Automated Verification of Soundness of DNN Certifiers

AVALJOT SINGH, University of Illinois Urbana-Champaign, USA
YASMIN CHANDINI SARITA, University of Illinois Urbana-Champaign, USA
CHARITH MENDIS, University of Illinois Urbana-Champaign, USA
GAGANDEEP SINGH, University of Illinois Urbana-Champaign, USA

The uninterpretability of Deep Neural Networks (DNNs) hinders their use in safety-critical applications. Abstract Interpretation-based DNN certifiers provide promising avenues for building trust in DNNs. Unsoundness in the mathematical logic of these certifiers can lead to incorrect results. However, current approaches to ensure their soundness rely on manual, expert-driven proofs that are tedious to develop, limiting the speed of developing new certifiers. Automating the verification process is challenging due to the complexity of verifying certifiers for arbitrary DNN architectures and handling diverse abstract analyses.

We introduce ProveSound, a novel verification procedure that automates the soundness verification of DNN certifiers for arbitrary DNN architectures. Our core contribution is the novel concept of a *symbolic DNN*, using which, ProveSound reduces the soundness property, a universal quantification over arbitrary DNNs, to a tractable symbolic representation, enabling verification with standard SMT solvers. By formalizing the syntax and operational semantics of ConstraintFlow, a DSL for specifying certifiers, ProveSound efficiently verifies both existing and new certifiers, handling arbitrary DNN architectures.

Our code is available at https://github.com/uiuc-focal-lab/constraintflow.git

CCS Concepts: • **Theory of computation → Logic and verification**; **Automated reasoning**; **Operational semantics**; **Program verification**.

Additional Key Words and Phrases: Abstract interpretation, Language design, Machine learning, Program analysis, Verification

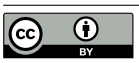
ACM Reference Format:
Avaljot Singh, Yasmin Chandini Sarita, Charith Mendis, and Gagandeep Singh. 2025. Automated Verification of Soundness of DNN Certifiers. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 144 (April 2025), 29 pages. https://doi.org/10.1145/3720509

1 Introduction

While DNNs can achieve impressive performance, there is a growing need for their safety and robustness in safety-critical domains like autonomous driving [8], healthcare [2], etc., due to their susceptibility to environmental and adversarial noise [31, 68]. Formal certification of DNNs can be used to assess their performance on a large, potentially infinite set of inputs, thereby providing guarantees on DNN behavior. Abstract Interpretation-based DNN certifiers are used widely for formally certifying DNNs, balancing cost and precision tradeoffs [3, 5–7, 9, 15, 16, 18, 20, 30, 32, 34, 37, 39, 43–46, 49, 51, 52, 54–57, 60–64, 66, 67, 70, 72, 73].

Abstract Interpretation-based DNN certifiers must satisfy the *over-approximation-based soundness* property to ensure correctness. Currently, when a new DNN certifier is proposed, its soundness is

Authors' Contact Information: Avaljot Singh, avaljot2@illinois.edu, University of Illinois Urbana-Champaign, USA; Yasmin Chandini Sarita, University of Illinois Urbana-Champaign, USA, ysarita2@illinois.edu; Charith Mendis, University of Illinois Urbana-Champaign, USA, charithm@illinois.edu; Gagandeep Singh, University of Illinois Urbana-Champaign, USA, ggnds@illinois.edu.



ACM 2475-1421/2025/4-ART144
https://doi.org/10.1145/3720509

proved manually using arduous pen-and-paper proofs. These proofs show that the outputs computed by abstract transformers over-approximate the outputs of the DNN on concrete inputs. Developing these proofs demands an expert-level understanding of abstract interpretation and substantial experience in proving mathematical lemmas and theorems. Consequently, the development of DNN certifiers is often confined to a small group of experts. Automating the verification of DNN certifiers would significantly reduce these barriers, enabling more widespread development of reliable certifiers. However, this automation presents several challenges, which we outline below.

Challenge 1: Imperative Programming. While one approach to verifying the mathematical soundness of DNN certifiers could be to use program verifiers such as Dafny [26], they are unsuitable because the commonly-used libraries implementing the DNN certifiers, such as auto_LiRPA [69], ELINA [48], and ERAN [45], are extensive code-bases in general-purpose programming languages, employing complex imperative programming paradigms, such as pointer arithmetic. Verifying the soundness of these libraries would require isolating the mathematical logic from their implementation and modeling the algorithm's behavior on an arbitrary DNN.

Challenge 2: Universal Quantification. Since a DNN is an input to a DNN certifier, the over-approximation-based soundness of the certifier is a universally quantified assertion over all possible DNNs, which significantly complicates its verification. To illustrate this, consider verifying the certifier for a fixed DNN, where the architecture is known. In this case, the soundness can be verified by representing all neurons and edges in the DNN (represented as a Directed Acyclic Graph) using symbolic variables and then executing the certifier symbolically. The difficulty arises when the input DNN is arbitrary and so, cannot be directly represented symbolically. A DNN might be a simple fully-connected network with ReLU activations, or a more complex architecture such as ResNet, with arbitrary residual connections and activations. These DNNs have drastically different architectures, and the DNN certifier may have different execution traces for them. So, verifying the soundness of the certifier for one architecture does not guarantee soundness for arbitrary DNNs.

Challenge 3: Complex DNN Certifiers. Popular DNN certifiers like [45, 65, 73] associate polyhedral bounds with each neuron, which makes it difficult to naively model the certifier behavior using symbolic execution. For example, a polyhedral lower bound for a neuron n might be expressed as $n \geq 5n_1 + n_2$, where the neurons n_1, n_2 are neurons located anywhere in the DNN, independent of the DNN architecture. This adds a structure over the neurons (beyond the DNN architecture) that is unknown before executing the certifier. Further, $n, n_1, n_2, \cdots$ are symbolic variables even during a concrete execution of the certifier. So, modeling the certifier behavior using symbolic execution entails modeling the symbolic variables (neurons) as SMT symbolic variables. The correctness of this modeling is unclear and is not explored in existing work [4, 53].

Challenge 4: Huge Query Size. One approach would be to represent a DNN as a complete DAG where each neuron is a vertex, but this results in massive graphs (i.e. 10^4 neurons in a modest-size DNN will have around $(10^4)^2$ edges), with a weight of zero in the DAG representing the absence of an edge in the DNN. However, a complete DAG would lead to a huge query, which would overwhelm current SMT solvers, making them either fail or take an impractically long time. So, naively modeling arbitrary DNNs as a complete DAG is impractical for realistic-size DNNs.

To the best of our knowledge, no existing technique can automatically verify the soundness of abstract interpretation-based DNN certifiers while accommodating a diverse range of certifiers, ensuring soundness for arbitrary DNNs, and maintaining efficiency and scalability.

This work. We design a novel automated bounded verification procedure—ProveSound—which can verify the soundness of DNN certifiers for arbitrary DNNs. ProveSound is based on the novel concept of a *symbolic DNN*—an abstract neural network that represents all subgraphs of any arbitrary DNN on which a DNN certifier can be applied (§ 5). By leveraging symbolic DNNs, we transform the universally quantified soundness conditions into a tractable symbolic representation,

verifying which is sufficient to prove the certifier's soundness on arbitrary DNNs. We offload the verification of this tractable symbolic representation to off-the-shelf SMT solvers. Recently, a preliminary design of a Domain Specific Language (DSL)—ConstraintFlow—was proposed for specifying the core mathematical logic of abstract interpretation-based DNN certifiers decoupling it from any implementation details [42]. However, its syntax and semantics are not formalized. So, we design a BNF grammar, type-system, and operational semantics for ConstraintFlow, which enables ProveSound to verify the soundness of certifier specifications within ConstraintFlow.

Main contributions.

- We develop a type-system for ensuring well-typed programs in ConstraintFlow and also provide operational semantics. We also develop symbolic semantics for ConstraintFlow and a novel concept of a symbolic DNN to devise a verification procedure—ProveSound—to automatically find bugs or verify the soundness of the specified DNN certifiers.
- We establish formal guarantees and provide proofs that include type-soundness, and the soundness of the automated verification procedure, ProveSound, w.r.t. the operational semantics of ConstraintFlow.
- We provide an extensive evaluation to demonstrate that ProveSound enables proving the correctness or detecting bugs in existing and new abstract transformers for contemporary DNN certifiers and new DNN certifiers with new abstract domains. Using ProveSound, for the first time, we can automatically verify the soundness of DNN certifiers for DNNs with an arbitrary number of layers, each with millions of learned parameters.

2 Background

In this section, we provide the necessary background needed for abstract interpretation-based DNN certifiers. While the concepts introduced are relevant to a broad range of certifiers, we describe the widely used DeepPoly certifier [45] and use it as our running example throughout the paper.

2.1 Abstract Interpretation-Based DNN Certifiers

We use a definition of DNNs similar to the one used in [42]. A DNN is represented as a Directed Acyclic Graph (DAG) with neurons as the vertices and edges corresponding to the non-zero weights in the DNN architecture. The value of each neuron is determined by a DNN operation f, which receives as input a set of neurons, referred to as the *previous* neurons p. DNN operations can be categorized into two categories: (i) primitive operations and (ii) composite operations. Primitive operations include the addition and multiplication of two neurons as well as non-linear activations like ReLU, sigmoid, etc. Composite operations are operations that can be expressed as combinations of primitive functions. Examples include affine transformation of neurons (fully connected layers or convolution layers) or activations like maxpool ,etc.

For a given DNN operation f, the input consists of m neurons, where m denotes the arity of f (e.g., $f_{add} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ has $m = 2$). Let $\mathbf{x}$ represent an m-dimensional input to a layer, with each dimension corresponding to a neuron. DNN certifiers take a potentially infinite set of inputs, represented as $c = \{\mathbf{x_i}\}$ and $c \in C$, where C is the concrete domain. Concrete elements $c_1, c_2 \in C$ are ordered by subset inclusion $\subseteq$. Certification involves defining an abstract domain $\mathcal{A}$ and abstract transformers $f^\sharp$ for each f. The DNN certifiers map concrete inputs to abstract elements via an abstraction function α and propagate these through the network using abstract transformers. Abstract elements $a \in \mathcal{A}$ can be mapped back to concrete values using a concretization function γ.

Definition 2.1. *An abstract transformer $f^\sharp$ is sound w.r.t. the DNN operation f if $\forall a \in \mathcal{A} \cdot \forall c \in C \cdot c \subseteq \gamma(a) \implies f(c) \subseteq \gamma(f^\sharp(a))$, where the semantics of f are lifted to the natural set semantics.*

2.2 DeepPoly DNN Certifier

We focus on abstract domains that associate fields with each neuron n to impose constraints on their values. These fields form an *abstract shape* s with corresponding constraints denoted as $\mathcal{P}(s, n)$. Popular abstract interpretation-based certifiers, including DeepPoly, use such domains. In the DeepPoly abstract domain, an abstract element $a \in \mathcal{A}$ is represented as a conjunction of constraints over the neurons' abstract shapes, i.e., $a = (s_1, \ldots, s_N)$, where N is the total number of neurons. For each neuron n, its abstract shape is $s_n = \langle l_n, u_n, L_n, U_n \rangle$, where $l_n, u_n \in \mathbb{R} \cup \{-\infty, \infty\}$, and L_n, U_n are affine expressions of neurons in the DNN. The associated over-approximation-based constraints are $\mathcal{P}(s, n) \triangleq (l_n \leq n \leq u_n) \wedge (L_n \leq n \leq U_n)$. Thus, the concretization function $\gamma(a) = \{(n_1, \ldots, n_m) \in \mathbb{R}^m \mid \forall i \in [m], (l_{n_i} \leq n_i \leq u_{n_i}) \wedge (L_{n_i} \leq n_i \leq U_{n_i})\}$

An abstract transformer updates the abstract shape of the output neuron based on the concrete operation f while leaving the others unchanged. For the Affine operation, the updated abstract shape is $s'_n = \langle l'_n, u'_n, L'_n, U'_n \rangle$, where $L'_n = U'_n = b + \sum_{i=1}^{l} w_i n_i$, where the bias ($b$) and the weights ($w_i$) are the DNN's learned parameters. To compute the lower concrete bound (l'_n), DeepPoly performs a backsubstitution step which starts with the lower polyhedral expression, $e = L'_n$. At each step, $e = c'_0 + \sum_{i=1}^{l} c'_i n_i$, each n_i in e is replaced with its own lower or upper polyhedral bound depending on the sign of the coefficient c'_i, i.e., $e \leftarrow c'_0 + \sum_{i=1}^{l} (c'_i \geq 0 \; ? \; c'_i L_{n_i} : c'_i U_{n_i})$. This step is repeated until all the neurons in e are in the input layer, after which the constituent neurons are replaced with their respective lower or upper concrete bounds, i.e., if $e = c''_0 + \sum_{i=1}^{l} c''_i n_i$, then $l'_n = c''_0 + \sum_{i=1}^{l} (c''_i \geq 0 \; ? \; c''_i l_{n_i} : c''_i u_{n_i})$. The upper concrete bound u'_n is also computed similarly.

3 Overview

We first provide an overview of DNN certifier specification in ConstraintFlow using the DeepPoly specification from [42] as a running example, followed by the novel type-system and semantics for ConstraintFlow. Finally, we show the soundness verification of the certifier specification.

3.1 ConstraintFlow

ConstraintFlow introduces datatypes specific to DNN certifiers including `Neuron`, `PolyExp`, and `Ct`. Neurons are represented as `Neuron`. The type `PolyExp` represents affine expressions over neurons and `Ct` represents symbolic constraints. Since some DNN certifiers use symbolic variables to specify constraints over the neuron values [44, 46, 55], we introduce the `sym` construct to declare a symbolic variable of the type `Sym`. We also introduce `SymExp` to capture symbolic expressions over these symbolic variables. By treating polyhedral and symbolic expressions as first-class members, we can define the operational semantics of constructs that can directly operate on these new types. These include (i) binary arithmetic operations like '+', (ii) `map`, which applies a function to each constituent neuron or symbolic variable in a polyhedral or symbolic expression, and (iii) `traverse`, which repeatedly applies `map` to a polyhedral expression until a termination condition is met. The formal semantics (discussed in detail in § 4.3) enable automated reasoning and verification.

In ConstraintFlow, a DNN certifier is specified through three main steps: (i) specifying the abstract shape for each neuron along with its soundness constraints, (ii) defining the abstract transformers for each DNN operation, and (iii) determining how constraints propagate through the network. We illustrate the different steps of specifying a DNN certifier in ConstraintFlow using the DeepPoly specification in Fig. 1.

