Engineering A Verifier for Deep Neural Networks

ThanhVu Nguyen

Hai Duong

August 29, 2025 (latest version available on Github)

Preface

Contents

Part I

Basics of Neural Networks and Verification

Chapter 1

Neural Networks

1.1 Basics of Neural Networks

A *neural network* (**NN**) [23] 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 (derived by training the network with data). 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. Figure 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:

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

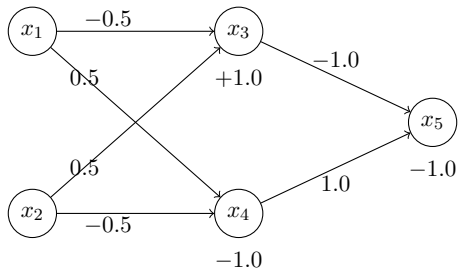


Fig. 1.1: A simple DNN with two inputs x_1, x_2, two hidden neurons x_3, x_4, and one output neuron x_5.

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

- 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:

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

1.2 Affine Transformation

The affine transformation (AF) of a neuron consists of a *linear combination*—i.e., a weighted sum—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[1] 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] Non-linear means that the output of the function is not a linear combination of its inputs.

Tab. 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 NNs. 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.

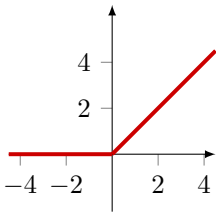


Fig. 1.2: ReLU (Rectified Linear Unit) function.

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

However, 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, verification NNS with ReLU is NP-complete as shown in §3.4. We will use ReLU throughout this book as the default activation function for hidden neurons in an NN.

1.3.2 Sigmoid

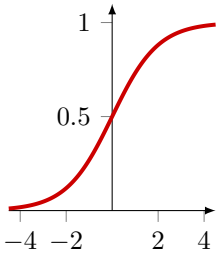


Fig. 1.3: Sigmoid function.

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 1.3.2. $\text{sigmoid}\,(-1.2) \approx 0.23$, $\text{sigmoid}\,(0) = 0.5$, 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.

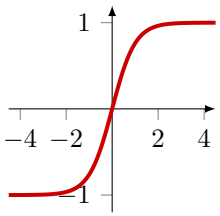


Fig. 1.4: tanh (hyperbolic tangent) activation function.

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

Example 1.3.3. $\tanh(-1.2) \approx -0.83$, $\tanh(0) = 0$, 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

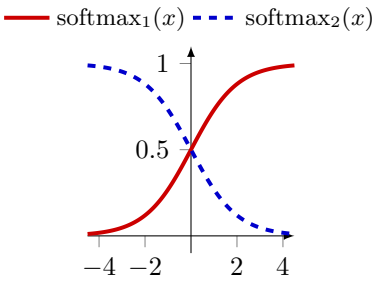


Fig. 1.5: Softmax function for 2 classes.

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 1.3.4. 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.

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.

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.

Tab. 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

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 In fully connected NNs, 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 showed a fully connected network with two inputs and one hidden layer with two neurons, and one output neuron. Fig. 1.6 below shows a fully connected network with four inputs, two hidden layers with five neurons each, and three output neurons (weights and biases not shown for simplicity).

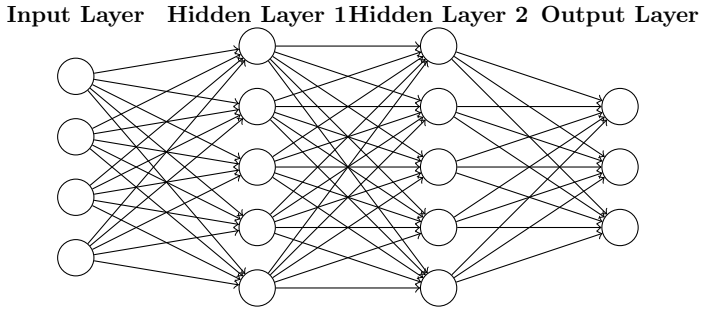


Fig. 1.6: A fully connected NN with two hidden layers.

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.

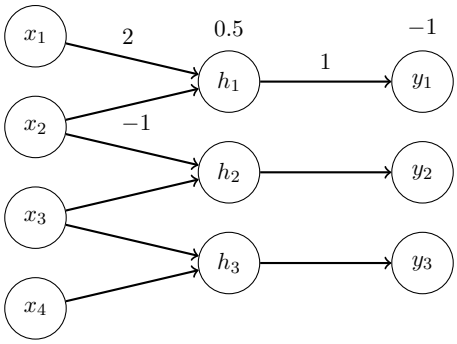


Fig. 1.7: 1-dimensional CNN

Example 1.4.2. 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.

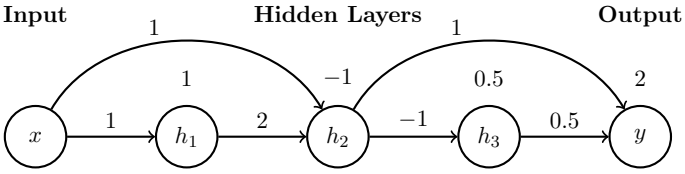


Fig. 1.8: ResNet block with weights on all connections and biases above nodes. Each node is labeled and typical ReLU is applied after each hidden sum.

Example 1.4.3. 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 data structures.

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.

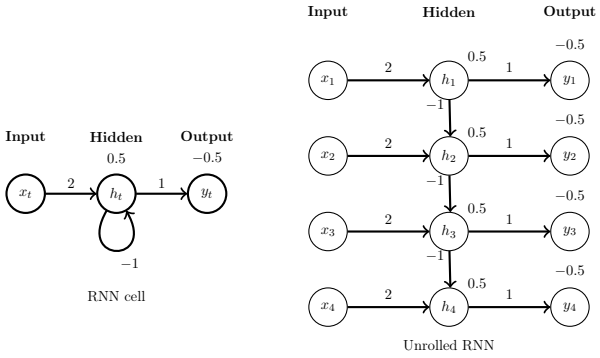


Fig. 1.9: RNN cell (left) and unrolled RNN sequence structure (right). Weights on connections; biases above hidden and output nodes.

Example 1.4.4. 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 $y_1 = h_1 - 0.5$. For the second time step, the hidden state is $h_2 = \text{ReLU}\,(2x_2 - h_1 + 0.5)$, 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) [38] 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:

[TVN]: check for accuracy, I make it very small just to illustrate what ONNX looks like

```
ir_version: 9
opset_import { version: 13 }
graph tiny_net {
  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 software programs, 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 [41].

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.

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.

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

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:

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

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.

Example 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:

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

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.

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 gender 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, while maintaining logical ordering based on academic merit.

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):

$$\text{if } f(x) \neq f(x') \text{ and } \|x - x'\| \leq \epsilon, \text{ then } x' \text{ is a counterexample.}$$

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.

2.5 The VNN-LIB Specification Language

The VNN-LIB standard [16, 44] 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. The international VNN-COMP competition [8] also uses VNN-LIB to evaluate the performance of 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 [38].
- **Property specification** in SMT-LIB format [4].

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. The VNN-LIB code for the property given in **??** 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))
```

Chapter 3

Verification of Neural Networks

3.1 The Neural Network Verification (NNV) Problem

Definition 3.1.1 (NNV)**.** Given a DNN N and a property ϕ, the *DNN verification problem* asks if ϕ is a valid property of N. Typically, ϕ is a formula of the form $\phi_{in} \Rightarrow \phi_{out}$, where ϕ_{in} is a property over the inputs of N and ϕ_{out} is a property 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 [25].

Example 3.1.1. A *valid* property for the DNN in Fig. 1.1 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 DNN evaluates to $x_5 = -3.5$.

3.2 Satisfiability Formulation

As with traditional software verification, DNN verification is often represented as a satisfiability problem, and check if a given property is satisfied by the DNN.

To do this, we first need to 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:

$$\alpha = \bigwedge_{\substack{i\in[1,L]\\ j\in[1,N]}} v_{i,j} \equiv ReLU\Big(\sum_{k\in[1,N]} (w_{i-1,j,k}\cdot v_{i-1,j}) + b_{i,j}\Big)$$
$$= \bigwedge_{\substack{i\in[1,L]\\ j\in[1,N]}} v_{i,j} \equiv \max\Big(\sum_{k\in[1,N]} (w_{i-1,j,k}\cdot v_{i-1,j}) + b_{i,j}, 0\Big)$$

With this definition the NNV problem can be formulated as checking the validity of the following formula:

$$\alpha \implies (\phi_{in} \implies \phi_{out}). \tag{3.2.1}$$

This formula checks if the network N satisfies (implies) the property ϕ, where ϕ_{in} is a precondition on the inputs and ϕ_{out} is a postcondition on the outputs. This validity checking can be reduced to checking the satisfiability of the formula:

$$\alpha \wedge \phi_{in} \wedge \neg\phi_{out} \tag{3.2.2}$$

If Eq. 3.2.2 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. 3.2.2.

Example 3.2.1. We represent the DNN in Fig. 1.1 as a formula α as follows:

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

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

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

We then check the satisfiability of $\alpha \wedge \phi_{in} \wedge \neg\phi_{out}$ using a constraint solver (e.g., an SMT solver). In this case, the formula is satisfiable and the solver returns `sat`, and we query the solver for a satisfying assignment to the input variables, e.g., $x_1 = -1$ and $x_2 = 2$. This assignment is a counterexample (§2.4) to the property, as it satisfies the input constraint ϕ_{in} but violates the output constraint ϕ_{out}, i.e., $x_5 = -3.5$, which is < 0.

3.3 Activation Pattern Search

For ReLU-based DNNs, NNV becomes a search for *activation patterns*, i.e., boolean assignments representing activation status of neurons, that lead to satisfaction the formula in Eq. 3.2.2.

Modern DNN verification techniques [1,10,17,19,21,28,39,48] all adopt this idea and search for satisfying activation patterns.

[TVN]: example of activation pattern search? using the above DNN example [HD]: I will pull the example from our paper, but I think we also should create a tree to demonstrate how it proceeds – I will work on this

[TVN]: Partial Activation Pattern

3.4 Complexity

ReLU-based NNV is NP-complete as shown in [26,40]. 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.

3.5 Challenges

3.5.1 Scalability

Neural networks can be very large, with millions of parameters. This makes it difficult to verify their correctness, as the number of possible inputs grows exponentially with the size of the network.

For DNN verification, AF is straightforward to reason about because it is a linear function. However, AFs are often followed by non-linear activation functions, described next in §1.3, which make the verification problem more challenging.

For DNN verification, these non-linear activation functions make verification difficult because it introduces multiple possible outcomes for any input, making it hard to reason about the output of the network. For example, ReLU has two possible outputs for any input: 0 if the input is less than zero, and the input itself otherwise, and Sigmoid has a smooth curve with infinite possible outputs for any input.

3.5.2 Formalization

3.5.3 Expressiveness

3.6 Challenges in Specifying Properties

[TVN]: TODO

Specifying desirable properties for neural networks is not straightforward. Some challenges include:

- **Incomplete Specifications:** We may not be able to specify all desirable properties.
- **Expressiveness** For tasks such as sentiment analysis or image recognition, functional correctness is hard to define precisely.

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 represented as a satisfiability problem. Specifically, we encode the network N as a logical formula α, and use a constraint solver to check that α satisfies the property ϕ of interest.

A straightforward way to do this encoding is using *symbolic execution* (SE) [2,29], 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.

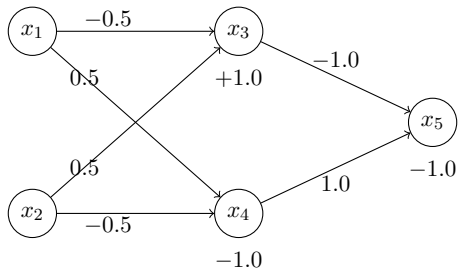


Fig. 4.1: A simple DNN (similar to Fig. 1.1).

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

4.1.2 SMT Solving

After obtaining the symbolic representation of the DNN, we can use an SMT solver [4] 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 $x_1 \in [-1, 1], x_2 \in [-2, 2]$ (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. This leads to the notorious *path explosion* problem and becoming intractable for large DNNs.
- **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 to demonstrate the correctness of small DNNs, e.g., in a classroom setting. Modern-based DNN verification tools do not use SMT solving, and instead, rely on more efficient techniques including abstraction and MILP solving (§4.2).

[TVN]: Hai, include some references or results, e.g., from Nguyen's, showing that SMT solvers are not scalable for DNN verification.[HD]: where can I find the results?[TVN]: Not sure, I thought Nguyen record the results/graphs in some TeX document. Could you ask him?

4.2 MILP

Due to scalability issues with SMT solvers (§4.1.3), modern DNN verification techniques [19, 45, 46] often rely on Mixed Integer Linear Programming (MILP) solving, which is specialized for linear constraints and more efficient for DNN reasoning.

The fundamental distinction between SMT and MILP solving lies in their problem formulation and solution approach. SMT solvers address satisfiability problems: determining whether there exist variable assignments that make a given logical formula true. SMT solvers can handle diverse mathematical theories including non-linear arithmetic. MILP solvers address optimization problems: finding the best solution (according to some objective function) that satisfy a given set of *linear* constraints.

MILP solvers cannot directly solve general SMT problems because they are restricted to linear constraints and cannot handle the non-linear theories and complex

logical structures that SMT problems often contain. While SMT solvers could theoretically encode MILP problems as satisfiability queries, they are not designed for optimization and would be highly inefficient. Despite both approaches having exponential worst-case complexity, MILP solvers often demonstrate superior practical performance due to their restriction to linear constraints, which enables several key optimizations such as linear relaxations, preprocessing techniques, and specialized algorithms.

