likely to be unsatisfiable. The default value in $\texttt{BaB}_{\texttt{ProofCheck}}$ is $N = 2$ is selected to enable a modest degree of pruning, while being close enough to a proven unsatisfiable node that it has a reasonable chance of itself being unsatisfiable.

Parallelization Finally, the structure of a prooflang proof trees is designed to be easily parallelized. Each tree path is an independent sub-proof and partitions of the tree allow checker to leverage multiprocessing to check large proof trees efficiently. $\texttt{BaB}_{\texttt{ProofCheck}}$ uses a parameter k to control the number of leaf nodes to be checked in parallel.

Chapter 9

Common Engineerings and Optimizations

In addition to adversarial attacks (§7), DNN verifiers often employ a range of optimizations and engineering techniques to improve performance. This chapter discusses some of the common techniques used in modern DNN verifiers.

9.1 Input Splitting

Many verifiers, e.g., [22, 32, 51, 52], use a technique called *input splitting* to quickly deal with networks with verification problems involving low-dimensional networks, such as those in the ACAS Xu benchmark **??** where the networks have a small number of inputs (e.g., ≤ 50).

The idea is to split the original verification problem into subproblems, each checking whether the DNN produces the desired output from a smaller input region and returns `unsat` if all subproblems are verified and `sat` if a counterexample is found in any subproblem. Input splitting avoids BaB search (§6)—which performs *neuron splitting*—and is often used to quickly eliminate easy cases.

Moreover, each subproblem now has a smaller input region, thus the verifier can often verify them more quickly than the original problem. Finally, because each task can be solved independently, the verifier can solve them in parallel to further speed UB the verification process.

Example 9.1.1. Given a problem with input region $\{x_1 \in [-1, 1], x_2 \in [-2, 2]\}$, input splitting splits the input region into four subregions: $\{x_1 \in [-1, 0], x_2 \in [-2, 0]\}$, $\{x_1 \in [-1, 0], x_2 \in [0, 2]\}$, $\{x_1 \in [0, 1], x_2 \in [-2, 0]\}$, and $\{x_1 \in [0, 1], x_2 \in [0, 2]\}$. The verifier then checks if the DNN produces the desired output from each of these subregions.

Note that the formula $-1 \leq x_1 \leq 1 \wedge -2 \leq x_2 \leq 2$ representing the original input region is equivalent to the formula $(-1 \leq x_1 \leq 0 \vee 0 \leq x \leq 1) \wedge (-2 \leq x_2 \leq$

$0 \vee 0 \leq x_2 \leq 2$) representing the combination of the created subregions.

9.2 Bounds Tightening

9.2.1 Input Bounds Tightening

When the verifier splits on a neuron (e.g., $\hat{x}_i \leq 0$ or $\hat{x}_i > 0$), it adds new linear constraints that further restrict the feasible input region. As a result, the original input bounds (e.g., $-1 \leq x_1 \leq 1$) may no longer reflect the true range of values that satisfy all current constraints. **Input Bound tightening** recomputes the smallest possible ranges for each input variable under these constraints. These tighter input bounds propagate through the network and yield tighter neuron and output bounds, which in turn helps prune subproblems earlier.

Example 9.2.1. Recall the DNN from Fig. 3.1 can be represented as:

$$\begin{aligned}
\hat{x}_3 &= -0.5x_1 + 0.5x_2 + 1.0 \wedge \\
\hat{x}_4 &= 0.5x_1 - 0.5x_2 + 1.0 \wedge \\
x_3 &= \text{ReLU}\left(\hat{x}_3\right) \wedge \\
x_4 &= \text{ReLU}\left(\hat{x}_4\right) \wedge \\
x_5 &= -x_3 + x_4 - 1.0,
\end{aligned}$$

and input property ϕ_{in} $-1 \leq x_1 \leq 1 \wedge -2 \leq x_2 \leq 2$

Suppose we make a split on x_3, which give us two subproblems, represented by the constraints: $-0.5x_1 + 0.5x_2 + 1.0 \leq 0$ and $-0.5x_1 + 0.5x_2 + 1.0 > 0$. We can use the new constraint to tighten the input bounds of x_1 and x_2 for each subproblem.

For the first subproblem, we can create an optimization problem to tighten the bounds. For example, to tighten the upper bounds of x_1, we can create an optimization problem as follows:

$$\begin{aligned}
\text{maximize } x_1 \quad & \text{s.t.} \\
& -0.5x_1 + 0.5x_2 + 1.0 \leq 0 \\
& -1 \leq x_1 \leq 1 \\
& -2 \leq x_2 \leq 2
\end{aligned} \tag{9.2.1}$$

We can apply the same approach to tighten the lower bounds of x_1 using a minimization problem. Similarly, we can create optimization problems to tighten both the bounds of x_2.

Note that, this approach works well for networks with small inputs, e.g., the ACAS Xu benchmark **??** with 5 inputs, and is adopted by many modern DNN verification tools, including `Marabou` and `nnenum`. For larger input dimensions networkswe, it will take much time as we need to run $2\times$ the number of inputs to tighten

all the input bounds (for each input, we need to tighten the upper and lower bounds individually).

To reduce the time complexity, we can prioritize the input variables to tighten first, e.g., deciding the variable with the largest bounds first, and only tighten UB to a maxinum number of inputs, e.g., 10 inputs.

9.2.2 Neuron (Hidden) Bounds Tightening

Instead of tightening the input bounds, we can tighten the bounds of the hidden neurons. There are a couple of advantages of tightening the bounds of the hidden neurons:

1. In some networks, numbers of hidden neurons are *fewer* than the number of inputs, e.g., the MNISTFC 2x256 network has $28 \times 28 = 784$ inputs and only 512 hidden neurons.

2. We don't necessarily tighten all hidden neurons, as some hidden neurons are already stable, i.e., their lower bounds are greater than 0 or their upper bounds are less than 0.

3. More importantly, if a hidden neuron is stabilized after tightening its bounds, e.g., its lower bound becomes greater than 0 or its upper bound becomes less than 0, we no longer need to branch on that particular neuron.

Example 9.2.2. Let's revisit Ex. 9.2.1, after splitting x_4, we can tighten the upper bounds of x_4 (suppose that $l_{x_4} < 0 < u_{x_4}$, e.g., unstable and not split yet) by solving the following optimization problem:

$$\begin{aligned} \text{maximize } x_4 \quad \text{s.t.} \\ 0.5x_1 - 0.5x_2 + 1.0 \leq 0 \\ -1 \leq x_1 \leq 1 \\ -2 \leq x_2 \leq 2 \end{aligned} \tag{9.2.2}$$

If the optimization result yields a maximum value of x_4 less than 0, e.g., $u_{x_4} < 0$, we can conclude that x_4 is stabilized and no longer need to branch on x_4, thus, reducing the number of branches made by the verifier. There will be many tweaks to make the optimization problem more efficient, e.g., setting a early stopping condition, e.g., if the maximum value of x_4 is less than 0, we can stop the optimization. It is because the fact that, if the upper bound of x_4 is already less than 0, x_4 will be considered as *inactive* no matter what the *optimal* upper bound is.

Note that in both cases optimization problems are independent (between inputs/neurons and lower/upper bounds) and thus can be solved in parallel, e.g., optimizing the both upper and lower bounds of x_1 (input) and x_4 (neuron) simultaneously.

9.3 Batch Processing

Matrix Form for Interval Computation While the min/max formulation, e.g., for abstraction §5.2.1, is intuitive, it can be efficiently implemented using matrix operations. We can decompose the weight matrix W into positive and negative components:

$$W^+ = \max(W, 0) \qquad W^- = \min(W, 0)$$

where the max and min operations are applied element-wise.

Then the interval bounds can be computed as:

$$f_L^a = W^+ \cdot \mathbf{l} + W^- \cdot \mathbf{u} + \mathbf{b}$$
$$f_U^a = W^+ \cdot \mathbf{u} + W^- \cdot \mathbf{l} + \mathbf{b}$$

where $\mathbf{l} = [l_1, l_2, \ldots, l_n]^T$ and $\mathbf{u} = [u_1, u_2, \ldots, u_n]^T$ are the vectors of lower and upper bounds.

Example 9.3.1. We can verify that the matrix form produces identical results to the min/max approach using the same network from Ex. 5.2.1.

Given the weight $w_1 = -0.5$, $w_2 = 0.5$, and bias $b = 1.0$, we decompose:

$$W^+ = [\max(-0.5, 0), \max(0.5, 0)] = [0, 0.5]$$

$$W^- = [\min(-0.5, 0), \min(0.5, 0)] = [-0.5, 0]$$

For inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$, the lower vector $\mathbf{l} = [1, -1]$ and the upper vector $\mathbf{u} = [2, 3]$:

$$\begin{aligned} f_L^a &= W^+ \cdot \mathbf{l} + W^- \cdot \mathbf{u} + b \\ &= [0, 0.5] \cdot [1, -1] + [-0.5, 0] \cdot [2, 3] + 1 \\ &= 0 \cdot 1 + 0.5 \cdot (-1) + (-0.5) \cdot 2 + 0 \cdot 3 + 1 \\ &= 0 - 0.5 - 1.0 + 0 + 1 = -0.5 \end{aligned}$$

$$\begin{aligned} f_U^a &= W^+ \cdot \mathbf{u} + W^- \cdot \mathbf{l} + b \\ &= [0, 0.5] \cdot [2, 3] + [-0.5, 0] \cdot [1, -1] + 1 \\ &= 0 \cdot 2 + 0.5 \cdot 3 + (-0.5) \cdot 1 + 0 \cdot (-1) + 1 \\ &= 0 + 1.5 - 0.5 + 0 + 1 = 2.0 \end{aligned}$$

This gives us $f^a([1, 2], [-1, 3]) = [-0.5, 2.0]$, which matches exactly the result from the min/max approach in Ex. 5.2.1.

The matrix formulation of interval arithmetic is equivalent to the min/max approach but enables efficient batch processing of multiple input regions simultaneously. This is particularly useful when splitting input regions into multiple subproblems and computing abstraction in parallel.

Consider an input region that needs to be split for improving abstraction precision. For example, we can split the input region $\{x_1 \in [-1, 1], x_2 \in [-2, 2]\}$ along the first dimension into two subproblems:

1. Subproblem 1: $\{x_1 \in [-1, 0], x_2 \in [-2, 2]\}$
2. Subproblem 2: $\{x_1 \in [0, 1], x_2 \in [-2, 2]\}$

Using the matrix form in Ex. 9.3.1, we can process both subproblems simultaneously by stacking the bounds into matrices:

$$\mathbf{L} = \begin{bmatrix} -1 & -2 \\ 0 & -2 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 & 2 \\ 1 & 2 \end{bmatrix}$$

where each row represents the lower and upper bounds for one subproblem.

Next, for a linear layer with weight matrix W and bias vector $\mathbf{b}$, we can compute the bounds for all subproblems simultaneously:

$$\mathbf{O}_L = \mathbf{L} \cdot (W^+)^T + \mathbf{U} \cdot (W^-)^T + \mathbf{b}$$
$$\mathbf{O}_U = \mathbf{U} \cdot (W^+)^T + \mathbf{L} \cdot (W^-)^T + \mathbf{b}$$

where $\mathbf{1}$ is a vector of ones with appropriate dimension and $\otimes$ denotes the outer product.

Example 9.3.2. Consider the network from Ex. 5.2.1 with weight $W = [-0.5, 0.5]$ and bias $b = 1.0$. We split the input region $\{x_1 \in [-1, 1], x_2 \in [-2, 2]\}$ into two subproblems as described above.

First, we decompose the weight:

$$W^+ = [0, 0.5] \qquad W^- = [-0.5, 0] \qquad \mathbf{b} = [1.0, 1.0]$$

The bias vector is the same for both subproblems, thus, just duplicate it for each subproblem. Then we compute bounds for both subproblems:

$$\mathbf{L} \cdot (W^+)^T = \begin{bmatrix} -1 & -2 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -1.0 \\ -1.0 \end{bmatrix}$$

$$\mathbf{U} \cdot (W^-)^T = \begin{bmatrix} 0 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ -0.5 \end{bmatrix}$$

$$\mathbf{O}_L = \begin{bmatrix} -1.0 \\ -1.0 \end{bmatrix} + \begin{bmatrix} 0.0 \\ -0.5 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.0 \\ -0.5 \end{bmatrix}$$

Similarly for upper bounds:

$$\mathbf{U} \cdot (W^+)^T = \begin{bmatrix} 0 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

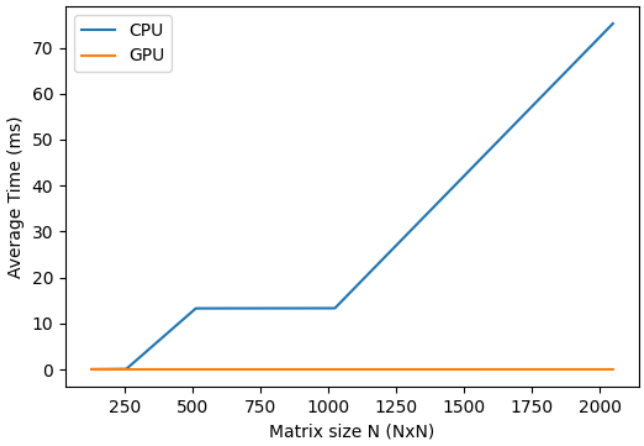


Figure 9.1: GPU Benchmarking Results.

$$\mathbf{L} \cdot (W^{-})^{T} = \begin{bmatrix} -1 & -2 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} -0.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.0 \end{bmatrix}$$

$$\mathbf{O}_{U} = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.0 \end{bmatrix} + \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 2.5 \\ 2.0 \end{bmatrix}$$

Therefore:

- Subproblem 1: $x_3 \in [0.0, 2.5]$
- Subproblem 2: $x_3 \in [-0.5, 2.0]$

We can verify these results by computing each subproblem individually using the original formulation, which yields identical results.

9.4 GPU Processing

By using matrix operations, we can leverage using GPUs natively to speed UB the computation. As expected, the computation time is reduced significantly using GPUs when the size of matrices (weights, bias, inputs, etc.) becomes larger Fig. 9.1.

```
import torch

# weights and bias
W = torch.tensor([[0, 0.5], [-0.5, 0]]).cuda()
b = torch.tensor([1.0, 1.0]).cuda()

% input bounds
L = torch.tensor([[-1, -2], [0, -2]]).cuda()
U = torch.tensor([[0, 2], [1, 2]]).cuda()
```

```
# decompose weights
W_plus = W.clamp(min=0)
W_minus = W.clamp(max=0)

# compute bounds
O_L = L @ W_plus.T + U @ W_minus.T + b
O_U = U @ W_plus.T + L @ W_minus.T + b
```

Part IV

Modern DNN Verification Tools

Chapter 10

The NeuralSAT Algorithm

NeuralSAT [22, 23] is a relatively new competitor in the DNN verification space, but it has quickly become a strong contender, consistently placed among the top tools at DNN verification competitions **??**.

At its core, NeuralSAT is BaB, but follows a DPLL(T) framework [16] and includes specialized optimizations and heuristics to improve its search. Thus, NeuralSAT is essentially an SMT solver (§B.3.1) with respect to a theory, in this case, the theory of DNNs.

10.1 Overview

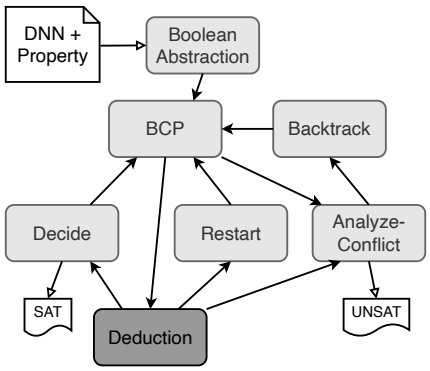


Figure 10.1: The NeuralSAT DPLL(T) Algorithm.

Fig. 10.1 gives an overview of NeuralSAT, which consists of standard DPLL components (light shades) and the theory solver (dark shade). NeuralSAT first constructs a propositional formula over Boolean variables that represent the activation status of neurons (*Boolean Abstraction*). Clauses in the formula assert that each neuron, e.g., neuron i, is active or inactive, e.g., $v_i \vee \overline{v_i}$. This representation enables using standard DPLL to search for truth values satisfying these clauses and a DNN-specific theory solver to check the feasibility of truth assignments—*activation patterns* (§6.1)— with

respect to the constraints encoding the DNN and the property of interest.

`NeuralSAT` now enters an iterative process to find activation pattern (truth assignment) satisfying the activation clauses. First, `NeuralSAT` assigns a truth value to an unassigned variable (*Decide*), detects unit clauses caused by this assignment, and infers additional assignments (*Boolean Constraint Propagation*). Next, `NeuralSAT` invokes the theory solver or T-solver (*Deduction*), which uses LP solving and abstraction to check the feasibility of the constraints of the current assignment with the property of interest.

If the T-solver confirms feasibility, `NeuralSAT` continues with new assignments (*Decide*). Otherwise, `NeuralSAT` detects a conflict (*Analyze Conflict*) and learns clauses to remember it and backtrack to a previous decision (*Backtrack*). If `NeuralSAT` detects local optima, it would restart (*Restart*) the search by clearing all decisions that have been made, but save the conflict clauses learned so far to avoid reaching the same state in the next runs. Restarting especially benefits challenging DNN problems by enabling better clause learning and exploring different decision orderings.

This iterative process repeats until `NeuralSAT` can no longer backtrack, and returns `unsat`, indicating the DNN has the property, or it finds a total assignment for all boolean variables, and returns `sat`.

§A provides more details on the `NeuralSAT` algorithm, describing the main components of `NeuralSAT` and how they work together to verify DNNs.

10.2 Illustration

Example 10.2.1. We use `NeuralSAT` to prove that for inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$ the DNN in Fig. 1.1 produces the output $x_5 \leq 0$. `NeuralSAT` takes as input the formula α representing the DNN:

$$\begin{aligned} x_3 &= \text{ReLU}\left(-0.5x_1 + 0.5x_2 + 1\right) \ \wedge \\ x_4 &= \text{ReLU}\left(x_1 + x_2 - 1\right) \ \wedge \\ x_5 &= -x_3 + x_4 - 1.0 \end{aligned}$$

and the formula ϕ representing the property:

$$\phi : -1 \leq x_1 \leq 1 \wedge -2 \leq x_2 \leq 2 \quad \Rightarrow \quad x_5 \leq 0.$$

To prove $\alpha \Rightarrow \phi$, `NeuralSAT` needs to show that *no* values of x_1, x_2 satisfying the input properties would result in $x_5 > 0$. In other words, `NeuralSAT` needs to return `unsat` for:

$$\alpha \ \wedge \ -1 \leq x_1 \leq 1 \ \wedge \ -2 \leq x_2 \leq 2 \ \wedge \ x_5 > 0. \tag{10.2.1}$$

Notation: In the following, we write $x \mapsto v$ to denote that the variable x is assigned with a truth value $v \in \{T, F\}$. This assignment can be either decided by

Table 10.1: `NeuralSAT`'s run producing `unsat`.

Iter	**BCP**	**DEDUCTION**		**DECIDE**	**ANALYZE-CONFLICT**	
		Constraints	Bounds		Bt	Learned Clauses
Init	-	$I = -1 \leq x_1 \leq 1;$ $-2 \leq x_2 \leq 2$	$-1 \leq x_1 \leq 1;$ $-2 \leq x_2 \leq 2$	-	-	$C = \{v_3 \vee \overline{v_3};\ v_4 \vee \overline{v_4}\}$
1	-	I	$x_5 \leq 1$	$\overline{v_4}$@1	-	-
2	-	$I; x_4 =$ `off`	$x_5 \leq -1$	-	0	$C = C \cup \{v_4\}$
3	v_4@0	$I; x_4 =$ `on`	$x_3 \geq 0.5; x_5 \leq 0.5$	v_3@0	-	-
4	-	$I; x_3 =$ `on`$; x_4 =$ `on`	-	-	**-1**	$C = C \cup \{\overline{v_4}\}$

`Decide` or inferred by `BCP`. We also write $x@dl$ and $\overline{x}@dl$ to indicate the respective assignments $x \mapsto T$ and $x \mapsto F$ at decision level dl.