3.1.1 Abstract Domain. The specification of a DNN certifier starts by defining the abstract domain used by the certifier (Line 1 of Fig. 1). In ConstraintFlow, this is done by defining the abstract shape (s) associated with each neuron and the constraints defining the over-approximation-based

```
1  Def shape as (Real l, Real u, PolyExp L, PolyExp U) {(curr[l] <= curr) and (curr[u] >= curr)
       and (curr[L] <= curr) and (curr[U] >= curr)};

2  Func priority(Neuron n) = n[layer];
3  Func concretize_lower(Neuron n, Real c) = (c >= 0) ? (c * n[l]) : (c * n[u]);
4  Func concretize_upper(Neuron n, Real c) = (c >= 0) ? (c * n[u]) : (c * n[l]);
5  Func replace_lower(Neuron n, Real c) = (c >= 0) ? (c * n[L]) : (c * n[U]);
6  Func replace_upper(Neuron n, Real c) = (c >= 0) ? (c * n[U]) : (c * n[L]);
7  Func backsubs_lower(PolyExp e, Neuron n) = (e.traverse(backward,priority,false,replace_lower)
       {e <= n}).map(concretize_lower);
8  Func backsubs_upper(PolyExp e, Neuron n) = (e.traverse(backward,priority,false,replace_upper)
       {e >= n}).map(concretize_upper);

9  Transformer DeepPoly{
10   Affine -> (backsubs_lower(prev.dot(curr[w]) + curr[b], curr),
11              backsubs_upper(prev.dot(curr[w]) + curr[b], curr),
12              prev.dot(curr[w]) + curr[b],
13              prev.dot(curr[w]) + curr[b]);
14   Relu -> prev[l] > 0 ?
15             (prev[l], prev[u], prev, prev) :
16             (prev[u] < 0 ?
17                 (0, 0, 0, 0) :
18                 (0, prev[u], 0, ((prev[u] / (prev[u] - prev[l])) * prev) - ((prev[u] * prev[l])
       / (prev[u] - prev[l]))));
19 }

20 Flow(forward, -priority, false, DeepPoly);
```

Fig. 1. DeepPoly specification in ConstraintFlow

soundness condition ($\mathcal{P}$). These are specified for the `curr` neuron, which serves as a syntactic placeholder for all neurons in the DNN. For example, the DeepPoly abstract shape and its constraints can be defined in ConstraintFlow as illustrated in Fig. 1, where l, u, L, U are user-defined members of the abstract shape, accessed via square bracket notation (`curr` $[\cdot]$). The DeepPoly soundness condition is encoded as: $(l \leq n) \wedge (u \geq n) \wedge (L \leq n) \wedge (U \geq n)$.

We formalize the syntax for ConstraintFlow (§ 4.1), allowing the users to define arbitrary abstract shapes. For instance, abstract domains can combine polyhedral and novel symbolic expressions. Symbolic variables (ϵ) are subject to default constraints, $-1 \leq \epsilon_i \leq 1$, defining multi-dimensional polyhedra. The constraint `curr` <> `curr`$[Z]$ indicates that `curr` is embedded in the polyhedron defined by `curr`$[Z]$, meaning there exists an assignment to the symbolic variables in `curr`$[Z]$ such that `curr` = `curr`$[Z]$:

```
Def shape as (Real l, Real u, PolyExp L, PolyExp U, SymExp Z) {curr[l] <= curr, curr[u] >= curr,
     curr[L] <= curr, curr[U] >= curr, curr <> curr[Z]};
```

3.1.2 Abstract Transformers. After defining the abstract domain, the second step is to specify the abstract transformers for different DNN operations. In Fig. 1, lines 2-8 show the user-defined functions used within the transformer definitions in lines 9-19 within the `Transformer` construct. The implicit inputs to the `Transformer` construct are `curr`, representing the current neuron, and `prev`, representing the previous neurons. `prev` is a list for DNN operations with multiple inputs, like

`Affine`, and a single neuron in case of operations with a single input, like `ReLU`. The transformer for each DNN operation specifies the computations for updating the four fields of the abstract shape: l, u, L, and U. The transformers for `Affine` and `ReLU` operations are shown in Fig. 1 in lines 10 and 14 respectively. Using the semantics of the ConstraintFlow constructs, we show how the DeepPoly specification in Fig. 1 simulates the mathematical logic of DeepPoly (explained in § 2). The ConstraintFlow semantics also allow us to explore variants of DeepPoly.

In the DeepPoly `Affine` transformer, the polyhedral bounds (L and U) are given by `prev.dot` (`curr` [w]) + `curr` [b]. There are many ways to compute the concrete lower l and upper bounds u. Consider `concretize_lower` and `replace_lower` functions from Fig. 1 that respectively replace a neuron with its lower or upper concrete and polyhedral bounds based on its coefficient. We can compute the lower concrete bound for `curr`, by applying the `concretize_lower` to all the neurons in the lower polyhedral expression, i.e., (`prev.dot` (`curr` [w]) + `curr` [b]).`map` (`concretize_lower`). We can compute a more precise polyhedral lower bound by first applying `replace_lower` to each constituent neuron, i.e., (`prev.dot` (`curr` [w]) + `curr` [b]).`map` (`replace_lower`). We can repeat this several times, following which, we can apply `concretize_lower` to concretize the bound. In the standard implementation, the number of applications of `replace_lower` is unknown because it is applied until the polyhedral bound only contains neurons from the input layer of the DNN. Although this is precise, it might be costly to perform this computation until the input layer is reached. So, custom stopping criteria can be decided, balancing the tradeoff between precision and cost. Note that the order in which the neurons are substituted with their bounds also impacts the output's precision.

To specify arbitrary graph traversals succinctly, we provide the `traverse` construct, which decouples the stopping criterion from the neuron traversal order. `traverse` operates on polyhedral expressions and takes as input the direction of traversal and three functions—a user-defined stopping function, a priority function over neurons specifying the order of traversal and a neuron replacement function. In each step, `traverse` applies the priority function to each constituent neuron in the polyhedral expression. Then, it applies the neuron replacement function to each constituent neuron with the highest priority among the neurons on which the stopping condition evaluates to false. The outputs are then summed up to generate a new polyhedral expression. This process continues until the stopping condition is true on all the constituent neurons or all the neurons are in the input or output layer depending on the traversal order. We can use `traverse` to specify the backsubstitution step and hence the DeepPoly `Affine` transformer as shown in Fig. 1.

3.1.3 Flow of Constraints. Existing DNN certifiers propagate constraints from the input to the output layer or in reverse [58, 65, 71]. Further, the order in which abstract shapes of neurons are computed impacts analysis precision. In ConstraintFlow, the specification of the order of application is decoupled from the actual transformer specification, so the soundness verification of the transformer remains independent of the traversal order. We formalize this syntax and semantics to provide adjustable knobs to define custom flow orders, using a direction, priority function, and a stopping condition. The user specifies these arguments and the transformer using the `Flow` construct, as demonstrated in Fig. 1, Line 20, for the DeepPoly certifier. This code assigns higher priority to lower-layer neurons, resulting in a BFS traversal. The stopping function is set to `false`, stopping only when reaching the output layer. We verify the soundness of all specified transformers in the `Transformer` construct. Based on the DNN operation, `Flow` applies the corresponding transformer, ensuring a composition of only sound transformers.

3.2 ProveSound: Automated Bounded Verification of the DNN Certifier

To establish the soundness of a certifier, it is necessary to verify the soundness of each abstract transformer $f^\sharp$ w.r.t. its concrete counterpart f, i.e.,

$$\forall a \in \mathcal{A} \cdot \forall c \in C \cdot c \subseteq \gamma(a) \implies f(c) \subseteq \gamma(f^\sharp(a)) \tag{1}$$

Equation 1 is universally quantified over both the abstract element a and the concrete element c. The abstract element, a tuple of abstract shapes, over-approximates the values of neurons in the DNN, while the concrete element represents specific valuations for the neurons. Since the DNN architecture—its topology, number of neurons, and consequently the number of abstract shapes—can vary, the universal quantification in equation 1 presents a challenge for verification.

So, we introduce the concept of a *Symbolic DNN* to represent an arbitrary DNN and the corresponding abstract shapes symbolically. The symbolic DNN is an abstract neural network representing all subgraphs of any arbitrary DNN on which the specified transformer can be applied. It consists of symbolic values representing only the necessary neurons for executing the transformer specification. So, verifying the soundness of the specified transformer on a finite symbolic DNN is sufficient to prove its soundness on an arbitrarily large DNN with any topology.

The symbolic DNN is initialized only with `curr` and `prev`, along with their abstract shapes so the specified abstract transformer can be symbolically executed. However, in some cases, the symbolic execution of a transformer requires more neurons to be initialized in the symbolic DNN. We do so by a *Symbolic DNN Expansion*, where we statically analyze the transformer and only introduce neurons and their abstract shapes necessary for the symbolic execution. We explain these steps using an example in § 3.2.1, § 3.2.2. After the creation and expansion steps, we have a symbolic representation of the DNN and corresponding abstract shapes sufficient for symbolic execution to generate the final verification query which can be off-loaded to an off-the-shelf SMT solver (§ 3.2.3).

To better illustrate these steps, we introduce a new DeepPoly transformer for `ReLU` which has a better runtime than the original transformer but is slightly less precise. We then show the above-mentioned steps for the verification of the new transformer. As introduced in § 2, the DeepPoly abstract shape consists of 4 fields—l, u, L, U, where l, u are the concrete bounds and L, U are the polyhedral bounds of the neuron. Consider the DeepPoly `ReLU` transformer. It takes in as input the abstract shape of the `prev` neuron and computes the new abstract shape for `curr` neuron. It has 3 cases based on the values $\texttt{prev}[l], \texttt{prev}[u]$ of the input abstract shape - (i) $\texttt{prev}[l] \geq 0$, (ii) $\texttt{prev}[u] \leq 0$, and (iii) $\texttt{prev}[l] < 0 < \texttt{prev}[u]$. We focus only on the first case for illustration. In this case, the concrete bounds are set to the input concrete bounds, i.e., $\texttt{curr}[l] \leftarrow \texttt{prev}[l]$ and $\texttt{curr}[u] \leftarrow \texttt{prev}[u]$. Both the lower and upper polyhedral bounds are set to `prev`, i.e., $\texttt{curr}[L] \leftarrow \texttt{prev}$ and $\texttt{curr}[U] \leftarrow \texttt{prev}$. In the new transformer for `ReLU`, instead of setting the polyhedral bounds of `curr` in terms of the neurons of the previous layer, i.e., `prev`, we set them using the lower and upper polyhedral bounds of `prev`, which are $\texttt{prev}[L]$ and $\texttt{prev}[U]$ respectively. In ConstraintFlow, these polyhedral bounds can be computed using `map`(`replace_lower`) and `map`(`replace_upper`) respectively. The user-defined functions `replace_lower` and `replace_upper` replace a neuron with its lower or upper polyhedral bounds based on its coefficient. The `map` construct applies a function to all neurons in a polyhedral expression. So, the expression for the upper polyhedral bound (and similarly for lower) can thus be written as $e \equiv \texttt{prev}[U].\texttt{map}(\texttt{replace\_upper})$.

3.2.1 Symbolic DNN Creation. For each DNN operation η (e.g., `ReLU` in this case), given the abstract transformer, we create a symbolic DNN (Fig. 2a) with neurons representing `prev` and `curr` that are respectively the input and output of η. These neurons are associated with symbolic variables μ_p and μ_c representing their valuations respectively. The edges are only between `curr` and `prev` neurons representing the `ReLU` operation. Here, `prev` represents only a single neuron. However,

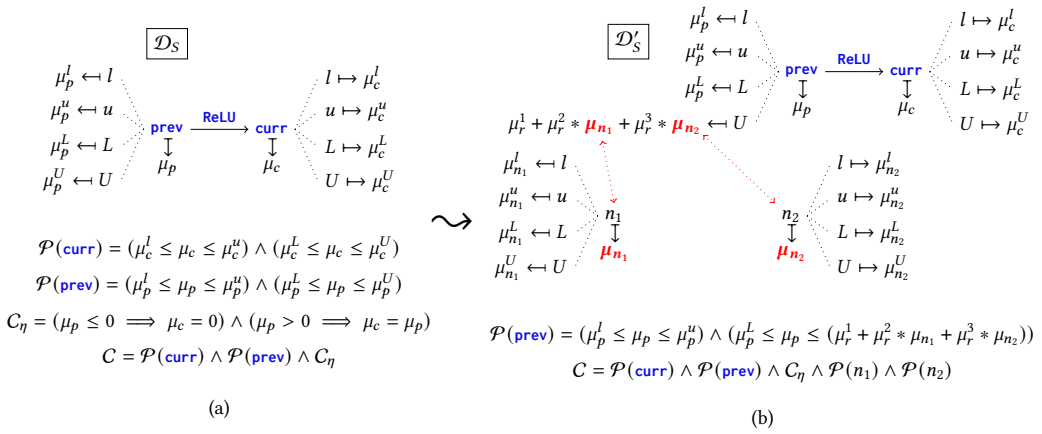


Fig. 2. Symbolic DNN creation and expansion for DeepPoly. $\mathcal{P}(n) \equiv (l \le n \le u) \wedge (L \le n \le U)$

for DNN operations like `Affine`, the symbolic DNN is initialized with $\texttt{prev}_1, \cdots \texttt{prev}_k$ where k is a sufficiently large parameter. We do not make any assumptions about the DNN's architecture, resulting in the absence of any extra neurons or edges between $\texttt{prev}_i$ and $\texttt{prev}_j$ and thus, no additional constraints over symbolic variables. Fig. 2a shows the symbolic DNN for the `ReLU` transformer for the DeepPoly certifier. The soundness property $\mathcal{P}$ for this certifier is that for each neuron n, $(l \le n \le u) \wedge (L \le n \le U)$. Each shape member and metadata associated with these neurons is also initialized with fresh symbolic variables. For instance, μ_p^l, μ_p^u represent the lower and upper concrete bounds respectively, and μ_p^L, μ_p^U are the lower and upper polyhedral bounds of `prev`. The symbolic DNN is associated with constraints representing the edge relations between the neurons and the soundness property assumptions before applying the transformer. In Fig. 2a, these constraints are presented as $C = \mathcal{P}(\texttt{curr}) \wedge \mathcal{P}(\texttt{prev}) \wedge C_\eta$, where $\mathcal{P}(\texttt{curr})$ and $\mathcal{P}(\texttt{prev})$ represent the soundness property over `curr` and `prev` respectively. C_η represents the semantics of the `ReLU` operation, i.e., $\texttt{curr} = 0$ when $\texttt{prev} < 0$, and $\texttt{curr} = \texttt{prev}$ otherwise. The formal definition and details of a symbolic DNN can be found in § 5.1.