4.2.1 ReLU Encoding

To use MILP solving, we encode non-linear activation functions like ReLU using *binary indicator* variables and linear constraints. For each neuron, we introduce a binary variable that indicates whether the neuron is "active" (output equals input) or "inactive" (output is zero). This transforms the non-linear $\max(x, 0)$ 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 ($z \leq 0$) or inactive ($z < 0$)
- l, u: lower and upper bounds on z ($l \leq z \leq u$) over the input region

The MILP encoding of $\hat{z} = \max(z, 0)$ is then:

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

These constraints enforce $\hat{z} = z$ when $a = 1$ (active, $z \geq 0$) and $\hat{z} = 0$ when $a = 0$ (inactive, $z < 0$), which capture the semantics of ReLU. Notice that these constraints are linear and involve both continuous variable $\hat{z}$ and binary variable a.

Note the importance of the upper and lower bounds u and l on z. These bounds are critical not only for making the MILP encoding tight, but also 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 can never be non-negative, so the active phase ($a = 1$) is infeasible (the constraints are not satisfiable) and should be ruled out by the MILP encoding; similarly, if $l \geq 0$, only the active phase is possible. Being able to eliminate infeasible cases is crucial for the efficiency of the MILP solver, as it reduces the search space. In general, the tightness of the bounds l and u is crucial for the efficiency of the MILP solver and a major focus of DNN reasoning is developing techniques to capture these bounds more precisely (§5).

Computing Bounds $l \leq e \leq u$ To determine the pairs of values that achieve the lower and upper bounds of a linear expression e:

$$e = a_1x_1 + a_2x_2 + \ldots + b,$$

where each variable x_i ranges over $[l_i, u_i]$, we proceed as follow:

- **For the lower bound** (l_3): For each variable x_i, use u_i (its upper bound) if $a_i < 0$, and use l_i (its lower bound) if $a_i \geq 0$.
- **For the upper bound** (u_3): For each variable 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, with the exact pair determined by the sign of each coefficient.

Example 4.2.1. Consider the neuron x_3 in the DNN in Fig. 4.1 from 4.1.2. 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 z_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.

Let $\hat{x}_3$ be the output of neuron x_3 after applying ReLU, and let a_3 be a binary variable indicating whether neuron x_3 is active (1) or inactive (0). We encode the ReLU activation of neuron x_3:

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

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. This is actually a worst-case scenario for the MILP solver, as it has to consider both cases. 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, and the MILP solver would only have to consider the active case.

4.2.2 DNN encoding

More generally, we can encode the DNN as a set of 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. We create the following MILP formulation for the example in 4.1.2. The DNN Fig. 4.1 in the example has two inputs x_1, x_2, 1 hidden layer with 2 neurons x_3, x_4, each with its own ReLU output $\hat{x}_3, \hat{x}_4$, and 1 output neuron x_5. We will use property $x_5 > 0$ for any inputs $x_1 \in [-1, 1], x_2 \in [-2, 2]$:

1. Encoding precondition representing input bounds:

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

2. Encoding hidden layer (pre- and post-activation):

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

3. Encoding output layer (post-activation):

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

4. Negating property

$$x_5 \leq 0$$

5. Computing upper and lower bounds over given input ranges. For example, 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 $l_3 = -0.5, u_3 = 2.5, l_4 = -2.5, u_4 = 0.5$.

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

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

7. **The final MILP encoding:**

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

8. **Solving** The MILP solver will find a satisfying assignment if one exists, such as $x_1 = -1, x_2 = 2$, 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 $x_5 \leq 0$, so the solver stops.

[TVN]: So far we talk about bounds for pre-ReLU values, right? is that the same as post-ReLU? In other words, when we talk about abstraction for ReLU, are we talking about these pre-ReLU bounds, which currently use intervals? or the abstraction for ReLU refer to something else and not these pre-ReLU bounds? [HD]: no, we are referring to post-ReLU bounds. For example, a pre-ReLU interval is $z_3 \in [-0.5, 2.5]$ — indicating an unstable neuron, and we will abstraction to compute bounds for the post-ReLU e.g., $\hat{z}_3$.

4.2.3 Limitations

- **Scalability:** While MILP is more efficient than SMT solving[TVN]: is it? due to what?, it still cannot be applied directly to real-world DNNs due to the exponential growth of the search space. Each ReLU introduces a binary variable, leading to 2^n possible branches, and realistic DNNs can have millions of ReLUs, making the search space intractable.

- **Limited exploitation of network structure and modern hardware** General MILP solvers are designed for arbitrary MILP problems and do not exploit DNN-specific structures such as[TVN]: Hai, like what? mention those that DNN verification tools exploit.

  Moreover, MILP solvers, even industrial-strength ones such as Gurobi [24], are primarily CPU-based and does not leverage the massive parallelism provided by modern GPUs.

- **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 (and in general program analyses) employ more advanced abstract domains such as zonotopes (which capture linear relationships) and polytopes (which represent arbitrary linear constraints) to improve precision.

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.

[TVN]: Hai, above you mention MILP also employs BaB algorithm, could you write a paragraph or so about how MILP solvers use BaB and briefly show how that would apply to example 4.2.2 (just briefly, no need a full step by step demonstration). Is this BaB algorithm used by MILP very similar to the one used by DNN verification tools (minus the DNN specific optimizations and heuristics)? like the BnB algorithm described in Alg. 1

Chapter 5

Abstractions

As mentioned in §4.2, and DNN verification in general, we need to compute the bounds of the neurons. However, computing these bounds precisely is often infeasible due to the complexity of the DNN structure and non-linear activation functions such as ReLU. To address this, modern DNN verifiers use *abstraction* techniques to approximate the set of possible values that a post-ReLU neuron can have.

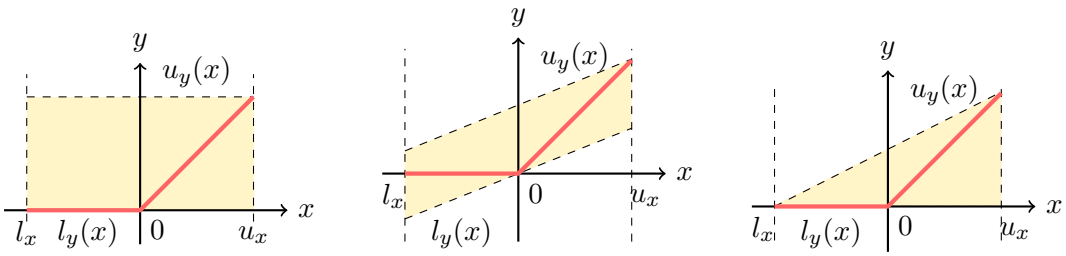
5.1 Overview of Abstractions for ReLU

We will focus on ReLU (§1.3.1) activation functions, which are the most common in DNNs. The goal is to compute the bounds of a post-ReLU neuron $\hat{z}$ given the bounds of its pre-ReLU value z over the input region. 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.

We want to compute the bounds to be as tight as possible, i.e., they should be the largest lower and lowest upper values that $\hat{z}$ can take given the bounds on z. This computation is called *abstraction* and is crucial for DNN verification, as it allows us to reason about the behavior of the network without having to enumerate all possible values of the neurons. There are several abstraction techniques that can be used to compute these bounds, each with its own trade-offs in terms of precision and computational complexity.

5.1.1 Common Abstractions for ReLU

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:



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

Fig. 5.1: Abstractions of $\mathrm{ReLU}(x) = \max(0, x)$ over $x \in [l_x, u_x]$: (a) interval, (b) zonotope, (c–d) polytope abstractions with different lower bounds.

- **Interval Abstraction:** Interval represents the output of the post-ReLU neuron as intervals $[l_y(x), u_y(x)]$ where $l_y(x) = 0$ and $u_y(x) = u_x$. It does not capture the relationship between the input and output of ReLU, and simply assumes that the output can take any value between 0 and the upper bound of the input. Interval is a simple and efficient abstraction, but it can be *too imprecise* for many cases.

  In Fig. 5.1a, the yellow rectangle represents the interval $[0, u_x]$. As can be seen, this interval region 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 can capture linear relationships between variables more effectively. We can represent ReLU as a zonotope that includes the linear segments. The lower bound $l_y(x)$ is $y = \lambda x$ and the upper bound $u_y(x)$ is $y = \lambda x + u_x(1 - \lambda)$ for some slope $\lambda \in [0, \frac{u_x}{u_x - l_x}]$. If we set $\lambda = 0$, the zonotope is a rectangle, which is the same as the interval abstraction. If we set $\lambda = \frac{u_x}{u_x - l_x}$, the zonotope's is upper bound is the same as polytope's upper bound in Fig. 5.1c.

  In Fig. 5.1b, the zonotope, depicted as a a parallelogram, 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 $u_y(x) = u_x$.

5.1.2 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 5.1.1 (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 5.1.2. 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:

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

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.

5.2.1 Interval

Interval is a very simple abstraction for DNN verification. Instead of keeping a single value for each variable, it maintains a lower and upper bound for it. Thus, we can use it to estimate the range of values that a neuron can take using a lower and upper bound $[l, u]$, e.g., an interval over the set of values $\{-2.5, -8.2, -10.7, 2, 4.7, 5.1\}$ can be represented as $[-10.7, 5.1]$.

Interval is very efficient, but treats each variable independently, so it cannot capture correlations between variables such as $x_2 = -x_1$.

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 cuboid 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:

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

Example 5.2.1. Consider the DNN in Fig. 5.2 with inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$. The affine function for the neuron x_3 is given by:

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

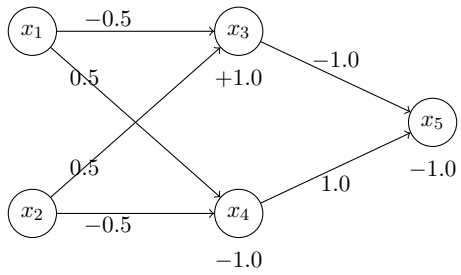


Fig. 5.2: A simple DNN (similar to Fig. 1.1).

Then

$$f^a([1,2],[-1,3]) = \Big[1 + \min(-0.5 \cdot 1, 0.5 \cdot 2) + \min(-0.5 \cdot -1, 0.5 \cdot 3)$$
$$1 + \max(-0.5 \cdot 1, 0.5 \cdot 2) + \max(-0.5 \cdot -1, 0.5 \cdot 3)\Big]$$
$$= [-0.5, 2.0]$$

5.2.1.2 ReLu Transformer

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

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

This is equivalent to three cases:

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

Example 5.2.2. For the example in 5.2.1, applying ReLU to the output bounds of neuron x^3 gives:

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

5.2.1.3 Efficiency and Precision

Intervals is very *efficient* and scales very 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. Intervals approximate each variable independently, which can lead to over-approximation errors when variables are correlated.

Example 5.2.3. Suppose we have $v_1 \in [0, 1]$, $v_2 = -v_1$, and $z = v_1 + v_2$[TVN]: Hai, do we have these kinds of correlation in DNN verification?. The concrete value of z is always 0, but the interval abstraction gives $z \in [-1, 1]$, which is a very loose over-approximation.

Moreover, if we apply ReLU, then the true output is $\text{ReLU}(z) = \text{ReLU}(0) = 0$, but the interval abstraction gives $\text{ReLU}^a([-1, 1]) = [0, 1]$, which is again a loose over-approximation.

This overapproximation error grows quickly as we propagate through many layers of a large network, i.e., it keeps "inflating" the bounds, leading to a very loose approximation of the output and becomes useless for verification tasks. For example, if we want to verify that $z \leq 0$, the interval, which gives $z \in [0, 1]$, cannot provide a tight enough bound to prove this property.

Nonetheless, interval-based methods remain popular due to their simplicity and efficiency. In some cases, despite being too coarse, they can still successfully verify properties of neural networks, e.g., if we want to verify that $z \leq 2$, then the interval result $[-1, 1]$ would suffice.

5.2.2 Zonotope

Zonotopes extend intervals (§5.2.1) by introducing *generators* that represent linear dependencies between variables. This allows zonotopes to capture correlations between variables, leading to more precise approximations compared to intervals.

A **zonotope** $\mathcal{Z}$ in $\mathbb{R}^n$ is formally defined as:

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

where :

- $c \in \mathbb{R}^n$ is the *center* (like the midpoint of intervals),
- $g_i \in \mathbb{R}^n$ are *generator vectors* (directions of variability), and
- ϵ_i are independent coefficients in $[-1, 1]$.

5.2.2.1 Affine Transformer

For the affine function f in §1.2

$$f(x) = Wx + b$$

where W is a weight matrix and b is a bias vector. Given an input zonotope $\mathcal{Z} = (c, G)$ where $G = [g_1, g_2, \ldots, g_m]$ is the generator matrix, the abstract transformer f^a for zonotope is:

$$f^a(\mathcal{Z}) = (Wc + b, WG)$$

The new center becomes $Wc + b$ and the new generator matrix becomes WG. Since affine transformations preserve zonotope structure, this transformation is exact with no over-approximation.