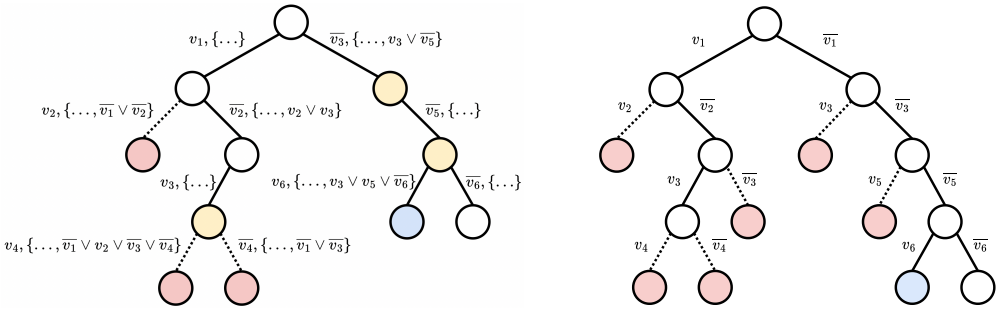
Boolean Abstraction First, `NeuralSAT` creates two Boolean variables v_3 and v_4 to represent the activation status of the hidden neurons x_3 and x_4, respectively. For example, $v_3 = T$ means x_3 is `active` and thus gives the constraint $-0.5x_1 + 0.5x_2 + 1 > 0$. Similarly, $v_3 = F$ means x_3 is `inactive` and therefore gives $-0.5x_1 + 0.5x_2 + 1 \leq 0$. Next, `NeuralSAT` forms two clauses $\{v_3 \vee \overline{v_3}\ ;\ v_4 \vee \overline{v_4}\}$ ensuring that these variables are either `active` or `inactive`.

DPLL(T) Iterations `NeuralSAT` searches for a satisfying *activation pattern*—truth assignment for the Boolean variables to satisfy the clauses and the constraints they represent with respect to the formula in Eq. 10.2.1. For this example, `NeuralSAT` uses four iterations, summarized in Tab. 10.1, to determine that no such assignment exists and the problem is thus `unsat`.

In *iteration 1*, as shown in Fig. 10.1, `NeuralSAT` starts with `BCP`, which has no effects because the current clauses and (empty) assignment produce no unit clauses. In `Deduction`, `NeuralSAT` uses an LP solver to determine that the current set of constraints, which contains just the initial input bounds, is feasible[1]. `NeuralSAT` then uses abstraction to approximate an output upper bound $x_5 \leq 1$ and thus deduces that satisfying the output $x_5 > 0$ might be feasible. `NeuralSAT` continues with `Decide`, which uses a heuristic to select the unassigned variable v_4 and sets $v_4 = F$—essentially a *guess* that neuron x_4 is inactive. `NeuralSAT` increments the decision level (dl) to 1 and associates $dl = 1$ to the assignment, i.e., $\overline{v_4}$@1.

In *iteration 2*, `BCP` again has no effect because it does not detect any unit clauses. In `Deduction`, `NeuralSAT` determines that current set of constraints, which contains $x_1 + x_2 - 1 \leq 0$ due to the assignment $v_4 \mapsto F$ (i.e., $x_4 =$ `off`), is feasible. `NeuralSAT` then approximates a new output upper bound $x_5 \leq -1$, which means satisfying the output $x_5 > 0$ constraint is *infeasible*.

[1]We use the terms feasible, from the LP community, and satisfiable, from the SAT community, interchangeably.



(a) `NeuralSAT` (b) Others

Figure 10.2: Search tree explored by `NeuralSAT` (a) and other verifiers (b) during a verification run. The notation {...} indicates learned clauses; red is infeasibility; white is feasibility; yellow is BCP application; and blue is current consideration. The search tree of `NeuralSAT` is smaller than the tree of the other techniques because `NeuralSAT` was able to prune various branches, e.g., through BCPs (e.g., v_3 and $\overline{v_5}$) and non-chronological backtracks (e.g., $\overline{v_3}$).

`NeuralSAT` now enters `AnalyzeConflict` and determines that v_4 causes the conflict (v_4 is the only variable assigned so far). From the assignment $\overline{v_4}$@1, `NeuralSAT` learns a "backjumping" clause v_4, i.e., v_4 must be T. `NeuralSAT` now backtracks to dl 0 and erases all assignments decided *after* this level. Thus, v_4 is now unassigned and the constraint $x_1 + x_2 - 1 \leq 0$ is also removed.

In *iteration 3*, `BCP` determines that the learned clause is also a unit clause v_4 and infers v_4@0. In `Deduction`, we now have the new constraint $x_1 + x_2 - 1 > 0$ due to $v_4 \mapsto T$ (i.e., $x_4 =$ `on`). With the new constraint, `NeuralSAT` approximates the output upper bound $x_5 \leq 0.5$, which means $x_5 > 0$ might be satisfiable. Also, `NeuralSAT` computes new bounds $0.5 \leq x_3 \leq 2.5$ and $0 < x_4 \leq 2.0$, and deduces that x_3 must be positive because its lower bound is 0.5. Thus, `NeuralSAT` has a new assignment v_3@0 (dl stays unchanged due to the implication). This new assignment inference from the T-solver is known as *theory propagation* in DPLL(T).

In *iteration 4*, `BCP` has no effects because we have no new unit clauses. In `Deduction`, `NeuralSAT` determines that the current set of constraints, which contains the new constraint $-0.5x_1 + 0.5x_2 + 1 > 0$ (due to $v_3 \mapsto T$), is *infeasible*. Thus, `NeuralSAT` enters `AnalyzeConflict` and determines that v_4, which was set at $dl = 0$ (by `BCP` in iteration 3), causes the conflict. `NeuralSAT` then learns a clause $\overline{v_4}$ (the conflict occurs due to the assignment $\{v_3 \mapsto T; v_4 \mapsto T\}$, but v_3 was implied and thus making v_4 the conflict). However, because v_4 was assigned at decision level 0, `NeuralSAT` can no longer backtrack and thus sets $dl = -1$ and returns `unsat`, i.e , the property is valid.

10.3 NeuralSAT vs. BaB

Note that this process of selecting and assigning (guessing) values to variables representing neurons is the *branching* phase in BaB. It is also commonly called *neuron splitting* because it splits the search tree into subtrees corresponding into the assigned values (e.g., see §10.3).

As mentioned in §3.3, ReLU-based DNN verification is NP-complete, and for difficult problem instances DNN verification tools often have to exhaustively search a very large space, making scalability a main concern for modern DNN verification.

Fig. 10.2 shows the difference between NeuralSAT and another DNN verification tool (e.g., using the popular Branch-and-Bound (BaB) approach) in how they navigate the search space. We assume both tools employ similar abstraction and neuron splitting. Fig. 10.2b shows that the other tool performs splitting to explore different parts of the tree (e.g., splitting v_1 and explore the branches with $v_1 = T$ and $v_1 = F$ and so on). Note that the other tool needs to consider the tree shown regardless if it runs sequentially or in parallel.

In contrast, NeuralSAT has a smaller search space shown in Fig. 10.2a. NeuralSAT follows the path v_1, v_2 and then $\overline{v_2}$ (just like the tool on the right). However, because of the learned clause $v_2 \vee v_3$, NeuralSAT performs a BCP step that sets v_3 (and therefore prunes the branch with $\overline{v_3}$ that needs to be considered in the other tree). Then NeuralSAT splits v_4, and like the other tool, determines infeasibility for both branches. Now NeuralSAT's conflict analysis determines from learned clauses that it needs to backtrack to v_3 (yellow node) instead of v_1. Without learned clauses and non-chronological backtracking, NeuralSAT would backtrack to decision v_1 and continues with the $\overline{v_1}$ branch, just like the other tool in Fig. 10.2b.

Thus, NeuralSAT was able to generate non-chronological backtracks and use BCP to prune various parts of the search tree. In contrast, the other tool would have to move through the exponential search space to eventually reach the same result.

Chapter 11

The Reluplex Algorithm

Reluplex [31] is a classical BaB (§6) approach for verifying neural networks. The technique extends the simplex method [41] to support the ReLU activation function (**Reluplex** = **Relu** + Sim**plex**). Reluplex has been succeeded by the Marabou [32] tool, which are more efficient and scalable. However, the core ideas of Reluplex are still relevant and therefore we present it here.

11.1 Algorithm Overview

Reluplex extends the classical simplex method so that it can handle the non-linear ReLU constraints $\hat{x} = \max(x, 0)$. Like simplex, Reluplex maintains a set of *basic* variables whose values are determined by other (non-basic) variables, and updates them through pivot operations.

Initialization The constraints representing the DNN are first rewritten into a basic form by introducing basic or *slack* variables. reluplex also forms the lower and upper bounds for each variable based the input constraints and semantics of ReLU functions, e.g., for a ReLU variable $\hat{x}_i$, the lower bound is 0.

All variable values are then initially set to some guess, such as 0, even if these values violate bounds. This initial configuration acts as the starting point for Reluplex to iteratively refine the variable values to satisfy all constraints.

Fixing bound violations Reluplex repeatedly checks all variables to see if any violate their bounds. If a non-basic variable violates its bounds, it can be directly updated to a valid value; the update is then propagated into the basic variables using the defining linear equations. If a basic variable violates its bounds, Reluplex cannot adjust it directly; instead it performs a *pivot*, swapping that basic variable with one of the non-basic variables appearing in its defining equation. After pivoting, the out-of-bounds variable becomes non-basic and can then be directly fixed.

Handling ReLU violations When all variable bounds are satisfied, Reluplex next checks the ReLU relations $\hat{x}_i = \max(x_i, 0)$. If a pair $(x_i, \hat{x}_i)$ is inconsistent (e.g., $x_i > 0$ but $\hat{x}_i = 0$), then the algorithm repairs it in the same way: if the violated

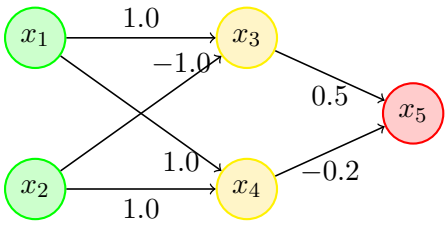


Figure 11.1: A simple DNN.

variable is non-basic, it is simply updated; if it is basic, a pivot is performed so it can be updated.

Termination If no variable violates bounds or ReLU relations, the current configuration is a feasible assignment that satisfies all constraints, and `Reluplex` returns `sat`. If `Reluplex` explores all possible pivoting options without finding any feasible configuration, it concludes `unsat`.

11.2 Illustration

Example 11.2.1. We demonstrate `Reluplex` using the DNN in Fig. 11.1. This network can be represented by the following equations:

$$\begin{aligned} x_3 &= x_1 - x_2, \quad \hat{x_3} = \mathrm{ReLU}\left(x_3\right), \\ x_4 &= x_1 + x_2, \quad \hat{x_4} = \mathrm{ReLU}\left(x_4\right), \\ x_5 &= 0.5\hat{x_3} - 0.2\hat{x_4} \end{aligned}$$

We want to check that the DNN has the property

$$(0 \leq x_1 \leq 0.5 \wedge -2 \leq x_2 \leq -1) \implies (x_5 < 0 \vee x_5 > 0.5). \tag{11.2.1}$$

That is, when the inputs x_1, x_2 fall within certain ranges, then the result x_5 has certain values. As shown in §3.2, we turn this into a satisfiability problem by negating the property:

$$(0 \leq x_1 \leq 0.5 \wedge -2 \leq x_2 \leq -1) \wedge 0 \leq x_5 \leq 0.5, \tag{11.2.2}$$

and checking whether there exists a counterexample satisfying the negated property.

We now use `Reluplex` to check whether there exists an assignment satisfying the negated property in Eq. 11.2.2. We start by introducing three *basic* (auxilliary or slack) variables to represent the relationships between the variables in the DNN:

$$a_1 = x_1 - x_2 - x_3, \tag{11.2.3}$$

$$a_2 = x_1 + x_2 - x_4, \tag{11.2.4}$$

$$a_3 = 0.5\hat{x_3} - 0.2\hat{x_4} - x_5 \tag{11.2.5}$$

These basic variables are used to maintain the relationships between the variables in the constraints and are updated during the search process. In contrast, a *non-basic* variable is one that does not represent the relationship between other variables in the constraints. In this example, all variables except a_1, a_2, a_3 are non-basic.

`Reluplex` first assigns 0 to all variables in the equations in Eq. 11.2.3– Eq. 11.2.5. This will likely violate some bounds, which `Reluplex` uses an iterative process, represented by a sequence of *configurations*, to refine.

Tab. 11.1 shows the initial configuration with all values assigned to 0. The lower (LB) and upper (UB) bounds of the inputs x_1, x_2 and output x_5 are specified in Eq. 11.2.2. The LBs of $\hat{n}_i$'s representing ReLUs are 0s, and the other hidden variables are unbounded (i.e., $-\infty$ to ∞).

Table 11.1: Configuration #1

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	x_5	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	**0**	0	0	0	0	0	0	0	0
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 11.1 also shows that x_2 is out-of-bounds because $0 \notin [-2, -1]$. To fix x_2, which is non-basic, `Reluplex` updates it to a valid value, e.g., x_2`+=` $-1.0 = -1.0$ by adding -1.0 to the current value of x_2.

Now, because $a_1 = x_1 - x_2 - x_3$ and $a_2 = x_1 + x_2 - x_4$ depend on x_2 as shown in Eq. 11.2.3 and Eq. 11.2.4, this update to x_2 also changes a_1, a_2:

$$a_1 \texttt{ += } 1.0 = 1.0,$$
$$a_2 \texttt{ += } -1.0 = -1.0.$$

Table 11.2: Configuration #2

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	x_5	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	**-1**	0	0	0	0	0	**1**	**-1**	0
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 11.2 shows the new configuration, which now has a_1, a_2 violating their bounds and need to be fixed. Assume `Reluplex` picks a_1. To fix the basic variable a_1 `Reluplex` pivots (swaps) it with a variable it depends on from the constraint $a_1 = x_1 - x_2 - x_3$ in Eq. 11.2.3. Assume `Reluplex` pivots a_1 with x_3, we get

$$x_3 = x_1 - x_2 - a_1. \tag{11.2.6}$$

`Reluplex` now updates the non-basic a_1 to 0 ($a_1 \texttt{+=} -1.0 = 0$). This also changes x_3 to 1.0 ($x_3 \texttt{+=} 1.0 = 1.0$) because x_3 depends on a_1 as shown in Eq. 11.2.6.

Tab. 11.3 shows the new configuration, which now has the basic variable a_2 out-of-bound. To fix it, `Reluplex` pivots a_2 with a variable it depends on from the

Table 11.3: Configuration #3

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	$\hat{x}_5$	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	-1	**1**	0	0	0	0	**0**	-1	0
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

constraint $a_2 = x_1 + x_2 - x_4$ in Eq 11.2.4. Assume `Reluplex` pivots a_2 with x_4, we get

$$x_4 = x_1 + x_2 - a_2 \tag{11.2.7}$$

Now a_2 becomes non-basic and is updated to 0 through $a_2 += 1.0 = 0$. As x_4 depends on a_1 as shown in Eq. 11.2.7, we make the change x_4 `-=` $1.0 = -1.0$.

Table 11.4: Configuration #4

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	x_5	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	-1	1	0	**-1**	0	0	0	**0**	0
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 11.4 shows the new configuration. At this point, we no longer have out-of-bound variables, but have inconsistent values for the pair of RELU variables $x_3, \hat{x}_3$. This is because $\hat{x}_3 = \max(x_3, 0)$ but we have $x_3 = 1$ thus $\max(1, 0) = 1$, which is not $\hat{x}_3 = 0$. Thus, `Reluplex` needs to fix either $\hat{x}_3$ or x_3.

Assume `Reluplex` picks $\hat{x}_3$. Because $\hat{x}_3$ is non-basic, we simply update it, i.e., $\hat{x}_3 = +1 = 1$. As a_3 depends on $\hat{x}_4$, i.e., $a_3 = 0.5\hat{x}_3 - 0.2\hat{x}_4 - x_5$, `Reluplex` also makes the change $a_3 += 0.5 \times 1.0 = 0.5$.

Table 11.5: Configuration #5

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	x_5	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	-1	1	**1**	-1	0	0	0	0	**0.5**
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 11.5 shows the new configuration. In the new configuration, a_3 is out-of-bound. To fix this basic variable, we pivot a_3 with one of the variables $\hat{x}_3, \hat{x}_4, x_5$ because of the constraint $a_3 = 0.5\hat{x}_3 - 0.2\hat{x}_4 - x_5$ in Eq. 11.2.5. Assume we pivot a_3 with x_5, we get

$$x_5 = 0.5\hat{x}_3 - 0.2\hat{x}_4 - a_3 \tag{11.2.8}$$

Now, a_3 becomes non-basic and we update it to 0 through $a_3 += -0.5 = 0$. As x_5 depends on a_3 as shown in Eq 11.2.8, we make the change $x_5 += 0.5 = 0.5$.

Tab. 11.6 shows the new configuration. At this point, `Reluplex` no longer sees any out-of-bound or inconsinstent values, and thus stops and returns `sat` with the

Table 11.6: Configuration #6

	x_1	x_2	x_3	$\hat{x}_3$	x_4	$\hat{x}_4$	x_5	a_1	a_2	a_3
LB	0	-2	$-\infty$	0	$-\infty$	0	0	0	0	0
Val	0	-1	1	**1**	-1	0	**0.5**	0	0	**0**
UB	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

values in the **Val** row in Tab. 11.6 as the satisfying assignment for the formula in Eq 11.2.2.

Thus, the property in Eq. 11.2.1 is *not valid* for the DNN in Fig. 11.1 because for inputs $x_1 = 0, x_2 = -1$, the network gives the output $x_5 = 0.5$, violating the property.

Problem 11.2.1. Read the example in Ex. 11.2.1 again carefully. Then:

- Rewrite the steps in Ex. 11.2.1 on paper, showing all the calculations for each step. The goal of handwriting the steps is to help you understand how `Reluplex` works in detail.
- Provide feedback on this example. Is it clear? Are there any steps or terminologies that are confusing? How can we improve the explanation? For this part you can type directly on a text file and submit.

Problem 11.2.2. Consider the description of `Reluplex` in §11.1 and the example in Ex. 11.2.1.

1. Explain the difference between a *basic* and a *non-basic* variable in `Reluplex`.
2. Why can `Reluplex` update a non-basic variable directly, but must perform a pivot before updating a basic variable? Use an example if necessary.
3. Explain why `Reluplex` must *still* consider ReLU violations after all bound violations have been fixed. Why is all variables (excluding ReLU ones) within bounds not sufficient? Use an example if necessary.

Part V

Background

Background A

Formal Methods

Neural network verification (NNV) is a rising topic that applies *formal methods* (FM) to machine learning systems, particularly neural networks (NN). This chapter explains what FM are, how verification differs from testing and debugging, and key concepts such as specifications and common FM techniques.

A.1 What Are Formal Methods (FM)?

FM are techniques for analyzing systems, e.g., computer software, using *mathematically precise models and reasoning*. In FM the behavior of a program is described and analyzed using logic and mathematics rather than informal reasoning or empirical testing. This allows formal methods to provide *guarantees* about system properties. For example, an FM might prove that a sorting algorithm always produces a correctly ordered list for any input list

Compared to Testing To motivate the need for FM, consider a simple function that computes the absolute value of an integer, where the goal is to show that the function always returns a non-negative number.

Listing A.1: Absolute value function

```
def abs_val(x):
    if x >= 0:
        return x
    else:
        return -x
```

We might *test* `abs_val` by running it on a few inputs:

```
abs_val(3) = 3
abs_val(-5) = 5
abs_val(0) = 0
...
```

The program works correctly on these inputs, returning non-negative numbers. We could run many more tests with different inputs to increase our confidence.

However, we can never test all possible integers, and therefore cannot be certain that the function works correctly for all possible inputs. This is precisely where bugs can hide, and why testing—no matter how extensive—alone cannot provide absolute guarantees about program correctness.