3.2.2 Symbolic DNN Expansion. Initially, polyhedral bounds such as $\texttt{prev}[L]$ and $\texttt{prev}[U]$ are represented as single symbolic variables. However, for operations like `map`, the polyhedral values need to be expanded into expressions of the form $x_0 + x_1 \cdot n_1 + x_2 \cdot n_2 \ldots$, where x_i are coefficients and n_i are neurons. This is necessary for the semantics of `map`, as functions like `replace_upper` are applied to each constituent neuron and coefficient within the polyhedral expression. For example, consider $e \equiv \texttt{prev}[U].\texttt{map}(\texttt{replace\_upper})$. Initially, $\texttt{prev}[U]$ is a single symbolic variable μ_p^U (Fig. 2a), but to symbolically evaluate e, the expression must be expanded into its constituent terms, e.g., $\mu_r^1 + \mu_r^2 \cdot \mu_{n_1} + \mu_r^3 \cdot \mu_{n_2}$, where μ_r^1, μ_r^2, and μ_r^3 are symbolic coefficients, and μ_{n_1}, μ_{n_2} represent new neurons. In this case, the expansion introduces two neurons, but in general, the number of neurons n_{sym} is a sufficiently large parameter. No architectural assumptions are made about the new neurons, but they must be added to the symbolic DNN along with their metadata, and the soundness property $\mathcal{P}$ must be assumed for them. Fig. 2b shows the updated symbolic DNN after one expansion step. Similarly, before executing the expression for the polyhedral lower bound $e \equiv \texttt{prev}[L].\texttt{map}(\texttt{replace\_lower})$, μ_p^L must also be expanded. This expansion is performed through static analysis of the transformer. Once the symbolic DNN is expanded, the associated constraints C are updated to reflect the new neurons and the expanded values. Detailed steps for Symbolic DNN Expansion are in § 5.2.

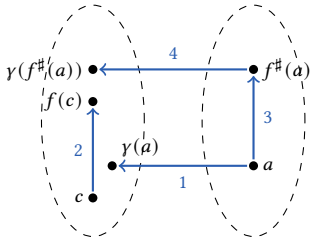


Fig. 3. Soundness of $f^\sharp$ w.r.t. f

$$
\begin{aligned}
& c \subseteq \gamma(a) \overset{?}{\implies} f(c) \subseteq \gamma(f^\sharp(a)) \\
\equiv\ & c \subseteq \gamma(a) \overset{?}{\implies} \Big((\cdot\cdot, p, c, \cdot\cdot) \in f(c) \implies (\cdot\cdot, p, c, \cdot\cdot) \in \gamma(f^\sharp(a))\Big) \\
\equiv\ & \Big(c \subseteq \gamma(a) \wedge (\cdot\cdot, p, c, \cdot\cdot) \in f(c)\Big) \overset{?}{\implies} \Big((\cdot\cdot, p, c, \cdot\cdot) \in \gamma(f^\sharp(a))\Big) \\
\equiv\ & \Big(\mathcal{P}(s_c, c) \wedge \mathcal{P}(s_p, p) \wedge c = f(p)\Big) \overset{?}{\implies} \Big(a' = f^\sharp(a) \implies (\mathcal{P}(s'_c, c))\Big) \\
\equiv\ & \Big(\varphi_0 \wedge \varphi_1 \wedge \varphi_2\Big) \overset{?}{\implies} \Big(\varphi_3 \implies \varphi_4\Big) \\
\equiv\ & \Big(\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3\Big) \overset{?}{\implies} \varphi_4
\end{aligned}
$$

Fig. 4. SMT query for Soundness of $f^\sharp$ w.r.t. f

Table 1. Generating SMT query for verifying one case of the ReLU transformer for DeepPoly certifier.

Steps in Fig. 3	**DeepPoly Translation for ReLU Operation**
Let $a = (\cdots, s_{n_1}, s_{n_2}, s_p, s_c, \cdots)$	Declare fresh symbolic variables for all neurons, metadata, and shape fields in the expanded symbolic DNN
(1) Let $(\cdots, n_1, n_2, p, c, \cdots) = \gamma(a), c \subseteq \gamma(a)$	$\varphi_1 \equiv \mathcal{P}(s_{n_1}, n_1) \wedge \mathcal{P}(s_{n_2}, n_2) \wedge \mathcal{P}(s_p, p) \wedge \mathcal{P}(s_c, c)$
(2) Apply f to c	$\varphi_2 \equiv c = f_r(p)$
(3) Let $a' = f^\sharp(a)$	Declare new symbolic variables for output: $\varphi_3 \equiv (a' == (\cdots, s_{n_1}, s_{n_2}, s_p, s'_c, \cdots))$
(4) Apply γ to a'	$\varphi_4 \equiv \mathcal{P}(s'_c, c)$

3.2.3 Generating the Verification Query. Once the symbolic DNN is expanded, we can translate the soundness check of a DNN certifier (Formula 1) into a closed-form SMT query. In the case of ReLU, the symbolic DNN corresponds to an abstract element a, a tuple of abstract shapes $a = (\cdots, s_{n_1}, s_{n_2}, s_p, s_c, \cdots)$, where s_{n_1}, s_{n_2}, s_p, and s_c represent the abstract shapes of n_1, n_2, prev, and curr, respectively. As shown in Fig. 3, the verification process consists of two steps (1, 2) to compute $f(c)$, and two steps (3, 4) to compute $\gamma(f^\sharp(a))$, starting from a. Table 1 outlines the computations for each step, with an example for the first case of the DeepPoly ReLU transformer ($\varphi_0 \equiv \texttt{prev}[l] \geq 0$).

1. $c \subseteq \gamma(a)$, representing the set of neuron value tuples satisfying $\mathcal{P}$. This is denoted by φ_1.
2. Applying f to prev to compute curr. Any $v \in f(c)$, with $v = (\cdots, p, c, \cdots)$, must satisfy $\varphi_2 \equiv c = f(p)$, where, in the case of ReLU, f_r is defined as $f_r(p) = \max(p, 0)$.
3. Applying $f^\sharp$ to a, updating only the abstract shape of curr: $a' = (\cdots, s_{n_1}, s_{n_2}, s_p, s'_c, \cdots)$. The new shape fields l, u, L, and U are computed symbolically. For example, $\texttt{curr}[U]$ is set to $\texttt{prev}[U].\texttt{map}(\texttt{replace\_upper})$.We start this computation by computing $\texttt{prev}[U]$ as $\mu_r^1 + \mu_r^2 * \mu_{n_1} + \mu_r^3 * \mu_{n_2}$. Then we apply replace_upper to each constituent summands to compute the final value as $\mu_r^1 + If(\mu_r^2 \geq 0, \mu_r^2 * \mu_{n_1}^U, \mu_r^2 * \mu_{n_1}^L) + If(\mu_r^3 \geq 0, \mu_r^3 * \mu_{n_2}^U, \mu_r^3 * \mu_{n_2}^L)$. Here, $If(c, l, r)$ is a Z3 construct. Similarly, the lower polyhedral bound is also computed.
4. Applying γ to a' results in $\varphi_4 \equiv \mathcal{P}(s'_c, c)$.

The verification reduces to checking if $(\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3) \implies \varphi_4$, as illustrated in Fig. 4. More details on the symbolic semantics and the steps to generate the final query can be found in § 5.3, 5.4.

3.2.4 Soundness and Completeness of ProveSound. The target of the verification procedure is to ensure that if using the operational semantics of ConstraintFlow, the abstract transformer is

$$
\begin{array}{lll}
\langle \textit{Expression} \rangle & e & ::= c \mid x \mid \texttt{sym} \mid e_1 \ \oplus \ e_2 \mid e[x] \mid f_c(e_1, \cdots) \mid x.\texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\} \mid \\
 & & \quad e.\texttt{map}(f_c) \mid \texttt{solver}(\texttt{minimize}, e_1, e_2) \mid \cdots \\
\langle \textit{Shape-decl} \rangle & d & ::= \texttt{Def shape as } (t_1\ x_1, t_2\ x_2, \cdots)\{e\} \\
\langle \textit{Function-def} \rangle & f & ::= \texttt{Func } x(t_1\ x_1, t_2\ x_2, \cdots) = e \\
\langle \textit{DNN-operation} \rangle & \eta & ::= \texttt{Affine} \mid \texttt{ReLU} \mid \texttt{MaxPool} \mid \texttt{DotProduct} \mid \texttt{Sigmoid} \mid \texttt{Tanh} \mid \cdots \\
\langle \textit{Transformer-decl} \rangle & \theta_d & ::= \texttt{Transformer } x \\
\langle \textit{Transformer-ret} \rangle & \theta_r & ::= (e_1, e_2, \cdots) \mid (e\ ?\ \theta_{r_1} : \theta_{r_2}) \\
\langle \textit{Transformer} \rangle & \theta & ::= \theta_d\ \{\eta_1 \rightarrow \theta_{r_1}; \eta_2 \rightarrow \theta_{r_2}; \cdots\} \\
\langle \textit{Statement} \rangle & s & ::= \texttt{Flow}(\delta, f_{c_1}, f_{c_2}, \theta_c) \mid f \mid \theta \mid s_1\ ;\ s_2 \\
\langle \textit{Program} \rangle & \Pi & ::= d\ ;\ s
\end{array}
$$

Fig. 5. A part of the BNF grammar for ConstraintFlow. The complete grammar can be found in Appendix A

applied to any concrete DNN along with its abstract element that satisfies the specified property, the updated abstract element still maintains the over-approximation-based soundness. For this, ProveSound creates a symbolic DNN and executes the specified transformer using symbolic semantics to generate an SMT query. We prove that verifying the transformer using symbolic semantics over a symbolic DNN ensures the verification using operational semantics over any concrete DNN. We explain this in detail in § 5.5.

Soundness. We introduce the notion of a symbolic DNN over-approximating a concrete DNN and symbolic semantics over-approximating the operational semantics. As a result, we use a bisimulation argument to prove that if the transformer is verified for a symbolic DNN, then it is also verified for all concrete DNNs that the symbolic DNN over-approximates.

Completeness. Symbolic execution is not complete for `traverse` because it involves loops with input-dependent termination conditions. So, to verify programs using `traverse`, we check the correctness and subsequently use the *inductive invariant* provided by the programmer. We also provide a construct `solver` in ConstraintFlow that can be used for calls to external solvers. For example, finding the minimum value of an expression e_1 under some constraints e_2 can be encoded as $\texttt{solver}(\texttt{minimize}, e_1, e_2)$. Since we do not have access to the solver, instead of symbolically executing it, we use *function contracts* to represent the output, i.e., a fresh variable x is declared that represents the output. Under the conditions e_2, the output x must be less than e_1, i.e., $e_2 \implies x \leq e_1$. Due to the invariants and contracts not being the strongest, the verification is not complete. However, it is complete for programs that do not use these constructs.

4 Formalising ConstraintFlow

We formally develop the syntax, type-system, and operational semantics of ConstraintFlow.

4.1 Syntax

4.1.1 Statements. In ConstraintFlow, a program Π starts with the shape declaration (d) and is followed by a sequence of statements (s), i.e., $\Pi ::= d\ ;\ s$. As shown in Fig. 5, statements include function definitions (f) - specified using `Func` construct, transformer definitions (θ) - specified using `Transformer` construct, the flow of constraints - specified using `Flow` construct, and sequence of statements separated by ;. The output of a function is an expression e, while the output of a transformer (θ_r) is either a tuple of expressions $t \equiv (e_1, \cdots)$, where e_i represents the output of each member of the abstract shape, or $(e\ ?\ \theta_{r_1} : \theta_{r_2})$, where _?_ : _ is the ternary operator.

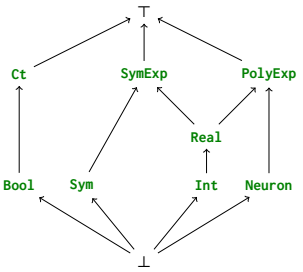


Fig. 6. Subtyping Lattice

$$\frac{\cdot \vdash \Pi : \Gamma, \tau_s \qquad \cdot \vdash d : \tau_s \qquad \Gamma, \tau_s \vdash s : \Gamma'}{\cdot \vdash d\ ;\ s\ :\ \Gamma, \tau_s}\ \textsc{T-program}$$

$$\frac{\tau_s = [x_1 \mapsto t_1, \cdots x_n \mapsto t_n] \qquad \forall i \in [n], \bot \sqsubset t_i \sqsubset \top}{\cdot \vdash \texttt{Def shape as}\ (t_1\ x_1, \cdots, t_n\ x_n) : \tau_s}\ \textsc{T-shape}$$

$$\frac{\begin{array}{c} x \notin \Gamma \quad \Gamma' = \Gamma[\texttt{curr} \mapsto \texttt{Neuron})][\texttt{prev} \mapsto \overline{\texttt{Neuron}}] \\ \forall i \in [m], \Gamma', \tau_s \vdash \theta_{r_i} : (t_i^1, \cdots, t_i^n) \\ \forall j \in [n], t^j = \sqcup_{i \in [m]}(t_i^j) \quad \forall j \in [n], t^j \sqsubseteq \tau_s^j \end{array}}{\begin{array}{l} \Gamma, \tau_s \vdash \texttt{Transformer}\ x = \{\eta_1 : \theta_{r_1}, \cdots\} : \\ \qquad \Gamma[x \mapsto (\texttt{Neuron} \times \overline{\texttt{Neuron}}) \rightarrow (t^1, \cdots)] \end{array}}\ \textsc{T-transformer}$$

Fig. 7. Type-checking Rules ($\mathcal{T}$)

4.1.2 Expressions. As shown in Fig. 5, apart from constants (c) and variables (x), `sym` is also an expression, which can be used to declare a new symbolic variable ϵ. For every symbolic variable, we implicitly add the constraint most commonly used in DNN certifiers, i.e., $-1 \leq \epsilon \leq 1$. We allow the standard binary operators, list operators, function calls, etc. Some operators like '+' are overloaded to also apply to polyhedral and symbolic expressions. Each neuron is associated with its abstract shape and metadata, which can be accessed by square bracket notation, for instance - `curr`$[l]$. The `map` construct takes in a function name and an expression of type `PolyExp` (or `SymExp`). The function is applied to all the constituent neurons (or symbolic variables) and adds the results to give a new polyhedral (or symbolic) expression. `traverse` is applied to a variable (x) representing a polyhedral expression, and takes in the direction of traversal (δ), a priority function (f_{c_1}), a stopping function (f_{c_2}), a replacement function (f_{c_3}), and a user-defined invariant (e), needed for verification. We also provide the `solver` construct in ProveSound, which allows calls to external solvers. For example, minimizing an expression e_1 under constraints e_2 can be expressed as `solver`(`minimize`, e_1, e_2).

4.1.3 Specifying Constraints. To verify a DNN certifier, one must provide the soundness property ($\mathcal{P}$) along the abstract shape. Also, for `traverse`, the programmer must provide an invariant. To define constraints in ConstraintFlow, the operators ==, $\leq$, $\geq$ are overloaded and can be used to compare polyhedral expressions as well as ConstraintFlow symbolic expressions. For example, the constraint $n_1 + n_2 \leq n_3$ means that for all possible values of n_1, n_2, and n_3 during concrete execution, the constraint must be true. Further, the construct <> can be used to define constraints such as e_1 <> e_2, where e_1 is a polyhedral expression, and e_2 is a symbolic expression. Mathematically, the constraint $n_1 + n_2$ <> $\texttt{sym}_1 + 2\ \texttt{sym}_2$ means $\forall n_1, n_2 \cdot \exists\ \texttt{sym}_1, \texttt{sym}_2 \in [-1, 1], s.t., n_1 + n_2 = \texttt{sym}_1 + 2\ \texttt{sym}_2$. In ConstraintFlow, the constraints are expressions of type `Ct`. The binary operators like $\wedge, \vee$ are also overloaded. For example, if e_1 and e_2 are of the type `Ct`, then $e_1 \wedge e_2$ is a constraint of type `Ct`.