Example 5.2.4. Consider the DNN in Fig. 5.2 with inputs $x_1 \in [1, 2]$ and $x_2 \in [-1, 3]$. The affine function for neuron x_3 is given by:

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

The center c represents the midpoint of each input interval:

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

The generator matrix G captures the half-widths of each interval as diagonal entries:

$$G_{11} = \frac{2-1}{2} = 0.5, \quad G_{22} = \frac{3-(-1)}{2} = 2.0$$

This construction ensures the zonotope exactly represents the input box: $x_1 \in [1.5 - 0.5, 1.5 + 0.5] = [1, 2]$ and $x_2 \in [1.0 - 2.0, 1.0 + 2.0] = [-1, 3]$. Thus, the input box can be encoded as a zonotope with center and generator matrix:

$$c = \begin{bmatrix} 1.5 \\ 1.0 \end{bmatrix}, \quad G = \begin{bmatrix} 0.5 & 0 \\ 0 & 2.0 \end{bmatrix}$$

For the weight $w = [-0.5, 0.5]^\top$ and bias $b = 1.0$, applying the zonotope transformer:

$$\begin{aligned} c' &= w^\top c + b = [-0.5, 0.5] \begin{bmatrix} 1.5 \\ 1.0 \end{bmatrix} + 1.0 = -0.75 + 0.5 + 1.0 = 0.75 \\ g_1' &= w^\top g_1 = [-0.5, 0.5] \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} = -0.25 \\ g_2' &= w^\top g_2 = [-0.5, 0.5] \begin{bmatrix} 0 \\ 2.0 \end{bmatrix} = 1.0 \end{aligned} \tag{5.2.1}$$

The output zonotope for x_3 is $(0.75, [-0.25, 1.0])$, thus the concrete bounds are:

$$\begin{aligned} f_L^a &= c' - |g_1'| - |g_2'| = 0.75 - 0.25 - 1.0 = -0.5 \\ f_U^a &= c' + |g_1'| + |g_2'| = 0.75 + 0.25 + 1.0 = 2.0 \end{aligned} \tag{5.2.2}$$

Note that, for this single affine operation over a box input, both interval and zonotope abstractions yield identical bounds $[-0.5, 2.0]$. The advantage of zonotopes becomes apparent when propagating through multiple layers, where zonotopes preserve linear correlations that intervals lose.

5.2.2.2 ReLU Transformer

Activation functions like ReLU are non-affine and do not preserve zonotope structure. For the ReLU function $\text{ReLU}(x) = \max(0, x)$, we must use conservative approximations. Given an input zonotope, we first compute its interval bounds $[l, u]$, then consider three cases:

- **Active case** ($l \geq 0$): All values are non-negative, so $\text{ReLU}^a(\mathcal{Z}) = \mathcal{Z}$ (identity).
- **Inactive case** ($u \leq 0$): All values are negative, so $\text{ReLU}^a(\mathcal{Z}) = \{0\}$ (zero zonotope).
- **Unstable case** ($l < 0 < u$): The range crosses zero, requiring over-approximation using a parallelogram that bounds the ReLU function over $[l, u]$.

For the unstable case, the zonotope approximation introduces a new generator to capture the ReLU's piecewise-linear behavior, typically using the slope $\lambda = \frac{u}{u-l}$ for the lower bound and maintaining the upper bound $y = u$.

Example 5.2.5 (ReLU on zonotope)**.** Continuing from the previous example, applying ReLU to the zonotope output $(0.75, [-0.25, 1.0])$ with bounds $[-0.5, 2.0]$:

Since $l = -0.5 < 0 < 2.0 = u$, this is an unstable neuron. The ReLU approximation introduces over-approximation, typically resulting in bounds $[0, 2.0]$ with additional generators to model the ReLU constraint.

5.3 Polytope

In the previous section, we have seen the abstract domain of zonotopes, which is more expressive than the interval domain. Specifically, instead of approximating functions using a hyper-rectangle, the zonotope domain allows us to approximate functions using a zonotope, e.g., a parallelogram, capturing relations between different dimensions. In this section, we look at an even more expressive abstract domain, the polyhedron (or polytope) domain.

Unlike the zonotope domain, the polyhedron domain allows us to approximate functions using arbitrary convex polyhedra. A polyhedron in $\mathbb{R}^n$ is a region made of straight (as opposed to curved) faces; a convex shape is one where the line between any two points in the shape is completely contained in the shape. Convex polyhedra can be specified as a set of linear inequalities of the form:

$$Ax \leq b$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ for some m, specifying m half-spaces whose intersection forms the polyhedron.

Using a set of convex polyhedra, we can more precisely approximate activation functions like ReLU. For instance, to approximate ReLU, we can describe the tightest convex over-approximation by the following constraints:

$$x \leq y, \quad 0 \leq y, \quad y \leq \lambda x + \mu$$

where λ and μ are parameters chosen based on the bounds of x.

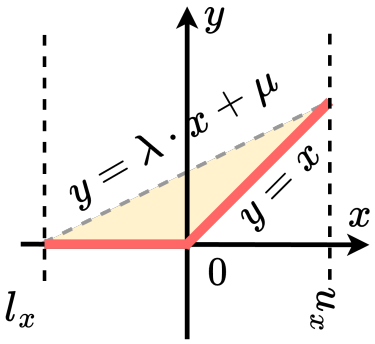


Fig. 5.3: The tightest Convex Polyhedral Approximation of ReLU.

This is the smallest convex polyhedron that soundly approximates ReLU, and is strictly more precise than the interval and zonotope abstractions, as can be seen visually.

5.3.1 Affine Functions

Affine transformations map convex polyhedra to convex polyhedra. Given an affine transformation $f(x) = Wx + b$, where W is a matrix and b is a bias vector, we can transform a polyhedron $\{x \mid Ax \leq b\}$ through f by substitution:

$$\{x \mid A(W^{-1}(x - b)) \leq b\}$$

if W is invertible. More commonly in practice, when applying affine transformations in neural networks (layer-wise), we modify the constraints accordingly by propagating through the weights and biases.

Thus, affine layers can be handled exactly under polyhedral abstraction without any need for relaxation or over-approximation.

5.3.2 Activation Functions

Handling non-linear activation functions, such as ReLU, with polyhedral abstraction is challenging because non-linearities generally map polytopes to non-convex shapes. Therefore, over-approximations are required.

The key idea is to approximate the graph of the non-linear function by a convex polyhedron that covers all possible cases. For ReLU, the previously mentioned three constraints:

$$x \leq y, \quad 0 \leq y, \quad y \leq \lambda x + \mu \tag{5.3.1}$$

construct a tight convex relaxation.

For other activations like sigmoid or tanh, the approximation involves linearizing the curve between the lower and upper bounds of the input variable, but the number of constraints can quickly grow, increasing computational complexity.

DeepPoly. By containing two lower polyhedra constraints for y, the approximation in 5.3.1 inherently suffers from a potential blow-up in number of constraints as the analysis proceeds. Due to scalability issues associated with general polyhedral representations, DeepPoly [43] was proposed as a precise and scalable abstract domain. DeepPoly introduces a specialized form of polyhedral abstraction combined with interval bounds.

More specifically, DeepPoly only allows for one lower bound in 5.3.1, the selection of which lower constraint depends on which constraint provides the tighter approximation.

Returning to our example, the area of the approximation in Fig. 4(b) is given by $0.5 \cdot u \cdot (u - l)$, while the area in Fig. 4(c) is given by $0.5 \cdot (-l_i) \cdot (u_i - l_i)$. To achieve a tighter relaxation, we select the approximation with the smaller area. Specifically, when $u \leq -l$, we apply the constraints and bounds derived from $x \leq y$. In addition, each neuron x_j is bounded above and below by affine expressions of the input neurons:

$$l_j(x) \leq x_j \leq u_j(x)$$

where l_j and u_j are affine functions.

5.3.3 DeepPoly Transformers

The DeepPoly abstract domain defines a scalable and precise framework for verifying deep neural networks by combining interval and polyhedral abstractions. Each neuron is bounded from above and below using affine expressions of the input neurons:

$$l_j(x) \leq x_j \leq u_j(x)$$

where $l_j(x)$ and $u_j(x)$ are affine functions representing lower and upper bounds respectively. DeepPoly then defines **abstract transformers** to propagate these bounds through different types of neural network layers.

5.3.3.1 Affine Layer Transformer

Given an affine transformation:

$$x^{(l+1)} = Wx^{(l)} + b$$

we define the transformation of the bounds by propagating the affine expressions directly:

$$l_j^{(l+1)}(x) = \sum_i w_{ji}^+ \cdot l_i^{(l)}(x) + w_{ji}^- \cdot u_i^{(l)}(x) + b_j$$
$$u_j^{(l+1)}(x) = \sum_i w_{ji}^+ \cdot u_i^{(l)}(x) + w_{ji}^- \cdot l_i^{(l)}(x) + b_j$$

Here, $w_{ji}^+ = \max(w_{ji}, 0)$, and $w_{ji}^- = \min(w_{ji}, 0)$, ensuring correct handling of sign-dependent propagation.

5.3.3.2 ReLU Transformer

The ReLU transformer in DeepPoly uses a convex relaxation that is tighter than previous abstractions by selecting only one lower bound constraint depending on the sign and tightness. Suppose $x_j = \text{ReLU}(x_i)$. Let $[l_i, u_i]$ be the lower and upper bounds for x_i.

- If $l_i \geq 0$: ReLU is linear, so

$$l_j(x) = l_i(x), \quad u_j(x) = u_i(x)$$

- If $u_i \leq 0$: ReLU is constant 0, so

$$l_j(x) = u_j(x) = 0$$

- If $l_i < 0 < u_i$: ReLU is approximated with:

$$u_j(x) = \frac{u_i}{u_i - l_i} \cdot (x_i - l_i)$$

  and only one lower bound is selected:

$$l_j(x) = \begin{cases} 0, & \text{if area under lower 0 line is smaller} \\ x_i, & \text{if area under identity line is smaller} \end{cases}$$

The decision is made based on which convex region has smaller area to achieve a tighter abstraction, following the principle:

$$\text{Choose constraint with smaller area:} \quad 0.5 \cdot u_i \cdot (u_i - l_i) \quad \text{vs} \quad 0.5 \cdot (-l_i) \cdot (u_i - l_i)$$

By maintaining only upper and lower affine bounds, DeepPoly avoids the full complexity of manipulating arbitrary polytopes while retaining significantly more precision than intervals or zonotopes.

...

Thus, DeepPoly achieves a practical balance between precision and scalability for verifying deep neural networks.

5.3.4 Example

Consider again the DNN in Fig. 1.1. Suppose the input set is defined by box constraints $x_1 \in [-1, 1]$ and $x_2 \in [-2, 2]$. These can be represented initially by 4 inequalities.

Applying the affine transformation for the hidden layer, we obtain a new set of inequalities describing the hidden layer nodes. Upon applying ReLU, we would use the convex polyhedral relaxation as depicted in **??**. The output layer similarly results from affine operations on the polyhedra describing the hidden layer.

In a DeepPoly setting, instead of carrying full inequalities, we track only the lower and upper affine bounds per neuron, leading to an efficient verification process.

5.3.5 Comparison to Zonotope Abstraction

While zonotope abstraction captures dependencies between variables better than intervals, it still assumes symmetrical dependencies around a center point and cannot easily model arbitrary convex shapes.

Polyhedral abstraction, on the other hand, allows representing arbitrary convex shapes precisely, enabling much tighter approximations, especially after ReLU activations.

However, traditional polyhedral methods suffer from:

- High computational complexity,
- Rapid growth in the number of constraints,
- Difficulty scaling to large networks.

DeepPoly addresses these challenges by:

- Restricting to simple upper and lower affine bounds,
- Maintaining polynomial scalability,
- Achieving higher precision than zonotopes or intervals,
- Providing efficient transformers for common layers in DNNs.

Therefore, DeepPoly achieves a balance, combining the expressiveness of polyhedral domains with the efficiency required for deep network verification.

5.4 Abstractions for Other Activation Functions

5.5 Exercises

5.5.1 Interval Abstraction

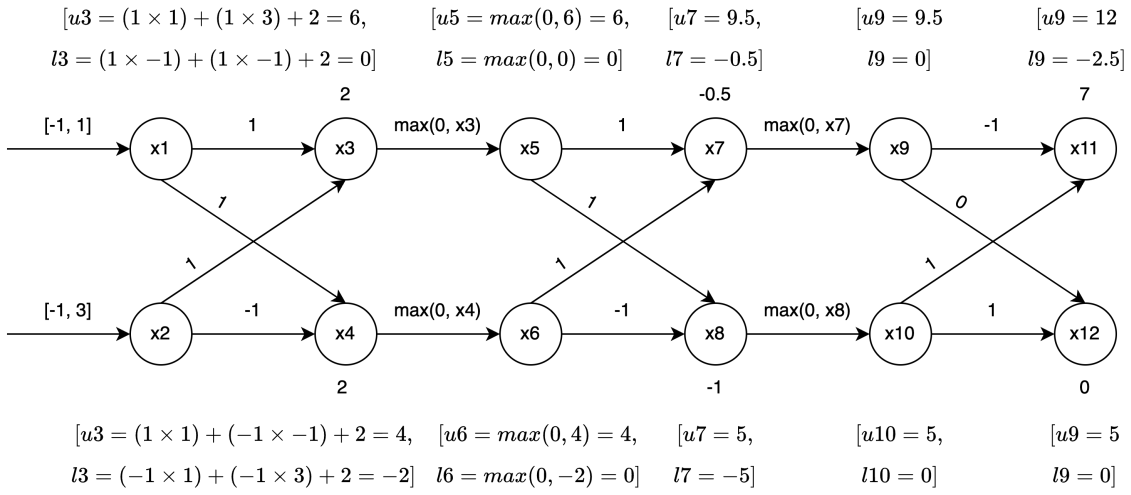


Fig. 5.4: Feed-forward neural network with 6 layers and 2 neurons per layer.

Example 5.5.1 (Full Example of Interval Abstraction)**.** In Fig. 5.4, we aim to verify whether $x_{11} \geq x_{12}$ given a bounded input range on x_1 and x_2. To do this using interval abstraction, we propagate intervals layer by layer.