An FM would instead reason directly about the structure of the code and atempts to prove the statement:

$$\text{For all integers } x, \text{abs\_val}(x) \geq 0. \tag{A.1.1}$$

If FM can prove this statement, we have a mathematical guarantee that the function behaves correctly for *all integers*—not just the ones we happened to test. Thus, FM are about this kind of reasoning. They aim to establish *guarantees* and *proofs*, not probabilities or empirical evidence.

A.2 Specifications

At the heart of FM is a *specification*—a precise description of what it means for a system to be correct (or safe, or robust, etc.). A specification defines the properties we want to verify about a system.

Software engineers often write specifications informally in natural language, e.g., an algorithm should be "corect" and "fast' or a web server should be "robust against attacks". However, these informal specifications are often ambiguous and imprecise, making them vulnerable to misinterpretation and error. For example, What is correct? how fast is fast enough? How much perturbation should a system be able to handle to be considered "robust"?

Problem A.2.1. Give the formal specification for a sorting function `sort(l1)` that takes a list of integers `l1` and returns a list of integers `l2`. The specification should define what it means for the output `l2` to be a correct sorting of the input `l1`.

In contrast, a formal specification must be *unambiguous* and *mathematically expressible*—typically using logic. It precisely defines the conditions under which the system is considered correct.

For example, a desirable property of neural networks used for image classification is being *robust* to small perturbations of the input image. For this property, an informal specification might say:

"Small changes to the input image shouldn't cause the neural network to change its classification label".

In contrast a formal specification of robustness would instead say:

"For an input image x, for all images x' such that the distance between x and x' is at most ϵ, the output label of a network N for x' must be the same as for x"

This statement is more precise. It defines exactly what inputs are allowed (those within distance ϵ of x) and exactly what behavior is required (the output label must

remain the same). Typically, we even go further and express this property in formal logic, e.g.,

$$\forall x', \|x - x'\| \leq \epsilon \implies N(x') = N(x) \tag{A.2.1}$$

Once having a formal specification like this, we can apply formal verification techniques to prove or disprove it.

A.3 FM Techniques

At a high level, an FM technique is an algorithm, often implemented as a software tool that analyzes a system. The FM tool takes as *inputs*

- A *model* of the system to be analyzed
- A *specification* of the desired property of the system or program

and applies a *reasoning process* to determine whether the model satisfies the specification. The tool produces an *output* indicating whether the property holds, does not hold, or is unknown.

System Model The FM tool takes as input a *model* of the system to be analyzed. This model is a mathematical abstraction of the system's behavior. It may be a program written in a language such as C or Python, a neural network in ONNX format, or other representations such as state machines or transition systems. This model, e.g., the program, is created by the programmer and is intended to capture the desired specification (§A.2) of the system. However, the model might contain bugs or inaccuracies that violate the intended specification, which is what the FM tool aims to detect.

Example A.3.1. For example, consider the following simple system that "clips" an input value into the range [0, 10]. This system is modeled by the following Python function:

```python
def clip(x):
    if x < 0:
        return 0
    elif x > 10:
        return 10
    else:
        return x
```

The formal specification of the system is:

$$\forall x \in \mathbb{R}, 0 \leq \mathit{clip}(x) \leq 10. \tag{A.3.1}$$

FM Reasoning Process Once the model and specification are given, a FM tool applies a *reasoning process*—an algorithm—to determine whether the model satisfies the specification. §A.4 describes several major classes of FM techniques, each with its own reasoning process.

Finally, the FM algorithm/tool produces one of the *three* possible results:

1. Proved: the model satisfies the specification.
2. Disproved: the model does not satisfy the specification. Typically a counterexample is also provided to show the violation (bug).
3. Unknown: the FM tool cannot determine whether the model satisfies the specification within its resource limits (e.g., time, memory).

A.4 Major Classes of Formal Methods Algorithms

We now introduce the major classes of formal-methods techniques. In contrast to dynamic or testing-based analysis, which execute the program on specific inputs, these formal methods statically analyze the program's structure and logic to reason about all possible behaviors.

We will use the `clip` function in Ex. A.3.1 as a running example demonstrate how each technique reasons about the same model and specification.

Model Checking Model checking (MC) works by *exploring all possible states* of a system and checking whether *bad states* which violate the specification are reachable.

MC works well for systems with a finite number of states such as hardware designs and communication protocols because these systems have finite state spaces that can be exhaustively explored. However, MC struggles with systems involving continuous variables or infinite state spaces, e.g., a program with loops.

Example A.4.1. For `clip`, MC would build a state machine representing all possible execution paths of the program for different inputs. It would then check whether any path leads to a state where the output is < 0 or > 10.

A standard (naïve) MC would struggle here because the number of states is huge (the input x is a real number). So MC would need to discretize or bound the input space, e.g., by considering only integer values of x from -1000 to 1000, to make the analysis feasible.

Interactive Theorem Proving An interactive theorem prover (ITP) or *proof assistant* attempts to reason about the program using *logical inference rules* to construct a formal proof that the specification holds for all possible inputs. The reason it is called "interactive" is that the user must guide the proof process by providing lemmas, definitions, and strategies to help the prover construct the proof.

ITPs are powerful and can prove deep rich properties about programs. However, they may require significant human guidance to construct the proofs and do not scale well to large or complex systems.

Example A.4.2. For `clip`, a TP might reason as follows:

1. Case 1: If `x < 0`, `clip` returns 0, which is in $[0, 10]$
2. Case 2: If `x > 10`, `clip` returns 10, which is in $[0, 10]$
3. Case 3: Otherwise, $0 \leq x \leq 10$, `clip` returns `x`, which is in $[0, 10]$.

Satisfaiblity Checking (SAT) and Satisfaiblity Modulo Theories (SMT) Solving SAT and SMT solving are automated techniques—also referred to as *automatic theorem proving*—that reduce the verification problem to a question of *logical satisfiability*. These solvers take as input a set of logical formulae representing the program and the *negation* of the specification, and determine whether there exists an assignment of variables that makes the formulae true (satisfiable) or not (unsatisfiable). If the formulae are unsatisfiable, the property is proved. If satisfiable, the property is disproved, and the solver provides a concrete counterexample demonstrating the violation.

SAT/SMT solving is crucial to modern formal methods and software verification tools due to its automation and ability to handle complex logical constraints. However, as we will see, pure SAT/SMT solving struggle with scalability for neural networks and thus leading to the development of hybrid techniques that combine SAT/SMT solving with other methods such as abstract interpretation.

Example A.4.3. For `clip`, a SAT/SMT solver would encode the program as logical constraints:

$$(x < 0 \implies y = 0) \wedge \tag{A.4.1}$$
$$(x > 10 \implies y = 10) \wedge \tag{A.4.2}$$
$$(0 \leq x \leq 10 \implies y = x) \tag{A.4.3}$$

and the negation of the specification

$$(y < 0) \vee (y > 10) \tag{A.4.4}$$

It would then check whether these constraints are satisfiable. In this case, the solver would find that the constraints are unsatisfiable, proving that `clip` always returns a value in [0, 10].

Abstract Interpretation Abstract interpretation (AbsInt) analyzes a program by automatically computing *over-approximations* of its behavior. Instead of tracking exact values, AbsInt tracks abstract properties such as ranges or signs of variables, e.g., instead of saying "x = 5", AbsInt might say "x is in [0, 10]" or "x is non-negative".

Because AbsInt is fully automated and uses over-approximations, it is efficient and scales well to large programs. However, overapproximation comes at the cost of precision: it can cause AbsInt to produce false positives by reporting potential violations that are not real bugs.

Example A.4.4. For `clip`, an AbsInt tool might reason:

1. Input `x` is in $(-\infty, \infty)$ and output `y` is unknown initially.
2. After `if x < 0` branch, output `y` is in $[0, 0]$.
3. After `if x > 10` branch, output `y` is in $[10, 10]$.
4. After the else branch, output `y` is in $[0, 10]$.
5. Combining all branches, output `y` is in $[0, 10]$.

For neural network verification, AbsInt is often used to compute bounds on the outputs of neural network layers given bounds on the inputs. AbsInt is necessarily because neural networksare are very large and have complex nonlinear functions such as ReLU that are difficult to analyze exactly.

A.5 Soundness and Completeness

Two core concepts in formal methods are *soundness* and *completeness*. These describe the correctness and power of an FM algorithm[1].

Definition A.5.1. An FM algorithm is **sound** if every property it claims to prove is actually true. In other words, if it says *"this property holds in this system,"* then the property truly holds. Conversely, an FM algorithm is **unsound** if it can claim that a false property is true.

Two things to note about soundness:

- While a sound FM algorithm never claims a false property is true, it may claims that a true property is false (i.e., it can give false counterexamples). Thus, an extremely conservative FM algorithm that claims every property is false is still sound. Similarly, an FM algorithm that returns "unknown" for every property is also sound.

[1] Mike Hicks also wrote a good blog post on this topic: http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/.

- It is important to distinguish between an FM algorithm and its implementation (the software tool). An FM algorithm may be sound, but its implementation—typically written by human (or AI nowadays)—may contain bugs that cause it to produce unsound results.

Definition A.5.2. An FM algorithm is **complete** if it can prove every true property. That is, if the property holds, the verifier will always be able to prove it. Conversely, an FM algorithm is **incomplete** if there exist true properties that it cannot prove.

Two things to note about completeness:

- While a complete FM algorithm can prove every true property, it may also claim that a false property is true (i.e., it can produce false proofs). Thus, an extremely reckless (and not trustworth) FM algorithm that claims every property is true is still complete. Similarly, an FM algorithm that returns "unknown" for every property is also complete.

- As with soundness, it is important to distinguish between an FM algorithm and its implementation. An FM algorithm may be complete, but its implementation may contain bugs that cause it to produce incomplete results.

Ideally, FM algorithm should be both sound and complete: never prove a false property and can prove every true property. However, we can only hope to achieve one of the two properties because achieving both is often computationally infeasible for complex systems. Between the two, soundness is typically prioritized in safety-critical settings because it is better to be unsure about a system (and say "unknown" or even claim that it is unsafe) than to incorrectly claim that it is safe when it is not. Thus, most practical FM algorithms are designed to be sound but incomplete.

Background B

Logics

The topic of neural network verification (NNV) relies heavily on two fundamental areas: *logic* (to formalize and reason about properties as logical formulae) and *optimization* (to formulate and reason about numerical constraints). This part provides the necessary background on both topics in detail, starting with propositional logic and satisfiability, and then moving to linear and mixed-integer programming.

B.1 Propositional Logic and Satisfiability

Propositional, or Boolean, logic is the branch of logic that studies formulae built from Boolean variables, where each variable can take only the values `True` or `False`. These formulae are combined using logical connectives such as `and`, `or`, and `not`, and their evaluation always reduces to a single truth value.

B.1.1 Syntax

A *propositional variable* is a symbol (e.g., x, y, z). Logical formulae are built *inductively* from these variables using logical connectives as follows:

- Variables: Any propositional variable p is a formula.
- Constants: $\top$ and $\bot$ are formulae.
- Negation: If ϕ is a formula, then so is $\neg\phi$.
- Binary connectives: If ϕ, ψ are formulae, then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \to \psi)$.

Example B.1.1.

$$(x \vee y) \wedge (\neg x \vee z)$$

is a formula involving three variables x, y, z.

Special connectives In addition to the standard connectives, we define some special connectives that are often useful:

- *Implication*: $\phi \to \psi \equiv \neg\phi \lor \psi$
- *Biconditional* (or *equivalence*): $\phi \leftrightarrow \psi \equiv (\phi \to \psi) \land (\psi \to \phi)$

> Notice here that we purely define the syntax of propositional logic, which allows us to construct valid formulae. We will discuss the semantics or meanings of formulae next (§B.1.2).

Conjunctive Normal Form (CNF) A formula is in CNF if it is a *conjunction* of *clauses*, where each clause is a *disjunction* of *literals*, where a literal is either a variable p or its negation $\neg p$.

Example B.1.2.

$$(x \lor y) \land (\neg x \lor z)$$

is in CNF. Each clause $(x \lor y)$, $(\neg x \lor z)$ is a disjunction of literals.

Transforming to CNF Any formula can be transformed into CNF. The process generally involves the following steps:

1. Eliminate biconditionals and implications.
2. Move negations inward using De Morgan's laws.
3. Distribute disjunctions over conjunctions.

Problem B.1.1. Transform the following formula into CNF:

$$(x \to y) \land (y \to z)$$

B.1.2 Semantics

A logical formula in propositional logic has a truth value, which can be either `True` or `False`. For example, the formula $x \lor \neg x$ is always `True`, while the formula $x \land y$ is `True` if both x and y are `True`, and `False` otherwise. This is the *semantics* of the formula.

An *assignment* σ maps each formula to `True` or `False`. The truth value of a formula is defined recursively:

$$\begin{aligned}
\sigma(\top) &= \texttt{True}, \\
\sigma(\bot) &= \texttt{False}, \\
\sigma(\neg\phi) = \texttt{True} &\quad \text{iff } \sigma(\phi) = \texttt{False}, \\
\sigma(\phi \wedge \psi) = \texttt{True} &\quad \text{iff } \sigma(\phi) = \texttt{True} \text{ and } \sigma(\psi) = \texttt{True}, \\
\sigma(\phi \vee \psi) = \texttt{True} &\quad \text{iff } \sigma(\phi) = \texttt{True} \text{ or } \sigma(\psi) = \texttt{True}, \\
\sigma(\phi \to \psi) = \texttt{True} &\quad \text{iff } \sigma(\phi) = \texttt{False} \text{ or } \sigma(\psi) = \texttt{True}.
\end{aligned}$$

Problem B.1.2. For the formula

$$(x \vee y) \wedge (\neg x \vee z),$$

evaluate its truth value under all $2^3 = 8$ possible assignments of x, y, z.

B.2 Satisfiability and Validity

Definition B.2.1 (Satisfiability)**.** A formula ϕ

- is *satisfiable* if there exists **some** σ such that $\sigma(\phi) = \texttt{True}$.
- is *unsatisfiable* if **no** assignment satisfies it.
- is *valid* if **all** assignments satisfy it.

Example B.2.1. The formula x is satisfiable (e.g., $\sigma(x) = \texttt{True}$).
The formula

$$(x) \wedge (\neg x)$$

is unsatisfiable (no assignment works). In contrast,

$$(x \vee \neg x)$$

is valid.

Problem B.2.1. Show that $(\phi \to \psi)$ is equivalent to $(\neg\phi \vee \psi)$. (Hint: construct a truth table.)

B.2.1 SAT Checking

A SAT solver takes a propositional formula, typically in CNF, as input and determines its satisfiability by searching for a satisfying assignment σ that makes the formula true. If one exists, it returns `sat` (and optionally σ); otherwise, it reports `unsat`.

Example B.2.2.

$$\phi_2 = (x \vee y) \wedge (\neg x \vee z) \quad \Rightarrow \texttt{sat}$$
$$\phi_2 = (x) \wedge (\neg x) \quad \Rightarrow \texttt{unsat}$$

Problem B.2.2. Use the Z3 solver to check whether

$$(x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$$

is satisfiable. Provide a satisfying assignment if it exists.

B.2.2 Validity Checking

By Def. B.2.1, a formula α is valid if all assignments satisfy it. This is equivalent to saying that its negation $\neg\alpha$ is unsatisfiable (no assignment satisfies $\neg\alpha$), i.e.,

$$\text{valid}(\alpha) \iff \text{unsat}(\neg\alpha)$$

Thus, to check that α is valid, we can use a SAT solver to check that $\neg\alpha$ is `unsat`.

B.2.3 Complexity of SAT

SAT checking (and therefore validity checking) of propositional formulae is NP-Complete [14,30]. This means that unlikely there exists an efficient (i.e., polynomial-time) algorithm that can solve all SAT instances, and that in the worst case, any SAT solving algorithm may need to explore an exponential number of assignments (in the number of variables) to determine satisfiability. Thus, SAT solvers typically employ heuristics and optimizations to handle practical instances efficiently, even though the worst-case complexity remains exponential. We will discuss DPLL, a popular SAT solving algorithm, next (§C.1).

B.3 Satisfiability Modulo Theories (SMT)

SAT solvers only reason about propositional formulae, which only contain Boolean variables and logical operators such as $\wedge, \vee$. However, in verification problems, we often need to reason about arithmetic constraints $2x + 3y \leq 7; x \geq 0; x, y \in \mathbb{Z}$.

This leads to *Satisfiability Modulo Theories (SMT)*:

- SMT generalizes propositional logic by adding background theories (e.g., linear arithmetic, bit-vectors, arrays).
- An SMT formula mixes Boolean structure with constraints from these theories.

Example B.3.1. Is the formula $2x = 1$ satisfiable?

The answer depends on the chosen theory. Here, over real numbers ($\mathbb{R}$), it is SAT (e.g., $x = 0.5$), but over integers ($\mathbb{Z}$) it is UNSAT.

B.3.1 SMT Solvers

Popular SMT solvers include Z3 (Microsoft Research), CVC5, and dReal. They combine:

- A SAT solver for the Boolean skeleton.
- A *theory solver* or T-solver (e.g., linear arithmetic) for the numeric parts.

The SAT solver guesses a truth assignment for the Boolean structure. The T-solver checks whether the corresponding arithmetic constraints are feasible. This loop continues until either a satisfying assignment is found or `unsat` is proven.

Problem B.3.1. Decide the satisfiability of:

$$(x > 3) \land (y < 2) \land (x + y < 1).$$

First, do the reasoning informally by hand, and then confirm with an SMT solver such as Z3.

B.4 Z3 SMT Solver

Z3 [17] is a well-known SMT solver developed by Microsoft Research. Here we'll show how to use it to check propositional formulae and SMT formulae involving linear arithmetic constraints.

Installing You can install Z3 using your package manager, for example

```
brew install z3        # using homebrew
pip install z3-solver   # using pip
apt install z3 python3-z3 # using apt (on Debian-based Linux systems)
conda install z3solver # using conda
```

Then try its Python interface:

```
from z3 import *
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

Example B.4.1 (Propositional Logic)**.** We use Z3 to check satisfiability of propositional formulae and generate counterexamples.

```
from z3 import *

x, y = Bools('x y')
formula = And(Or(x, y), Or(Not(x), y))
s = Solver()
s.add(formula)
print(s.check())   #output: sat
print(s.model())   #a possible model is {x=True, y=True}
```

Problem B.4.1. Modify the code in Ex. B.4.1 to check

$$(x \wedge \neg x).$$

Confirm that the result is `unsat`.

Example B.4.2 (Linear Constraints.)**.** We now use Z3 as an SMT solver to check linear constraints.

```
from z3 import *
x, y = Reals('x y')

# example 1
s = Solver()
s.add(x > 0, y > 0, 2*x + y <= 1)
print(s.check())  # Output: unsat

# example 2
s = Solver()
s.add(x + y <= 4, x >= 0, y >= 0)
print(s.check()) # Output: sat
print(s.model())  # Output: {x=0, y=0}


# example 3
x, y = Ints('x y')
s = Solver()
s.add(Implies(x > 3, y > 2))
print(s.check()) # Output: sat
print(s.model()) # Output: {x=0, y=0}
```

Problem B.4.2. Use Z3 to check whether the constraints

$$x + y = 5, \quad x \geq 0, \quad y \geq 0$$

are satisfiable, and if so, ask Z3 for a model.

Problem B.4.3. Use Z3 to check the formula given in Ex. B.3.1. Do it for both the reals and integers.

Problem B.4.4. Use Z3 to formulate and check the statement "`If` $x > 3$ `then` $y > 2$ `AND` $x \leq 1$". Is it satisfiable? What model does Z3 return?

Example B.4.3 (DNN-style checking)**.** Consider a one-neuron ReLU: $y = \max(0, x-1)$. We want to check if the property "`For all` $x \leq 0$`, we have` $y = 0$" holds.