4.2 Type Checking

We define a subtyping relation $\sqsubset$ for the basic types in ConstraintFlow, organized as a lattice (Fig. 6). An expression is type-checked to ensure that it has a type other than $\top$ or $\bot$. Type-checking involves recording the types of the members of the abstract shape in a record τ_s (referred to as T-shape in Fig. 7). A static environment Γ maps program identifiers to their respective types, and the tuple (Γ, τ_s) forms the typing context in ConstraintFlow (T-program). We utilize standard function types of the form $t_1 \times \cdots \times t_n \rightarrow t$, where t_i are the argument types and t is the return type. The `Transformer` construct encapsulates the abstract transformers associated with

$$\boxed{\langle \Pi, \mathcal{D}_C\rangle \Downarrow \mathcal{D}'_C \qquad \langle s, F, \Theta, \mathcal{D}_C\rangle \Downarrow F', \Theta', \mathcal{D}'_C \qquad \langle e, F, \rho, \mathcal{D}_C\rangle \Downarrow v}$$

OP-MAP

$$\frac{\langle e, F, \rho, \mathcal{D}_C\rangle \Downarrow c_0 + \sum_{i=0}^{i=l} c_i \cdot v_i \qquad \forall i \in [l],\ \langle f_c(v_i, c_i), F, \rho, \mathcal{D}_C\rangle \Downarrow v_i}{\langle e.\texttt{map}(f_c), F, \rho, \mathcal{D}_C\rangle \Downarrow c_0 + \sum_{i=0}^{i=l} v_i}$$

OP-TRAVERSE-2

$$\frac{\begin{array}{c}\mathsf{V}' = \mathsf{P}(\mathsf{V}, f_{c_1}, F, \rho, \mathcal{D}_C) \qquad v = c + v_{\mathsf{V}'} + v_{\overline{\mathsf{V}'}} \\ \langle v_{\mathsf{V}'}.\texttt{map}(f_{c_3}), F, \rho, \mathcal{D}_C\rangle \Downarrow v' \\ v'' = c + v' + v_{\overline{\mathsf{V}'}} \\ \mathsf{V}'' = \mathsf{Ft}((\mathsf{V} \setminus \mathsf{V}') \cup \mathsf{N}(\mathsf{V}', \delta), f_{c_2}, F, \rho, \mathcal{D}_C) \\ \langle v''.\texttt{traverse}, F, \rho, \mathcal{D}_C, \mathsf{V}''\rangle \Downarrow v'''\end{array}}{\langle v.\texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3}), F, \rho, \mathcal{D}_C, \mathsf{V}\rangle \Downarrow v'''}$$

Fig. 8. Big-step Operational Semantics ($\mathcal{OP}$) of ConstraintFlow

each DNN operation. In rule T-TRANSFORMER, the output of an abstract transformer θ_r is a tuple of expressions that undergo recursive type-checking to ensure consistency with τ_s. The implicit inputs to `Transformer` are `curr` and `prev`. For n members in the user-defined abstract shape and m DNN operations, the corresponding abstract transformers yield tuples of types $(t_i^1, \cdots, t_i^n)$. For each abstract shape element, we define the type $t^j = \sqcup_{i\in[m]} t_i^j$. The transformer type checks if $j \in [n]$ and $t_i^j \sqsubseteq \tau_s^j$, where τ_s^j is the type of the j-th shape member. The type of `curr` is `Neuron`, while the type of `prev` depends on the DNN operation; for simplicity, we assume `prev` is of type $\overline{\texttt{Neuron}}$. If all abstract transformers in the `Transformer` construct pass type-checking, a new binding is created in Γ mapping the transformer name to the type $\texttt{Neuron} \times \overline{\texttt{Neuron}} \rightarrow (t^1, \cdots, t^m)$. The detailed description of type-checking in ConstraintFlow can be found in Appendix B.

4.3 Operational Semantics

The input concrete DNN is represented as a record $\mathcal{D}_C$ that maps the metadata and abstract shape members of all neurons to their respective values. While executing statements in ConstraintFlow, two stores are maintained: (i) F, which maps function names to their arguments and return expressions, and (ii) Θ, which maps transformer names to their definitions. The general form for the operational semantics of statements in ConstraintFlow is given by: $\langle s, F, \Theta, \mathcal{D}_C\rangle \Downarrow F', \Theta', \mathcal{D}'_C$. Function definitions add entries to F, while transformer definitions add entries to Θ. The `Flow` construct applies transformer θ_c to the neurons in the DNN $\mathcal{D}_C$, modifying it to $\mathcal{D}'_C$.

Each expression in ConstraintFlow evaluates to a value (v), with the formal definition of values provided in Appendix C. A record ρ maps variables in ConstraintFlow to concrete values. The general form for the operational semantics of expressions in ConstraintFlow is: $\langle e, F, \rho, \mathcal{D}_C\rangle \Downarrow v$, with most operations, including unary and binary, following their natural operational semantics.

The operational semantics of `map` (OP-MAP in Fig. 8) begins by recursively evaluating the input expression e, yielding a polyhedral or symbolic expression denoted as v_b. The input function f is then applied to each component of v_b, resulting in individual outputs v_i that are summed to produce the final output. For `traverse`, the input expression e is first evaluated to yield a polyhedral value v. Then, an active vertex set V is established by retrieving constituent neurons from v and filtering out neurons that satisfy the stopping condition f_{c_2}, i.e., $\mathsf{V} \leftarrow \mathsf{Ft}(\text{neurons}(v), f_{c_2}, F, \rho, \mathcal{D}_C)$. This set initializes V and is iterated upon until it is empty. In each iteration, shown in OP-TRAVERSE-2 (Fig. 8), the priority function f_{c_1} is applied to each neuron in V, selecting the highest-priority neurons: $\mathsf{V}' \leftarrow \mathsf{P}(\mathsf{V}, f_{c_1}, F, \rho, \mathcal{D}_C)$. The value v can be decomposed into three parts: a constant c, the value associated with neurons in V', and the value for neurons not in V': $v = c + v_{\mathsf{V}'} + v_{\overline{\mathsf{V}'}}$. The replacement function f_{c_3} is applied only to $v_{\mathsf{V}'}$, retaining the coefficients of the other neurons, resulting in a new polyhedral value: $v'' = c + v' + v_{\overline{\mathsf{V}'}}$. The active set is updated by removing

neurons from V′ and adding their neighbors, filtered again to satisfy the stopping condition: $V'' = \mathsf{Ft}((V \setminus V') \cup \mathsf{N}(V', \delta), f_{c_2}, F, \rho, \mathcal{D}_C)$. This process continues until the final value is computed. More detailed operational semantics for `traverse` and other constructs can be found in Appendix D.

4.4 Type Soundness

We demonstrate that if a program type-checks according to the rules of ConstraintFlow, then applying the program according to operational semantics produces an updated abstract element for the input neural network (Theorem 4.1). Lemmas 4.1 and 4.2 establish that if an expression or statement type-checks, it will evaluate according to operational semantics, with the output type consistent with the type computed during type-checking. Detailed proofs are in Appendix E.

Lemma 4.1. *Given* (Γ, τ_s) *and* $(F, \rho, \mathcal{D}_C)$ *with finite* $\mathcal{D}_C$ *such that* $(F, \rho, \mathcal{D}_C)$ *is consistent with* (Γ, τ_s)*, if* $\Gamma, \tau_s \vdash e : t$ *and* $\bot \sqsubset t \sqsubset \top$*, then* $\langle e, F, \rho, \mathcal{D}_C\rangle \Downarrow v$ *and* $\vdash v : t'$ *s.t.* $t' \sqsubseteq t$.

Lemma 4.2. *Given* (Γ, τ_s) *and* $(F, \rho, \mathcal{D}_C)$ *with finite* $\mathcal{D}_C$ *such that* $(F, \rho, \mathcal{D}_C)$ *is consistent with* (Γ, τ_s)*, if* $\Gamma, \tau_s \vdash s : \Gamma'$*, then* $\langle s, F, \rho, \mathcal{D}_C\rangle \Downarrow F', \rho', \mathcal{D}'_C$ *s.t.* $(F', \rho', \mathcal{D}'_C)$ *is consistent with* (Γ', τ_s).

Theorem 4.1. *A well-typed program in ConstraintFlow successfully terminates according to the operational semantics, i.e.,* $\mathcal{T} \models \mathcal{OP}$*. Formally, if* $\cdot \vdash \Pi : \Gamma, \tau_s$ *then* $\langle \Pi, \mathcal{D}_C\rangle \Downarrow \mathcal{D}'_C$

Proof sketch. Theorem 4.1 follows directly from Lemmas 4.1 and 4.2. The lemmas are proved by induction on the structures of e and s. For Lemma 4.1, the case where $e \equiv x \cdot \texttt{traverse}(\delta, f_1, f_2, f_3)\{\_\}$ is particularly intricate as it involves traversing the DNN. We demonstrate this by constructing a bit vector B representing the neurons in the DNN, ordered topologically (as a DAG), where 1 indicates the presence in the active set and 0 indicates absence. We show that the value of B is bounded and decreases by at least 1 in each iteration. □

5 ProveSound—Bounded Automatic Verification

We present bounded automated verification for the soundness verification of every abstract transformer specified for a DNN certifier. Bounds are assumed on the maximum number of neurons in the previous layer (n_{prev}), and the maximum number of ProveSound symbolic variables used by the certifier (n_{sym}). We reduce this verification task to a first-order logic query which can be handled with an off-the-shelf SMT solver. In this section, the terms *symbolic variables* and *constraints* refer to SMT symbolic variables and constraints over them, not the ProveSound symbolic variables ϵ or constraints unless stated otherwise. When executing the certifier using operational semantics, the input is a concrete DNN. So, the soundness of the certifier must be verified for all possible inputs, i.e., all possible DNNs. Our key insight is a *Symbolic DNN* that can represent arbitrary concrete DNNs within the above-stated bounds. In a nutshell, given a ProveSound program, we perform the following steps: (i) create a symbolic DNN (§ 5.1), (ii) expand the symbolic DNN to be able to execute the program (§ 5.2), (iii) execute the program on the symbolic DNN using symbolic semantics (§ 5.3), (iv) generate the verification query and verify the query using an off-the-shelf SMT solver (§ 5.4). We prove the *soundness* of the symbolic semantics w.r.t. the operational semantics (§ 5.5). So, verifying the soundness of a certifier for a symbolic DNN ensures the soundness of any concrete DNN within the bounds.

5.1 Symbolic DNN Creation

We introduce the concept of a *Symbolic DNN* to represent an arbitrary DNN and the corresponding abstract shapes symbolically. It represents all subgraphs of any arbitrary neural network on which the specified transformer can be applied. So, it consists of symbolic values representing neurons necessary for executing the transformer.

$$\textsc{E-shape-b}\quad \frac{\langle e, F, \sigma, \mathcal{D}_s, C\rangle \downarrow n, \_ \quad \mathsf{expandN}(n, x, \tau_s, \mathcal{D}_S, C, \mathcal{P}) = \mathcal{D}'_S, C'}{\mathsf{expand}(e[x], \tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P}) = \mathcal{D}'_S, C'}$$

$$\textsc{G-map}\quad \frac{\begin{array}{c}\tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P} \models e \rightsquigarrow \mathcal{D}'_S, C' \\ \mathsf{expand}(e, \tau_s, F, \sigma, \mathcal{D}'_S, C', \mathcal{P}) = \mathcal{D}_{S_0}, C_0 \\ \langle e, F, \sigma, \mathcal{D}_{S_0}, C_0\rangle \downarrow \mu_{b_0} + \sum_{i=1}^{j} n_i * \mu_{b_i} \\ \forall i \in [j] \quad \tau_s, F, \sigma_i, \mathcal{D}_{S_{i-1}}, C_{i-1} \models f_c(n_i, \mu_{b_i}) \rightsquigarrow \mathcal{D}_{S_i}, C_i\end{array}}{\tau_S, F, \sigma, \mathcal{D}_S, C, \mathcal{P} \models e \cdot \texttt{map}(f_c) \rightsquigarrow \mathcal{D}_{S_j}, C_{S_j}}$$

$$\textsc{Expand-poly-r}\quad \frac{\begin{array}{c}\tau_s(x) = \texttt{PolyExp} \qquad \mathcal{D}_S[n[x]] = \mu_{b_r} \qquad \mathcal{N} = [n'_1, \cdots n'_j] \\ \mathcal{D}_{S_0} = \mathcal{D}_S \qquad \forall i \in [j], \mathcal{D}_{S_i}, C_i = \mathsf{add}(n'_i, \tau_s, \mathcal{D}_{S_{i-1}}, \mathcal{P}, C_{i-1}) \\ \mu_b = \mu_{r_0} + \sum_{i=1}^{j} \mu_{r_i} * n'_i \qquad \mathcal{D}'_S = \mathcal{D}_{S_j}[n[x] \mapsto \mu_b]\end{array}}{\mathsf{expandN}(n, x, \tau_s, \mathcal{D}_S, C_0, \mathcal{P}) = \mathcal{D}'_S, C_j}$$

$$\textsc{G-traverse}\quad \frac{\begin{array}{c}\mathcal{N} = [n_1, \cdots n_j] \\ \tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P} \models e \rightsquigarrow \mathcal{D}_{S_0}, C_0 \qquad \forall i \in [j], \mathcal{D}_{S_i}, C_i = \mathsf{add}(n_i, \tau_s, \mathcal{D}_{S_{i-1}}, \mathcal{P}, C_{i-1}) \\ \mu_b = \mu_{b_0} + \sum_{i=1}^{j} \mu_{b_i} * n_i \quad \mu_b, \mu_{b_0}, \mu_{b_i} \text{ are fresh symbolic variables} \\ \mathcal{D}'_{S_0} = \mathcal{D}_{S_j} \qquad C'_0 = C_j \qquad \forall i \in [j], \tau_s, F, \sigma, \mathcal{D}'_{S_{i-1}}, C'_{i-1}, \mathcal{P} \models f_{c_2}(n_i, \mu_{b_i}) \rightsquigarrow \mathcal{D}'_{S_i}, C'_i \\ \mathcal{D}''_{S_0} = \mathcal{D}'_{S_j} \qquad C''_0 = C'_j \qquad \forall i \in [j], \tau_s, F, \sigma, \mathcal{D}''_{S_{i-1}}, C''_{i-1}, \mathcal{P} \models f_{c_3}(n_i, \mu_{b_i}) \rightsquigarrow \mathcal{D}''_{S_i}, C''_i\end{array}}{\tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P} \models x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\} \rightsquigarrow \mathcal{D}''_{S_j}, C''_j}$$

Fig. 9. Symbolic DNN Expansion

Definition 5.1. *A symbolic DNN is a graph $\langle V, E, \mathcal{D}_S, C\rangle$, where V is the set of neurons and E is the set of edges representing the DNN operations (e.g., `Affine`, `ReLU`). Each node is associated with an abstract shape and metadata. $\mathcal{D}_S$ is a record that maps each neuron, its shape members, and metadata to symbolic variables and C represents constraints over the symbolic variables.*

As explained in § 3.2.1, 3.2.2, for each DNN operation η (e.g., `ReLU`), we initialize a symbolic DNN with neurons representing `prev` and `curr` that are respectively the input and output of η. The edges are only between `curr` and `prev` neurons and represent the operation η. C encodes η and the assumption of the user-specified property $\mathcal{P}$ over all of the neurons in the symbolic DNN. Each shape member and metadata associated with these neurons is set to symbolic variables in $\mathcal{D}_S$. In subsequent sections, we omit V and E and refer to $\mathcal{D}_S, C$ as a symbolic DNN. Next, to enable symbolic execution of the specified transformer, we may need to expand the symbolic DNN. For example, in the case of the expression $e \equiv \texttt{prev}[U].\texttt{map}(\texttt{foo})$, where `foo` is a user-defined function, $\texttt{prev}[U]$ must be expanded before we can apply `foo`. The symbolic DNN expansion step is written in the form $\tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P} \models e \rightsquigarrow \mathcal{D}'_S, C'$ (§ 5.2). After the symbolic DNN expansion step of an expression e, it can be symbolically executed using the symbolic semantics. The symbolic semantics are defined in the form $\langle e, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu, C'$ (§ 5.3).