Suppose the inputs x_1 and x_2 satisfy:

$$x_1 \in [-1, 1], \quad x_2 \in [-1, 2]$$

Let an affine neuron (e.g., x_3) compute:

$$x_3 = 1x_1 + 1x_2 + 2$$

Then we apply the affine abstract transformer:

$$\begin{aligned} x_3 \in [&\min(1 \times -1, 1 \times 1) + \min(1 \times -1, 1 \times 3) + 2, \\ &\max(1 \times -1, 1 \times -1) + \max(-1 \times -1, -1 \times 3) + 2] = \\ &[0, 6] \end{aligned} \tag{5.5.1}$$

So we over-approximate x_3 as:

$$x_3 \in [0, 6]$$

This process is repeated for each layer until we reach the output layer. Here we can see that by using Interval abstraction, the property $x_{11} \geq x_{12}$ or $x_{11} - x_{12} \geq 0$ is *UNSAT*[HD]: seems incorrect due to

$$-2.5 - 0 \leq x_{11} - x_{12} \leq 12 - 5$$

Part III

DNN Verification Algorithms

Chapter 6

The Branch and Bound Search Algorithm

Modern DNN verifiers typically adopt the Branch-and-Bound (BaB) approach to search for activation patterns (§3.3). At its core, a BaB search consists of two main components: (branch) splitting into the problem smaller subproblems by using *neuron splitting*, which decides boolean values representing neuron activation status, and (bound) using abstraction and LP solving to approximate bounds on neuron values to determine the satisfiability of the partial activation pattern (§3.3) formed by the split.

6.1 The Algorithm

Alg. 1. The $\texttt{BaB}_{\texttt{NV}}$ algorithm.

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

```
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, v_i) ← Decide(N, φ_in, φ_out, σ_i)
6         if cex then // found a valid counter-example
7             return (sat, cex)
          // create new activation patterns
8         ActPatterns ← ActPatterns ∪ {σ_i ∧ v_i ; σ_i ∧ ¬v_i}
9 return unsat
```

Alg. 1 shows $\texttt{BaB}_{\texttt{NV}}$, a reference BaB architecture [35] for modern DNN verifiers. $\texttt{BaB}_{\texttt{NV}}$ takes as input a ReLU-based DNN $\mathcal{N}$ and a formulae $\phi_{in} \Rightarrow \phi_{out}$ representing the property of interest. $\texttt{BaB}_{\texttt{NV}}$ maintains a set of activation patterns (`ActPatterns`) that represent the current activation pattern of the DNN. Initially, `ActPatterns`

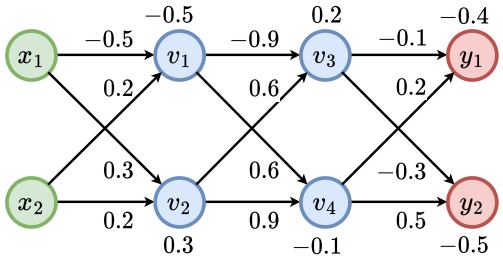


Fig. 6.1: A simple DNN.

is initialized with an empty activation pattern (line 1). In each iteration, $\texttt{BaB}_{\texttt{NV}}$ selects an activation pattern σ_i from `ActPatterns` ([TVN]: also remove σ_i from `ActPatterns`?)[HD]: yes. It then calls `Deduce` to check the feasibility of the problem based on the current activation pattern. (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.)

If `Deduce` determines that the problem is feasible (using the current activation pattern σ_i), it calls `Decide` to select a neuron v_i to split, which essentially means the problem is split into two independent subproblems: one with v_i (active) and the other with $\overline{v_i}$ (inactive). $\texttt{BaB}_{\texttt{NV}}$ then adds the two new activation patterns $\sigma_i \wedge v_i$ and $\sigma_i \wedge \overline{v_i}$ to `ActPatterns`. $\texttt{BaB}_{\texttt{NV}}$ then loops back to line 6 to process the next activation pattern. [TVN]: briefly describe what Decide do? in particular what does it do to determine/create an cex when the activation pattern is partially assigned?

If `Deduce` determines that the problem is infeasible, it loops back to line 6 to process the next activation pattern. $\texttt{BaB}_{\texttt{NV}}$ terminates when there are no more activation patterns to process, at which point it returns `unsat`, indicating that the property is valid.

[TVN]: shows that $\texttt{BaB}_{\texttt{NV}}$ often does not exhaust all activation patterns, but rather prunes the search space by deducing infeasibility of some activation patterns. May be use a tree example to illustrate this?

Example 6.1.1. [TVN]: We should change this example to the 5-neuron DNN we have always been using.Fig. 6.1a illustrates a DNN and how $\texttt{BaB}_{\texttt{NV}}$ verifies that this DNN has the property

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

First, $\texttt{BaB}_{\texttt{NV}}$ initializes the activation pattern set `ActPatterns` with an empty activation pattern $\emptyset$ (line 1). Then $\texttt{BaB}_{\texttt{NV}}$ enters a loop (line 6–line 20) to search for a satisfying assignment. In the first iteration, $\texttt{BaB}_{\texttt{NV}}$ selects the only available activation pattern $\emptyset \in$ `ActPatterns`. It calls `Deduce` to check the feasibility of the problem based on the current activation pattern. `Deduce` uses abstraction to approximate that from the input constraints the output values are feasible for the

given network. Since `Deduce` cannot decide infeasibility, $\texttt{BaB}_{\texttt{NV}}$ randomly selects a neuron to split (`Decide`). Let us assume that it chooses v_4 to split, which essentially means the problem is split into two independent subproblems: one with v_4 active and the other with v_4 inactive. $\texttt{BaB}_{\texttt{NV}}$ then adds $\{v_4\}$ and $\{\overline{v_4}\}$ to `ActPatterns`.

In the second iteration, $\texttt{BaB}_{\texttt{NV}}$ has two subproblems (that can be processed in parallel). For the first subproblem with v_4, `Deduce` cannot decide infeasibility, so it selects v_2 to split. It then conjoins v_4 with v_2 and then with $\overline{v_2}$ and adds both conjuncts to `ActPatterns`. For the second subproblem with $\overline{v_4}$ inactive, `Deduce` determines that the problem is unsatisfiable.

In the third iteration, $\texttt{BaB}_{\texttt{NV}}$ has two subproblems for $v_4 \wedge v_2$ and $v_4 \wedge \overline{v_2}$. For the first subproblem, `Deduce` cannot decide infeasibility, so it selects v_1 to split. It then conjoins v_1 and then $\overline{v_1}$ to the current activation pattern and adds them to `ActPatterns`. For the second subproblem, `Deduce` determines that the problem is unsatisfiable.

In the fourth iteration, $\texttt{BaB}_{\texttt{NV}}$ has two subproblems for $v_4 \wedge v_2 \wedge v_1$ and $v_4 \wedge v_2 \wedge \overline{v_1}$. Both subproblems are determined to be unsatisfiable.

Finally, $\texttt{BaB}_{\texttt{NV}}$ has an empty `ActPatterns`, stops the search, and returns `unsat`—the property is valid.

6.2 Beyond the Basic BaB

What described above is the basic and minimal BaB algorithm. However, 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 $\texttt{BaB}_{\texttt{NV}}$ variant (e.g., the `NeuralSAT` tool described in §8) 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 [18]. In addition, heuristics are also used to select, e.g., which neuron to split next (`Decide`). We explore these optimizations and strategies in Part VI.

Chapter 7

Common Engineerings and Optimizations

A full branch and bound (BaB search (§6) is typically expensive and slow. Thus, DNN verifiers often begin by applying a range of optimizations and engineering techniques that aim to quickly eliminate easy cases, such as when the network has a small number of inputs.

7.1 Input Splitting

Modern DNN verifiers [1, 18, 27, 28] often 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 §13.1.1 where the networks have a small number of inputs. Currently many work set the threshold to 50 inputs, i.e., if the network has more than 50 inputs, it is considered large and input splitting is not applied.

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—also called *neuron splitting* and focuses on individual neurons—and is often used as a fast-path optimization to quickly eliminate easy cases.

Example 7.1.1. Given k available threads, `NeuralSAT` splits the original input region to obtain subproblems as described and and runs DPLL(T) on k subproblems in parallel. `NeuralSAT` returns `unsat` if it verifies all subproblems and `sat` if it found a counterexample in any subproblem. For example, we split the input region $\{x_1 \in [-1, 1], x_2 \in [-2, 2]\}$ 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 $\{x_1 \in [0, 1], x_2 \in [0, 2]\}$. 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.

7.2 Input Bounds Tightening

For networks with small inputs, DNN verification tools use a more aggressive abstraction process to achieve more precise computation. Specifically, they use LP solving to compute the tightest bounds for all input variables from the generated linear constraints. This computation is efficient when the number of inputs is small.

After tightening input bounds DNN verification tools apply abstraction to approximate the output bounds, which can be more precise with better input bounds. For networks with large number of inputs, we obtain input bounds from the input property ϕ_{in}.[TVN]: Vu: to rewrite

7.3 Adversarial Attacks (§H)

Like other DNN verifiers [21, 52], `NeuralSAT` tool implements a fast-path optimization that attempts to disprove or falsify the property before running DPLL(T). `NeuralSAT` uses two *adversarial attack* algorithms to find counterexamples to falsify properties. First, we try a randomized attack approach [12], which is a derivative-free sampling-based optimization [51], to generate a potential counterexample. If this approach fails, we then use a gradient-based approach [32] to create another potential counterexample.

If either attack algorithm gives a valid counterexample, `NeuralSAT` returns `sat`, indicating that property is invalid. If both algorithms cannot find a counterexample or they exceed a predefined timeout, `NeuralSAT` continues with its DPLL(T) search.

7.4 Multiprocessing

[TVN]: Do we still use this? or we use the Parallel DPLL(T) in §8.3.1.1 instead? For networks with small inputs, `NeuralSAT` uses a simple approach to create and solve subproblems in parallel. Given a verification problem $N_{orig} = (\alpha, \phi_{in}, \phi_{out})$, where α is the DNN and $\phi_{in} \Rightarrow \phi_{out}$ is the desired property, `NeuralSAT` creates subproblems $N_i = (\alpha, \phi_{in_i}, \phi_{out})$, where ϕ_{in_i} is the i-th subregion of the input region specified by ϕ_{in}. Intuitively, each subproblem checks if the DNN produces the output ϕ_{out} from a smaller input region ϕ_{in_i}. The combination of these subproperties $\bigwedge \phi_{in_i} \Rightarrow \phi_{out}$ is logically equivalent to the original property $\phi_{in} \Rightarrow \phi_{out}$.

Given k available threads, `NeuralSAT` splits the original input region to obtain subproblems as described and and runs DPLL(T) on k subproblems in parallel. `NeuralSAT` returns `unsat` if it verifies all subproblems and `sat` if it found a counterexample in any subproblem. For example, we split the input region $\{x_1 \in [-1, 1], x_2 \in$

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

7.5 GPU Processing

[TVN]: Hai, talk about how modern tools like crown and neuralsat leverage GPU. Give details. Provide small CUDO code and examples as needed.

Chapter 8

The NeuralSAT Algorithm

NeuralSAT [18, 19] is a relatively late competitor in the DNN verification space, but it has quickly become a strong contender, consistently placed among the top tools at DNN verification competitions §14.

At its core, NeuralSAT is BaB, but follows a DPLL(T) framework [13] and includes unique optimizations and heuristics to improve the search performance. Thus, NeuralSAT is essentially an SMT solver with respect to a theory, in this case, the theory of DNNs.

8.1 Overview

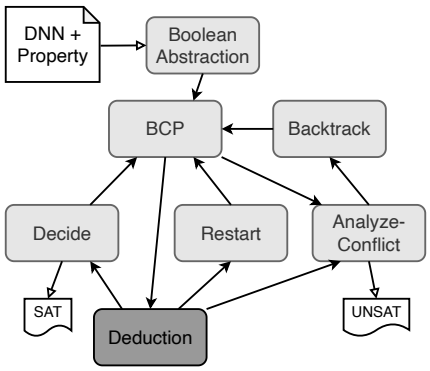


Fig. 8.1: The NeuralSAT DPLL(T) Algorithm.

Fig. 8.1 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 with respect to the constraints

encoding the DNN and the property of interest.

`NeuralSAT` now enters an iterative process to find a truth assignment—*activation pattern* (§3.3)—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 satisfiability of the constraints of the current assignment with the property of interest.

If the T-solver confirms satisfiability, `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`.

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

8.2 Illustration

Example 8.2.1. 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:

$$\alpha \ \wedge \ -1 \leq x_1 \leq 1 \ \wedge \ -2 \leq x_2 \leq 2 \ \wedge \ x_5 > 0. \tag{8.2.1}$$

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

Tab. 8.1: `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. 8.2.1. For this example, `NeuralSAT` uses four iterations, summarized in Tab. 8.1, to determine that no such assignment exists and the problem is thus `unsat`.

In *iteration 1*, as shown in Fig. 8.1, `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.

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., $x_4 = \texttt{on}$). 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.

8.3 NeuralSAT's Optimizations

NeuralSATimplements several optimizations to improve the performance of the search. First are the common optimizations used by other DNN verifiers, such as input splittings (§7) and adversarial attacks (§H). In addition, NeuralSAT implements several unique optimizations [19] to improve the performance of the search. These are neuron stability, restart tree, and restart.

8.3.1 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, [TVN]: any neuron stable? or they are all unstable?

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.