```
x, y = Reals('x y')
s = Solver()

# y = max(0, x-1)
s.add(y >= x-1, y >= 0)
s.add(Or(y == x-1, y == 0)) # ReLU
s.add(x <= 0, y != 0) # Negation of property

print(s.check()) # Output: unsat, property holds
```

Problem B.4.5. Modify the above to check property: "`For all` $x \geq 2$`, we have` $y \geq 1$." What does Z3 return?

Background C

SAT Solving Algorithms

C.1 DPLL

Fig. C.1 gives an overview of **DPLL**, a SAT solving technique introduced in 1961 by Davis, Putnam, Logemann, and Loveland [16]. DPLL is an iterative algorithm that takes as input a propositional formula and (i) decides an unassigned variable and assigns it a truth value, (ii) performs Boolean constraint propagation (BCP or also called Unit Propagation), which detects single literal clauses that either force a literal to be true in a satisfying assignment or give rise to a conflict; (iii) analyzes the conflict to backtrack to a previous decision level `dl`; and (iv) erases assignments at levels greater than `dl` to try new assignments.

These steps repeat until DPLL finds a satisfying assignment and returns `sat`, or decides that it cannot backtrack (`dl`=-1) and returns `unsat`.

C.1.1 Decide

The *Decide* step selects an unassigned variable and assigns it a truth value. Decision is the main source for the state space explosion in DPLL because it uses random choices or heuristics (e.g., VSIDS [39]) to select the variable and truth value. Thus, value assignments can be *wrong* (e.g., x should be False but is assigned True), leading to conflicts later on.

In addition, value assignment creates a new *decision level*. In the beginning, the decision level is 0 (no decisions made yet). Each time a new variable is assigned, the decision level increases by 1. Decision level is used to manage backtracking and erasing assignments discussed later.

Example C.1.1 (Simple Decision)**.** Consider the CNF:

$$\varphi = (x_1 \vee x_2) \ \wedge \ (\neg x_1 \vee x_3).$$

All variables are initially unassigned. DPLL may choose:

$$\text{Decide: } x_1 = \text{True} \qquad \text{(Decision level 1)}.$$

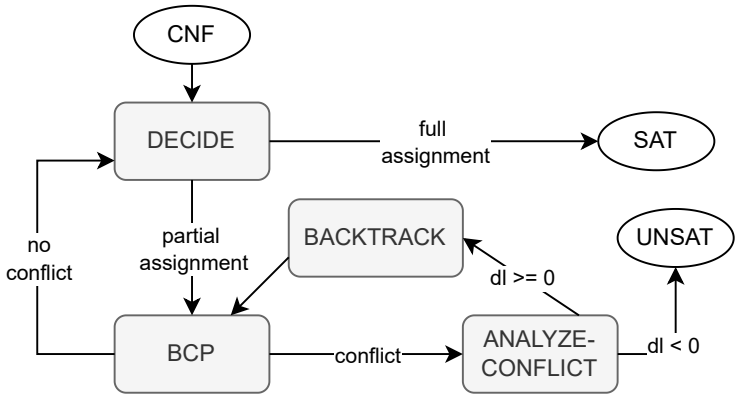


Figure C.1: The classicial DPLL Algorithm.

Example C.1.2 (Decision Using a Heuristic)**.** Given:

$$\varphi = (x_1 \vee x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_3 \vee x_2),$$

a heuristic such as VSIDS may choose the most frequent variable:

$$\text{Decide: } x_3 = \text{False}.$$

C.1.2 Boolean Constraint Propagation (BCP)

BCP detects *unit clauses*, i.e., clauses with exactly one unassigned literal and all other literals set to False. Such clauses force the remaining literal to be assigned True.

BCP is very useful because it can determine variable assignments directly without using Decide, which can guesses wrong. For example, if we have the clause $(\neg x_1 \vee x_2)$ and we know x_1 is True, then we don't need to guess x_2; it must be True to satisfy the clause.

BCP is often invoked after each decision to propagate the consequences of the assignment.

Example C.1.3 (Single Propagation)**.**

$$\varphi = (x_1) \wedge (\neg x_1 \vee x_2).$$

The unit clause (x_1) forces:

$$x_1 = \text{True}.$$

Now $(\neg x_1 \vee x_2)$ becomes $(\text{False} \vee x_2)$, hence:

$$x_2 = \text{True}.$$

Example C.1.4 (Propagation Leading to Conflict)**.**

$$\varphi = (x_1) \wedge (\neg x_1).$$

BCP assigns $x_1 = \text{True}$, immediately contradicting $(\neg x_1)$.

Example C.1.5 (Cascade of Propagations)**.**

$$\varphi = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3).$$

BCP yields:

$$x_1 = \text{True} \Rightarrow x_2 = \text{True} \Rightarrow x_3 = \text{True}.$$

C.1.3 Conflict Analysis and Backtracking

A *conflict* occurs the formula is False under the current assignment σ. More specifically, since the formula is in CNF, a conflict arises when at least one clause is False (i.e., all its literals are False).

Example C.1.6.

$$\varphi = (\neg x_1 \vee x_2) \ \wedge \ (\neg x_2) \ \wedge \ (x_1).$$

Assignments $x_1 = \text{True}$ and $x_2 = \text{False}$ make $(\neg x_1 \vee x_2) = (\text{False} \vee \text{False})$, producing a conflict.

Conflict Analysis Conflict analysis explains why the conflict occurred and generates a *learned clause* that prevents the solver from revisiting the same conflicting assignment. Most modern solvers use the 1st-UIP (Unique Implication Point) learning scheme.

Example C.1.7 (Simple Learned Clause)**.**

$$(x_1) \wedge (\neg x_1).$$

The conflict directly yields the learned clause:

$$\neg x_1 \vee x_1.$$

Because this is always false, the solver concludes the formula is unsatisfiable.

Example C.1.8 (UIP Conflict Analysis)**.** Assume decision levels:

$$x_1 = \text{True} \ (dl = 1), \qquad x_2 = \text{False} \ (dl = 2).$$

Clauses:

$$(\neg x_1 \vee x_2 \vee x_3), \qquad (\neg x_2 \vee x_3), \qquad (\neg x_3).$$

Propagation yields $x_3 = \text{False}$, causing a conflict with $(\neg x_3)$.

The implication graph is:

The learned clause (1st-UIP) is:

$$(\neg x_2 \vee \neg x_1).$$

Backtracking Given a learned clause, DPLL backtracks to the decision level at which the clause becomes unit.

Example C.1.9 (Non-Chronological Backtracking)**.** Decision levels:

$$dl = 1 : x_1 = \text{True}, \qquad dl = 2 : x_2 = \text{True}, \qquad dl = 3 : x_3 = \text{False}.$$

A conflict analysis produces the learned clause $(\neg x_1 \vee x_3)$. The highest decision level in the clause except the most recent UIP is 1, so the solver backtracks to level 1, undoing assignments to x_2 and x_3.

C.2 CDCL

Modern DPLL solving improves the original version with Conflict-Driven Clause Learning (*CDCL* [8,37,38]). DPLL with CDCL can *learn new clauses* to avoid past conflicts and backtrack more intelligently (e.g., using non-chronologically backjumping). Due to its ability to learn new clauses, CDCL can significantly reduce the search space and allow SAT solvers to scale to large problems. In the following, whenever we refer to DPLL, we mean DPLL with CDCL.

C.2.1 DPLL(T)

DPLL(T) [42] extends DPLL for propositional formulae to check SMT formulae involving non-Boolean variables, e.g., real numbers and data structures such as strings, arrays, lists. DPLL(T) combines DPLL with dedicated *theory solvers* to analyze formulae in those theories[1]. For example, to check a formula involving linear arithmetic over the reals (LRA), DPLL(T) may use a theory solver that uses linear programming to check the constraints in the formula.

Modern DPLL(T)-based SMT solvers such as Z3 [17] and CVC4 [5] include solvers supporting a wide range of theories including linear arithmetic, nonlinear arithmetic, string, and arrays [34].

[1]SMT is Satisfiability Modulo Theories and the T in DPLL(T) stands for Theories.

Background D

Linear Programming (LP)

Linear Programming (LP) and its generalization Mixed-Integer Linear Programming (MILP) form the optimization backbone of many neural network verification techniques. This chapter provides an overview of LP and MILP that are related to DNN verification.

D.1 Linear Constraints and Objectives

At a high level, LP is a method for optimizing an objective with respect to certain constraints. For example, we want to maximize profit while keeping production costs within budget. In LP, both the objective and constraints are *linear*.

Linear Constraints Linear constraints are inequalities that involve linear combinations of variables. A linear inequality has the general form

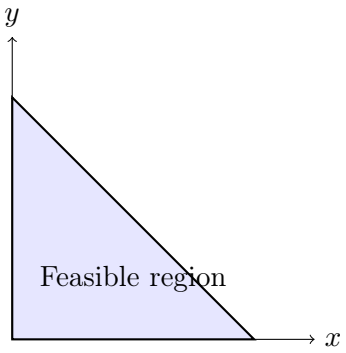
$$c_1x_1 + c_2x_2 + \cdots + c_nx_n \leq b,$$

where $c_i, b \in \mathbb{R}$. These constraints limit the feasible values that the variables x_i can take. The set of points satisfying all constraints is called the *feasible region*.

Example D.1.1. Consider the following linear constraints:

$$x + y \leq 4, \qquad x \geq 0, \qquad y \geq 0.$$

The feasible region is a triangle with vertices at $(0, 0), (4, 0), (0, 4)$.



Linear Objective Functions A linear objective function z has the general form

$$z(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n,$$

where $c_i \in \mathbb{R}$ are coefficients, and $x_i \in \mathbb{R}$ are decision variables.

Optimizing Goals Our goal is to optimize (maximize or minimize) z with respect to a set of linear constraints over the decision variables.

$$\begin{aligned}&\text{maximize } z\\&\text{subject to}\\&\{c_1x_1 + c_2x_2 + \cdots + c_nx_n \leq b, \ldots\}\end{aligned}$$

Example D.1.2. Maximize

$$1.5x + 2y$$

subject to:

$$\{x + y \leq 4, x \geq 0, y \geq 0\}.$$

First, we can solve for the intersection points of the constraints:

$$\begin{aligned}x + y &= 4\\x &= 0\\y &= 0\end{aligned}$$

Solving these equations gives us the corner points or vertices of the feasible region:

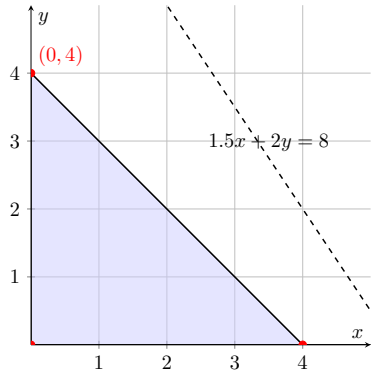
$$\begin{aligned}&(0, 0)\\&(4, 0)\\&(0, 4)\end{aligned}$$

Now, we can evaluate the objective function $z = 1.5x + 2y$ at each vertex:

x	y	z
0	0	0
4	0	6
0	4	8

Thus, the maximum value of z is 8, achieved at the vertex $(0, 4)$.

Geometric interpretation The constraints represent the feasible region as a triangle with vertices at $(0, 0)$, $(4, 0)$, and $(0, 4)$:



The objective function $z = 1.5x + 2y$ is maximized at vertex $(0, 4)$.

Problem D.1.1. Maximize $z = 4x + 5y$ subject to $\{x + y \leq 20,\ 2x + 4y \leq 72\}$.

1. Identify the corner points.
2. Evaluate the objective function at each corner point.

D.2 Mixed-Integer Linear Programming (MILP)

LP (§D.1) assumes continuous variables over real numbers. A mixed-integer linear program (MILP) extends LP by requiring some variables to be integers (often binary 0 or 1). This is useful because it allows for modeling discrete decisions and logical constraints, e.g., on/off decisions, yes/no choices, and active/inactive ReLU.

Linear Constraints In MILP, a linear constraint can involve both integer and continuous decision variables, and have the general form

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n + d_1y_1 + d_2y_2 + \cdots + d_my_m \leq b,$$

where $x_i \in \mathbb{R}$ are continuous variables, $y_j \in \mathbb{Z}$ are integer variables, and $c_i, d_j, b \in \mathbb{R}$.

Objective Function A linear objective function z in MILP can be expressed as:

$$z(x_1, \ldots, x_n, y_1, \ldots, y_m) = c_1x_1 + c_2x_2 + \cdots + c_nx_n + d_1y_1 + d_2y_2 + \cdots + d_my_m,$$

where $x_i \in \mathbb{R}$ are continuous variables, $y_j \in \mathbb{Z}$ are integer variables, and $c_i, d_j \in \mathbb{R}$.

Optimizing Goals In MILP, the goal is to find the optimal values of the decision variables x and y that maximize or minimize the objective function z, while satisfying all linear constraints.

$$\begin{aligned}
&\text{maximize } z(x_1, \ldots, x_n, y_1, \ldots, y_m) \\
&\text{subject to} \\
&\{c_1x_1 + c_2x_2 + \cdots + c_nx_n + d_1y_1 + d_2y_2 + \cdots + d_my_m \leq b\}
\end{aligned}$$

Example D.2.1. A company sells a chair for \$50 and a table for \$120. The production costs of the chair and the table are \$20 and \$70, respectively. It also takes 3 hours to produce a chair and 1 hour to produce a table. Moreover, it takes 5 units of wood to produce a chair and 20 units of wood to produce a table.

How many chairs and tables should the company produce to *maximize* profit within a week without exceeding its monthly budget of 200 units of woods, 80 hours of labor, and \$800 in production costs?

First, the decision variables are defined as follows:

- x: the number of chairs produced.
- y: the number of tables produced.

The objective function to maximize profit P is given by:

$$P = 50x + 120y - 20x - 70y = 30x + 50y$$

with subject to the constraints:

$$\begin{aligned}
&5x + 20y \leq 200 \implies x + 4y \leq 40 \\
&3x + y \leq 80 \\
&20x + 70y \leq 800 \implies 2x + 7y \leq 80 \\
&x \geq 0 \\
&y \geq 0.
\end{aligned}$$

Solving First, we can find the corner points from the constraints

$$\begin{aligned}
&x + 4y = 40 \\
&3x + y = 80 \\
&2x + 7y = 80 \\
&x = 0 \\
&y = 0
\end{aligned}$$

Solving these equations gives us the corner points or vertices of the feasible region: (0,0); (0, 10); $(\frac{80}{3}, 0)$; and $(\frac{280}{11}, \frac{40}{11})$ (from x + 4 y = 40 & 3x + y = 80). Note that all of these are feasible with $2x + 7y \leq 800$.

Now, we can evaluate the objective function P at each vertex:

x	y	P = 30x + 50y
0	0	0
0	10	500
$\frac{80}{3}$	0	800
$\frac{280}{11}$	$\frac{40}{11}$	945.45

Thus, the maximum value of P is 945.45, achieved at the vertex $(\frac{280}{11}, \frac{40}{11})$.

Of course, since we cannot produce fractional chairs or tables, we need to round these numbers to the nearest integers. This means the company should produce 24 chairs and 4 tables per week, and get a profit of $920.00

D.2.1 Encoding Binary Variables

MILP problems often involve disjunction or condition where the answer depends on some condition. We can encode such condition using binary variables (0-1 integers), e.g., using $z \in \{0, 1\}$ to indicate whether a certain condition is met. Moreover, we also use a "trick" with a large constant M, e.g., ∞, to encode logical implications.

Example D.2.2. To encode "if $z =$ `True` then $x \geq 5$", use:

$$x \geq 5 - M(1 - z), \quad z \in \{0, 1\}$$

This works because

- $z = 1 \implies x \geq 5 - M(0) \implies x \geq 5$.
- $z = 0 \implies x \geq 5 - M \implies x \geq -\infty$ (because M is large this is vacuously true and can be discarded).

Problem D.2.1. Encode the disjunction $x \geq 5 \vee x \leq 3$ in MILP.

Example D.2.3 (Encoding ReLU in MILP)**.** To encode the ReLU (§1.3.1) activation function $y = \max(0, x)$, use

$$\begin{aligned} &y \geq x, \\ &y \geq 0, \\ &y \leq x + M(1 - z), \\ &y \leq Mz, \\ &z \in \{0, 1\}. \end{aligned} \tag{D.2.1}$$

This works because

- $z = 1 \implies y = x$.
- $z = 0 \implies y = 0$.

Problem D.2.2. Consider again the ReLU example in Ex. D.2.3 but now x has the bounds $L \leq x \leq U$. Modify the MILP encoding so that L and U are used *directly* instead of a large constant M.

D.3 Using Z3 to Solve LP and MILP

We can use Z3's optimization capabilities to solve both LP and MILP problems. Note that in practice, we often use dedicated solvers such as Gurobi or CPLEX for large problems, but Z3 is effective for demonstration purposes.

Example D.3.1. We can model and solve Ex. D.1.2 using Z3 as follows:

```
from z3 import *

x, y = Reals("x y")
opt = Optimize() # setUB the optimization problem
opt.add(x + y <= 4, x >= 0, y >= 0)

# Objective
z = 1.5*x + 2*y
h = opt.maximize(z)

# Solve
if (opt.check() == sat):
    print("Optimal sol:", opt.model())  # output [x = 0, y = 4]
    print("Max value:", opt.upper(h))   # output 8
```

Problem D.3.1. Do Prob. D.1.1 using Z3.

Problem D.3.2. Do Ex. D.2.1 using Z3.

Problem D.3.3. For this problem, you will find some interesting MILP problem online, formulate it, and solve it using Z3. Specifically, do the following:

- Find *two* interesting MILP problems online. These can be from textbooks, research papers, or online resources (Youtube, Google, online tutorials).

  Note: Interesting here means the problem has a non-trivial structure, resembles real-world applications (like the table and chair example in Ex. D.2.1), or involves complex constraints.

- For each problem:
  - Write down the problem and *cite* the sources (e.g., exact URL or Author(s)/Title/Year if from a paper).

- Formulate the problem as a MILP (clearly indicate the objective and constraints).
- Solve them by hand (you can type this out, but show all the steps, if possible, draw the graph showing the feasible region).
- Implement the formulation in Z3 and solve it.

Problem D.3.4. Minimize $z = x + y$ subject to:

$$\begin{aligned} x + 2y &\geq 3, \\ 3x + y &\geq 3, \\ x, y &\geq 0, \\ x, y &\in \mathbb{R}. \end{aligned}$$

You can either do this by hand on paper (and take picture) or write Python code with Z3. Either way, show all the steps (e.g., as comments in the Z3 code):

1. The constraints and objective function.
2. The corner or candidate points (this problem is very simple that you do need an algorithm like Simplex to solve. You can simply "guess" these points, e.g., the intersection of the given constraints)
3. The objective value at each corner point.
4. The optimal solution.

D.3.1 Using LP as Feasibility Checking

In addition to optimization, LP can be used for *feasibility* (or satisfiability) checking of linear constraints, i.e., are there any values for the variables that satisfy all the constraints? This is achieved by running an LP solver with a constant objective (e.g., "minimize 0"), in which case it simply tries to find *any* point in the feasible region or shows that none exists (infeasible). This thus makes it usefful for property checking and verification (e.g., showing that no counterexample exists).

The main difference between LP feasibility and SMT satisfiability is the type of constraints they can handle. SMT satisfiability can handle richer logics involving booleans, integers, nonlinear arithmetic, and other theories while LP feasibility is limited to linear equalities/inequalities over real numbers. But for problems that involve only linear real arithmetic, LP feasibility is sufficient and often more efficient. For problems involving more complex logic or theories, SMT solvers are necessary.

Example D.3.2. For the constraints in Ex. D.1.1:

$$x + y \leq 4, \quad x \geq 0, \quad y \geq 0 \tag{D.3.1}$$

Running an LP solver with objective $\min 0$ will return a point, e.g., (x=0,y=0), within the feasible region.

Part VI

Appendices

Advanced Topics A

NeuralSAT Algorithm

Algorithm 4. The NeuralSAT DPLL(T) algorithm.

input : DNN α, property $\phi_{in} \Rightarrow \phi_{out}$
output : unsat if the property is valid and sat otherwise