5.2 Symbolic DNN Expansion

The expansion step is done by statically analyzing the transformer specification and expanding the symbolic DNN accordingly. A subset of the rules for symbolic DNN expansion is shown in Fig. 9. The complete set of rules can be found in Appendix G. This step analyzes the expression e for the presence of one of three constructs - `map`, function call, or `traverse`. The rules for `map` and `traverse`

are shown in rules G-map and G-traverse (in Fig. 9). In the rule G-map, the graph expansion is recursively applied to the input expression e in the first line. Then, since it is a `map` construct, it must be ensured that the output of e is in expanded form. This is done in the second line. The third line asserts that the output from the symbolic execution of e is already in the expanded form $\mu_{b_0} + \sum_{i=1}^{j} n_i * \mu_{b_i}$. Since the `map` construct applies the function call to all the individual summands of the output, the DNN expansion step is applied to each function call before symbolically executing it. This is shown in the fourth line of G-map rule.

Now, we explain the $\text{expand}(e, \tau_s, F, \sigma, \mathcal{D}_S, C, \mathcal{P})$ rules used to ensure that the output of symbolically executing e is in expanded form. Here, expand takes in an expression, e, τ_S, F, σ, $\mathcal{D}_S$, C, and the abstract shape constraint definition $\mathcal{P}$. The output of expand is $\mathcal{D}'_S$, which can contain new shape members and expanded versions of existing shape members, and C', which is extended to include the soundness property assumptions on any new neurons added to the symbolic DNN or the constraint $-1 \leq \epsilon \leq 1$ for any new ProveSound symbolic variables. In Fig. 9, we show one of the base cases of this step, Expand-poly-r, where we expand the accessed polyhedral shape member of the input neuron. In the first line, we symbolically execute e to get the neuron n. Then, if x is of the type `PolyExp` or `SymExp`, we add new symbolic variables to the symbolic DNN accordingly.

Another interesting case for graph expansion is the expressions $x.\texttt{traverse}(d, f_{c_1}, f_{c_2}, f_{c_3})$ shown in the rule G-traverse, where we recursively call the graph expansion for the invariant e in line 1. Since we cannot symbolically execute the `traverse` construct due to it being a loop with an undetermined number of iterations at the analysis time, we declare new neurons to represent the output. In line 2, these new neurons and their corresponding metadata are added to the symbolic DNN. So, the output of symbolically executing `traverse` is represented as $\mu_b = \mu_{b_0} + \sum_{i=1}^{j} \mu_{b_i} * n_i$ in line 3. When generating the query, we also need to assume that the stopping condition (f_{c_2}) is true on all summands of the final output, and also the function f_{c_3} is applied to all the summands. So, in lines 4-5, we recursively apply the symbolic DNN expansion on all the summands using f_{c_2} and f_{c_3}.

5.3 Symbolic Semantics

Like operational semantics, symbolic semantics ($\mathcal{S}$) use F which maps function names to their formal arguments and return expressions. However, instead of the concrete store ρ used in operational semantics, it uses a symbolic store, σ, which maps the identifiers to their symbolic values μ in expanded form. Symbolic semantics output a symbolic value μ, and also add additional constraints to C, i.e., $\langle e, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu, C'$. Constants, variables, and the introduction of new ProveSound symbolic variables using the ϵ construct are the base cases of the symbolic semantics of ProveSound. Unary, binary, and ternary operations are straightforward recursive cases. We show Sym-ternary in Fig. 10, where the three expressions e_1, e_2, and e_3 are recursively executed to output μ_1, μ_2, μ_3, respectively. The output value of the ternary operation is thus returned as $\text{If}(\mu_1, \mu_2, \mu_3)$, where If is a Z3 construct. Also, the constraints are accumulated in the recursive calls. The symbolic semantics for `map` construct are similar to the operational semantics and are therefore omitted here. We now discuss the semantics for the more challenging `traverse` construct. Detailed semantics for other constructs are available in Appendices F and G.

Due to the lack of DNN architecture information, full symbolic execution of the loop specified by the `traverse` construct is not feasible. So, we validate the user-provided invariant's soundness and subsequently use it for the symbolic semantics of `traverse`. In the rule Sym-traverse in Fig. 10, e is the user-defined invariant for the traversal, μ_b is the output symbolic polyhedral expression, and μ is the result of applying the invariant e to μ_b. We check the soundness of this invariant in two steps (Check-invariant in Fig. 10). First, we verify that the invariant is satisfied at the initial state. Here, μ represents the evaluated invariant expression e applied to the input state of

$$\textsc{Sym-ternary}\quad \frac{\langle e_1, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu_1, C_1 \quad \langle e_2, F, \sigma, \mathcal{D}_S, C_1\rangle \downarrow \mu_2, C_2 \quad \langle e_3, F, \sigma, \mathcal{D}_S, C_2\rangle \downarrow \mu_3, C_3}{\langle (e_1 ? e_2 : e_3), F, \sigma, \mathcal{D}_S, C\rangle \downarrow If(\mu_1, \mu_2, \mu_3), C_3}$$

$$\textsc{Check-induction}\quad \frac{\begin{array}{c}\mathcal{N} = [n'_1, \cdots, n'_j] \qquad \mu_b = \mu_0^{\text{real}} + \sum_{i=1}^{j} \mu_i^{\text{real}} * n'_i \qquad \sigma' = \sigma[x \mapsto \mu_b] \\ \langle e, F, \sigma', \mathcal{D}_S, C\rangle \downarrow \mu'_b, C_0 \qquad \forall i \in [j], \langle f_{c_2}(n_i, \mu_{r_i}), F, \sigma', \mathcal{D}_S, C_{i-1}\rangle \downarrow \mu'_i, C_i \\ C'_0 = C_j \qquad \forall i \in [j], \langle f_{c_3}(n_i, \mu_{r_i}), F, \sigma', \mathcal{D}_S, C'_{i-1}\rangle \downarrow \mu''_i, C'_i \\ \mu'' = \mu_{r_0} + \sum_{i=1}^{j} If(\mu'_i, \mu''_i, \mu_{r_i} * n_i) \qquad \sigma'' = \sigma[x \mapsto \mu''] \qquad \langle e, F, \sigma'', \mathcal{D}_S, C'_j\rangle \downarrow \mu''', C''\end{array}}{\mathsf{Ind}(x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\}, F, \sigma, \mathcal{D}_S, C) = \mathsf{unsat}(\neg(C_0 \wedge \mu'_b \implies C''_j \wedge \mu'''))}$$

$$\textsc{Check-invariant}\quad \frac{\begin{array}{c}\langle e, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu, C' \quad \mu_b = \mathsf{unsat}(\neg(C' \implies \mu)) \\ \mu'_b = \mathsf{Ind}(x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\}, F, \sigma, \mathcal{D}_S, C)\end{array}}{\mathsf{Inv}(x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\}, F, \sigma, \mathcal{D}_S, C) = \mu_b \wedge \mu'_b, C'}$$

$$\textsc{Sym-traverse}\quad \frac{\begin{array}{c}\mathsf{Inv}(x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\}, F, \sigma, \mathcal{D}_S, C) = \mathsf{true}, C' \\ \mu_b = \mu_0 + \sum_{i=1}^{j} \mu_i * \mu'_i \qquad \sigma' = \sigma[x \mapsto \mu_b] \qquad \langle e, F, \sigma', \mathcal{D}_S, C'\rangle \downarrow \mu, C''\end{array}}{\langle x \cdot \texttt{traverse}(\delta, f_{c_1}, f_{c_2}, f_{c_3})\{e\}, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu_b, \mu \wedge C''}$$

Fig. 10. Symbolic Semantics ($\mathcal{S}$) for ProveSound expressions: $\langle e, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu, C'$

`traverse`. $\mathsf{unsat}(\neg(C' \implies \mu))$ implies that μ is true under the conditions, C', which are valid before executing `traverse`. Second, we verify that the invariant is inductive (Check-induction). In Ind, $\mathsf{unsat}(\neg(C_0 \wedge \mu'_b \implies C''_j \wedge \mu'''))$ means that under the assumption that the invariant holds before an iteration of `traverse`, the invariant must hold after the iteration of `traverse`. If the invariant is validated, we create a symbolic value of the form $\mu_0 + \sum_{i=1}^{j} \mu_i * \mu'_i$ to represent the output of $x.\texttt{traverse}(d, f_{c_1}, f_{c_2}, f_{c_3})\{e\}$ and assume, in C, that the invariant holds on this output.

5.4 Queries for Verification

Initially, it is assumed that the property $\mathcal{P}$ holds for all the neurons in the symbolic DNN. To compute the new abstract shape, the user-specified abstract transformer is executed using the symbolic semantics as described in § 5.3. This results in the new abstract shape (for `curr`) - a tuple of symbolic values $(\mu_1, \cdots, \mu_n)$ and a condition, C' that encodes constraints over μ_i. To verify the soundness of the abstract transformer, we need to check that if the property $\mathcal{P}$ holds for all the neurons in the symbolic DNN ($\forall n \in \mathcal{D}_S, \mathcal{P}(\alpha_n, n)$), then it also holds for the new symbolic abstract shape values, $\mathcal{P}(\alpha'_{\texttt{curr}}, \texttt{curr})$, where $\alpha'_{\texttt{curr}} = (\mu_1, \cdots, \mu_n)$. We split the query into two parts: (i) antecedent p—encoding the initial constraints on the symbolic DNN, the computations of the new abstract shape for `curr`, represented by $\mathcal{R}$, the semantic relationship η between `curr` and `prev`, and any path conditions relevant to the specific computations we are verifying, C'. $p \triangleq (\forall n \in \mathcal{D}_S, \mathcal{P}(\alpha_n, n)) \wedge \texttt{curr} = \eta(\texttt{prev}) \wedge \mathcal{R} \wedge C'$ (ii) consequent q—encoding the property $\mathcal{P}$ applied to the new abstract shape of `curr`. $q \triangleq \mathcal{P}(\alpha'_{\texttt{curr}}, \texttt{curr})$. So, the final query is $\mathsf{checkValid}(p \implies q)$.

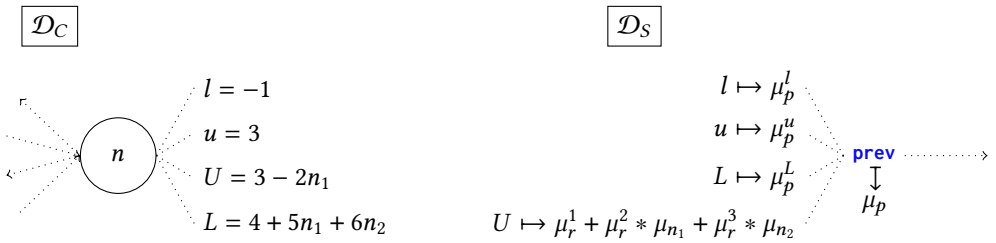


Fig. 11. Parts of Concrete DNN $\mathcal{D}_C$ and Symbolic DNN $\mathcal{D}_S$

5.5 Correctness of Verification Procedure

We define a notion of *over-approximation* of a concrete DNN by a symbolic DNN, a concrete value by a symbolic value, etc. So, any property proved by our verification algorithm for a symbolic DNN also holds for any concrete DNN that is over-approximated by the symbolic DNN. This notion lets us establish the correctness of the ProveSound verification procedure.

5.5.1 Over-Approximation. Fig. 11 shows parts of a concrete DNN $\mathcal{D}_C$ and a symbolic DNN $\mathcal{D}_S$ from Fig. 2b. The neuron `prev` in $\mathcal{D}_S$ over-approximates the neuron n in the concrete DNN $\mathcal{D}_C$ if φ is satisfiable, where $\varphi \equiv (\mu_p^l = -1) \wedge (\mu_p^u = 3) \wedge (\mu_p^L = 4+5n_1+6n_2) \wedge (\mu_r^1+\mu_r^2 * \mu_{n_1} + \mu_r^3 * \mu_{n_2} = 3-2n_1)$. Further, if $\mu_p, \mu_{n_1}, \mu_{n_2}$ represent n, n_1, n_2 respectively, they must also be equal, i.e., $\varphi_1 \equiv \varphi \wedge (\mu_p = n) \wedge (\mu_{n_1} = n_1) \wedge (\mu_{n_2} = n_2)$ must be satisfiable. Note that the neurons in $\mathcal{D}_C$ are not assigned any values and are therefore symbolic themselves. So, φ_1 must be satisfiable for all possible values of n, n_1, n_2 in $\mathcal{D}_C$. Further, the symbolic DNN has another component C which imposes constraints on μ_i. So, the formula must be satisfiable under the constraints C, i.e., φ_2 must be true.

$$\varphi_2 = \forall\{n, n_1, n_2\} \cdot \exists\{\mu_p, \mu_{n_1}, \mu_{n_2}, \mu_p^l, \cdots\} \cdot \big(\varphi \wedge (\mu_p = n) \wedge (\mu_{n_1} = n_1) \wedge (\mu_{n_2} = n_2) \wedge C\big) \quad (2)$$

In the symbolic DNN, C contains (i) the constraints encoded by the property $\mathcal{P}$ assumed on all the neurons in the symbolic DNN, and (ii) the edge relationship between `curr` and `prev`.