Alg. 2. `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)

```
model ← MIP(α, φ_in, φ_out, σ) // create model ( Eq. 12.3.1) with current assignment
[v_1, ..., v_m] ← GetUnassignedVariable(σ) // get all m current unassigned variables
[v'_1, ..., v'_m] ← Sort([v_1, ..., v_m]) // prioritize tightening order
[v'_1, ..., v'_k] ← Select([v'_1, ..., v'_m], k) // select top-k unassigned variables, k ≤ m
// stabilize k neurons in parallel
parfor v_i in [v'_1, ..., v'_k] do
    if (v_i.lower + v_i.upper) ≥ 0 then // lower is closer to 0 than upper, optimize lower first
        Maximize(model, v_i.lower) // tighten lower bound of v_i
        if v_i.lower < 0 then // still unstable
            Minimize(model, v_i.upper) // tighten upper bound of v_i
    else // upper is closer to 0 than lower, optimize upper first
        Minimize(model, v_i.upper) // tighten upper bound of v_i
        if v_i.upper > 0 then // still unstable
            Maximize(model, v_i.lower) // tighten lower bound of v_i
```

Stabilization involves the solution of a mixed integer linear program (MILP) system [45]:

$$\begin{aligned}
&\text{(a)} \quad z^{(i)} = W^{(i)}\hat{z}^{(i-i)} + b^{(i)};\\
&\text{(b)} \quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&\text{(c)} \quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0;\\
&\text{(d)} \quad a_j^{(i)} \in \{0, 1\};\\
&\text{(e)} \quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned} \tag{8.3.1}$$

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. 5 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. 12.3.1. Note that the neuron lower ($l_j^{(i)}$) and upper bounds ($u_j^{(i)}$) 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 11 or line 17) and `Minimize` (line 13 or line 15), are performed by an external LP solver (e.g., Gurobi [24]).

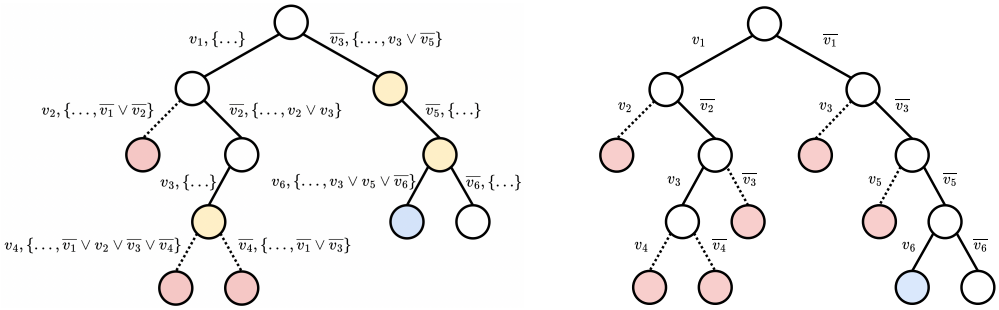
Example 8.3.1. [TVN]: Need a concrete example here. Use the DNN in Fig. 1.1 and create the MILP system in Eq. 12.3.1.

8.3.1.1 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 [20] 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 up 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. 6 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 [**?**, 31].

In addition to raw speedup 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. 6 line **??**, to inform the next round of search. As we show in S**??**, the engineering of this form of parallelism in DPLL(T) leads to substantial performance improvement.



(a) `NeuralSAT` (b) Others

Fig. 8.2: Search tree explored by `NeuralSAT` (a) and other verifiers (b) during a verification run. [TVN]: Hai, instead of other verifiers can we say this search tree is for a native/general BaB approach? 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., $\overline{v_3}$).

8.3.2 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 [15,21,48]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [22] 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.

8.4 `NeuralSAT` vs. BaB

[TVN]: TODO: talk about how `NeuralSAT` is BaB but has things such as CDCL to help prune the search space.

[TVN]: rewrite: technically we don't split if the guess was right 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 §8.4).

As mentioned in §3.4, 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. 8.2 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. 8.2b 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. 8.2a. `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. 8.2b.

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 as `NeuralSAT`.

Chapter 9

The Reluplex Algorithm

Reluplex [26] is a classical BnB approach for verifying neural networks. The technique extends the simplex method [36] to support the ReLU activation function (**Reluplex** = **Relu** + Sim**plex**). Reluplex has been succeeded by the Marabou [28] tool, which are more efficient and scalable. However, the core ideas of Reluplex are still relevant and therefore we present it here.

9.1 Illustration

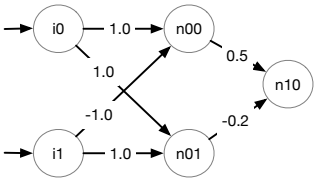


Fig. 9.1: A DNN example

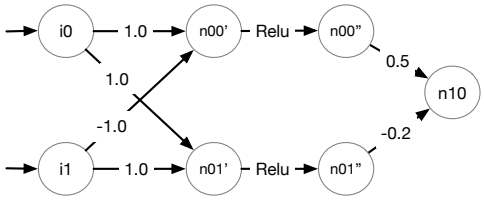


Fig. 9.2: Separating nodes to represent RELU

Example 9.1.1. We use the DNN in Fig. 9.1 to demonstrate Reluplex. Assume we want to check that the DNN has the property

$$(0 \leq i_0 \leq 0.5 \wedge -2 \leq i_1 \leq -1) \implies (n_{10} < 0 \vee n_{10} > 0.5). \tag{9.1.1}$$

That is, when the inputs i_0, i_1 fall within certain ranges, then the result n_{10} has certain values. As mentioned in 3.2, we turn this into a satisfiability problem by negating the property and checking if the negation is unsatisfiable. In this case, we want to check if the negation of Eq. 9.1.1 is unsat:

$$(0 \leq i_0 \leq 0.5 \wedge -2 \leq i_1 \leq -1) \wedge 0 \leq n_{10} \leq 0.5. \tag{9.1.2}$$

If Eq. 9.1.2 is unsat, then the DNN satisfies the property in Eq. 9.1.1—there exists no assignment of the inputs i_0, i_1 such that the output n_{10} is within the range $[0, 0.5]$. Otherwise, the DNN does not have the property and we can return a counterexample.

Problem Encoding A DNN can be encoded as a conjunctions of constraints. The ReLU node v can be encoded as a pair of variables v' and v'', where v' is used to connect the nodes of previous layers to v, and v'' is used to connect v to the next layer.

For example, in Fig. 9.1, the ReLU node $n00 = \max(i0 - 1i1, 0)$ is encoded using two variables $n00' = i0 - 1i1$ and $n00'' = \max(n00', 0)$. The DNN inFig. 9.2 shows the network in Fig. 9.1 with additional nodes representing ReLU encoding.

The first step to use Repluplex is to encode the constraints representing the DNN and the problem into a format that Reluplex accepts. The DNN in Fig. 9.1 can be encoded as a conjunction of equalities:

$$\begin{aligned} n00' = i_0 - i_1, \quad n00'' = \max(n00', 0), \\ n01' = i_0 + i_1, \quad n01'' = \max(n01', 0), \\ n_{10} = 0.5n00'' - 0.2n01'' \end{aligned}$$

Basic Variables We also introduce three new *basic* (auxilliary or slack) variables to store the relationships of the DNN's constraints:

$$a_1 = i_0 - v_{12}^b - n00', \tag{9.1.3}$$

$$a_2 = i_0 + v_{12}^b - n01', \tag{9.1.4}$$

$$a_3 = 0.5n00'' - 0.2n01'' - n_{10} \tag{9.1.5}$$

In simplex terminology, a basic variable is a variable that is used to represent the relationship between other variables in the constraints. For example, a_1 represents the difference between i_0, v_{12}^b, and $n00'$. The 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 a variable that is not used to represent the relationship between other variables in the constraints. For example, i_0 and i_1 are non-basic because they are not used to represent the relationship between other variables in the constraints. Non-basic variables are typically the input variables of the DNN.

Iteration `Reluplex` first assigns 0 to all [TVN]: basic? variables in the constraints in Eq. 9.1.3– Eq. 9.1.5. This assignment—essentially an initial guess—would likely cause issues such as variables violating their bounds. `Reluplex` works by iteratively fixing these invalid values until it finds a feasible assignment (and returns `sat`) or cannot do so (and returns `unsat`).

This iterative updating process can be demonstrated through a sequence of configuration updates over the variables. Tab. 9.1 shows the initial configuration with all values assigned to 0. Observe that the lower and upper bounds of the inputs i_0, i_1

and output n_{10} are specified in the property in Eq. 9.1.2, the lower bounds of v_f's representing ReLU are 0, and the other hidden variables are unbounded.

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.1: Configuration #1

Tab. 9.1 shows that i_1 is out-of-bounds because $0 \notin [-2, -1]$. To fix i_1, which is non-basic, Reluplex simply updates it to a valid value, e.g., i_1+= $-1.0 = -1.0$, which means adding -1.0 to the current value of i_1 and thus having $i_1 = -1.0$. Now, because a_1, a_2 depend on i_1 as shown in Eq. 9.1.3 and Eq. 9.1.4, respectively, this update to i_1 also changes a_1, a_2:

$$a_1 \mathrel{+}= 1.0 = 1.0 \text{ because } a_1 = i_0 - i_1 - n00',$$
$$a_2 \mathrel{+}= -1.0 = -1.0 \text{ because } a_2 = i_0 + i_1 - n01'.$$

Tab. 9.2 shows the new configuration. Tab. 9.2 also shows a_1, a_2 violate their bounds and need to be fixed. Assume Reluplex picks a_1. To fix a_1, which is a basic variable, Reluplex pivots (swaps) it with one of the variables it depends on as shown in the constraint $a_1 = i_0 - v_{12}^b - n00'$ in Eq. 9.1.3. Assume Reluplex pivots a_1 with $n00'$, we get

$$n00' = i_0 - v_{12}^b - a_1. \tag{9.1.6}$$

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.2: Configuration #2

Reluplex now updates the non-basic a_1 to 0 ($a_1 \mathrel{+}= -1.0 = 0$). This also changes $n00'$ to 1.0 ($n00' \mathrel{+}= 1.0 = 1.0$) because $n00'$ depends on a_1 as shown in Eq. 9.1.6.

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.3: Configuration #3

Tab. 9.3 shows the new configuration. Tab. 9.3 also shows the basic variable a_2 is out-of-bound. To fix it, Reluplex pivots a_2 with a variable it depends on as shown

in the constraint $a_2 = i_0 + v^b_{12} - n01'$ in Eq 9.1.4. Assume Reluplex pivots a_2 with $n01'$, we get

$$n01' = i_0 + v^b_{12} - a_2 \tag{9.1.7}$$

Now a_2 becomes non-basic and is updated to 0 through $a_2 += 1.0 = 0$. As $n01'$ depends on a_1 as shown in Eq. 9.1.7, we make the change $n01'$ `-=` $1.0 = -1.0$.

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.4: Configuration #4

Tab. 9.4 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 $n00', n00''$. This is because $n00'' = \max(n00', 0)$ but we have $n00' = 1$ thus $\max(1, 0) = 1$, which is not $n00'' = 0$. Thus, Reluplex needs to fix either $n00''$ or $n00'$.

Assume Reluplex picks $n00''$. Because $n00''$ is non-basic, we simply update it, i.e., $n00'' = +1 = 1$. As a_3 depends on $n01''$, i.e., $a_3 = 0.5n00'' - 0.2n01'' - n_{10}$, Reluplex also makes the change $a_3 += 0.5 \times 1.0 = 0.5$.

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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**
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.5: Configuration #5

Tab. 9.5 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 $n00'', n01'', n_{10}$ because of the constraint $a_3 = 0.5n00'' - 0.2n01'' - n_{10}$ in Eq. 9.1.5. Assume we pivot a_3 with n_{10}, we get

$$n_{10} = 0.5n00'' - 0.2n01'' - a_3 \tag{9.1.8}$$

Now, a_3 becomes non-basic and we update it to 0 through $a_3 += -0.5 = 0$. As n_{10} depends on a_3 as shown in Eq 9.1.8, we make the change $n_{10} += 0.5 = 0.5$.

Tab. 9.6 shows the new configuration. At this point, Reluplex no longer has any out-of-bound or inconsinstent values, and thus stops and returns `sat` with the values in the **Val** row in Tab. 9.6 as the satisfying assignment for the formula in Eq 9.1.2.

Thus, in this example, we conclude that property in Eq. 9.1.1 is *not valid* for the DNN in Fig. 9.1 because for the inputs $i_0 = 0, v_{-12} = -1$, the DNN gives the output $n_{10} = 0.5$, which violates the property in Eq. 9.1.1.

	i_0	i_1	$n00'$	$n00''$	$n01'$	$n01''$	n_{10}	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**
UP	0.5	-1	∞	∞	∞	∞	0.5	0	0	0

Tab. 9.6: Configuration #6

9.2 Excercises

Exercise 9.2.1. Consider the DNN in Fig. 9.1 and the property in Eq. 9.1.1. Use `Reluplex` to verify the property. If you find a counterexample, provide it. You will need to provide all the steps in the `Reluplex` algorithm, e.g., DNN encoding, basic variable encoding, and the `Reluplex` search through a series of configurations as shown in 9.1.1.

Chapter 10

GPU and Multicore Parallelism

Part IV

Survey of DNN Verification Tools

Chapter 11

Popular Techniques and Tools

Part V

Advanced Topics

Chapter 12

Proof Generation and Checking

As DNN tools become more complex (e.g., SOTA tools have 20K LoCs), they are more prone to bugs. VNN-COMP'23 [9] 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`. This is a serious issue, as it can lead to unsafe DNNs being deployed in safety-critical applications, such as autonomous driving and medical diagnosis.

While checking counterexamples is relatively straightforward (we can just evaluate the DNN on the input), checking `unsat` results—proving no counterexample exists—is far more challenging. This would require verifiers to track their decision steps, which are often complex and large.

12.1 Proof Generation

A proof of satisfiability (sat) is an input that violates the property, i.e., a counterexample. We can easily check such a counterexample c by evaluating $\phi(c, N(c))$ (i.e., running the DNN on the counterexample). In fact, VNN-COMPs already requires competing DNN verification tools to return counterexamples demonstrating satisfiability.

In contrast, the proof of an unsatisfiability result (which explains why *no possible inputs* can violate the property) is inherently more complex to generate (§12.1), requires a more sophisticated encoding (§12.2), and an efficient checking algorithm (§12.3). We are mainly interested in `unsat` proofs.