```
1  clauses ← BooleanAbstraction(α)
2  while true do
3      σ ← ∅ // initial assignment
4      dl ← 0 // initial decision level
5      igraph ← ∅ // initial implication graph
6      while true do
7          is_conflict ← true
8          if BCP(clauses, σ, dl, igraph) then
9              if Deduction(σ, dl, α, φ_in, φ_out) then
10                 is_sat, v_i ← Decide(α, φ_in, φ_out, dl, σ) // decision heuristic
11                 if is_sat then return sat // total assignment
12                 σ ← σ ∧ v_i
13                 dl ← dl + 1
14                 is_conflict ← false // mark as no conflict
15         if is_conflict then
16             if dl ≡ 0 then return unsat // conflict at decision level 0
17             clause ← AnalyzeConflict(igraph)
18             dl ← Backtrack(σ, clause)
19             clauses ← clauses ∪ {clause} // learn conflict clauses
20         if Restart() then break // restart heuristic
```

Alg. 4 shows the NeuralSAT algorithm, which takes as input the formula α representing the ReLU-based DNN N and the formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property ϕ to be proved. Internally, NeuralSAT checks the satisfiability of the formula given in Eq. 3.2.3

$$\alpha \wedge \phi_{in} \wedge \neg\phi_{out}.$$

NeuralSAT returns unsat if the formula unsatisfiable, indicating that ϕ is a valid

property of N, and `sat` if it is satisfiable, indicating the N is not a valid property.

`NeuralSAT` uses a DPLL(T)-based algorithm to check unsatisfiability. First, the input formula in Eq. 3.2.3 is abstracted to a propositional formula with variables encoding neuron activation status (`BooleanAbstraction`). Next, `NeuralSAT` assign values to Boolean variables (`Decide`) and checks for conflicts the assignment has with the real-valued constraints of the DNN and the property of interest (`BCP` and `Deduction`). If conflicts arise, `NeuralSAT` determines the assignment decisions causing the conflicts (`AnalyzeConflict`), backtracks to erase such decisions (`Backtrack`), and learns clauses to avoid those decisions in the future. `NeuralSAT` repeats these decisions and checking steps until it finds a total or full assignment for all Boolean variables, in which it returns `sat`, or until it no longer can backtrack and returns `unsat`.

A.1 Boolean Abstraction

`BooleanAbstraction` (Alg. 4 line 1) encodes the DNN verification problem into a Boolean constraint to be solved by DPLL. This step creates Boolean variables to represent the *activation status* of hidden neurons in the DNN. Observe that when evaluating the DNN on any concrete input, the value of each hidden neuron *before* applying ReLU is either > 0 (the neuron is *active* and the input is passed through to the output) or ≤ 0 (the neuron is *inactive* because the output is 0). This allows partial assignments to these variables to represent neuron activation patterns within the DNN.

From the given network, `NeuralSAT` first creates Boolean variables representing the activation status of neurons. Next, `NeuralSAT` forms a set of initial clauses ensuring that each status variable is either `T` or `F`, indicating that each neuron is either active or inactive, respectively. A truth assignment over the variable v_i creates a constraint on the pre-ReLU neuron x_i.

Example A.1.1. For the DNN in Fig. 1.1, `NeuralSAT` creates two status variables v_3, v_4 for neurons x_3, x_4, respectively, and two initial clauses $v_3 \vee \neg v_3$ and $v_4 \vee \neg v_4$. The assignment $\{x_3 = T, x_4 = F\}$ creates the constraint $0.5x_1 - 0.5x_2 - 1 > 0 \wedge x_1 + x_2 - 2 \leq 0$.

A.2 DPLL

After `BooleanAbstraction`, `NeuralSAT` iteratively searches for an assignment satisfying the status clauses (Alg. 4, lines 6– 20). `NeuralSAT` combines DPLL components (e.g., `Decide`, `BCP`, `AnalyzeConflict`, `Backtrack` and `Restart`) to assign truth values with a theory solver (SA.3), consisting of abstraction and linear programming solving, to check the feasibility of the constraints implied by the assignment with respect to the network and property of interest.

`NeuralSAT` maintains several variables (Alg. 4, lines 1– 5). These include clauses, a set of *clauses* consisting of the initial activation clauses and learned clauses; σ, a *truth assignment* mapping status variables to truth values; *igraph*, an *implication graph* used for analyzing conflicts; and *dl*, a non-zero *decision level* used for assignment and backtracking.

A.2.1 Decide

From the current assignment, `Decide` (Alg. 4, line 10) uses a heuristic to choose an unassigned variable and assigns it a random truth value at the current decision level. `NeuralSAT` applies the Filtered Smart Branching (FSB) heuristic [12, 18]. For each unassigned variable, FSB assumes that it has been decided (i.e., the corresponding neuron has been split) and computes a fast approximation of the lower and upper-bounds of the network output variables. FSB then prioritizes unassigned variables with the best differences among the bounds that would help make the input formula unsatisfiable (which helps prove the property of interest). Note that if the current assignment is full, i.e., all variables have assigned values, `Decide` returns `False` (from which `NeuralSAT` returns `sat`).

A.2.2 Boolean Constraint Propagation (BCP)

From the current assignment and clauses, `BCP` (Alg. 4, line 8) detects *unit clauses*[1] and infers values for variables in these clauses. For example, after the decision $a \mapsto F$, `BCP` determines that the clause $a \vee b$ becomes unit, and infers that $b \mapsto T$. Moreover, each assignment due to `BCP` is associated with the current decision level because instead of being "guessed" by `Decide` the chosen value is logically implied by other assignments. Moreover, because each BCP implication might cause other clauses to become unit, `BCP` is applied repeatedly until it can no longer find unit clauses. BCP returns `False` if it obtains contradictory implications (e.g., one BCP application infers $a \mapsto F$ while another infers $a \mapsto T$), and returns `True` otherwise.

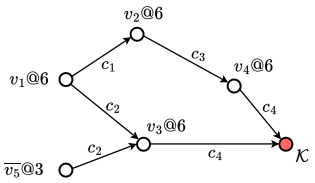
Implication Graph `BCP` uses an *implication graph* [7] to represent the current assignment and the reason for each BCP implication. In this graph, a node represents the assignment and an edge $i \xrightarrow{c} j$ means that `BCP` infers the assignment represented in node j due to the unit clause c caused by the assignment represented by node i. The implication graph is used by both BCP, which iteratively constructs the graph on each BCP application and uses it to determine conflict, and `AnalyzeConflict` (§A.2.3), which analyzes the conflict in the graph to learn clauses.

Example A.2.1. Assume we have the clauses in Fig. A.1(a), the assignments $\overline{v_5}$@3 and v_1@6 (represented in the graph in Fig. A.1(b) by nodes $\overline{v_5}$@3 and v_1@6, respectively), and are currently at decision level *dl* 6. Because of assignment v_1@6,

[1] A unit clause is a clause that has a single unassigned literal.

$c_1 = (\overline{v_1} \vee v_2)$
$c_2 = (\overline{v_1} \vee v_3 \vee$
$c_3 = (\overline{v_2} \vee v_4)$
$c_4 = (\overline{v_3} \vee \overline{v_4})$

(a)



(b)

name	cl	lit	var	ante
c_4	$\overline{v_3} \vee \overline{v_4}$	$\overline{v_3}$	v_3	c_2
	$\overline{v_4} \vee \overline{v_1} \vee v_5$	$\overline{v_4}$	v_4	c_3
	$\overline{v_1} \vee v_5 \vee \overline{v_2}$	$\overline{v_2}$	v_2	c_1
c_5	$\overline{v_1} \vee v_5$			

(c)

Figure A.1: (a) A set of clauses, (b) an implication graph, and (c) learning a new clause.

`BCP` infers v_2@6 from the unit clause c_1 and captures that implication with edge v_1@6 $\xrightarrow{c_1}$ v_2@6. Next, because of assignment v_2@6, `BCP` infers v_4@6 from the unit clause c_3 as shown by edge v_2@6 $\xrightarrow{c_3}$ v_4@6.

Similarly, `BCP` creates edges v_1@6 $\xrightarrow{c_2}$ v_3@6 and $\overline{v_5}$@ $\xrightarrow{c_2}$ v_3@6 to capture the inference v_3@6 from the unit clause c_2 due to assignments $\overline{v_5}$@3 and v_1@6. Now, `BCP` detects a conflict because clause $c_4 = \overline{v_3} \vee \overline{v_4}$ cannot be satisfied with the assignments v_4@6 and v_3@6 (i.e., both v_3 and v_4 are T) and creates two edges to the (red) node κ: v_4@6 $\xrightarrow{c_4}$ κ and v_3@6 $\xrightarrow{c_4}$ κ to capture this conflict.

Note that in this example `BCP` has the implication order v_2, v_4, v_3 (and then reaches a conflict). In the current implementation, `NeuralSAT` makes an arbitrary decision and thus could have a different order, e.g., v_3, v_4, v_2.

A.2.3 Conflict Analysis

Given an implication graph with a conflict (e.g., Fig. A.1b), `AnalyzeConflict` learns a new *clause* to avoid past decisions causing the conflict. The algorithm traverses the implication graph backward, starting from the conflicting node κ, while constructing a new clause through a series of resolution steps. `AnalyzeConflict` aims to obtain an *asserting* clause, which is a clause that will force an immediate BCP implication after backtracking.

`AnalyzeConflict`, shown in Alg. 5, first extracts the conflicting clause *cl* (line 1), represented by the edges connecting to the conflicting node κ in the implication graph. Next, the algorithm refines this clause to achieve an asserting clause (lines 2– 6). It obtains the literal *lit* that was assigned last in *cl* (line 3), the variable *var* associated with *lit* (line 4), and the antecedent clause *ante* of that *var* (line 5), which contains $\overline{lit}$ as the only satisfied literal in the clause. Now, `AnalyzeConflict` resolves *cl* and

Algorithm 5. AnalyzeConflict

```
   input  : implication graph igraph
   output : clause
1  clause ←
     CurrentConflictClause(igraph)
2  while ¬StopCriterion(clause) do
3  |  lit ← LastLiteral(igraph, clause)
4  |  var ← LiteralToVariable(lit)
5  |  ante ← Antecedent(igraph, lit)
6  |  clause ← BinRes(clause, ante, var)
7  return clause
```

ante to eliminate literals involving *var* (line 6). The result of the resolution is a clause, which is then refined in the next iteration.

Resolution. We use the standard *binary resolution rule* to learn a new clause implied by two (*resolving*) clauses $a_1 \vee ... \vee a_n \vee \beta$ and $b_1 \vee ... \vee b_m \vee \overline{\beta}$ containing complementary literals involving the (*resolution*) variable β:

$$\frac{(a_1 \vee ... \vee a_n \vee \beta) \qquad (b_1 \vee ... \vee b_m \vee \overline{\beta})}{(a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_m)} \quad \text{(Binary-Resolution)} \qquad \text{(A.2.1)}$$

The resulting (*resolvant*) clause $a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_m$ contains all the literals that do not have complements β and $\neg\beta$.

Example A.2.2. Fig. A.1(c) demonstrates `AnalyzeConflict` using the example in §A.2.2 with the BCP implication order v_2, v_4, v_3 and the conflicting clause cl (connecting to node κ in the graph in Fig. A.1(b)) $c_4 = \overline{v_3} \vee \overline{v_4}$. From c_4, we determine the last assigned literal is $lit = \overline{v_3}$, which contains the variable $var = v_3$, and the antecedent clause containing v_3 is $c_2 = \overline{v_1} \vee v_3 \vee v_5$ (from the implication graph in Fig. A.1(b), we determine that assignments $v_1@6$ and $\overline{v_5}@3$ cause the BCP implication $v_3@6$ due to clause c_2). Now we resolve the two clauses cl and c_2 using the resolution variable v_3 to obtain the clause $\overline{v_4} \vee \overline{v_1} \vee v_5$. Next, from the new clause, we obtain $lit = \overline{v_4}, var = v_4, ante = c_3$ and apply resolution to get the clause $\overline{v_1} \vee v_5 \vee \overline{v_2}$. Similarly, from this clause, we obtain $lit = \overline{v_2}, var = v_2, ante = c_1$ and apply resolution to obtain the clause $v_1 \vee v_5$.

At this point, `AnalyzeConflict` determines that this is an asserting clause, which would force an immediate BCP implication after backtracking. As will be shown in §A.2.4, `NeuralSAT` will backtrack to level 3 and erases all assignments after this level (so the assignment $\overline{v_5}@3$ is not erased, but assignments after level 3 are erased). Then, `BCP` will find that c_5 is a unit clause because $\overline{v_5}@3$ and infers $\overline{v_1}$. Once obtaining the asserting clause, `AnalyzeConflict` stops the search, and `NeuralSAT` adds $v_1 \vee v_5$ as the new clause c_5 to the set of existing four clauses.

The process of learning clauses allows `NeuralSAT` to learn from its past mistakes. While such clauses are logically implied by the formula in Eq. 3.2.3 and therefore do not change the result, they help prune the search space and allow DPLL and therefore `NeuralSAT` to scale. For example, after learning the clause c_5, together with assignment $v_5@3$, we immediately infer $v_1 \mapsto F$ through BCP instead of having to guess through `Decide`.

A.2.4 Backtrack

From the clause returned by `AnalyzeConflict`, `Backtrack` (Alg. 4, line 18) computes a backtracking level and erases all decisions and implications made after that level. If the clause is *unary* (containing just a single literal), then we backtrack to level 0.

Currently, NeuralSAT uses the standard *conflict-drive backtracking* strategy [7], which sets the backtracking level to the *second most recent* decision level in the clause. Intuitively, by backtracking to the second most recent level, which means erasing assignments made *after* that level, this strategy encourages trying new assignments for more recently decided variables.

Example A.2.3. From the clause $c_5 = \overline{v_1} \vee v_5$ learned in AnalyzeConflict, we backtrack to decision level 3, the second most recent decision level in the clause (because assignments v_1@6 and $\overline{v_5}$@3 were decided at levels 6 and 3, respectively). Next, we erase all assignments from decision level 4 onward (i.e., the assignments to v_1, v_2, v_3, v_4 as shown in the implication graph in Fig. A.1). This thus makes these more recently assigned variables (after decision level 3) available for new assignments (in fact, as shown by the example in §A.2.2, BCP will immediately infer $v_1 = T$ by noticing that c_5 is now a unit clause).

A.2.5 Restart

As with any stochastic algorithm, NeuralSAT can perform poorly if it gets into a subspace of the search that does not quickly lead to a solution, e.g., due to choosing a bad sequence of neurons to split [12, 18]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [26] to avoid being stuck in such a *local optima*.

NeuralSAT uses a simple restart heuristic (Alg. 4, line 20) that triggers a restart when either the number of processed assignments (nodes) exceeds a pre-defined number (e.g., 300 nodes) or the current runtime exceeds a pre-defined threshold (e.g., 50 seconds). After a restart, NeuralSAT avoids using the same decision order of previous runs (i.e., it would use a different sequence of neuron splittings). It also resets all internal information (e.g., decisions and implication graph) except the learned conflict clauses, which are kept and reused as these are *facts* about the given constraint system. This allows a restarted search to quickly prune parts of the space of assignments.

We found the combination of clause learning and restarts effective for DNN verification. In particular, while restart resets information it keeps learned clauses, which are *facts* implied by the problem, and therefore enables quicker BCP applications and non-chronological backtracking (e.g., as illustrated in Fig. 10.2).

A.3 Deduction (Theory Solving)

Deduction (Alg. 4, line 9) is the theory or T-solver, i.e., the T in DPLL(T). The main purpose of the T-solver is to check the feasibility of the constraints represented by the current propositional variable assignment; as shown in the formalization in S?? this amounts to just *linear equation* solving for verifying piecewise linear DNNs. However, NeuralSAT is able to leverage specific information from the DNN problem, including

Algorithm 6. DEDUCTION

input : DNN α, input property ϕ_{in}, output property ϕ_{out}, decision level dl and current assignment σ

output : false if infeasibility occurs, true otherwise

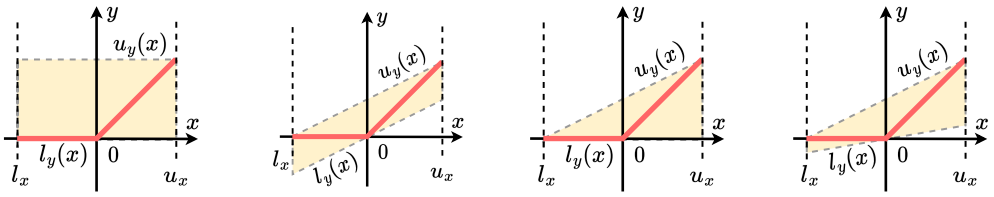
```
1  solver ← LPSolver(σ, α ∧ φ_in ∧ ¬φ_out)
2  if Solve(solver) ≡ INFEASIBLE then return false
3  if isTotal(σ) then return true // orig prob ( Eq. 3.2.3) is satisfiable
4  input_bounds ← TightenInputBounds(solver, φ_in)
5  output_bounds, hidden_bounds ← Abstract(α, σ, input_bounds)
6  if Check(output_bounds, ¬φ_out) ≡ INFEASIBLE then return false
7  for v ∈ hidden_bounds do
8      x ← ActivationStatus(v)
9      if x ∈ σ ∨ ¬x ∈ σ then continue
10     if LowerBound(v) > 0 then σ ← σ ∪ x@dl
11     else if UpperBound(v) ≤ 0 then σ ← σ ∪ x̄@dl
12 return true
```

input and output properties, for more aggressive feasibility checking. Specifically, `Deduction` has three tasks: (i) checking feasibility using linear programming (LP) solving, (i) further checking feasibility with input tightening and abstraction, and (iii) inferring literals that are unassigned and are implied by the abstracted constraint.

Alg. 6 describes `Deduction`, which returns `False` if infeasibility occurs and `True` otherwise. First, it creates a linear constraint system from the input assignment σ and $\alpha \wedge \phi_{in} \wedge \overline{\phi_{out}}$, i.e., the formula in Eq. 3.2.3 representing the original problem (line 1). The key idea is that we can remove ReLU activation for hidden neurons whose activation status have been decided. For constraints in α associated with variables that are not in the σ, we ignore them and just consider the cutting planes introduced by the partial assignment. For example, for the assignment $v_3 \mapsto T, v_4 \mapsto F$, the non-linear ReLU constraints $x_3 = ReLU(-0.5x_1 + 0.5x_2 + 1)$ and $x_4 = ReLU(x_1 + x_2 - 1)$ for the DNN in Fig. 1.1 become linear constraints $x_3 = -0.5x_1 + 0.5x_2$ and $x_4 = 0$, respectively.

Next, an LP solver checks the feasibility of the linear constraints (line 2). If the solver returns infeasible, `Deduction` returns `False` so that `NeuralSAT` can analyze the assignment and backtrack. If the constraints are feasible, then there are two cases to handle. First, if the assignment is total (i.e., all variables are assigned), then that means that the original problem is satisfiable (line 3) and `NeuralSAT` returns `sat`.

ReLU Abstraction. Second, if the assignment is not total then `Deduction` applies abstraction to check satisfiability (lines 4–6). Specifically, we over-approximate ReLU computations to obtain the upper and lower bounds of the output values and check if the output properties are feasible with respect to these bounds. For example, the output $x_5 > 0$ is *not* feasible if the upperbound is $x_5 \leq 0$ and *might be*



(a) interval (b) zonotope (c) DeepPoly (d) LiRPA

Figure A.2: Abstractions for ReLU: (a) interval, (b) zonotope, and (c-d) polytopes. Notice that ReLU is a non-convex region (red line) while all abstractions are convex regions. Note that (c) and (d) are both polytopes.

feasible if the upperbound is $x_5 \leq 0.5$ ("might be" because this is an upper-bound). If abstraction results in infeasibility, then `Deduction` returns `False` for `NeuralSAT` to analyze the current assignment (line 6).

`NeuralSAT` uses abstraction to approximate the lower and upper bounds of hidden and output neurons. Fig. A.2 compares the (a) interval [51], (b) zonotope [47], and (c, d) polytope [48,52,54] abstraction domains to compute the lower $l_y(x)$ and upper $u_y(x)$ bounds of a ReLU computation $y =$ `ReLU(x)` (non-convex red line). `NeuralSAT` can employ any existing abstract domains, though currently it adopts the *LiRPA* polytope (Fig. A.2d) [52–54] because it has a good trade-off between precision and efficiency.

Inference If abstraction results in feasible constraints, `Deduction` next attempts to infer implied literals (lines 7– 11). To obtain the bounds of the output neurons, abstraction also needs to compute the bounds of hidden neurons, including those with undecided activation status (i.e., not yet in σ). This allows us to assign the activation variable of a hidden neuron the value `True` if the lowerbound of that neuron is greater than 0 (the neuron is active) and `False` otherwise. Since each literal is considered, this would be considered exhaustive theory propagation. Whereas the literature [34,42] suggests that this is an inefficient strategy, we find that it does not incur significant overhead (average overhead is about 4% and median is 2%).

Example A.3.1. For the illustrative example in Ex. 10.2.1, in iteration 3, the current assignment σ is $\{v_4 = 1\}$, corresponding to a constraint $x_1 + x_2 - 1 > 0$. With the new constraint, we optimize the input bounds and compute the new bounds for hidden neurons $0.5 \leq x_3 \leq 2.5$, $0 < x_4 \leq 2.0$ and output neuron $x_5 \leq 0.5$ (and use this to determine that the postcondition $x_5 > 0$ might be feasible). We also infer $v_3 = 1$ because of the positive lower bound $0.5 \leq x_3$.

A.4 NeuralSAT's Optimizations

NeuralSAT implements several optimizations to improve the performance of the search. First are the common optimizations used by other DNN verifiers, such as input splittings (§9) and adversarial attacks (§7). In addition, NeuralSAT implements several unique optimizations [23] to improve the performance of the search. These are neuron stability, restart tree, and restart.

A.4.1 Neuron Stability

A neuron is *stable* if its activation status does not change regardless of the input values. In contrast, a neuron is *unstable* if its activation status can change depending on the input values. For example, in the DNN in Fig. 1.1,

The key idea in using neuron stability is that if we can determine that a neuron is stable, we can assign the exact truth value for the corresponding Boolean variable instead of having to guess. This has a similar effect as BCP—reducing mistaken assignments by Decide—but it operates at the theory level instead of the propositional Boolean level.

Stabilization involves the solution of a mixed integer linear program (MILP) system [50]:

$$\begin{aligned}
&\text{(a)} \quad z^{(i)} = W^{(i)}\hat{z}^{(i-i)} + b^{(i)};\\
&\text{(b)} \quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&\text{(c)} \quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0;\\
&\text{(d)} \quad a_j^{(i)} \in \{0,1\};\\
&\text{(e)} \quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned} \tag{A.4.1}$$

where x is input, y is output, and $z^{(i)}$, $\hat{z}^{(i)}$, $W^{(i)}$, and $b^{(i)}$ are the pre-activation, post-activation, weight, and bias vectors for layer i. The equations encode the semantics of a DNN as follows: (a) defines the affine transformation computing the pre-activation value for a neuron in terms of outputs in the preceding layer; (b) defines the inputs and outputs in terms of the adjacent hidden layers; (c) asserts that post-activation values are non-negative and no less than pre-activation values; (d) defines that the neuron activation status indicator variables that are either 0 or 1; and (e) defines constraints on the upper, $u_j^{(i)}$, and lower, $l_j^{(i)}$, bounds of the pre-activation value of the jth neuron in the ith layer. Deactivating a neuron, $a_j^{(i)} = 0$, simplifies the first of the (e) constraints to $\hat{z}_j^{(i)} \leq 0$, and activating a neuron simplifies the second to $\hat{z}_j^{(i)} \leq z_j^{(i)}$, which is consistent with the semantics of $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

Alg. 7 describes Stabilize solves this equation system. First, a MILP problem is created from the current assignment, the DNN, and the property of interest using formulation in Eq. A.4.1. Note that the neuron lower ($l_j^{(i)}$) and upper bounds ($u_j^{(i)}$)

Algorithm 7. `Stabilize`

input : DNN α, property $\phi_{in} \Rightarrow \phi_{out}$, current assignment σ, number of neurons for stabilization k

output : Tighten bounds for variables **not** in σ (unassigned variables)