Definition 5.2. *A symbolic DNN $\mathcal{D}_S, C$ over-approximates a concrete DNN $\mathcal{D}_C$ if $\forall Y \cdot \exists W \cdot (C \bigwedge_{t \in \mathsf{dom}(\mathcal{D}_S)} \mathcal{D}_S(t) = \mathcal{D}_C(t))$, where Y is the set of neurons and ProveSound symbolic variables in $\mathcal{D}_C$ and W is the set of all SMT symbolic variables in $\mathcal{D}_S$.*

Further, in Equation 2, all the variables inside universal quantifier (n, n_1, n_2) are set equal to variables in the existential quantifier $\mu_p, \mu_{n_1}, \mu_{n_2}$. So, the equation can be rewritten by simply replacing the variables within the universal quantifier with corresponding variables in the existential quantifier, and removing the corresponding equality constraints, i.e., $\varphi_3 = \varphi_2$, where $\varphi_3 = \forall\{\mu_p, \mu_{n_1}, \mu_{n_2}\} \cdot \exists\{\mu_p^l, \mu_p^u, \mu_p^L, \mu_r^1, \mu_r^2, \mu_r^3\} \cdot (\varphi \wedge C)$.

In our example in Fig. 11, $Y = \{\mu_p, \mu_{n_1}, \mu_{n_2}\}$, and W is the set of all the other symbolic variables used in $\mathcal{D}_S$. So, a symbolic DNN $\mathcal{D}_S, C$ over-approximates a concrete DNN $\mathcal{D}_C$ if $\forall Y \cdot \exists W \cdot (C \wedge \bigwedge_{t \in \mathsf{dom}(\mathcal{D}_S)} (\mathcal{D}_S(t) = \mathcal{D}_C(t)))$. There are two types of symbolic variables in W—ones that represent constants during concrete execution and ones that represent polyhedral or symbolic expressions. So, we partition W into two sets, X and Z, where X contains the symbolic variables representing constants, while Z contains the other symbolic variables. So, we can then re-write φ_3 as $\varphi_4 = \forall Y \cdot \exists X \cdot \exists Z \cdot (C \wedge \bigwedge_{t \in \mathsf{dom}(\mathcal{D}_S)} (\mathcal{D}_S(t) = \mathcal{D}_C(t)))$. Note that in the example above, $X = \{\mu_p^l, \mu_p^u, \mu_r^1, \mu_r^2, \mu_r^3\}, Z = \{\mu_p^L\}$. From Equation 2, since $\mu_p^l, \mu_p^u, \mu_r^1, \mu_r^2, \mu_r^3$ are independent of n, n_1, n_2, we bring the set X out of the $\forall$ quantifier. Generalizing this notion, we use the definition Over-approx-DNN in Fig. 12. The over-approximation of a concrete DNN $\mathcal{D}_C$ by a symbolic DNN $\mathcal{D}_S, C$

$$\textsc{Over-approx-DNN}\quad \frac{\mathrm{dom}(\mathcal{D}_S) \subseteq \mathrm{dom}(\mathcal{D}_C) \qquad X = \mathsf{Constants}(\mathcal{D}_S, C) \qquad Y = \mathsf{Neurons}(\mathcal{D}_S, C) \cup \mathsf{SymbolicVars}(\mathcal{D}_S, C) \\ Z = \mathsf{PolyExps}(\mathcal{D}_S, C) \cup \mathsf{SymExps}(\mathcal{D}_S, C) \cup \mathsf{Constraints}(\mathcal{D}_S, C) \\ \exists X \cdot \forall Y \cdot \exists Z \cdot \left(C \wedge \bigwedge_{t \in \mathrm{dom}(\mathcal{D}_S)} \mathcal{D}_S(t) = \mathcal{D}_C(t)\right)}{\mathcal{D}_C \prec_C \mathcal{D}_S}$$

$$\textsc{Bisumation}\quad \frac{\langle e, F, \rho, \mathcal{D}_C\rangle \Downarrow v \qquad \langle e, F, \sigma, \mathcal{D}_S, C\rangle \downarrow \mu, C' \\ \exists! X \cdot \forall Y \cdot \exists! Z \cdot \left(CS(\mu, v) \wedge C' \wedge \mathcal{M} \wedge \bigwedge_{t \in \mathrm{dom}(\mathcal{D}_S)} \mathcal{D}_S(t) = \mathcal{D}_C(t)\right)}{\langle\!\langle e, F, \rho, \sigma, \mathcal{D}_C, \mathcal{D}_S, C\rangle\!\rangle \updownarrow v, \mu, C', \mathcal{M}}$$

Fig. 12. Definitions for Over-approximation and Bisimulation

is represented as $\mathcal{D}_C \prec_C \mathcal{D}_S$. The definitions for a symbolic value over-approximating a concrete value and a symbolic store over-approximating a concrete store can be found in (Appendix H).

Using these definitions of over-approximation, we prove two important properties. First, if a symbolic DNN over-approximates the concrete DNN, then expanding the symbolic DNN maintains the over-approximation. Second, we show that given a ProveSound expression that type-checks, if one starts with a symbolic DNN $\mathcal{D}_S, C$ and a concrete DNN $\mathcal{D}_C$ such that $\mathcal{D}_C \prec_C \mathcal{D}_S$, then the output of applying symbolic semantics on $\mathcal{D}_S, C$ over-approximated the output of applying operational semantics on $\mathcal{D}_C$. We prove this using bisimulation (rule Bisimulation in Fig. 12), where we simultaneously apply the operational semantics to the concrete DNN and the symbolic semantics to the symbolic DNN $\mathcal{D}_S, C$. The complete details can be found in Appendix I.

5.5.2 Soundness and Completeness. We show that if ProveSound concludes that the abstract transformers specified in the program are verified to maintain the user-defined property $\mathcal{P}$, then executing the program on any concrete DNN also maintains the property $\mathcal{P}$. We prove this by initially creating a symbolic DNN with only the neurons representing `curr` and `prev` and edges representing their corresponding DNN operation (η - for example `ReLU`). This over-approximates any part of an arbitrary concrete DNN (within the bounds of verification) which is the output of η. Next, the over-approximation is maintained during symbolic DNN expansion and executing symbolic semantics. Finally, the query is generated over symbolic values that overapproximate the corresponding concrete values. So, if the SMT solver concludes that the property $\mathcal{P}$ is maintained over the symbolic DNN, then we can conclude that $\mathcal{P}$ will also be maintained over all over-approximated concrete DNNs. Further, since the symbolic semantics are not exact only for `traverse` and `solver` constructs, ProveSound is complete, excluding these constructs.

Theorem 5.1 (Soundness). *For a well-typed program Π, if ProveSound verification procedure proves it maintains the property $\mathcal{P}$, then upon executing Π on all concrete DNNs within the bounds of verification, the property $\mathcal{P}$ will be maintained at all neurons in the DNN.*

Theorem 5.2 (Completeness). *If executing a well-typed program Π that does not use `traverse` and `solver` constructs on all concrete DNNs within the bounds of verification maintains the property $\mathcal{P}$ for all neurons in the DNN, then it can be proved by the ProveSound verification procedure.*

6 Evaluation

We demonstrate that designing the formal semantics for ConstraintFlow and the verification procedure ProveSound enables users to design and verify new DNN certifiers. The new designs

include—(i) variations to the existing certifiers, (ii) supporting new DNN operations within the existing abstract domains, and (iii) completely new abstract domains and transformers. In practice, the implementations of existing DNN certifiers [12, 44–46, 65, 73] employ various techniques to adjust the scalability vs precision tradeoff. Incorporating such modifications to the original algorithms unintentionally alters their mathematical logic. However, the original pen-and-paper proofs do not ensure the correctness of the certifiers with these modifications. In § 6.1, we demonstrate that these modified certifiers can be verified using ProveSound by specifying them in ConstraintFlow. In § 6.2, we extend DNN certifiers to support new operations such as `Abs`, `HardSigmoid`, etc. by designing abstract transformers, which has not been addressed by any existing work [45]. We also show the verification of their soundness using ProveSound. In § 6.3, we design new abstract domains and their corresponding transformers in ConstraintFlow and verify their soundness using ProveSound.

Finally, in § 6.4, we show that ConstraintFlow can specify and verify the above-mentioned diverse existing DNN certifiers, covering various abstract domains, transformers, and flow directions. We evaluated a diverse set of state-of-the-art DNN certifiers, including IBP [12], DeepPoly [45], CROWN [73], DeepZ [44], RefineZono [46], Vegas [65], and Hybrid Zonotope [33]. For all our experiments, we demonstrate that our verification procedure, ProveSound, can automatically prove the soundness of the certifiers specified in ConstraintFlow or detect unsoundness. The benchmarks for testing the unsoundness detection using ProveSound were created by introducing random bugs programmatically in the DNN certifiers, following a methodology established in prior research [11]. The details are provided in Appendix K.1.

DNN Operations. We focus on the widely used DNN operations, including primitive operations like `ReLU`, `Max`, `Min`, `Add`, `Mult`, etc., and composite operations like `Affine`, `MaxPool`, etc. The primitive operations are the ones that take a small, fixed number of inputs, like the addition or multiplication of 2 neurons. Since these can be composed to define composite operations, such as Attention layers, the corresponding abstract transformers can also be composed accordingly. Although verifying transformers for primitive operations directly implies the soundness of arbitrary compositions, in some cases, transformers can be more precise if specified directly for composite operations. In such cases, we show the specification and verification for composite operations.

We focus on the abstract transformers where the verification problem is known to be decidable. Although it is possible to express transformers for activation functions like Sigmoid and Tanh in ConstraintFlow (Appendix K.2), their verification may become undecidable [21]. In the future, ProveSound verification can be extended to handle these transformers using δ-complete decision procedures [17]. Currently, our verification queries fall under SMT of Nonlinear Real Arithmetic (NRA), decidable with a doubly exponential runtime in the worst case [24].

Verification Bounds. For verification of composite operations - `Affine` and `MaxPool`, the parameters, n_{prev} (maximum number of neurons in a layer) and n_{sym} (maximum length of a polyhedral or symbolic expression) are used during the graph expansion step and impact the verification times. For our experiments, we set $n_{sym} = n_{prev}$. Note that n_{prev} is an upper bound for the maximum number of neurons in a single layer, without restricting the total neuron count in the DNN. Therefore, the DNN can have an arbitrary number of layers, each with at most n_{prev} neurons, thereby, allowing for an arbitrary total number of neurons in the DNN. We set these parameters based on the sizes of layers within DNNs that existing certifiers currently handle [29, 34, 45, 73]. For `MaxPool`, `MinPool`, and `AvgPool`, existing certifiers handle at most 10 neurons at a time, so we set $n_{prev} = n_{sym} = 10$. The `Affine` layer includes DNN operations like convolution layers and fully-connected layers. In Table 3b, we present the computation times for `Affine` with $n_{prev} = n_{sym} = 2048$. In Fig. 16, we show how the verification time scales with parameter values ($n_{prev} = n_{sym}$), ranging from 32 to

8192, for `Affine` transformers. Note that $n_{prev} = 8192$ corresponds to over 64 million parameters per layer. Existing DNN certifiers [29, 34, 45, 73] usually do not operate on larger sizes than this, but the verification time for larger sizes can be extrapolated from the graph for higher values.

Experimental setup. We implemented the automated verification procedure in Python and used Z3 SMT solver [14] to verify the generated queries. All our experiments were run on a 2.50 GHz 16 core 11th Gen Intel i9-11900H CPU with a main memory of 64 GB.

6.1 Verifying Modified DNN Certifiers

Implementations of DNN certifiers often include modifications to balance the scalability vs. precision tradeoff. It is crucial to ensure the soundness of the modified certifiers. Verifying them using pen-and-paper proofs can be complicated. In contrast, ConstraintFlow and ProveSound provide a way to specify and verify these certifiers respectively. For illustration purposes, we focus on the DeepPoly abstract domain and key DNN operations—`Affine`, `MaxPool`, and `ReLU`. However, the concepts introduced can be applied to other certifiers and DNN operations. We present two case studies: BALANCE Cert and REUSE Cert, and show the evaluation results in Table 2a.

6.1.1 BALANCE Cert (Balanced Efficiency and Precision Certifier). We use the same abstract shape as the DeepPoly certifier and design transformers that balance precision and efficiency.
Affine. The most expensive part of the DeepPoly certifier is the backsubstitution step in the `Affine` transformer. To improve efficiency, albeit with reduced precision, BALANCE Cert employs a custom stopping function within the `traverse` construct to stop the backsubstitution at an intermediate layer, specifically, two layers back rather than always proceeding to the input layer.
ReLU. In the case of unstable neurons, there are two commonly used lower polyhedral bounds - 0 and `prev`. In BALANCE Cert, a heuristic determines which polyhedral lower bound to store based on `prev`$[l]$ and `prev`$[u]$.
MaxPool. For `MaxPool`, we use the new abstract transformer designed in [42], which is more precise than DeepPoly. We compute a list of neurons whose concrete lower bound is greater than or equal to the concrete upper bounds of all other neurons in `prev`. If this list is non-empty, we set the polyhedral lower and upper bounds to the average of the neurons in this list. Otherwise, we use the same polyhedral bounds used in DeepPoly. The complete code can be found in Appendix K.4.

6.1.2 REUSE Cert (Reused Bounds for Enhanced Efficiency). In an existing implementation of DeepPoly [47], the certifier stores previously computed polyhedral bounds from earlier layers to reuse them instead of recalculating them for current layer bounds. This approach prioritizes efficiency while accepting a slight trade-off in precision. In ConstraintFlow, this can be easily specified by additionally storing the cached polyhedral bounds as separate members of the abstract shape L_c, U_c. For the `Affine` abstract transformer, the users can first use the new polyhedral bounds. If the results are not sufficiently precise (based on a heuristic), then the computation falls back to the original computation using the `traverse` construct. This transformer significantly boosts efficiency by leveraging cached values of previous `Affine` layer backsubstitutions rather than computing them anew at each layer. The transformers for `ReLU` and `MaxPool` can be similarly defined for REUSE Cert. The complete code can be found in Appendix K.4.

6.2 Abstract Transformers for New DNN Operations

As deep learning frameworks continually introduce new activations, the need for designing *sound* abstract transformers becomes increasingly critical. We demonstrate the effectiveness of ConstraintFlow syntax and formal semantics and ProveSound verification procedure in this context

Table 2. Query generation time (G), verification time (V) for correct implementation, and bug-finding time for randomly introduced bugs (B) in seconds for new DNN certifiers (§ 6.1, § 6.2).