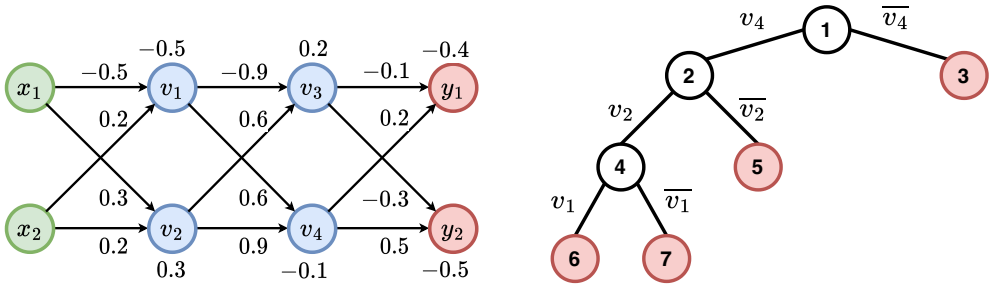
12.1.1 Proof Generation for Branch and Bound (BaB) Algorithms

As mentioned in §6, major DNN verification techniques share the common "branch and bound" (BaB) search algorithm. The BaB structure, shown in Alg. 1, splits the problem into smaller subproblems and use abstraction to compute bounds to prune

Alg. 3. 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)

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



(a) (b)

Fig. 12.1: (a) A simple DNN (a redrawn of Fig. 6.1), and (b) A proof tree produced verifying the property $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2)$.

the search space. This commonality allows us to bring proof generation capabilities with minimal overhead to existing DNN verification tools.

Alg. 3 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 the proof tree (line 11). The proof tree is a binary tree 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 We reuse the example in §6 to illustrate $\texttt{BaB}_{\texttt{ProofGen}}$. Recall the goal is to verify that the DNN in Fig. 12.1(a) (a redraw of Fig. 6.1) has the property $(x_1, x_2) \in [-2.0, 2.0] \times [-1.0, 1, 0] \Rightarrow (y_1 > y_2)$. $\texttt{BaB}_{\texttt{ProofGen}}$ generates the proof tree

in Fig. 12.1(b) to show unsatisfiability, i.e., the property is valid.

First, $\texttt{BaB}_{\texttt{ProofGen}}$ initializes the activation pattern set `ActPatterns` with an empty activation pattern $\emptyset$. Then $\texttt{BaB}_{\texttt{ProofGen}}$ enters a loop (line 6-line 20) to search for a satisfying assignment or a proof of unsatisfiability. In the first iteration, $\texttt{BaB}_{\texttt{ProofGen}}$ selects the only available activation pattern $\emptyset \in$ `ActPatterns`. It calls `Deduce` to check the feasibility of the problem based on the current activation pattern. `Deduce` uses abstraction to approximate that from the input constraints the output values are feasible for the given network. Since `Deduce` cannot decide infeasibility, $\texttt{BaB}_{\texttt{ProofGen}}$ randomly selects a neuron to split (`Decide`). Let us assume that it chooses v_4 to split, which essentially means the problem is split into two independent subproblems: one with v_4 active and the other with v_4 inactive. $\texttt{BaB}_{\texttt{ProofGen}}$ then adds v_4 and $\overline{v_4}$ to `ActPatterns`.

In the second iteration, $\texttt{BaB}_{\texttt{ProofGen}}$ has two subproblems (that can be processed in parallel). For the first subproblem with v_4, `Deduce` cannot decide infeasibility, so it selects v_2 to split. It then conjoins v_4 with v_2 and then with $\overline{v_2}$ and adds both conjuncts to `ActPatterns`. For the second subproblem with $\overline{v_4}$ inactive, `Deduce` determines that the problem is unsatisfiable and $\texttt{BaB}_{\texttt{ProofGen}}$ saves the node v_4 to the proof tree, as node 3, to indicate one unsatisfiable pattern, i.e., whenever the network has v_4 being inactive, the problem is unsatisfiable.

In the third iteration, $\texttt{BaB}_{\texttt{ProofGen}}$ has two subproblems for $v_4 \wedge v_2$ and $v_4 \wedge \overline{v_2}$. For the first subproblem, `Deduce` cannot decide infeasibility, so it selects v_1 to split. It then conjoins v_1 and then $\overline{v_1}$ to the current activation pattern and adds them to `ActPatterns`. For the second subproblem, `Deduce` determines that the problem is unsatisfiable and $\texttt{BaB}_{\texttt{ProofGen}}$ saves the node $v_4 \wedge \overline{v_2}$ to the proof tree, as node 5.

In the fourth iteration, $\texttt{BaB}_{\texttt{ProofGen}}$ has two subproblems for $v_4 \wedge v_2 \wedge v_1$ and $v_4 \wedge v_2 \wedge \overline{v_1}$. Both subproblems are determined to be unsatisfiable, and $\texttt{BaB}_{\texttt{ProofGen}}$ saves them to the proof tree as nodes 6 and 7, respectively.

Finally, $\texttt{BaB}_{\texttt{ProofGen}}$ has an empty `ActPatterns`, stops the search, and returns `unsat` and the proof tree.

12.2 Proof Language

In §12.1 we have shown that the BaB class of DNN verification techniques can generate a binary tree that represents a proof of unsatisfiability. Rather than record such proofs 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.

To meet this goal, we introduce $\texttt{BaB}_{\texttt{ProofLang}}$, a proof language to specify DNN proofs. This language is inspired by the SMTLIB format [4] used for SMT solving, which has also been adopted by the VNNLIB language [44] to specify DNNs and their properties for verification.

⟨*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*⟩ ::= $<$ | $\leq$ | $>$ | $\geq$

⟨*input-var*⟩ ::= X_⟨*constant*⟩

⟨*output-var*⟩ ::= Y_⟨*constant*⟩

⟨*hidden-var*⟩ ::= N_⟨*constant*⟩

⟨*constant*⟩ ::= **Int** | **Real**

Fig. 12.2: 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))))
```

Fig. 12.3: $\texttt{BaB}_{\texttt{ProofLang}}$ example format of the proof tree in Fig. 12.1b.

Fig. 12.2 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* are declared with specific piece-wise linear (PWL) activation functions, such as ReLU or Leaky ReLU. These hidden variables correspond to the internal nodes of the neural network that process the input data through various activation functions.

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*. Each assertion is composed of a *formula*, which can involve terms and logical operators. Formulas include simple comparisons between terms (e.g., less than, greater than) or more complex logical combinations using `and` and `or` operators. The terms used in these formulas can be input variables, output variables, hidden variables, or constants.

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 12.2.1. The proof in Fig. 12.3 corresponds to the proof tree in Fig. 12.1b. 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 [38] 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 §12.3.

12.3 Proof Checker

Finally, we need to check that the generated proof is correct and that the original NN verification problem is indeed unsatisfiable. The checker must be efficient to handle large proofs and trusted of its results (if it verifies the proof, then the original NNV problem is proved).

To achieve this, we present $\texttt{BaB}_{\texttt{ProofCheck}}$, a proof checker for $\texttt{BaB}_{\texttt{ProofLang}}$ proofs. $\texttt{BaB}_{\texttt{ProofCheck}}$ 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.

12.3.1 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. 3.2.2 (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. 4 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 (§12.2). 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 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.

Alg. 4. $\texttt{BaB}_{\texttt{ProofCheck}}$ algorithm.

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

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

Example 12.3.1. For the $\texttt{BaB}_{\texttt{ProofLang}}$ proof in Fig. 12.3, we need to check that the four leaf nodes 3, 5, 6, and 7 of the proof tree in Fig. 12.1b 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. 3.2.2 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.

12.3.1.1 MILP Formulation

$\texttt{BaB}_{\texttt{ProofCheck}}$ formulates MILP problems [45] and check for feasible solutions using off-the-shelf LP solving. Formally, the MILP problem is defined as:

$$\begin{aligned}
&\text{(a)} \quad z^{(i)} = W^{(i)}\hat{z}^{(i-i)} + b^{(i)};\\
&\text{(b)} \quad y = z^{(L)}; x = \hat{z}^{(0)};\\
&\text{(c)} \quad \hat{z}_j^{(i)} \geq z_j^{(i)}; \hat{z}_j^{(i)} \geq 0;\\
&\text{(d)} \quad a_j^{(i)} \in \{0,1\};\\
&\text{(e)} \quad \hat{z}_j^{(i)} \leq a_j^{(i)} u_j^{(i)}; \hat{z}_j^{(i)} \leq z_j^{(i)} - l_j^{(i)}(1 - a_j^{(i)});
\end{aligned} \tag{12.3.1}$$

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, respectively. This encodes the semantics of a ReLU-based DNN: (a) the affine transformation computing the pre-activation value for a neuron in terms of outputs in the preceding layer; (b) the inputs and outputs in terms of the adjacent hidden layers; (c) assertion that post-activation values are non-negative and no less than pre-activation values; (d) neuron

activation status indicator variables that are either 0 or 1; and (e) 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 $\hat{z}_j^{(i)} = max(z_j^{(i)}, 0)$.

12.3.1.2 Correctness

Alg. 4 returns certified iff the input $\texttt{BaB}_{\texttt{ProofLang}}$ proof tree is unsatisfiable. This proof tree encodes a disjunction of constraints, one per tree path, where each constraint represents an activation pattern of the network (the leaf node). The algorithm checks each constraint using LP solving and only returns certified iff every one of them is unsatisfiable. We note that this correctness argument assumes that the LP solver is correct – in practice multiple solvers could be used to guard against errors in that component. We note that it is standard for proof checkers to assume the correctness of a small set of external tools, e.g., checkers that use theorem provers assume the correctness of the underlying prover [**?**].

12.3.2 Optimizations

While the core $\texttt{BaB}_{\texttt{ProofCheck}}$ algorithm in Alg. 4 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.

12.3.2.1 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* [19] 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.

$\texttt{BaB}_{\texttt{ProofCheck}}$ uses the algorithm in Alg. 5 to traverse the DNN and compute stable neurons. The algorithm initializes the MILP model with input constraints (line 1) and then iterates over each layer of the network. Next, for each layer, it creates constraints (line 4 or line 6) depending on the layer type. Moreover, it uses approximation to estimate bounds of neuron values to determine neuron stability (line 7). Next, it filters unstable neurons (line 8) and attempt to make them stable by optimizing either their lower (`Maximize`) or upper (`Minimize`) bounds.

Alg. 5. `CreateStabilizedMILP` procedure.