```
1  model ← MIP(α, φ_in, φ_out, σ) // create model ( Eq. A.4.1) with current
       assignment
2  [v_1, ..., v_m] ← GetUnassignedVariable(σ) // get all m current unassigned
       variables
3  [v'_1, ..., v'_m] ← Sort([v_1, ..., v_m]) // prioritize tightening order
4  [v'_1, ..., v'_k] ← Select([v'_1, ..., v'_m], k) // select top-k unassigned variables,
       k ≤ m
   // stabilize k neurons in parallel
5  parfor v_i in [v'_1, ..., v'_k] do
6      if (v_i.lower + v_i.upper) ≥ 0 then // lower is closer to 0 than upper,
        optimize lower first
7          Maximize(model, v_i.lower) // tighten lower bound of v_i
8          if v_i.lower < 0 then // still unstable
9              Minimize(model, v_i.upper) // tighten upper bound of v_i
10     else // upper is closer to 0 than lower, optimize upper first
11         Minimize(model, v_i.upper) // tighten upper bound of v_i
12         if v_i.upper > 0 then // still unstable
13             Maximize(model, v_i.lower) // tighten lower bound of v_i
```

can be quickly computed by polytope abstraction.

Next, it collects a list of all unassigned variables which are candidates being stabilized (line 2). In general, there are too many unassigned neurons, so `Stabilize` restricts consideration to k candidates. Because each neuron has a different impact on abstraction precision we prioritize the candidates. In `Stabilize`, neurons are prioritized based on their interval boundaries (line 3) with a preference for neurons with either lower or upper bounds that are closer to zero. The intuition is that neurons with bounds close to zero are more likely to become stable after tightening.

We then select the top-k (line 4) candidates and seek to further tighten their interval bounds. The order of optimizing bounds of select neurons is decided by its boundaries, e.g., if the lower bound is closer to zero than the upper bound then the lower bound would be optimized first. These optimization processes, i.e., `Maximize` (line 7 or line 13) and `Minimize` (line 9 or line 11), are performed by an external LP solver (e.g., Gurobi [28]).

Example A.4.1.

A.4.1.1 Parallel Search

The DPLL(T) process in `NeuralSAT` is designed as a tree-search problem where each internal node encodes an *activation pattern* defined by the variable assignments

from the root. To parallelize DPLL(T), we adopt a beam search-like strategy which combines distributed search from Distributed Tree Search (DTS) algorithm [24] and Divide and Conquer (DNC) [**?**] paradigms for splitting the search space into disjoint subspaces that can be solved independently. At every step of the search algorithm, we select UB to n nodes of the DPLL(T) search tree to create a beam of width n. This splits (like DNC) the search into n subproblems that are independently processed. Each subproblem extends the tree by a depth of 1.

Our approach simplifies the more general DNC scheme since the n bodies of the **parfor** on line **??** of Alg. 4 are roughly load balanced. While this is a limited form of parallelism, it sidesteps one of the major roadblocks to DPLL parallelism – the need to efficiently synchronize across load-imbalanced subproblems [**?**, 35].

In addition to raw speedUB due to multiprocessing, parallelism accelerates the sharing of information across search subspaces, in particular learned clause information for DPLL. In `NeuralSAT`, we only generate independent subproblems which eliminates the need to coordinate their solution. When all subproblems are complete, their conflicts are accumulated, Alg. 4 line **??**, to inform the next round of search.

A.4.2 Restart

As with any stochastic algorithm, `NeuralSAT` would perform poorly if it gets into a subspace of the search that does not quickly lead to a solution, e.g., due to choosing a bad sequence of neurons to split [18, 25, 52]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [26] to avoid being stuck in such a *local optima*.

`NeuralSAT` uses a simple restart heuristic that triggers a restart when either the number of processed assignments (nodes) exceeds a pre-defined number or the number of remaining assignments that need be checked exceeds a pre-defined threshold. After a restart, `NeuralSAT` avoids using the same decision order of previous runs (i.e., it would use a different sequence of neuron splittings). It also resets all internal information except the learned conflict clauses, which are kept and reused as these are *facts* about the given constraint system. This allows a restarted search to quickly prune parts of the space of assignments. Although restarting may seem like an engineering aspect, it plays a crucial role in stochastic algorithms, and helps `NeuralSAT` reduce verification time for challenging problemsk.

Part VII

Programming Assignments and Schedule

Assignments A

Programming Assignments

A.1 PA1: Symbolic Execution of Neural Networks

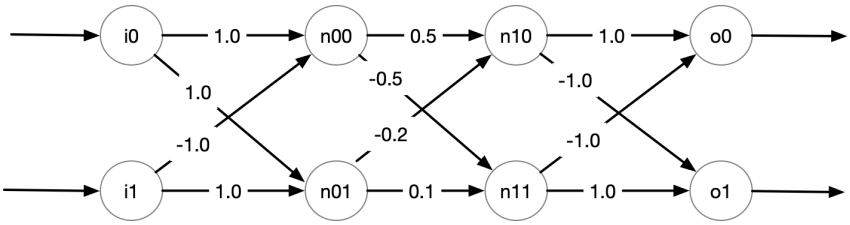
In this PA you will be implementing a neural network (NN) verifier using symbolic execution (SE) as described in §4.1. You can consider NNs as a specific type of programs and thus can "execute" it. SE runs the NN on symbolic inputs and returns symbolic outputs, i.e., a constraint representation. You will represent the symbolic outputs as a logical formula and use a constraint solver to check *assertions*, i.e., properties about the DNN. This PA has two parts: (1) implement SE on a given DNN, and (2) evaluate the scalability of your SE on various DNNs.

You will use Python and the Z3 SMT solver (§B.4) for this assignment. **Do not** use external libraries or any extra tools.

Finally, you will need to answer some questions about your implementation and findings in the provided `README` template. https://github.com/dynaroars/book_nnv/blob/main/pa/templates/README_PA1.md

A.1.1 Part1: Symbolic Execution

Consider the following fully-connected network with 2 inputs, 2 hidden layers, and 2 outputs.



1. **DNN Encoding**: We can encode this DNN using Python:

```
# (weights, bias, use activation function relu or not)
n00 = ([1.0, -1.0], 0.0, True)
```

```
n01 = ([1.0, 1.0], 0.0, True)
hidden_layer0 = [n00,n01]

n10 = ([0.5, -0.2], 0.0, True)
n11 = ([-0.5, 0.1], 0.0, True)
hidden_layer1 = [n10, n11]

# don't use relu for outputs
o0 = ([1.0, -1.0], 0.0, False)
o1 = ([-1.0, 1.0], 0.0, False)
output_layer = [o0, o1]

dnn = [hidden_layer0, hidden_layer1, output_layer]
```

In this DNN, the outputs of the neurons in the hidden layers (prefixed with `n`) are applied with the `relu` activation function, but the outputs of the DNN (prefixed with `o`) are not. These settings are controlled by the `True`, `False` parameters as shown above. Also, this example does not use `bias`, i.e., bias values are all 0.0's as shown.

Note that all of these settings are parameterized and I deliberately use this example to show these how these parameters are used (e.g., `relu` only applies to hidden neurons, but not outputs).

2. **Symbolic States**: After performing symbolic execution on `dnn`, we obtain symbolic states, represented by a logical formula relating input and output variables.

```
# my_symbolic_execution is something you implement,
# it returns a single (but large) formula representing the symbolic states.
symbolic_states = my_symbolic_execution(dnn)
...
``done, obtained symbolic states for DNN in 0.1s''
assert z3.is_expr(symbolic_states)  #symbolic_states is a Z3 expression

# your symbolic states should look something like this
print(symbolic_states)
# And(n0_0 == If(i0 + -1*i1 <= 0, 0, i0 + -1*i1),
#     n0_1 == If(i0 + i1 <= 0, 0, i0 + i1),
#     n1_0 ==
#     If(1/2*n0_0 + -1/5*n0_1 <= 0, 0, 1/2*n0_0 + -1/5*n0_1),
#     n1_1 ==
#     If(-1/2*n0_0 + 1/10*n0_1 <= 0, 0, -1/2*n0_0 + 1/10*n0_1),
#     o0 == n1_0 + -1*n1_1,
#     o1 == -1*n1_0 + n1_1)
```

3. **Constraint Solving**: We will use `Z3` to query various things about this DNN from the obtained symbolic states. Examples include:

   (a) Generating random inputs and obtain outputs. Note that these are random values (that satisfy the symbolic states), so your results might look different.

```
z3.solve(symbolic_states)
[n0_1 = 15/2,
    o1 = 1/2,
    o0 = -1/2,
    i1 = 7/2,
    n1_1 = 1/2,
    n1_0 = 0,
    i0 = 4,
    n0_0 = 1/2]
```

(b) Simulating a concrete execution.

```
i0, i1, n0_0, n0_1, o0, o1 = z3.Reals("i0 i1 n0_0 n0_1 o0 o1")

# finding outputs when inputs are fixed [i0 == 1, i1 == -1]
g = z3.And([i0 == 1.0, i1 == -1.0])
z3.solve(z3.And(symbolic_states, g))  # you should get o0, o1 = 1, -1
[n0_1 = 0,
o1 = -1,
o0 = 1,
i1 = -1,
n1_1 = 0,
n1_0 = 1,
i0 = 1,
n0_0 = 2]
```

(c) Checking assertions, i.e., verifying/proving properties about the DNN or disproving/finding counterexamples

```
print("Prove that if (n0_0 > 0.0 and n0_1 <= 0.0) then o0 > o1")
g = z3.Implies(z3.And([n0_0 > 0.0, n0_1 <= 0.0]), o0 > o1)
print(g)  #  Implies (And(n0_0 > 0, n0_1 <= 0), o0 > o1)
z3.prove(z3.Implies(symbolic_states, g))  # proved

print("Prove that when (i0 - i1 > 0 and i0 + i1 <= 0), then o0 > o1")
g = z3.Implies(z3.And([i0 - i1 > 0.0, i0 + i1 <= 0.0]), o0 > o1)
print(g)  # Implies(And(i0 - i1 > 0, i0 + i1 <= 0), o0 > o1)
z3.prove(z3.Implies(symbolic_states, g))
# proved

print("Disprove that when i0 - i1 >0, then o0 > o1")
g = z3.Implies(i0 - i1 > 0.0, o0 > o1)
print(g)  # Implies(And(i0 - i1 > 0, i0 + i1 <= 0), o0 > o1)
z3.prove(z3.Implies(symbolic_states, g))
# counterexample  (you might get different counterexamples)
# [n0_1 = 15/2,
# i1 = 7/2,
# o0 = -1/2,
# o1 = 1/2,
# n1_0 = 0,
# i0 = 4,
# n1_1 = 1/2,
# n0_0 = 1/2]
```

A.1.2 TIPS

1. You should apply symbolic execution on this DNN example **by hand** first before attempt any coding. This example is small enough that you can work

out step by step. For example, you can do these two steps

(a) First, obtain the symbolic states by hand (e.g., on a paper) for the given DNN

(b) Then, put what you have in code but also for the given DNN. Use Z3 format (e.g., you would declare the inputs as symbolic values `i0 = z3.Real("i0")`, then compute the neurons and outputs, etc)

(c) Finally, convert what you have into a general program that would work for any DNN inputs.

2. You can use the below method to construct Z3 formula for ReLU

```
@classmethod
def relu(cls, v):
   return z3.If(0.0 >= v, 0.0, v)
```

3. Two ways of doing symbolic execution to obtain symbolic states

   (a) You can follow the traditional SE method which produces the symbolic execution trees in which each condition representing ReLU forks into two paths. You can decide whether to do a depth-first search or breadth-first search here (they will have the same results). At the end, the symbolic states are a disjunction of the path conditions (i.e., `z3.And[path_conds]`).

   (b) But since we are using Z3, a much much simpler way, is to encode all those forked paths directly as a formula. For example

```
if (x + y > 0):
   r = 3
else:
   r = 4
```

   Using traditional SE, you would have 2 paths, e.g., path 1: `x+y > 0 && r = 3` and path 2 : `x+y <= 0 && r ==4`, from which you take the disjunction and get `(x+y > 0 && r == 3) || (x+y <= 0 && r ==4)`. But for this assignment, instead of having to fork into two paths, you can use Z3 to encode both branches using the `If` function, e.g., `If(x+y>0, r==3, r==4)` or `r==If(x+y>0,3,4)`. The results of these two methods are exactly the same at the end, just that the prior, traditional one you do more work while the later you do less. It is up to you.

4. When `bias` is none-zero, then it will simply be added to the neuron computation, i.e., `neuron = relu(sum(value_i * weight_i) + bias)`. For example, if `bias` is `0.123` then for neuron `n0_0` we would obtain `n0_0 == If(i0 + -1*i1 + 0.123 <= 0, 0, i0 + -1*i1 + 0.123)`.

A.1.3 Part 2: Evaluation

In this part you will show that symbolic execution *does not* scale on large DNNs.

1. You will create a function `create_random` that takes as input a list of positive integers representing the shape of the networks `[#inputs, #neurons, ..., #outputs]`, and create a neural network of that size with *random* weights and bias (e.g., between say -5 and 5).
2. You will run symbolic execution on this network as you did in Part 1, get its results, and time the solving process.
3. You will do this for various sizes of networks **until your computer cannot handle it anymore** (e.g., until it takes more than a minute or two).
4. You will **save these networks** because you will reuse them to measure and compare execution time using abstraction in Part 2.
5. Finally, you will plot the results: the x-axis will show the size of the networks (for simplicity, the product of all numbers in the input list) and the y-axis shows the time.

Feel free to be creative in this process (e.g., play around with different sizes/layers), and discuss your findings in the `README` file. Below gives some code example:

```
# Create a random network from a given
# number of inputs, hidden neurons, and outputs
l = [2,3,4,3,3]   # 2 inputs, 3 hidden layers (with 3,4,3 hidden neurons for
                  # respective hidden layer 1, 2, 3), and 3 outputs.
dnn = create_random_nn(l)

symbolic_state = my_symbolic_execution(dnn)

# time the execution
import time
st = time.time()
_ = z3.solve(symbolic_state)
print('time to solve: ', time.time() - st)
```

A.1.4 Conventions and Requirements

Your program must have the following:

Part 1

1. a function `my_symbolic_execution(dnn)` that
   - takes as input the DNN, whose specifications are given as example above.

  - This goes without saying: do not hard-code the DNN in your program (e.g., do not hardcode the example given above in your code). Your program should work with any DNN input.
- returns *symbolic states* of the DNN represented as a Z3 expression (a formula) as shown above
- this goes without saying but do NOT write this function only to work with the given example, i.e., do not hard-code the weight values etc in your program. This function should work with any DNN input (though it could be slow for big DNN's).

2. in your resulting formula, you must name
   - the nth input as i_n (e.g., the first input is i_0)
   - the nth output as o_n (e.g., the second output is o_1)
   - the jth neuron at the ith layer as n_{i_j}. Note that the first layer is the 0th layer
3. a testing function `test()` where you copy and paste pretty much the complete examples given above to demonstrate that your SE works for the given example. In summary, your `test` will include
   - the specification of the 2x2x2x2 DNN in the Figure
   - run the symbolic execution on the dnn (as shown above, it should output the dimension of the dnn and runtime)
   - output the symbolic states results
   - apply Z3 to the symbolic states to obtain
     - random inputs and associated outputs
     - simulate concrete execution
     - checking the 3 assertions as shown
4. another testing function `test2()` where you create another DNN:
   - The DNN will have
     - the same number of 2 inputs and 2 outputs, but 3 hidden layers where each layer has 4 neurons, i.e., a 2x4x4x4x2 DNN.
     - non-0.0 bias values.
   - then on this DNN, do every single tasks you did in `test()` (run SE on it, output results, apply Z3 etc). You will need to have 2 assertions that are true and 1 assertion that is false (just like above).

  - Initially you might randomly generate weights and bias values, but it is likley that you will need to manually adjust them so that you can prove some correct assertions (randomly generated values probably will not give you a meaningful DNN with any asserted properties).

You can be as creative as you want, but your SE must not run for too long and must not take up too much space (e.g., do not generate over 50MB of files). Use the `README` to tell me exactly how to compile/run your program, e.g., `python3 se.py`.

Part 2

1. a function `create_random(sizes:int)` that
   - takes as input the list `sizes` and returns a DNN of that size as descibed above.
2. a testing function `test3()` that runs `create_random` on various sizes of DNNs, runs `my_symbolic_execution` on them as you did with `test()` and `test2()` above, and time the execution.
3. a plot (in pdf, png, or jpeg) showing the scalability of `my_symbolic_execution` on various DNNs. You need to use **at least 25 DNNs of different sizes** to generate the plot, a plot with very few datapoints will not be sufficient to show the scalability trend.

A.1.5 What to Turn In

You must submit a **zip file** containing the following files. Use the following names:

1. The zip file must be named `pa1-[yourname/groupname].zip` (1 submission per group)
2. One single main source file `se.py`. Indicate in the `README` file on how I can run it.
   - Your file should include at least a `my_symbolic_execution` function and the test functions `test()`, `test2()`, and `test3()` as indicated above
3. The completed `README.md` file. You *must* use the provided template and answer all questions in it.
4. Nothing else, do not include any binary files or other directory like `__pycache__`, `.venv`.

A.1.6 Grading Rubric (out of 30 points)

- 12 points — for complete SE implementation.
  - 4 points for `my_symbolic_execution`
  - 2 points for `test1`
  - 2 points for `test2`
  - 4 points for `create_random` and `test3`
- 2 points — for code quality and submission format.
- 16 points — for completing the `README` file.
  - 4 points for Part 1
  - 10 points for Part 2 (plot and discussion)
  - 2 points for the running instructions and screenshots

A.2 PA2: Abstract Domain Analysis of Neural Networks

In this PA you will implement neural network (NN) verification using (1) *interval abstraction* as described in §5.2.1 and (2) *zonotope abstraction* as described in §5.2.2. While symbolic execution (PA1 §A.1) computes exact symbolic representations, abstract interpretation uses over-approximations to make verification more scalable at the cost of some precision. You will implement interval and zonotope abstract transformers and compare their precision-scalability trade-offs with symbolic execution.

You will use Python and PyTorch for this assignment. **Do not** use external verification libraries (other than standard numerical libraries like `torch`, `numpy`).

This PA has three main parts:

1. implement interval abstraction for linear layers and ReLU activations.
2. implement zonotope abstraction for linear layers and ReLU activations.
3. evaluate the precision/scalability compared to symbolic execution from PA1.

A README template is provided. Please answer the questions in the template and include it in your submission. https://github.com/dynaroars/book_nnv/blob/main/pa/templates/README_PA2.md

A.2.1 Interval Abstraction

Interval abstraction is the simplest form of abstract interpretation for neural networks. Each variable is represented by an interval $[l, u]$ indicating its lower and upper bounds.

A.2.1.1 Part 1: Interval for Linear Layers

Linear transformations on intervals can be computed exactly using interval arithmetic.