(a) New Transformers introduced in § 6.1

Certifiers	Affine			MaxPool			ReLU		
	G	V	B	G	V	B	G	V	B
BALANCE Cert	0.230	1.921	0.318	0.172	0.844	0.069	0.252	1.397	0.099
REUSE Cert	0.263	2.843	0.667	0.176	1.029	0.073	0.242	2.895	0.359

(b) New DNN operations introduced in § 6.2

Certifiers	ReLU6			Abs			HardSigmoid			HardTanh			HardSwish		
	G	V	B	G	V	B	G	V	B	G	V	B	G	V	B
DeepPoly/CROWN	0.299	2.454	0.543	0.199	5.252	0.069	0.319	2.238	0.147	0.304	3.016	0.354	0.277	2.963	0.383
Vegas(Backward)	0.216	1.264	0.145	0.078	0.237	0.102	0.206	0.900	0.076	0.166	1.154	0.095	0.186	0.812	0.065
DeepZ	0.150	1.25	0.363	0.116	0.462	0.369	0.172	1.634	0.550	0.148	2.677	0.526	0.290	3.457	0.886
RefineZono	0.233	2.084	0.347	0.165	0.870	0.128	0.259	2.847	0.150	0.178	2.444	0.657	0.542	2.42	0.564
IBP	0.102	0.237	0.289	0.147	0.455	0.059	0.098	0.228	0.071	0.123	0.269	0.065	0.205	0.653	0.218
Hybrid Zonotope	0.109	0.388	0.456	0.125	0.930	0.121	0.118	0.369	0.403	0.175	0.405	0.197	0.238	2.256	0.065
BALANCE Cert	0.230	1.921	0.318	0.172	0.844	0.069	0.252	1.397	0.099	0.229	2.433	0.083	0.198	2.070	0.462
REUSE Cert	0.263	2.843	0.667	0.176	1.029	0.073	0.242	2.895	0.359	0.227	4.354	0.446	0.234	3.733	0.121

by specifying and verifying abstract transformers for novel DNN operations not currently supported by existing DNN certifiers. These new operations include `ReLU6`, `Abs`, `HardSigmoid`, `HardTanh`, and `HardSwish`. Detailed transformers for each operation can be found in Appendix K. Evaluation results across different DNN certifiers are presented in Table 2b, demonstrating that most transformers for these operations can be verified (or disproved) within 1 second. For illustration, we show the DeepPoly transformer for `HardSwish` ($\texttt{HardSwish}(x) = x \cdot \min(1, \min(0, \frac{x+3}{6}))$).

```
Func slope(Real x1, Real x2) = ((x1 * (x1 + 3)) - (x2 * (x2 + 3))) / (6 * (x1-x2));
Func intercept(Real x1, Real x2) = x1 * ((x1 + 3) / 6) - (slope(x1, x2) * x1);
Func f1(Real x) = x < 3 ? x * ((x + 3) / 6) : x;
Func f2(Real x) = x * ((x + 3) / 6);
Func f3(Neuron n) = max(f2(n[l]), f2(n[u]));
Transformer DeepPoly{
    HardSwish ->
    (prev[l] < -3) ?
        (prev[u] < -3 ?
            (0, 0, 0, 0) :
            (prev[u] < 0 ?
                (-3/8, 0, -3/8, 0) :
                (-3/8, f1(prev[u]), -3/8, f1(prev[u]) * (prev - prev[l])))) :
            ((prev[l] < 3) ? ((prev[u] < 3) ?
                (-3/8, f3(prev), -3/8, prev*slope(prev[u], prev[l]) + intercept(prev[u],prev[l
    ])):
                (-3/8, prev[u], -3/8, prev[u] * ((prev + 3) / (prev[u] + 3)))) :
        (prev[l], prev[u], prev, prev));
}
```

6.3 Designing New DNN Certifiers with New Abstract Domains

We show that ProveSound allows verifying the soundness of new DNN certifiers based on completely new abstract domains and transformers. Specifying them in ConstraintFlow is only possible due to the novel formalism including type system and semantics introduced in this work.

```
Def shape as (Real l, Real u, PolyExp symL, PolyExp symU) {(curr[l]<=curr) and (curr[u]>=curr)
    and (curr[symL]<=curr) and (curr[symU]>=curr)};
Transformer SymPoly{
  Relu -> prev[l] > 0 ? (prev[l], prev[u], prev, prev) :
            (prev[u] < 0 ? (0, 0, 0, 0) :
                (0, prev[u], ((1+sym)/2) * prev, ((prev[u] / (prev[u] - prev[l])) * prev) - ((
    prev[u] * prev[l]) / (prev[u] - prev[l]))));
}
```

(a) SymPoly

```
Def shape as (Real l, Real u, PolyExp L, PolyExp U, SymExp Z) {curr[l]<=curr and curr[u]>=curr
    and curr[L]<=curr and curr[U]>=curr and curr <> curr[Z]};
Func min_symexp(Sym e, Real c) = c > 0 ? -c : c;
Func lower_sym(Neuron List prev, Neuron curr) = (prev[Z] * curr[w] + curr[b]).map(min_symexp);
Func lower_poly(Neuron List prev, Neuron curr) = backsubs_lower(prev * curr[w] + curr[b]);
Transformer PolyZ{
   Affine ->  (max(lower_sym(prev, curr), lower_poly(prev, curr)),
               min(upper_sym(prev, curr), upper_poly(prev, curr)),
               prev * curr[w] + curr[b], prev * curr[w] + curr[b], prev[Z] * curr[w] + curr[b
    ]);
}
```

(b) PolyZ

Fig. 13. Code Sketches for new DNN certifiers. The complete codes can be found in Appendix K.4

SymPoly DNN Certifier. Several state-of-the-art DNN certifiers, including DeepPoly, CROWN, etc., approximate the value of each neuron in the DNN by imposing polyhedral constraints over each of them. However, in the case of piecewise-linear activation functions, these certifiers rely on heuristics to choose appropriate polyhedral bounds from more than one possible choice. For instance, in the case of an unstable `ReLU` neuron, there are infinite possibilities for a potential lower polyhedral bound. We argue that in general, the lower polyhedral bound can be of the form $c \cdot$ `prev` where c is any real coefficient s.t. $0 \leq c \leq 1$. The two most commonly used lower bounds - `prev` and 0 are only two extreme cases of the general lower bound. Using the ConstraintFlow syntax and semantics, the users can directly specify the general transformer, i.e., $\texttt{curr}[L] \leftarrow \frac{1+\texttt{sym}}{2} * \texttt{prev}$. ProveSound can be used to prove the soundness of this lower bound. In this way, ProveSound allows a user to verify the soundness of a space of abstract transformers, which can be leveraged to automatically synthesize the optimal transformer using a cost function encoding the precision of the transformer based on the DNN certification problem. Further, since each invocation of the `sym` construct outputs a new symbolic value, different values of the symbolic coefficient can be chosen for different neurons in the DNN. A slightly different version is explored in the DNN certifier α−CROWN [70], where α is a concrete but learnable coefficient, learned using gradient descent.

Based on this idea, the DNN certifier SymPoly can be found in Fig. 13a. The abstract domain consists of two concrete bounds `l, u` and two polyhedral bounds with symbolic coefficients `symL, symU`. The abstract transformer for `ReLU` is specified in 3 cases - (i) $\texttt{curr}[u] < 0$, (ii) $\texttt{curr}[l] > 0$, and (iii) $\texttt{curr}[l] \leq 0 \leq \texttt{curr}[u]$. In the more challenging third case, the lower polyhedral bound is set to $\frac{1+\texttt{sym}}{2} * \texttt{prev}$. The abstract transformers can be similarly designed for activations such as `HardTanh`, `HardSigmoid`, `HardSwish`, `Abs`, etc. These can be found in Appendix K.4. Notably, we can verify the soundness of these transformers in runtimes similar to the DeepPoly certifier.

```
Func lower(Neuron n1, Neuron n2) = min([n1[l]*n2[l], n1[l]*n2[u], n1[u]*n2[l], n1[u]*n2[u]]);
Func upper(Neuron n1, Neuron n2) = max([n1[l]*n2[l], n1[l]*n2[u], n1[u]*n2[l], n1[u]*n2[u]]);
Transformer DeepPoly{
    Max -> (prev0[l] >= prev1[u]) ? (prev0[l], prev0[u], prev0, prev0) : ((prev1[l] >= prev0[u
     ]) ?
                    (prev1[l], prev1[u], prev1, prev1) :
                    (max(prev0[l], prev1[l]), max(prev0[u], prev1[u]), max(prev0[l], prev1[l]),
                        max(prev0[u], prev1[u])));
    Mult -> (lower(prev0, prev1), upper(prev0, prev1), lower(prev0, prev1), upper(prev0, prev1)
     );
}
```

Fig. 14. Max and Mult transformers for DeepPoly Certifier

```
Def shape as (Real l, Real u, PolyExp L, PolyExp U)
      {...};

Transformer DeepPoly_forward{ReLU -> ... ;}
Transformer DeepPoly_backward{rev_ReLU -> ... ;}

Flow(forward, ..., ... , DeepPoly_forward);
Flow(backward, ..., ..., DeepPoly_backward);
```

Fig. 15. Code Sketch for Vegas Certifier

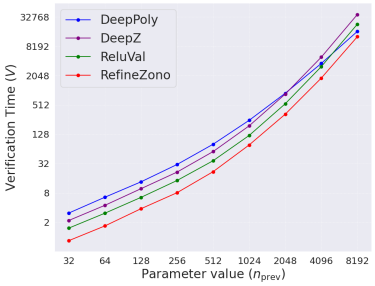


Fig. 16. Verification time (in s) for `Affine` transformers.

PolyZ DNN Certifier. We show another new abstract domain - PolyZ - a reduced product of the popular DeepZ and DeepPoly domains using polyhedral and symbolic constraints. The abstract shape consists of 5 members - two concrete interval bounds, l and u of the type `Real`, two polyhedral bounds L and U of the type `PolyExp`, and a symbolic expression Z of the type `SymExp`. The shape constraints state that the neuron's value satisfies the bounds l, u, L, and U and `curr` <> Z. We also define the abstract transformers for this new domain. The `Affine` transformer is shown in Fig. 13b and the complete specification is in Appendix K.4. PolyZ is more precise than both DeepPoly and DeepZ, and we can verify its soundness using the ProveSound verification procedure.

6.4 State-of-the-Art DNN Certifiers

The existing DNN certifiers evaluated in this section include IBP [12] (Interval Bound Propagation), DeepPoly [45] (or CROWN [73]), DeepZ [44], RefineZono [46], Vegas [65], and Hybrid Zonotope [33]. The abstract shapes of DeepPoly, CROWN, and Vegas include polyhedral expressions represented by the `PolyExp` datatype and use the `traverse` construct to compute the concrete bounds. DeepZ, RefineZono, and Hybrid Zonotope use symbolic expressions represented by `SymExp` in their abstract shapes. RefineZono uses `Ct` to encode constraints over the possible values of the neurons. RefineZono and Vegas use the `solver` construct to compute the concrete bounds. The users can define functions using the `Func` construct, promoting code reusability and facilitating a modular design. The ConstraintFlow codes for these DNN certifiers are presented in Appendix K.

Notably, with the formal syntax and the operational semantics, ConstraintFlow can handle various flow directions effectively. For instance, the Vegas certifier [65], which employs both

Table 3. Query generation time (G), verification time (V) for correct implementation, and bug-finding time for randomly introduced bugs (B) in seconds for transformers of existing DNN certifiers (§ 6.4).

(a) Primitive operations

Certifiers	ReLU			Max			Min			Add			Mult		
	G	V	B	G	V	B	G	V	B	G	V	B	G	V	B
DeepPoly/CROWN	0.196	1.526	0.066	0.095	2.618	0.074	0.128	2.829	0.601	0.0812	0.136	0.205	0.209	2.104	0.129
Vegas(Backward)	0.142	0.584	0.221	0.047	0.139	0.084	0.052	0.115	0.087	0.056	0.097	0.153	0.388	0.486	0.110
DeepZ	0.0832	0.534	0.336	0.115	0.703	0.145	0.119	0.691	0.215	0.0815	0.091	0.256	0.234	0.498	0.427
RefineZono	0.158	0.980	0.071	0.199	1.235	0.262	0.213	1.263	0.331	0.089	0.117	0.242	0.404	17.197	0.468
IBP	0.112	0.493	0.364	0.132	0.508	0.081	0.136	0.545	0.333	0.0716	0.060	0.158	0.217	1.160	0.259
Hybrid Zonotope	0.260	1.003	0.341	0.132	0.775	0.292	0.132	0.724	0.626	0.086	0.286	0.204	0.209	0.520	1.397

(b) Composite operations

Certifiers	Affine			MaxPool			MinPool			AvgPool		
	G	V	B	G	V	B	G	V	B	G	V	B
DeepPoly / CROWN	5.496	889.607	9.825	14.744	196.651	1396.132	13.917	194.871	1419.119	0.137	0.363	0.131
Vegas (Backward)	2.436	25.447	25.898	-	-	-	-	-	-	-	-	-
DeepZ	4.569	854.548	833.314	54.217	364.859	1780.938	52.140	292.806	1366.977	0.0818	0.265	0.763
RefineZono	5.436	329.994	152.825	54.788	376.177	1451.729	56.427	308.570	1799.091	0.095	0.306	0.301
IBP	2.997	540.865	183.707	0.089	4.077	0.253	0.090	4.114	4.605	0.067	0.0117	0.921
Hybrid Zonotope	-	-	-	1.816	10.610	2.892	1.503	10.598	3.395	0.318	11.499	2.568

forward and backward flows, is easily expressed in ConstraintFlow. We provide the code for the Vegas certifier in Fig. 15. The abstract shape and the transformer for the forward direction are the same as the DeepPoly analysis, while the transformer for the backward analysis replaces operations like `ReLU` with `rev_ReLU`. We can also verify its soundness using ProveSound (Tables 3a, 3b).

For primitive operations like `Max`, `Mult`, etc., there are two implicit inputs to the transformer definitions, namely the input neurons - `prev0` and `prev1`. DeepPoly transformers for `Max` and `Mult` are shown in Fig. 14. The primitive operations - `ReLU`, `Max`, `Min`, `Add`, `Mult` shown in Table 3a can be verified in fractions of a second. In Table 3b, we show the evaluation results for the composite operations. For `MaxPool` and `MinPool`, the DeepZ and RefineZono transformers are harder to verify because their queries are doubly quantified due to the <> operator in their specifications. IBP is the easiest to verify because the limited abstract shape does not allow it to be as precise as other transformers for `MaxPool` and `MinPool`. Also, for Vegas, the backward transformers for `MaxPool`, `MinPool`, and `AvgPool` are not available in existing works. Similarly, for the Hybrid Zonotope, the transformer for `Affine` is defined in terms of transformers for primitive operations. So, we skip these in Table 3b. For `Affine`, DeepPoly is the hardest because it uses the `traverse` construct, which requires additional queries to check the validity of the invariant. Vegas takes the least time because of a relatively simpler verification query. Note that the verification times are not correlated to the runtimes of certifiers on concrete DNNs. In Appendix K.3, we provide the ConstraintFlow code for several of these certifiers. The complexity inherent in these certifiers and their implementations suggests that verifying them solely through pen-and-paper proofs or automated theorem provers is impractical.