```
input  : DNN 𝒩, property ϕ_in ⇒ ϕ_out, parallel factor k
output : MILP model
model ← AddInputConstrs(ϕ_in) // input property
// Add MILP constraints for each layer of network
for layer in 𝒩 do
    if isPiecewiseLinear(layer) then
        // add constraints Eq. 12.3.1 (c), (d), (e)
        model ← AddConstrsPWL(layer, ϕ_in, ϕ_out)
    else // this layer is linear
        // add constraints Eq. 12.3.1 (a), (b)
        model ← AddConstrsLinear(layer, ϕ_in, ϕ_out)
        // estimate upper and lower bounds
        layer_bounds ← EstimateBounds(layer)
        // select unstable neurons to be stabilized
        [v_1, ..., v_k] ← GetUnstableNeurons(layer_bounds)
        // stabilize selected neurons in parallel
        parfor v_i in [v_1, ..., v_k] do
            // optimize lower first
            if (v_i.lower + v_i.upper) ≥ 0 then
                Maximize(model, v_i.lower)
                if v_i.lower < 0 then // still unstable
                    Minimize(model, v_i.upper)
            else // optimize upper first
                Minimize(model, v_i.upper)
                if v_i.upper > 0 then // still unstable
                    Maximize(model, v_i.lower)
model ← AddObjectives(model, ϕ_out) // output property
return model
```

12.3.2.2 Pruning Leaf Nodes

Another optimization $\texttt{BaB}_{\texttt{ProofCheck}}$ employs is that it does not check child nodes if the parent node is unsatisfiable. In an $\texttt{BaB}_{\texttt{ProofLang}}$ 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. 12.1b). 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.

We 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. Future work will explore tuning N to a given verification problem.

12.3.2.3 Parallelization

Finally, the structure of $\texttt{BaB}_{\texttt{ProofLang}}$ proof tree 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.

12.4 Rounding Errors

[TVN]: Give concrete examples of rounding errors in VNN-COMP

Chapter 13

DNN Verification Benchmarks

Here we survey the latest benchmarks in DNN verification and how SOTA tools perform on them. These results are taken from the Verification Neural Network Competitions (VNN-COMP) [8] and the recent work by Duong et al. [19].

13.1 VNN-COMP Benchmarks

[TVN]: check this table with VNN-COMP'24 reports. The Report seems to have a lot more benchmarks for regular tracks. Also many numbers don't quite match. We should probably use the benchmarks from VNN-COMP'24 report. I also ? on place for you to fill in

13.1.1 ACAS Xu

Networks The ACASXu benchmark consists of 10 properties defined over 45 neural networks used to issue turn advisories to aircraft to avoid collisions. The neural

Tab. 13.1: Benchmark instances. U: `unsat`, S: `sat`, ?: `unknown`.

Benchmarks	**Networks**			**Per Network**		**Tasks**	
	Type	Networks	Neurons	Parameters	Input Dim	Properties	Instances (U/S/?)
cGan	Conv. + Vision Trans.			500K–68M	5		
NN4Sys	ReLU + Sigmoid			33k–37M	1–308		
LinearizeNN	FC. + Conv. + Vision Trans. + Res. + ReLU			203k	4		
Collins RUL CNN	Conv. + ReLU, Dropout			60k–262k	400–800		
cifar100	FC + Conv. + Res., ReLU + BatchNorm			2.5M–3.8M	3072		
tinyimagenet	FC + Conv. + Res., ReLU + BatchNorm			3.6M	9408		
Metaroom	Conv. + FC, ReLU			466k–7.4M	5376		
TLL Verify Bench	Two-Level Lattice NN (FC. + ReLU)			17k–67M	2		
Acas XU	FC. + ReLU	45	300	13k	5	10	139/47/0
Dist Shift	FC. + ReLU + Sigmoid			342k – 855k	792		
safeNLP	FC. + ReLU			4k	30		
CORA	FC. + ReLU			575k, 1.1M	784, 3072		

networks have 300 neurons arranged in 6 layers, with ReLU activation functions. There are 5 inputs corresponding to the aircraft states, and 5 network outputs, where the minimum output is used as the turn advisory the system ultimately produces.

Specifications VNN-COMP uses the original 10 properties [26], where properties 1–4 are checked on all 45 networks as was done in later work by the original authors [28]. Properties 5–10 are checked on a single network. The total number of benchmarks is therefore 186.[TVN]: does not match table

13.1.2 Cifar2020

Motivation This benchmark combines two convolutional CIFAR10 networks from last year's VNN-COMP 2020 with a new, larger network with the goal to evaluate the progress made by the whole field of Neural Network verification.

Networks The two ReLU networks `cifar_10_2_255` and `cifar_10_8_255` with two convolutional and two fully-connected layers were trained for ℓ_∞ perturbations of $\epsilon = \frac{2}{255}$ and $\frac{8}{255}$, respectively, using COLT [**?**] and the larger `ConvBig` with four convolutional and three fully-connected networks, was trained using adversarial training [**?**] and $\epsilon = \frac{2}{255}$.

Specifications We draw the first 100 images from the CIFAR10 test set and for every network reject incorrectly classified ones. For the remaining images, the specifications describe a correct classification under an ℓ_∞-norm perturbation of at most $\frac{2}{255}$ and $\frac{8}{255}$ for `cifar_10_2_255` and `ConvBig` and `cifar_10_8_255`, respectively and allow a per sample timeout of 5 minutes.

13.1.3 VGGNET16

Proposed by Stanley Bak, Stony Brook University

Motivation This benchmark tries to scale up the size of networks being analyzed by using the well-studied VGGNET-16 architecture [**?**] that runs on ImageNet. Input-output properties are proposed on pixel-level perturbations that can lead to image misclassification.

Networks All properties are run on the same network, which includes 138 million parameters. The network features convolution layers, ReLU activation functions, as well as max pooling layers.

Specifications Properties analyzed ranged from single-pixel perturbations to perturbations on all 150528 pixles (L-infinity perturbations). A subset of the images was used to create the specifications, one from each category, which was randomly chosen to attack. Pixels to perturb were also randomly selected according to a random seed.

Link https://github.com/stanleybak/vggnet16_benchmark2022/

13.1.4 cGAN

Proposed by Feiyang Cai, Ali Arjomandbigdeli, Stanley Bak (Stony Brook Univ.). Link: https://github.com/feiyang-cai/cgan_benchmark2023

This benchmark targets robustness verification for generative models—an area often overlooked compared to discriminative networks. It uses conditional GANs trained to generate images of vehicles at specific distances. The generator takes a 1D distance input and a 4D noise vector; the discriminator outputs a real/fake score and a predicted distance. Models vary in architecture (CNNs, vision transformers) and image size (32×32, 64×64). The verification task checks whether the predicted distance from the generated image matches the input condition, under small input perturbations.

Motivation While existing neural network verification benchmarks focus on discriminative models, the exploration of practical and widely used generative networks remains neglected in terms of robustness assessment. This benchmark introduces a set of image generation networks specifically designed for verifying the robustness of the generative networks.

Networks The generative networks are trained using conditional generative adversarial networks (cGAN), whose objective is to generate camera images that contain a vehicle obstacle located at a specific distance in front of the ego vehicle, where the distance is controlled by the input distance condition. The network to be verified is the concatenation of a generator and a discriminator. The generator takes two inputs: 1) a distance condition (1D scalar) and 2) a noise vector controlling the environment (4D vector). The output of the generator is the generated image. The discriminator takes the generated image as input and outputs two values: 1) a real/fake score (1D scalar) and 2) a predicted distance (1D scalar). Several different models with varying architectures (CNN and vision transformer) and image sizes (32x32, 64x64) are provided for different difficulty levels.

Specifications The verification task is to check whether the generated image aligns with the input distance condition, or in other words, verify whether the input distance condition matches the predicted distance of the generated image. In each specification, the inputs (condition distance and latent variables) are constrained in

small ranges, and the output is the predicted distance with the same center as the condition distance but with slightly larger range.

Chapter 14

VNN-COMPs

Chapter 15

Benchmarks Generation

Chapter 16

Conclusion

Appendix A

Schedule

1. Week 1: Introduction (videos of neural networks, basic terminologies and concepts).
2. Week 2: HW: Introducing yourself on Canvas.
3. Week 3:
4. Week 3:
5. Week 4:
6. Week 5:
7. Week 6:
8. Week 7:
9. Week 8:
10. Week 9:
11. Week 10:

Appendix B

Comparing neural networks with software

Tab. B.1: Similarities

Aspect	Description
Representation	Represented using if-then-else statements
Input-Output Mapping	Take Inputs and produce outputs
Determinism	Can produce deterministic outputs
Execution Path	Paths from input to output
Specifications	Can have specifications involving preconditions over inputs and postconditions over outputs
Logical Constraints	Can be represented as logical constraints
Bugs and Errors	Can have bugs that violate specifications

Tab. B.2: Differences

Aspect	Traditional Software	Neural Networks
Explainability	Can be explained through code inspection	Lacks explicit explainability mechanisms
Designing	Written by human	Created by machine through learning from training
Size	Small to medium-sized codebases	Large number of neurons and layers
Error Causing	arise from bugs in logic or implementation	arise from insufficient training data, overfitting, etc
Error Incurring	Can cause crashes, incorrect outputs, or unexpected behavior	Can cause incorrect predictions or unexpected beha
Error Handling	try-catch blocks for exception handling	Lacks explicit error-handling mechanisms

Appendix C

Logic and Linear Programming Foundations for DNN Verification

The field of neural network verification (NNV) relies heavily on two fundamental areas: *logic* (to reason about properties as logical formulae) and *optimization* (to reason about numerical constraints). This chapter provides the necessary background on both topics in detail, starting with propositional logic and satisfiability, and then moving to linear and mixed-integer programming.

C.1 Propositional Logic and Satisfiability

C.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:

- Constants: $\top$ and $\bot$ are formulae.
- Variables: Any propositional variable p is a formula.
- Negation: If ϕ is a formula, then $\neg\phi$ is a formula.
- Binary connectives: If ϕ, ψ are formulas, then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \to \psi)$.

Example C.1.1.

$$(x \vee y) \wedge (\neg x \vee z)$$

is a formula involving three variables x, y, z.

Notice here that we purely define the syntax of propositional logic. We have not talked about semantics (meanings) such as truth values.

C.1.2 Semantics

A logical formula has a truth value, which can be either `True` or `False`. This is the *semantics* of the formula.

An *assignment* α maps each formula to `True` or `False`. The truth value of a formula is defined recursively

$$\begin{aligned}
\alpha(\top) &= \texttt{True}, \\
\alpha(\bot) &= \texttt{False}, \\
\alpha(p) &\text{ iff } \alpha(p) = \texttt{True}, \\
\alpha(\neg\phi) = \texttt{True} &\text{ iff } \alpha(\neg\phi) = \texttt{True}, \\
\alpha(\phi \wedge \psi) = \texttt{True} &\text{ iff } \alpha(\phi) = \texttt{True} \text{ and } \alpha(\psi) = \texttt{True}, \\
\alpha(\phi \vee \psi) = \texttt{True} &\text{ iff } \alpha(\phi) = \texttt{True} \text{ or } \alpha(\psi) = \texttt{True}, \\
\alpha(\phi \to \psi) = \texttt{True} &\text{ iff } \alpha(\phi) = \texttt{False} \text{ or } \alpha(\psi) = \texttt{True}.
\end{aligned}$$

Exercise C.1.1. 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. Identify the satisfying assignments.

C.1.3 Satisfiability and Validity

- A formula ϕ is *satisfiable* if there exists **some** α such that $\alpha(\phi) = \texttt{True}$.
- A formula ϕ is *unsatisfiable* if **no** assignment satisfies it.
- A formula ϕ is *valid* if **all** assignments satisfy it.

Example C.1.2.

$$x$$

is satisfiable (e.g., $\alpha(x) = \texttt{True}$).

$$(x) \wedge (\neg x)$$

is unsatisfiable (no assignment works). In contrast,

$$(x \vee \neg x)$$

is valid.

Exercise C.1.2. Show that $(\phi \to \psi)$ is equivalent to $(\neg\phi \vee \psi)$. (Hint: construct a truth table.)

C.1.4 SAT Solvers

A SAT solver takes a propositional formula as input and determines its satisfiability by searching for a satisfying assignment. If one exists, it returns the assignment; otherwise, it reports `unsat`. Modern solvers use search algorithms such as *DPLL (Davis–Putnam–Logemann–Loveland)* and *CDCL (Conflict-Driven Clause Learning)*.

Example C.1.3.

$$\begin{aligned}\phi_1 &= (x \vee y) \wedge (\neg x \vee z) \quad \Rightarrow \text{SAT},\\ \phi_2 &= (x) \wedge (\neg x) \quad \Rightarrow \text{UNSAT}.\end{aligned}$$

Exercise C.1.3. Use a SAT solver (such as `pysat` in Python) 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.

Conjunctive Normal Form (CNF) In practice, SAT solvers do not take arbitrary formulae as input. Instead, they require the formula to be in *Conjunctive Normal Form (CNF)*. A formula is in CNF if it is a conjunction of disjunctions of literals, where a literal is either a variable p or its negation $\neg p$.

Example C.1.4.

$$(x \vee y) \wedge (\neg x \vee z)$$

is in CNF. Each clause $(x \vee y)$, $(\neg x \vee z)$ is a disjunction of literals.

Convert 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.

Exercise C.1.4. Transform the formula

$$(x \to y) \wedge (y \to z)$$

into CNF.

C.2 Satisfiability Modulo Theories (SMT)

SAT solvers only reason about propositional formulae, which only contain Boolean variables and operators. 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 SAT by adding background theories (e.g., linear arithmetic, bit-vectors, arrays).
- An SMT formula mixes Boolean structure with constraints from these theories.

Example C.2.1. 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.

C.2.1 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.

Exercise C.2.1. 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.

C.2.2 Z3 SMT Solver

Z3 [14] is a well-known SMT solver developed by Microsoft Research. Here we'll show how to use it for various logical and arithmetic problems.

Example C.2.2 (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}
```

Exercise C.2.2. Modify the code in C.2.2 to check

$$(x \wedge \neg x).$$

Confirm that the result is `unsat`.

Example C.2.3 (Linear Constraints.)**.** We now use Z3 as an SMT solver to check linear constraints.

```
from z3 import *
x, y = Reals('x y')

# an example
s = Solver()
s.add(x > 0, y > 0, 2*x + y <= 1)
print(s.check())  # Output: unsat

# another example
s = Solver()
s.add(x + y <= 4, x >= 0, y >= 0)
print(s.check()) # Output: sat
print(s.model())  # Output: {x=0, y=0}


# another example
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}
```

Exercise C.2.3. 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.

Exercise C.2.4. Use Z3 to check the formula given in C.2.1. Do it for both the reals and integers.

Example C.2.4. Use Z3 to check the formula `If` $x > 3$ `then` $y > 2$'' `AND` ``$x \leq 1$. Is it satisfiabile? What model does Z3 return?

C.2.3 Checking DNN-style Properties with Z3

Example 5. Consider a one-neuron ReLU: $y = \max(0, x - 1)$. We want to check if the property holds: "For all $x \leq 0$, we have $y = 0$."

Encode negation: find an $x \leq 0$ with $y \neq 0$.

```
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))

# Negation of property
s.add(x <= 0, y != 0)

print(s.check())
```

- Z3 returns `unsat`, so the property holds.

Exercise 14 (Homework). Modify the above to check property: "For all $x \geq 2$, we have $y \geq 1$." What does Z3 return?

—

—

Appendix D

Linear Programming (LP)

D.1 Linear Constraints and Feasible Regions

A *linear constraint* has the form:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b.$$

The set of points satisfying all constraints is called the *feasible region*.

Example. Constraints:

$$x + y \leq 4, \quad x \geq 0, \quad y \geq 0.$$

The feasible region is a triangle bounded by the axes and the line $x + y = 4$.

Exercise 5 (In-class). Sketch the feasible region defined by:

$$2x + y \leq 6, \quad x \geq 0, \quad y \geq 0.$$

D.2 Linear Programming Problems

An LP is an optimization problem:

$$\max c^T x \quad \text{subject to } Ax \leq b,\ x \geq 0.$$

Example. Maximize $x + y$ subject to

$$x + y \leq 4, \quad x \geq 0, \quad y \geq 0.$$

Solution: maximum is 4 at $(4, 0)$ or $(0, 4)$.

Exercise 6 (Homework). Solve the LP:

$$\max 3x + 2y \quad \text{subject to} \quad 2x + y \leq 8,\ x + 2y \leq 6,\ x, y \geq 0.$$

D.2.1 Duality (Optional Enrichment)

Every LP has a *dual* problem, and strong duality holds under mild conditions. Understanding duality is useful in verification because it enables certificates of infeasibility.

—

D.3 Mixed-Integer Linear Programming (MILP)

D.3.1 From Logic to MILP

SAT handles logic, LP handles linear inequalities. MILP combines both: some variables are required to be integers (often binary).

Example. Constraint:

$$x \in \{0, 1\}, \quad y \geq 2x.$$

- If $x = 0$, then $y \geq 0$. - If $x = 1$, then $y \geq 2$.

This encodes logical branching into optimization.

Exercise 7 (In-class). Formulate "if $x = 1$, then $y = 0$" as linear constraints using a binary variable and a big-M method.