```
class LinearTransformer:
    def __init__(self, W, b):
        # TODO: store weight matrix and bias vector

    def forward(self, input_lower, input_upper):
        """
        Apply affine transformation f(x) = Wx + b to intervals [l, u]

        Mathematical formulation:
        - W+: positive part of weights
        - W-: negative part of weights
        - output_lower = W+ @ input_lower + W- @ input_upper + bias
        - output_upper = W+ @ input_upper + W- @ input_lower + bias

        Args:
            input_lower: lower bounds of input intervals
            input_upper: upper bounds of input intervals

        Returns:
            (output_lower, output_upper): transformed interval bounds
        """
        # TODO:
        # 1. Split weights into positive and negative parts
        # 2. Compute output bounds using interval arithmetic
        # 3. Add bias to both bounds
        # 4. Return (output_lower, output_upper)

        return output_lower, output_upper
```

A.2.1.2 Part 2: Interval for ReLU Activations

ReLU transformation on intervals is straightforward: clamp lower bounds to 0.

```
class ReluTransformer:

    def forward(self, input_lower, input_upper):
        """
        Apply ReLU transformation to intervals: ReLU(x) = max(0, x)
        Args:
            input_lower: lower bounds of input intervals
            input_upper: upper bounds of input intervals

        Returns:
            (output_lower, output_upper): intervals after ReLU
        """
        # TODO: For each dimension:
        # 1. Clamp lower bounds to be >= 0
        # 2. Clamp upper bounds to be >= 0
        # 3. Return transformed bounds

        return output_lower, output_upper
```

A.2.1.3 Part 3: Putting It All Together

```
class IntervalAbstraction:

    def __init__(self, DNN):
        # TODO: convert DNN to corresponding interval abstraction transformers
        self.transformers = ...

    def forward(self, lb, ub):
        # propagate zonotope through layers
        for transformer in self.transformers:
            lb, ub = ...

        return (lb, ub)
```

A.2.1.4 Part 4: Test Your Implementation

In this DNN, the outputs of the neurons in the hidden layers (prefixed with `n`) are applied with the `relu` activation function, but the outputs of the DNN (prefixed with `o`) are not. These settings are controlled by the `True`, `False` parameters as shown above. Also, this example does not use `bias`, i.e., bias values are all 0.0's as shown.

```
# DNN specification
n00 = ([1.0, 1.0], 2.0, True)
n01 = ([1.0, -1.0], 2.0, True)
hidden_layer0 = [n00, n01]

n10 = ([1.0, 1.0], -0.5, True)
n11 = ([1.0, -1.0], -1.0, True)
hidden_layer1 = [n10, n11]

n20 = ([-1.0, 1.0], 7.0, False)
n21 = ([0.0, 1.0], 0.0, False)
hidden_layer2 = [n20, n21]

o0 = ([1.0, -1.0], 0.0, False)
output_layer = [o0]

dnn = [hidden_layer0, hidden_layer1, hidden_layer2, output_layer]
```

Note that all of these settings are parameterized and I deliberately use this example to show these how these parameters are used (e.g., `relu` only applies to hidden neurons, but not outputs).

```
net = IntervalAbstraction(dnn)
input_lower = torch.tensor([-1, -1], dtype=torch.float)
input_upper = torch.tensor([1, 3], dtype=torch.float)
output_lower, output_upper = net(input_lower, input_upper)
print(f'{output_lower=}')
print(f'{output_upper=}')
# output_lower=tensor([-7.5000])
# output_upper=tensor([12.])
```

A.2.2 Zonotope Abstraction

A.2.2.1 Part 1: Zonotope Representation

From Bounds to Zonotope A zonotope is represented by a tuple of a center vector and a generator matrix. Given the bounds of a zonotope, we can convert it to a zonotope. For a concrete example, consider Ex. 5.2.6.

```
def zonotope(lb, ub):
    """
    Computes zonotope from interval bounds [l, u].
    TODO:
    - compute center from bounds
    - generator from bounds
    """
    return center, generator
```

From Zonotope to Bounds Given the center and generators of a zonotope, we can concretize the bounds of the zonotope.

```
def concrete(center, generator):
    """
    Computes interval bounds [l, u] from zonotope (center, generator).

    A zonotope $\mathcal{Z} = \{c + \sum_{i=1}^{m} \epsilon_i g_i \mid \epsilon_i \in [-1, 1]\}$ can be converted to interval bounds:
    Lower bound: $l = c - \sum_{i=1}^{m} |g_i|$ (when all $\epsilon_i = -\text{sign}(g_i)$)
    Upper bound: $u = c + \sum_{i=1}^{m} |g_i|$ (when all $\epsilon_i = +\text{sign}(g_i)$)

    Args:
        center: Center vector c of the zonotope
        generator: Generator matrix G

    Returns:
        (lb, ub): Interval bounds for each dimension

    # TODO: compute interval bounds from zonotope
    # 1. compute lower bound
    # 2. compute upper bound
    """
    return lb, ub
```

A.2.2.2 Part 2: Zonotope for Linear Layers

Linear (affine) transformations can be handled exactly in zonotope abstraction. You will implement the missing components in the provided zonotope framework.

```
class LinearTransformer(nn.Module):
    def __init__(self, W, b):
        # TODO: store weights and bias

    def forward(self, center, generator):
        """
        Apply affine transformation f(x) = Wx + b to zonotope Z = (c, G)
        Result: f^a(Z) = (Wc + b, WG)

        Args:
```

```
            center: center vector c of input zonotope
            generator: generator matrix G of input zonotope

        Returns:
            (new_center, new_generator): transformed zonotope
        """
        # TODO:
        # 1. Transform center using weight matrix and bias
        # 2. Transform generator matrix using weight matrix
        # 3. Return transformed (center, generator)

        return new_center, new_generator
```

A.2.2.3 Part 3: Zonotope for ReLU Activations

ReLU activations are non-linear and require over-approximation in zonotope abstraction. For each neuron with bounds $[l, u]$, there are three cases to handle.

```
class ReluTransformer(nn.Module):

    def forward(self, center, generator):
        """
        Apply ReLU abstraction to zonotope representation: ReLU(x) = max(0, x)

        Three cases for each neuron i with bounds [l_i, u_i]:
        1. Active case (l_i >= 0): ReLU is identity, Z' = Z
        2. Inactive case (u_i <= 0): ReLU outputs 0, Z' = {0}
        3. Unstable case (l_i < 0 < u_i): requires over-approximation

        Args:
            center: center vector of input zonotope
            generator: generator matrix of input zonotope

        Returns:
            (new_center, new_generator): over-approximated zonotope after ReLU
        """
        # TODO:
        # 1. Compute current bounds using concrete(center, generator)
        # 2. Create new generator matrix with extra row for ReLU error
        # 3. For each neuron, handle based on bounds [l, u]:
        #    - Active (l >= 0): keep unchanged
        #    - Inactive (u <= 0): set to zero
        #    - Unstable (l < 0 < u): apply slope approximation
        # 4. Return transformed (center, generator)

        return new_center, new_generator
```

A.2.2.4 Part 4: Putting It All Together

```
    class ZonotopeAbstraction(nn.Module):

        def __init__(self, DNN):
            # TODO: convert DNN to corresponding zonotope abstraction transformers
            self.transformers = ...

        def forward(self, lb, ub):
            # 1. convert bounds to zonotope
```

```
        center, generator = ...
        # 2. propagate zonotope through layers
        for transformer in self.transformers:
            center, generator = ...

        # 3. convert zonotope to bounds
        lb, ub = ...
        return (lb, ub)
```

A.2.2.5 Part 5: Test Your Implementation

```
net = ZonotopeAbstraction(dnn) # same DNN specification as interval
input_lower = torch.tensor([-1, -1], dtype=torch.float)
input_upper = torch.tensor([1, 3], dtype=torch.float)
output_lower, output_upper = net(input_lower, input_upper)
print(f'{output_lower=}')
print(f'{output_upper=}')
# output_lower=tensor([0.1667])
# output_upper=tensor([6.1667])
```

A.2.3 Evaluation

- Make sure you use the same set of DNNs for the evaluation of both abstraction methods and the symbolic execution for fair comparison.
- Same requirement as PA1: use at least 25 DNNs of varying sizes (number of layers, number of neurons per layer, etc.) in your evaluation.
- A good proxy of precision is the width of the output bounds (i.e., upper - lower), but you can be creative with other metrics as well.
- All text in your plots must be legible.
- If the DNNs you use are too small or too few to show the trend, you will receive a low score.
- Answer the questions in the README template.

A.2.4 What to Turn In

You must submit a **zip file** containing the following files. Use the following names:

1. The zip file must be named `pa2-[yourname/groupname].zip` (1 submission per group)
2. One single main source file `pa2.py`. Indicate in the `README` file on how I can run it.

- Your code should use the provided class/function signatures as indicated above. E.g., `LinearTransformer`, `ReluTransformer`, `IntervalAbstraction`, `ZonotopeAbstraction`, `zonotope`, `concrete`.

3. If you imported anything from your previous PAs, please include those files as well (e.g., `p1.py`). We will not run your code if they are missing and you will lose points.

4. A completed `README.md` file. You *must* use the provided template and answer all questions in it.

5. Nothing else, do not include any binary files or other directory like `__pycache__`, `.venv`.

A.2.5 Grading Rubric (out of 30 points)

- 12 points — for complete interval and zonotope abstraction implementation
  - 2 points for `LinearTransformer` for Interval Abstraction
  - 2 points for `ReluTransformer` for Interval Abstraction
  - 2 points for zonotope representation functions (`zonotope` and `concrete`)
  - 3 points for `LinearTransformer` for Zonotope Abstraction
  - 3 points for `ReluTransformer` for Zonotope Abstraction
- 16 points — for completed README with evaluation and analysis
- 2 points — for code quality and submission format.

A.3 PA3: Branch and Bound with Abstract Interpretation

In this PA you will implement the branch and bound (BaB) algorithm as described in §6.2 with abstract interpretation from PA2 §A.2 (using either interval §5.2.1 or zonotope §5.2.2).

While symbolic execution (PA1 §A.1) provides exact analysis but does not scale, and abstract interpretation (PA2 §A.2) scales well but may lose precision, Branch and Bound provides a framework that can *refine* abstract domains when needed, achieving a balance between precision and scalability.

You will use Python and PyTorch for this assignment, building upon your implementations from PA2. **Do not** use external verification libraries.

This PA has three main parts:

1. implement the core BaB algorithm with a priority *queue*-based approach.

2. implement utility functions: `deduce`, `decide`, `bound`, and `prune`.

3. evaluate the precision and scalability of BaB compared to pure abstract interpretation and symbolic execution.

A README template is provided. Please answer the questions in the template and include it in your submission. https://github.com/dynaroars/book_nnv/blob/main/pa/templates/README_PA3.md

A.3.1 Part 1: Branch and Bound Algorithm Framework

BaB systematically explores the space by maintaining a queue of subproblems and iteratively refining them until verification is complete or a counterexample is found. You will implement four core utility functions that form the building blocks of the BaB algorithm: `deduce`, `decide`, `bound`, and `prune`.

A.3.1.1 Algorithm Skeleton

```
class BranchAndBound:
    def __init__(self, network, abstraction_type):
        """
        Initialize BaB verifier
        Args:
            network: DNN to verify (same format as PA2)
            abstraction_type: Interval or Zonotope from PA2
        """

        # TODO: Initialize your abstraction from PA2
        self.abstraction = ...
        self.network = network

    def verify(self, input_lower, input_upper, property_fn, timeout=60):
        """
        Main BaB verification algorithm following Algorithm 1 from the book

        Args:
            input_lower: lower bounds of input domain
            input_upper: upper bounds of input domain
            property_fn: function that takes (output_lower, output_upper)
                         and returns VERIFIED, FALSIFIED, or UNKNOWN
            timeout: maximum time in seconds

        Returns:
            result: VERIFIED, FALSIFIED, or TIMEOUT
            counterexample: input that violates property (if FALSIFIED)
        """
        start_time = time.time()

        # Initialize verification problems (Line 5 in Algorithm 1)
        problems = Queue()
        initial_problem = (input_lower, input_upper)
        problems.put(initial_problem)

        # Main loop
        while not problems.empty():
            if time.time() - start_time > timeout:
                return TIMEOUT, None
```

```
            # Select a subproblem
            lb, ub = ...  # Hint: use problems.get()

            # TODO: Use self.deduce() to determine
            # if the problem is feasible

            if self.deduce(lb, ub, property_fn):
                # Check satisfiability using adversarial attack
                # instead of LP
                cex = ...  # Hint: use self.attack()

                if cex is not None: # Found valid counterexample
                    return FALSIFIED, cex

                # Decide: pick a input dimension to split on
                # and create new subproblems using input splitting
                subproblems = ...  # Hint: use self.decide()

                # Add new subproblems to queue
                # TODO: Add each subproblem to the queue
                for sub_lb, sub_ub in subproblems:
                    ...  # Hint: use problems.put()

        # All subproblems are verified
        return VERIFIED, None
```

A.3.1.2 Deduce Function

The `deduce` function checks if the current problem is feasible by combining the abstract interpretation (bound).

```
def deduce(self, input_lower, input_upper, property_fn):
    """
    Deduce: combine bound + prune to check if we can make progress

    This function implements the Deduce step
    It combines abstract interpretation (bound) with pruning logic

    Args:
        input_lower: lower bounds of input domain
        input_upper: upper bounds of input domain
        property_fn: property function to check

    Returns:
        bool: False if verified, True othewise, need to branch
    """
    # TODO:
    # 1. Compute abstract bounds using self.bound() (PA2)
    # 2. Check pruning using self.prune()
    # 3. Return True if prune_result == UNKNOWN (need to branch)
    # 4. Return False if prune_result == VERIFIED (can prune)
    # 5. This function should NOT handle FALSIFIED case directly

    return prune_result == UNKNOWN
```

A.3.1.3 Attack Function

The `attack` function finds a counterexample using an adversarial attack. You can use simpler approch like random sampling, or more advanced approach like PGD, FGSM, etc.

```
def attack(self, input_lower, input_upper, property_fn):
    """
    Find counterexample using adversarial attack

    Uses random sampling or adversarial attacks instead of LP solving

    Args:
        input_lower: lower bounds of input domain
        input_upper: upper bounds of input domain
        property_fn: property function to check

    Returns:
        counterexample: cex that violates property, or None if not found

    Example approaches:
        # Random sampling
        for _ in range(100):
            # Randomly sample points within the input bounds
            random_input = ...
            output = self.network(random_input)
            if violates_property(output):
                return random_input

        # Or use PGD attack, FGSM, etc.
    """
    # TODO: Implement adversarial attack to find counterexample
    # 1. Generate candidate inputs within [input_lower, input_upper]
    # 2. Run network on candidates
    # 3. Check if any output violates the property
    # 4. Return first violating input, or None if none found

    return counterexample
```

A.3.1.4 Branch (Decide) Function

The `decide` function splits a domain into smaller subdomains. For this PA, we use input splitting where we divide one dimension at a time.

```
def decide(self, input_lower, input_upper):
    """
    Split input domain into 2 subproblems by splitting one dimension

    Args:
        input_lower: lower bounds of input domain
        input_upper: upper bounds of input domain

    Returns:
        list of (sub_lower, sub_upper) tuples representing subproblems

    Example:
        # Input domain: [-1, 1] x [-2, 2]
        input_lower = torch.tensor([-1.0, -2.0])
```

```
        input_upper = torch.tensor([1.0, 2.0])

        # Splitting the 1st dimension at 0 should return:
        # Subproblem 1: [-1, 0] x [-2, 2]
        # Subproblem 2:  [0, 1] x [-2, 2]

        subproblems = decide(input_lower, input_upper)
        # subproblems = [
        #     (torch.tensor([-1.0, -2.0]), torch.tensor([0.0, 2.0])),
        #     (torch.tensor([0.0, -2.0]), torch.tensor([1.0, 2.0]))
        # ]
    """
    # TODO:
    # 1. Find desired dimension to split on
    #       (e.g., first dimension, max width, random, etc.)
    # 2. Compute split point for that dimension
    #       (e.g., midpoint, random, etc.)
    # 3. Create two subproblems by splitting at the computed point
    # 4. Return list of subproblems

    return subproblems
```

A.3.1.5 Bound Function

The bound function computes abstract output bounds using your abstraction from PA2 (§A.2).

```
def bound(self, input_lower, input_upper):
    """
    Compute output bounds using abstract interpretation

    Args:
        input_lower: lower bounds of input domain
        input_upper: upper bounds of input domain

    Returns:
        (output_lower, output_upper): abstract bounds on network outputs

    Example:
        # Example with a simple 2-input 1-output DNN
        # Input domain: [-1, 1] x [-2, 2]
        input_lower = torch.tensor([-1.0, -2.0])
        input_upper = torch.tensor([1.0, 2.0])

        # Using abstraction from PA2 (interval or zonotope):
        output_lower, output_upper = bound(input_lower, input_upper)
        # output_lower = tensor([-10.0])
        # output_upper = tensor([10.0])
    """
    # TODO: Use your abstraction from PA2 (interval or zonotope)
    # to compute output bounds

    return output_lower, output_upper
```

A.3.1.6 Prune Function

The `prune` function determines whether a subproblem can be eliminated based on the property being verified.