7 Related Work

DNN Certification. The recent advancements in DNN certification techniques [1] have led to the organization of competitions to showcase DNN certification capabilities [10], the creation of benchmark datasets [13], the introduction of a DSL for specifying certification properties [19, 40], and the development of a library for DNN certifiers [27, 36]. However, these platforms lack formal soundness guarantees and do not offer a systematic approach to designing new certifiers.

DSL for Abstract Interpretation. Although [42] proposed a preliminary design for ConstraintFlow using a few examples, the absence of formal semantics hinders its use for designing and verifying new DNN certifiers. We equip ConstraintFlow with a BNF grammar, type-system, operational semantics, and symbolic semantics that enable users to specify existing DNN certifiers, design new ones, and verify their soundness using ProveSound.

Similarly, [28] designs TSL—a DSL for abstract interpreters for conventional programs. TSL allows users to specify the concrete semantics and the abstract domain and automatically produces an abstract interpreter based on these specifications. However, it does not provide any specialized datatypes needed to specify DNN certifiers easily. It also does not guarantee the soundness of the abstract interpreter. In contrast, ProveSound can verify the soundness of the certifier specification.

Symbolic Execution. Similar to ProveSound DNN expansion step, [25, 59] employ lazy initialization for symbolic execution of complex data structures like lists, trees, etc. The object fields are initialized with symbolic values only when accessed by the program. Unlike these works, which possess prior knowledge of the exact structure of the objects, DNN certifiers deal with arbitrary DAGs representing DNNs. The graph nodes (neurons) are intricate data structures with unknown graph topology. We believe that we are the first to create a symbolic DNN with sufficient generality to represent arbitrary graph topologies to verify the soundness of DNN certifiers.

Correctness of Symbolic Execution. Some existing works prove the correctness of the symbolic execution w.r.t. the language semantics [23]. However, these methods do not establish correctness in cases where symbolic execution also represents symbolic variables used in concrete executions. On the other hand, we provide elaborate proofs establishing the correctness of ProveSound where we encode the symbolic variables within the program as SMT symbolic variables.

8 Discussion and Future Work

We develop ProveSound, a novel bounded automated verification procedure to automatically verify the overapproximation-based soundness of abstract interpretation-based DNN certifiers. We also develop a formal syntax, type-system, operational semantics, and symbolic semantics for ConstraintFlow. For the first time, we can verify the soundness of DNN certifiers for arbitrary (but bounded) DAG topologies. Given the growing concerns about AI safety, we believe that ProveSound, coupled with ConstraintFlow, allows the development of new DNN certifiers without proving their soundness manually. This work allows the following future directions:

Multi-neuron specifications. - ProveSound can be extended to verify multi-neuron abstract shapes [35] by allowing their specification in ConstraintFlow.

Sequence of Operations. - ProveSound can also be extended to automatically verify a sequence of DNN operations, like `Affine + ReLU`. To do so, while generating the final query, we would execute the concrete semantics of the composition of `Affine` and `ReLU`.

Automating Abstract Interpretation. - ProveSound and ConstraintFlow facilitate the automated generation of abstract transformers [22, 38, 50] by offering all the basic components - (i) a DSL for defining the search space of candidate transformers, (ii) the semantics of the DSL, and (iii) a procedure for verifying the soundness of each candidate. This can be explored in future research.

Verification Property. - Currently, the verification property is the over-approximation-based soundness of a DNN certifier. Nevertheless, given that all the necessary formalism for verification has been established, the property can be extended to encompass more intricate aspects, such as encoding precision of a DNN certifier w.r.t. a baseline.

9 Data-Availability Statement

The artifact[41] consists of ProveSound implementation and the ConstraintFlow specifications of the DNN certifiers presented in Section 6 and Appendix K. The code, accompanied by the instructions to run it, can be found here.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF Grants No. CCF-2238079, CCF-2316233, CNS-2148583.

References

[1] Aws Albarghouthi. 2021. *Introduction to Neural Network Verification*. verifieddeeplearning.com. arXiv:2109.10317 [cs.LG] http://verifieddeeplearning.com.

[2] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. 2013. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine* 11, 2 (2013).

[3] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. Programming Language Design and Implementation (PLDI)*. 731–744.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657

[5] Debangshu Banerjee, Avaljot Singh, and Gagandeep Singh. 2024. Interpreting Robustness Proofs of Deep Neural Networks. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=Ev10F9TWML

[6] Debangshu Banerjee and Gagandeep Singh. 2024. Relational DNN Verification With Cross Executional Bound Refinement. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=HOG80Yk4Gw

[7] Debangshu Banerjee, Changming Xu, and Gagandeep Singh. 2024. Input-Relational Verification of Deep Neural Networks. *Proc. ACM Program. Lang.* 8, PLDI, Article 147 (June 2024), 27 pages. doi:10.1145/3656377

[8] Mariusz Bojarski, Davide Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Larry Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. (04 2016).

[9] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2019. CNN-Cert: An Efficient Framework for Certifying Robustness of Convolutional Neural Networks. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 398, 8 pages.

[10] Christopher Brix, Mark Niklas Müller, Stanley Bak, Taylor T. Johnson, and Changliu Liu. 2023. First Three Years of the International Verification of Neural Networks Competition (VNN-COMP). *CoRR* abs/2301.05815 (2023). doi:10.48550/arXiv.2301.05815 arXiv:2301.05815

[11] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 768–778. doi:10.1145/3238147.3240464

[12] Patrick Cousot and Radhia Cousot. 1977. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Raleigh, North Carolina). Association for Computing Machinery, New York, NY, USA, 77–94. doi:10.1145/800022.808314

[13] Armando Tacchella Dario Guidotti, Stefano Demarchi and Luca Pulina. 2023. The Verification of Neural Networks Library (VNN-LIB). https://www.vnnlib.org,2023.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.

[15] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *NASA Formal Methods*, Aaron Dutle, César Muñoz, and Anthony Narkawicz (Eds.). Springer International Publishing, Cham, 121–138.

[16] Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis*.

[17] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. 2012. δ-Complete Decision Procedures for Satisfiability over the Reals. In *International Joint Conference on Automated Reasoning*. https://api.semanticscholar.org/CorpusID:4508719

[18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 3–18. doi:10.1109/SP.2018.00058

[19] Chuqin Geng, Nham Le, Xiaojie Xu, Zhaoyue Wang, Arie Gurfinkel, and Xujie Si. 2023. Towards Reliable Neural Specifications. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 449, 17 pages.

[20] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 3–29.

[21] Omri Isac, Yoni Zohar, Clark W. Barrett, and Guy Katz. 2023. DNN Verification, Reachability, and the Exponential Function Problem. *CoRR* abs/2305.06064 (2023). doi:10.48550/arXiv.2305.06064 arXiv:2305.06064

[22] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing abstract transformers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 171 (oct 2022), 29 pages. doi:10.1145/3563334

[23] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming). *Proc. ACM Program. Lang.* 6, ICFP, Article 97 (aug 2022), 31 pages. doi:10.1145/3547628

[24] To Van Khanh and Mizuhito Ogawa. 2012. SMT for Polynomial Constraints on Real Numbers. In *TAPAS@SAS*. https://api.semanticscholar.org/CorpusID:13959185

[25] Sarfraz Khurshid, Corina S. PÃsÃreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, Hubert Garavel and John Hatcliff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 553–568.

[26] Rustan Leino and Michal Moskal. 2013. *Co-Induction Simply: Automatic Co-Inductive Proofs in a Program Verifier*. Technical Report MSR-TR-2013-49. https://www.microsoft.com/en-us/research/publication/co-induction-simply-automatic-co-inductive-proofs-in-a-program-verifier/

[27] Linyi Li, Tao Xie, and Bo Li. 2023. SoK: Certified Robustness for Deep Neural Networks. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, 22-26 May 2023*. IEEE.

[28] Junghee Lim and Thomas Reps. 2013. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 4 (April 2013), 59 pages. doi:10.1145/2450136.2450139

[29] T. Lorenz, A. Ruoss, M. Balunovic, G. Singh, and M. Vechev. 2021. Robustness Certification for Point Cloud Models. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Los Alamitos, CA, USA, 7588–7598. doi:10.1109/ICCV48922.2021.00751

[30] Zhaoyang Lyu, Ching-Yun Ko, Zhifeng Kong, Ngai Wong, Dahua Lin, and Luca Daniel. 2020. Fastened CROWN: Tightened Neural Network Robustness Certificates. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 5037–5044. https://ojs.aaai.org/index.php/AAAI/article/view/5944

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJzIBfZAb

[32] Denis Mazzucato and Caterina Urban. 2021. Reduced Products of Abstract Domains for Fairness Certification of Neural Networks. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 308–322. doi:10.1007/978-3-030-88806-0_15

[33] Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 3578–3586. https://proceedings.mlr.press/v80/mirman18b.html

[34] Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2021. Scaling Polyhedral Neural Network Verification on GPUs. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/ca46c1b9512a7a8315fa3c5a946e8265-Abstract.html

[35] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2022. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proc. ACM Program. Lang.* 6, POPL, Article 43 (jan 2022), 33 pages. doi:10.1145/3498704

[36] Long H. Pham, Jiaying Li, and Jun Sun. 2020. SOCRATES: Towards a Unified Platform for Neural Network Verification. *ArXiv* abs/2007.11206 (2020).

[37] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Semidefinite Relaxations for Certifying Robustness to Adversarial Examples. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*

(Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 10900–10910.

[38] Thomas Reps and Aditya Thakur. 2016. Automating Abstract Interpretation. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583* (St. Petersburg, FL, USA) *(VMCAI 2016)*. Springer-Verlag, Berlin, Heidelberg, 3–40. doi:10.1007/978-3-662-49122-5_1

[39] Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. 2019. *A Convex Relaxation Barrier to Tight Robustness Verification of Neural Networks*. Curran Associates Inc., Red Hook, NY, USA.

[40] David Shriver, Sebastian Elbaum, and Matthew B. Dwyer. 2021. DNNV: A Framework for Deep Neural Network Verification. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 137–150.

[41] A. Singh. 2025. *ProveSound: OOPSLA-2025-AEC-final*. https://doi.org/10.5281/zenodo.14597703

[42] Avaljot Singh, Yasmin Sarita, Charith Mendis, and Gagandeep Singh. 2024. ConstraintFlow: A DSL for Specification and Verification of Neural Network Analyses. In *Static Analysis*. Springer Nature Switzerland.

[43] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. 2019. *Beyond the Single Neuron Convex Barrier for Neural Network Certification*. Curran Associates Inc., Red Hook, NY, USA.

[44] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. *Advances in Neural Information Processing Systems* 31 (2018).

[45] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019).

[46] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2018. Boosting Robustness Certification of Neural Networks. In *International Conference on Learning Representations*.

[47] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2018. ETH Robustness Analyzer for Neural Networks (ERAN). https://github.com/eth-sri/eran.

[48] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast Polyhedra Abstract Domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 46–59. doi:10.1145/3009837.3009885

[49] Matthew Sotoudeh, Zhe Tao, and Aditya Thakur. 2023. SyReNN: A tool for analyzing deep neural networks. *International Journal on Software Tools for Technology Transfer* 25 (02 2023), 1–21. doi:10.1007/s10009-023-00695-1

[50] Aditya V. Thakur, Akash Lal, Junghee Lim, and T. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. In *TAPAS@SAS*. https://api.semanticscholar.org/CorpusID:8700802

[51] Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, Krunal Patel, and Juan Pablo Vielma. 2020. The Convex Relaxation Barrier, Revisited: Tightened Single-Neuron Relaxations for Neural Network Verification. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1819, 12 pages.

[52] Vincent Tjeng, Kai Yuanqing Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=HyGIdiRqtm

[53] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. doi:10.1145/2509578.2509586

[54] Hoang-Dung Tran, Stanley Bak, Weiming Xiang, and Taylor T. Johnson. 2020. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 18–42. doi:10.1007/978-3-030-53288-8_2

[55] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Star-Based Reachability Analysis of Deep Neural Networks. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 670–686.

[56] Shubham Ugare, Debangshu Banerjee, Sasa Misailovic, and Gagandeep Singh. 2023. Incremental Verification of Neural Networks. *Proc. ACM Program. Lang.* 7, PLDI, Article 185 (June 2023), 26 pages. doi:10.1145/3591299

[57] Shubham Ugare, Tarun Suresh, Debangshu Banerjee, Gagandeep Singh, and Sasa Misailovic. 2024. Incremental Randomized Smoothing Certification. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=SdeAPV1irk

[58] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Perfectly Parallel Fairness Certification of Neural Networks. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 185 (nov 2020), 30 pages. doi:10.1145/3428253

[59] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts,

USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/1007512.1007526

[60] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*.

[61] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks Using Symbolic Intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1599–1614.

[62] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J Zico Kolter. 2021. Beta-CROWN: Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Verification. *arXiv preprint arXiv:2103.06624* (2021).

[63] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5276–5285. https://proceedings.mlr.press/v80/weng18a.html

[64] Eric Wong and J. Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 5283–5292. http://proceedings.mlr.press/v80/wong18a.html

[65] Haoze Wu, Clark Barrett, Mahmood Sharif, Nina Narodytska, and Gagandeep Singh. 2022. Scalable Verification of GNN-Based Job Schedulers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 162 (oct 2022), 30 pages. doi:10.1145/3563325

[66] Haoze Wu, Alex Ozdemir, Aleksandar Zeljić, Kyle Julian, Ahmed Irfan, Divya Gopinath, Sadjad Fouladi, Guy Katz, Corina Pasareanu, and Clark Barrett. 2020. Parallelization Techniques for Verifying Neural Networks. In *2020 Formal Methods in Computer Aided Design (FMCAD)*. 128–137. doi:10.34727/2020/isbn.978-3-85448-042-6_20

[67] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. 2017. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *CoRR* abs/1708.03322 (2017). arXiv:1708.03322 http://arxiv.org/abs/1708.03322

[68] Changming Xu and Gagandeep Singh. 2023. Robust Universal Adversarial Perturbations. https://openreview.net/forum?id=VpYBxaPLaj-

[69] Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. 2020. Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond. (2020).

[70] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Sekhar Jana, Xue Lin, and Cho-Jui Hsieh. 2020. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. *ArXiv* abs/2011.13824 (2020).

[71] Pengfei Yang, Renjue Li, Jianlin Li, Cheng-Chao Huang, Jingyi Wang, Jun Sun, Bai Xue, and Lijun Zhang. 2021. *Improving Neural Network Verification through Spurious Region Guided Refinement*. 389–408. doi:10.1007/978-3-030-72016-2_21

[72] Tom Zelazny, Haoze Wu, Clark W. Barrett, and Guy Katz. 2022. On Optimizing Back-Substitution Methods for Neural Network Verification. *2022 Formal Methods in Computer-Aided Design (FMCAD)* (2022), 17–26.

[73] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient Neural Network Robustness Certification with General Activation Functions. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 4944–4953.

Received 2024-10-16; accepted 2025-02-18