D.3.2 Encoding ReLU Constraints

A ReLU is $y = \max(0, x)$. We introduce a binary variable z to encode activation:

$$\begin{aligned} &y \geq x, \\ &y \geq 0, \\ &y \leq x + M(1 - z), \\ &y \leq Mz. \end{aligned}$$

Here M is a large constant.

- If $z = 0$, $y = 0$. - If $z = 1$, $y = x$.

Exercise 8 (Homework). Write MILP constraints for $y = \max(0, 2x + 1)$ with $-10 \leq x \leq 10$.

D.3.3 Example: Verifying a Tiny Network

Network: input $x \in [0, 1]$, hidden neuron $h = \max(0, 2x - 1)$, output $y = h$.

Property. Verify: for all $x \in [0, 0.4]$, $y = 0$.

Encoding.

- Input constraint: $0 \leq x \leq 0.4$.
- ReLU encoding for $h = \max(0, 2x - 1)$.
- Negation of property: $y > 0$.

Result. MILP is infeasible $\Rightarrow$ property holds.

Exercise 9 (Homework). Modify the property to "for all $x \in [0.6, 1]$, $y > 0$." Check if the MILP is feasible.

—

—

D.4 Summary and Further Exercises

- Logic: propositional formulas, CNF, satisfiability.
- LP: optimization over linear constraints.
- MILP: integrates logic and LP, used for ReLU encoding.
- Application: encoding DNN verification as SAT/LP/MILP.

Homework Project. Take the network $y = \max(0, w_1x_1 + w_2x_2 + b)$ with $w_1 = 1, w_2 = -1, b = 0$. Verify the property: "For all $x_1, x_2 \in [0, 1]$, we have $y \leq 1$." Formulate the MILP and solve it using an MILP solver (e.g., Gurobi, CPLEX, or open-source solvers).

Appendix E

Logics and Linear Programming

E.1 Logics and Satisfiability

E.1.1 Satisfiability (SAT)

The classical satisfiability (SAT) problem asks if a given propositional formula over Boolean variables can be satisfied [7]. Given a formula f, a SAT solver returns `sat` if it can find a satisfying assignment that maps truth values to variables of f that makes f evaluate to true, and `unsat` if it cannot find any satisfying assignments. The problem is NP-Complete and research into methods for efficiently solving problem instances has been ongoing for multiple decades.

DPLL Fig. E.1 gives an overview of **DPLL**, a SAT solving technique introduced in 1961 by Davis, Putnam, Logemann, and Loveland [13]. 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`.

CDCL Modern DPLL solving improves the original version with Conflict-Driven Clause Learning (*CDCL* [6, 33, 34]). 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.

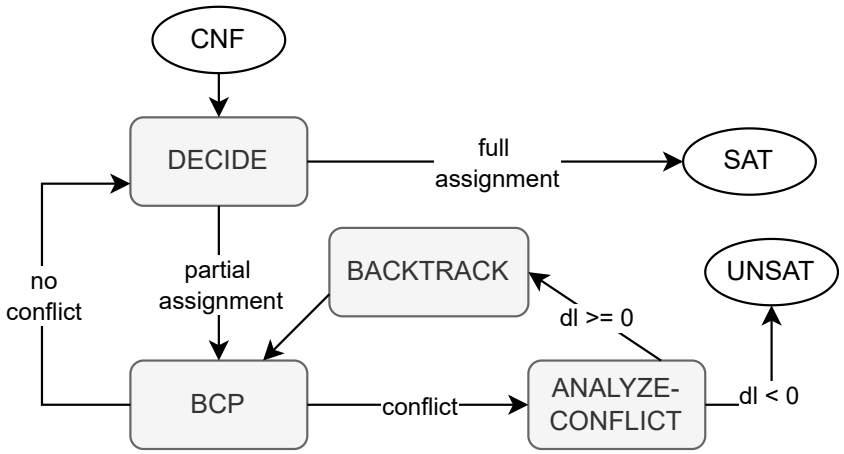


Fig. E.1: The classicial DPLL Algorithm.

DPLL(T) DPLL(T) [37] 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 [**?**] and CVC4 [3] include solvers supporting a wide range of theories including linear arithmetic, nonlinear arithmetic, string, and arrays [30].

[1]SMT is Satisfiability Modulo Theories and the T in DPLL(T) stands for Theories.

Appendix F

Software vs DNN Verification

Appendix G

NeuralSAT Algorithm

Alg. 6 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

$$\alpha \wedge \phi_{in} \wedge \overline{\phi_{out}}. \tag{G.0.1}$$

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. G.0.1 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.

G.1 Boolean Abstraction

BooleanAbstraction (Alg. 6 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.

Alg. 6. 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
```

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 G.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 $v_3 \vee \overline{v_3}$ and $v_4 \vee \overline{v_4}$. The assignment $\{x_3 = T, x_4 = F\}$ creates the constraint $0.5x_1 - 0.5x_2 - 1 > 0 \wedge x_1 + x_2 - 2 \leq 0$.

G.2 DPLL

After `BooleanAbstraction`, `NeuralSAT` iteratively searches for an assignment satisfying the status clauses (Alg. 6, lines 6– 20). `NeuralSAT` combines DPLL components (e.g., `Decide`, `BCP`, `AnalyzeConflict`, `Backtrack` and `Restart`) to assign truth values with a theory solver (SG.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. 6, 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.

G.2.1 Decide

From the current assignment, `Decide` (Alg. 6, 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 [11, 15]. 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`).

G.2.2 Boolean Constraint Propagation (BCP)

From the current assignment and clauses, `BCP` (Alg. 6, 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.

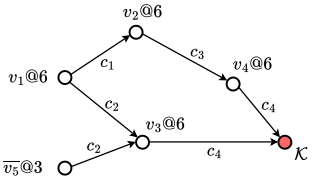
Implication Graph `BCP` uses an *implication graph* [5] to represent the current assignment and the reason for each BCP implication. In this graph, a node represents the assignment and an edge $i \xrightarrow{c} j$ means that `BCP` infers the assignment represented in node j due to the unit clause c 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` (§G.2.3), which analyzes the conflict in the graph to learn clauses.

Example G.2.1. Assume we have the clauses in Fig. G.1(a), the assignments $\overline{v_5}@3$ and $v_1@6$ (represented in the graph in Fig. G.1(b) by nodes $\overline{v_5}@3$ and $v_1@6$, re-

[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)

Fig. G.1: (a) A set of clauses, (b) an implication graph, and (c) learning a new clause.

spectively), and are currently at decision level *dl* 6. Because of assignment v_1@6, `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.

G.2.3 Conflict Analysis

Given an implication graph with a conflict (e.g., Fig. G.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. 7, 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

Alg. 7. 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
```

$\overline{lit}$ as the only satisfied literal in the clause. Now, `AnalyzeConflict` resolves cl and $ante$ 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:

$$\frac{(a_1 \vee ... \vee a_n \vee \beta) \qquad (b_1 \vee ... \vee b_m \vee \overline{\beta})}{(a_1 \vee ... \vee a_n \vee b_1 \vee ... \vee b_m)} \quad \text{(Binary-Resolution)} \qquad \text{(G.2.1)}$$

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 G.2.2. Fig. G.1(c) demonstrates `AnalyzeConflict` using the example in §G.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. G.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. G.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 §G.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. 3.2.2 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`.

G.2.4 Backtrack

From the clause returned by `AnalyzeConflict`, `Backtrack` (Alg. 6, 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 [5], 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 G.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. G.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 §G.2.2, `BCP` will immediately infer $v_1 = T$ by noticing that c_5 is now a unit clause).

G.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 [11, 15]. This problem, which has been recognized in early SAT solving, motivates the introduction of restarting the search [22] to avoid being stuck in such a *local optima*.

`NeuralSAT` uses a simple restart heuristic (Alg. 6, 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. 8.2).

G.3 Deduction (Theory Solving)

`Deduction` (Alg. 6, 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

Alg. 8. 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

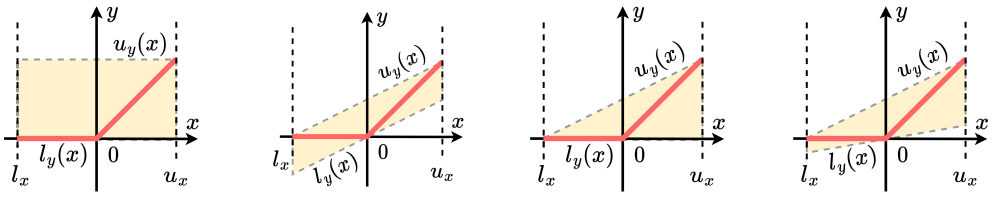
```
1  solver ← LPSolver(σ, α ∧ φ_in ∧ ¬φ_out)
2  if Solve(solver) ≡ INFEASIBLE then return false
3  if isTotal(σ) then return true // orig prob ( Eq. 3.2.2) is satisfiable
4  input_bounds ← TightenInputBounds(solver, φ_in)
5  output_bounds, hidden_bounds ← Abstract(α, σ, input_bounds)
6  if Check(output_bounds, ¬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. 8 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. 3.2.2 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

Fig. G.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. G.2 compares the (a) interval [47], (b) zonotope [42], and (c, d) polytope [43,48,50] 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. G.2d) [48–50] 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 [30,37] 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 G.3.1. For the illustrative example in 8.2.1, 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 $0.5 \leq x_3$.

Part VI

Optimizations and Strategies

Appendix H

Adversarial Attacks

Modern DNN verifiers ($\alpha\beta$-`CROWN`, `NeuralSAT`, etc) often first run an adversarial attack technique to check for obvious counterexamples. If one is found, the property is violated (and the verifier returns `sat`). If no counterexample is found, the verifier proceeds to the more expensive search-based verification algorithm (e.g., $\texttt{BaB}_{\texttt{NV}}$ as shown in **??**).

Thus, in the first phase of running adversarial attacks, the goal to *falsify* the property, i.e., find a counterexample that violates the property. If this fails, then the goal is to *verify* the property, i.e., prove that no counterexample exists. Of course, during the verification phase, the verifier may also discover counterexamples that the falsify phase misses.

H.1 Random Search Attack

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

Example H.1.1. Suppose the DNN input ranges are:

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

and the output is:

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

We wish to use RS 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: $(x_1 = -1,\ x_2 = 2)$.

2. Projected Gradient Descent (PGD)

Projected Gradient Descent (PGD) is a strong first-order adversarial attack that iteratively moves the input in the direction that maximizes property violation, while projecting (clipping) the input back into the allowed domain after every step.

1. **Initialize** at a valid input, e.g., $(x_1^{(0)}, x_2^{(0)}) = (0, 0)$.
2. **For each step t:**
   (a) Compute the gradient:

$$\nabla_x x_5 = \left(\frac{\partial x_5}{\partial x_1}, \frac{\partial x_5}{\partial x_2} \right)$$

   For our example,

$$\nabla_x x_5 = (2, -1.5)$$

   (b) Update:

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

   (c) **Project (Clip)**:

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

   (d) If $x_5 \leq 0$, **return** $(x_1^{(t+1)}, x_2^{(t+1)})$ as a counterexample.

Example:
Let $\eta = 0.5$, initial point $(x_1, x_2) = (0, 0)$.

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

Clip to the ranges: $x_1^{(1)} = -1$, $x_2^{(1)} = 0.75$.
Compute:

$$x_5 = 2 \times (-1) - 1.5 \times 0.75 + 1 = -2 - 1.125 + 1 = -2.125 < 0$$

So, the PGD attack finds a counterexample at $(x_1 = -1,\ x_2 = 0.75)$.

Bibliography

[1] S. Bak. nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement. In *NASA Formal Methods Symposium*, pages 19–36. Springer, 2021.

[2] 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.

[3] 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.

[4] 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.

[5] C. W. Barrett. Decision Procedures: An Algorithmic Point of View. *J. Autom. Reason.*, 51(4):453–456, 2013.

[6] 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.

[7] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[14] 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.

[15] 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.

[16] 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.

[17] H. Duong, T. Nguyen, and M. Dwyer. A DPLL(T) Framework for Verifying Deep Neural Networks. *arXiv preprint arXiv:2307.10266*, 2024.

[18] 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.

[19] 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.

[20] C. Ferguson and R. E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI*, volume 88, pages 128–132, 1988.

[21] 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.

[22] C. P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[23] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. https://www.deeplearningbook.org, last accessed August 29, 2025.

[24] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[25] 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.

[26] 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.

[27] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, 60(1):87–116, 2022.

[28] 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.

[29] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[30] D. Kroening and O. Strichman. *Decision procedures*. Springer, 2008.

[31] 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.

[32] 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.

[33] 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.

[34] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[35] 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.

[36] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[37] 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.

[38] ONNX Community. ONNX: Open neural network exchange. https://onnx.ai/, 2017. Accessed: 2025-04-01.

[39] OVAL-group. OVAL - Branch-and-Bound-based Neural Network Verification, 2023. https://github.com/oval-group/oval-bab.

[40] M. Sälzer and M. Lange. Reachability in simple neural networks. *Fundamenta Informaticae*, 189, 2023.

[41] 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.

[42] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev. Fast and effective robustness certification. *Advances in neural information processing systems*, 31, 2018.

[43] 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.

[44] A. Tacchella, L. Pulina, D. Guidotti, and S. Demarchi. The international benchmarks standard for the Verification of Neural Networks, 2023.

[45] V. Tjeng, K. Y. Xiao, and R. Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[46] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems*, 31, 2018.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Y. Yu, H. Qian, and Y.-Q. Hu. Derivative-free optimization via classification. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[52] H. Zhang, S. Wang, K. Xu, L. Li, B. Li, S. Jana, C.-J. Hsieh, and J. Z. Kolter. General cutting planes for bound-propagation-based neural network verification. *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.