```
def prune(self, output_lower, output_upper, property_fn):
    """
    Determine if subproblem can be pruned based on abstract bounds

    Args:
        output_lower: lower bounds on network outputs
        output_upper: upper bounds on network outputs
        property_fn: function that checks if bounds satisfy property

    Returns:
        VERIFIED: property holds for all points in this subproblem
        FALSIFIED: property violated for all points in this subproblem
        UNKNOWN: cannot determine, need to branch further

    Example:
        # Property: network output should be < threshold (safety property)
        def safety_property(out_lb, out_ub, threshold):
            if torch.all(out_ub < threshold):
                return VERIFIED    # All outputs < threshold
            elif torch.all(out_lb >= threshold):
                return FALSIFIED   # All outputs >= threshold
            else:
                return UNKNOWN     # Outputs cross threshold

        # Property 1: output should be < 5
        prop_1 = lambda out_lb, out_ub: safety_property(out_lb, out_ub, 5)


        # Example 1: Can be pruned as VERIFIED
        output_lower = torch.tensor([-10.0])
        output_upper = torch.tensor([3.0])  # All outputs < 5
        result = prune(output_lower, output_upper, prop_1)
        # result = VERIFIED -> prune this subproblem

        # Example 2: Cannot be pruned
        output_lower = torch.tensor([-2.0])
        output_upper = torch.tensor([8.0])  # Outputs span threshold
        result = prune(output_lower, output_upper, prop_1)
        # result = UNKNOWN -> need to branch further
    """
    # TODO:
    # 1. Check if property_fn gives definitive answer
    # 2. Return appropriate result based on property satisfaction

    return result
```

A.3.2 Part 2: Evaluation

- Make sure you use the same set of DNNs and properties for the evaluation of both abstraction methods and the symbolic execution for fair comparison.
- Use at least 30 diverse DNN and property pairs in your evaluation.

- All text in your plots must be legible.
- If the DNNs you use are too small or too few to show the trend, you will receive a low score.
- Answer the questions in the README template.

A.3.2.1 Test Your Implementation

This function is used to test whether your BaB implementation is correct. Make sure there is *at least one perturbed dimension*, e.g., input_lower should not be equal to input_upper, or otherwise, the input space is just a single point and no need to use abstraction.

```
def test_bab():
    """Test BaB on a DNN"""
    dnn = ...

    # Test with interval abstraction
    bab = BranchAndBound(dnn, 'interval')

    # Input domain
    input_lower = ...
    input_upper = ...

    # Property:
    property_fn = ...


    result, counterexample, stats = bab.verify(
        input_lower, input_upper, property_fn, timeout=30
    )

    print(f"Result: {result}")
```

A.3.2.2 Analysis

Create two additional test functions:

1. `bab_vs_z3`: Compare BaB with Z3 on a local robustness property
2. `bab_vs_abstraction`: Compare BaB with pure abstraction from PA2

Note that, property_fn can be an arbitrary function, make up your own property function (e.g., $y_0 > 0$ or $y_0 > y_1$, etc.,), just make sure to use *same property function* for different methods so that the comparison is fair.

For each testcase, you should:

1. Create networks of varying sizes
2. Use BaB with different configurations (e.g., abstraction types, branching strategies, etc.)

3. For each network, measure:
    - Symbolic execution time/result (PA1 §A.1)
    - Pure abstraction time/result (PA2 §A.2)
    - BaB verification time/result
    - Use appropriate timeouts (e.g., 30s)
4. Plot results showing precision vs scalability trade-offs
5. Analyze when BaB is most beneficial

A.3.3 What to Turn In

You must submit a **zip file** containing the following files. Use the following names:

1. The zip file must be named `pa3-[yourname/groupname].zip` (1 submission per group)
2. One single main source file `pa3.py`. Indicate in the `README` file on how I can run it.
    - Your code should use the provided class/function signatures as indicated above. E.g., `BranchAndBound`, `deduce`, `decide`, `bound`, `prune`...
3. If you imported anything from your previous PAs, please include those files as well (e.g., `p1.py`). We will not run your code if they are missing and you will lose points.
4. A completed `README.md` file. You *must* use the provided template and answer all questions in it.
5. Nothing else, do not include any binary files or other directory like `__pycache__`, `.venv`.

A.3.4 Grading Rubric (out of 30 points)

- 12 points — Core BaB implementation
    - 3 points for correct `verify` algorithm implementation
    - 3 points for `branch` function with proper input splitting
    - 3 points for `bound` function integration with PA2 abstractions
    - 3 points for `prune` function with correct pruning logic
- 16 points — for completed README with evaluation and analysis
- 2 points — for code quality and submission format.

A.4 PA4: Verifying ACAS XU benchmarks

In this PA, you will use your existing implementations you have done in previous PAs to verify properties of a real benchmark using the standard format (ONNX for networks, and VNNLIB for properties).

A.4.1 Part 1: Handling ONNX Networks

In order to handle ONNX networks, the simplest way is to use the ONNX library to load the network and convert to Pytorch model. You can install the conversion library "onnx2pytorch" using pip:

```
pip install onnx2pytorch
```

Then you can use the library to load the network and convert to Pytorch model as follows:

```
import onnx
import onnx2pytorch

# load the ONNX network
model = onnx.load("path/to/your/network.onnx")

# convert to Pytorch model
# experimental=True is optional to use batch processing
model = onnx2pytorch.convert(model, experimental=True)

# since ACAS XU benchmarks contain simple feed-forward networks,
# we can simply enumerate each layer of a network
for layer_name, layer in enumerate(layers):
    if isinstance(layer, nn.Linear):
        # TODO handle linear layer
        pass
    elif isinstance(layer, nn.ReLU):
        # TODO handle ReLU layer
        pass
    elif isinstance(layer, sub):
        # handle subtraction layer
        continue # this layer is y = x - 0, which is identity, so skip it
    else:
        # TODO handle other layers
        pass
```

A.4.2 Part 2: Handling VNNLIB Properties

A.4.2.1 Parsing VNNLIB Properties

VNNLIB properties are designed to represent properties of neural networks in a formal language. It specifies the precondition and postcondition of the network. To extract the preconditions and postconditions from a VNNLIB property, you can follow the provided code https://github.com/dynaroars/book_nnv/blob/main/code/pa4/test.py:

```
# input shape is (1, 5) for ACAS XU benchmarks
input_shape = (1, 5)

# parse the VNNLIB property
properties = parse_vnnlib("path/to/your/property.vnnlib", input_shape)
```

A.4.2.2 VNNLIB Disjunctive Properties

Note that, one VNNLIB file can contain disjunctive properties. Let's take a look at an example VNNLIB property 7 of ACAS XU benchmarks:

```
; ACAS Xu property 7

(declare-const X_0 Real)
(declare-const X_1 Real)
(declare-const X_2 Real)
(declare-const X_3 Real)
(declare-const X_4 Real)

(declare-const Y_0 Real)
(declare-const Y_1 Real)
(declare-const Y_2 Real)
(declare-const Y_3 Real)
(declare-const Y_4 Real)

; Unscaled Input 0: (0, 60760)
(assert (<= X_0 0.679857769))
(assert (>= X_0 -0.328422877))

; Unscaled Input 1: (-3.141592, 3.141592)
(assert (<= X_1 0.499999896))
(assert (>= X_1 -0.499999896))

; Unscaled Input 2: (-3.141592, 3.141592)
(assert (<= X_2 0.499999896))
(assert (>= X_2 -0.499999896))

; Unscaled Input 3: (100, 1200)
(assert (<= X_3 0.5))
(assert (>= X_3 -0.5))

; Unscaled Input 4: (0, 1200)
(assert (<= X_4 0.5))
(assert (>= X_4 -0.5))

; unsafe if strong left is minimial or strong right is minimal
(assert (or
    (and (<= Y_3 Y_0) (<= Y_3 Y_1) (<= Y_3 Y_2))
    (and (<= Y_4 Y_0) (<= Y_4 Y_1) (<= Y_4 Y_2))
))
```

This file specifies the properties in form of $X \wedge (P_0 \vee P_1)$, where X is the precondition on input variables (X_0, X_1, X_2, X_3, X_4) and P_0 and P_1 are the postconditions in *negation form* (do not negate them!) on output variables (Y_0, Y_1, Y_2, Y_3, Y_4).

Specifically,

$$\begin{aligned} X \equiv & (-0.328422877 \leq X_0 \leq 0.679857769) \\ & \wedge (-0.499999896 \leq X_1 \leq 0.499999896) \\ & \wedge (-0.499999896 \leq X_2 \leq 0.499999896) \\ & \wedge (-0.5 \leq X_3 \leq 0.5) \\ & \wedge (-0.5 \leq X_4 \leq 0.5) \end{aligned}$$

$$\begin{aligned} P_0 &\equiv (Y_3 \leq Y_0) \wedge (Y_3 \leq Y_1) \wedge (Y_3 \leq Y_2) \\ P_1 &\equiv (Y_4 \leq Y_0) \wedge (Y_4 \leq Y_1) \wedge (Y_4 \leq Y_2) \end{aligned}$$

To deal with disjuntive postconditions ($P_0 \vee P_1$), the code above from `NeuralSAT` simply converts $X \wedge (P_0 \vee P_1)$ to a list of conjunctive postconditions:

$$\underbrace{(X \wedge P_0)}_{\text{property 1}} \vee \underbrace{(X \wedge P_1)}_{\text{property 2}}$$

Now each sub-property (e.g., property 1 and property 2) has the same form as PA3, to verify a sub-property, you can use the code from PA3.

A.4.2.3 Output Constraints in VNNLIB

VNNLIB always represents the output constraints in the form of:

$$cs \cdot Y \leq rhs$$

where Y is the output vector.

For example, $Y_3 \leq Y_0$ is equivalent to $-Y_0 + Y_3 \leq 0$, which is represented as:

$$\underbrace{\begin{bmatrix} -1 & 0 & 0 & 1 & 0 \end{bmatrix}}_{cs} \underbrace{\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix}}_{Y} \leq \underbrace{\begin{bmatrix} 0 \end{bmatrix}}_{rhs}$$

If you are using VNNLIB extraction code from `NeuralSAT`, each sub-property is an object with following attributes:

- `lower_bounds`: input lower bounds (e.g., lower bounds of all input variables)
- `upper_bounds`: input upper bounds (e.g., upper bounds of all input variables)

- `cs`: coefficient matrix for output constraints
- `rhs`: constant vector for output constraints

```
# number in pop(), e.g., pop(1), means numbers of sub-properties
# we want to verify simultaneously, not the index of the sub-property
sub_property = properties.pop(1) # get one sub-property

print(sub_property.lower_bounds)
# tensor([[-0.3284, -0.5000, -0.5000, -0.5000, -0.5000]])

print(sub_property.upper_bounds)
# tensor([[0.6799, 0.5000, 0.5000, 0.5000, 0.5000]])

print(sub_property.cs)
# tensor([[[-1.,  0.,  0.,  1.,  0.],   # Y_3 <= Y_0
#          [ 0., -1.,  0.,  1.,  0.],   # Y_3 <= Y_1
#          [ 0.,  0., -1.,  1.,  0.]]])  # Y_3 <= Y_2

print(sub_property.rhs)
# tensor([[0., 0., 0.]])
```

A.4.2.4 Verifying VNNLIB Properties

Remember that we have a list of disjunctive sub-properties (e.g., property 1 and property 2 for property 7). To verify the original property is UNSAT, we need to verify all sub-properties are UNSAT. But to show the original property is SAT, disproving one sub-property is enough. The high-level idea should look like this:

```
while len(properties): # make sure to verify all sub-properties
    sub_property = properties.pop(1) # get one sub-property
    # disprove one sub-property, return counterexample immediately
    if verify(network, sub_property) == SAT:
        return SAT, cex

    # run out of time
    if verify(network, sub_property) == TIMEOUT:
        return TIMEOUT

    # prove one sub-property
    if verify(network, sub_property) == UNSAT:
        continue # move to next sub-property

# all sub-properties are verified, the original property is UNSAT
return UNSAT
```

A.4.3 Part 3: Putting Everything Together

Now you have all the components to buid the verifier. You need to follow the following naming conventions for your implementation:

1. A single python file `verify.py` that accepts three command line arguments, and run the verification. This provides a unified interface for us to test your implementation.

```
python verify.py [path/to/network.onnx] [path/to/property.vnnlib] [timeout_in_seconds]

E.g.
python verify.py acasxu/Network1_1.onnx acasxu/Property7.vnnlib 116
Would run the verifier on the given network and property with a timeout of 116 seconds
```

Your output should be either `UNSAT`, `SAT`, or `TIMEOUT`. If the output is `SAT`, you should also print the counterexample found.

2. You actual implementation should be in `p4.py`.

A.4.4 Part 4: Verifying ACAS XU benchmarks

Download the ACAS XU benchmarks from https://github.com/dynaroars/neuralbench/tree/main/instances/acasxu. In this folder, you will find the following files/folders:

- Folder `onnx`: contains the ONNX networks
- Folder `vnnlib`: contains the VNNLIB properties
- File `instances.csv`: contains list of instances of the ACAS XU benchmarks, where each line is a tuple of (network path, property path, timeout in seconds).

You can use `acasxu_instances_w_ground_truth.csv` instead of the `instances.csv` in the repo. It contains the same instances as `instances.csv`, but also includes the ground truth results for each instance. Its avaliable from https://github.com/dynaroars/book_nnv/blob/main/code/pa4/acasxu_instances_w_ground_truth.csv

Your task is to:

1. enumerate first 90 instances in `instances.csv` and verify the properties, the rest are optional.
2. report the result for each instance, either, UNSAT, SAT, or TIMEOUT.
3. some timeouts are expected, we do not expect your verifier to verify all instances.
4. check the results with the ground truth at https://github.com/dynaroars/book_nnv/blob/main/code/pa4/acasxu_instances_w_ground_truth.csv.

You should debug your implementation using a few instances first before running all 90 instances, as the complete run may take a long time.

A.4.5 What to Turn In

You must submit a **zip file** containing the following files. Use the following names:

1. The zip file must be named `pa4-[yourname/groupname].zip` (1 submission per group)

2. If you imported anything from your previous PAs, please include those files as well (e.g., `p1.py`). We will not run your code if they are missing and you will lose points.

3. One `verify.py` file as described above.

4. A completed `README.md` file. You *must* use the provided template and answer all questions in it.

5. It is recommended include any logs or results your verifier generated in a seperate folder. You can name the folder as you like, e.g., `results/`.

    - This is not mandatory, but it will show that you have done the experiments and help us to grade your PA.

6. If you used any additional scripts/files as needed to help your with Part 4, include those as well. Also include instructions on how to use those scripts/files in the `README.md` file.

7. Nothing else, do not include any binary files or other directory like `__pycache__`, `.venv`.

A.4.6 Grading Rubric (out of 30 points)

- 20 points — Verifier implementation
    - 4 points — for parsing ONNX networks correctly
    - 4 points — for parsing VNNLIB properties correctly
    - 6 points — for implementing the verifier (`verify.py`)
    - 6 points — for running your verifier on ACAS XU benchmarks (first 90 instances)
- 10 points — for completed README with analysis

Assignments B

Schedule

- Week 1 (Aug 27):
  - Introduction / Syllabus Overview
  - Formal Methods Overview (§A)
- Week 2 (Sept 3):
  - HW due: Post a short introduction about yourself on Canvas
  - Topics:
    - Short Videos
      - Formal Verification: A Quick Primer [13]
      - But what is a neural network? [1]
      - Large Language Models [2]
    - Background on Logic §B and Linear Programming §D
  - Quiz: no quiz this week

Bibliography

[1] 3Blue1Brown. But what is a neural network? | deep learning chapter 1. https://www.youtube.com/watch?v=aircAruvnKk, 2017.

[2] 3Blue1Brown. Large language models explained briefly. https://www.youtube.com/watch?v=LPZh9BOjkQs, 2024.

[3] S. Bak. nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer, 2021.

[4] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[6] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[7] C. W. Barrett. Decision Procedures: An Algorithmic Point of View. *J. Autom. Reason.*, 51(4):453–456, 2013.

[8] R. J. Bayardo Jr and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Aaai/iaai*, pages 203–208. Providence, RI, 1997.

[9] C. Brix, S. Bak, T. T. Johnson, and H. Wu. The Fifth International Verification of Neural Networks Competition (VNN-COMP 2024): Summary and Results, 2024.

[10] C. Brix, S. Bak, C. Liu, and T. T. Johnson. The Fourth International Verification of Neural Networks Competition (VNN-COMP 2023): Summary and Results, 2023.

[11] R. Bunel, P. Mudigonda, I. Turkaslan, P. Torr, J. Lu, and P. Kohli. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research*, 21(2020), 2020.

[12] R. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. K. Mudigonda. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[13] A. F. V. Channel. Formal verification: A quick primer. https://www.youtube.com/watch?v=J4hwfTbfhVU, 2020.

[14] S. A. Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023.

[15] M. Das, R. Ray, S. K. Mohalik, and A. Banerjee. Fast falsification of neural networks using property directed testing. *arXiv preprint arXiv:2104.12418*, 2021.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[17] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] A. De Palma, R. Bunel, A. Desmaison, K. Dvijotham, P. Kohli, P. H. Torr, and M. P. Kumar. Improved branch and bound for neural network verification via lagrangian decomposition. *arXiv preprint arXiv:2104.06718*, 2021.

[19] S. Demarchi, D. Guidotti, L. Pulina, A. Tacchella, N. Narodytska, G. Amir, G. Katz, and O. Isac. Supporting standardization of neural networks verification with vnnlib and coconet. In *FoMLAS@ CAV*, pages 47–58, 2023.

[20] H. Duong, T. Nguyen, and M. Dwyer. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266*, 2024.

[21] H. Duong, T. Nguyen, and M. Dwyer. Generating and checking dnn verification proofs. page to appear, 2025.

[22] H. Duong, T. Nguyen, and M. B. Dwyer. Neuralsat: A high-performance verification tool for deep neural networks. In *International Conference on Computer Aided Verification*, page to appear, 2025.

[23] H. Duong, D. Xu, T. Nguyen, and M. B. Dwyer. Harnessing neuron stability to improve dnn verification. *Proc. ACM Softw. Eng.*, 1(FSE), jul 2024.

[24] C. Ferguson and R. E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI*, volume 88, pages 128–132, 1988.

[25] C. Ferrari, M. N. Mueller, N. Jovanović, and M. Vechev. Complete Verification via Multi-Neuron Relaxation Guided Branch-and-Bound. In *International Conference on Learning Representations*, 2022.

[26] C. P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org, last accessed January 28, 2026.

[28] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[29] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International conference on computer aided verification*, pages 3–29. Springer, 2017.

[30] R. M. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2009.

[31] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[32] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[33] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] D. Kroening and O. Strichman. *Decision procedures*. Springer, 2008.

[35] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I 25*, pages 135–151. Springer, 2019.

[36] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[37] J. Marques Silva and K. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[38] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[39] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[40] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo. Consolidating a process for the design, representation, and evaluation of reference architectures. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 143–152. IEEE, 2014.

[41] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[42] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[43] ONNX Community. ONNX: Open neural network exchange. https://onnx.ai/, 2017. Accessed: 2025-04-01.

[44] OVAL-group. OVAL - Branch-and-Bound-based Neural Network Verification, 2023. https://github.com/oval-group/oval-bab.

[45] M. Sälzer and M. Lange. Reachability in simple neural networks. *Fundamenta Informaticae*, 189, 2023.

[46] S. A. Seshia, A. Desai, T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue. Formal specification for deep neural networks. In *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16*, pages 20–34. Springer, 2018.

[47] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 31, 2018.

[48] G. Singh, T. Gehr, M. Püschel, and M. Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[49] A. Tacchella, L. Pulina, D. Guidotti, and S. Demarchi. The international benchmarks standard for the Verification of Neural Networks, 2023.

[50] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[51] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1599–1614, 2018.

[52] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and J. Z. Kolter. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Robustness Verification. *Advances in Neural Information Processing Systems*, 34:29909–29921, 2021.

[53] K. Xu, Z. Shi, H. Zhang, Y. Wang, K.-W. Chang, M. Huang, B. Kailkhura, X. Lin, and C.-J. Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. *Advances in Neural Information Processing Systems*, 33:1129–1141, 2020.

[54] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh. Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. *arXiv preprint arXiv:2011.13824*, 2020.