SyReNN: A Tool for Analyzing Deep Neural Networks

Matthew Sotoudeh (✉)*, Zhe Tao (✉)*, and Aditya V. Thakur (✉)

University of California, Davis CA 95616, USA
{masotoudeh,zhetao,avthakur}@ucdavis.edu

Abstract. Deep Neural Networks (DNNs) are rapidly gaining popularity in a variety of important domains. Unfortunately, modern DNNs have been shown to be vulnerable to a variety of attacks and buggy behavior. This has motivated recent work in formally analyzing the properties of such DNNs. This paper introduces SyReNN, a tool for understanding and analyzing a DNN by computing its *symbolic representation*. The key insight is to decompose the DNN into linear functions. Our tool is designed for analyses using *low-dimensional subsets* of the input space, a unique design point in the space of DNN analysis tools. We describe the tool and the underlying theory, then evaluate its use and performance on three case studies: computing Integrated Gradients, visualizing a DNN's decision boundaries, and repairing buggy DNNs.

Keywords: Deep Neural Networks · Symbolic representation · Integrated Gradients · Repair

1 Introduction

Deep Neural Networks (DNNs) [20] have become the state-of-the-art in a variety of applications including image recognition [58,37] and natural language processing [11]. Moreover, they are increasingly used in safety- and security-critical applications such as autonomous vehicles [35] and medical diagnosis [9,42,32,41]. These advances have been accelerated by improved hardware and algorithms.

DNNs (Section 2) are programs that compute a vector-valued function, i.e., from $\mathbb{R}^n$ to $\mathbb{R}^m$. They are loop-free programs written as a concatenation of alternating linear and non-linear *layers*. The coefficients of the linear layers are learned from data via *gradient descent* during a training process. A number of different non-linear layers (called *activation functions*) are commonly used, including the *rectified linear* and *maximum pooling* functions.

Owing to the variety of application domains and deployment constraints, DNNs come in many different sizes. For instance, large image-recognition and natural-language processing models are trained and deployed using cloud resources [37,11], medium-size models could be trained in the cloud but deployed on hardware with limited resources [35], and finally small models could be trained

* Equal contribution.

and deployed directly on edge devices [52,8,24,38,39]. There has also been a recent push to compress trained models to reduce their size [27]. Such smaller models play an especially important role in privacy-critical applications, such as wake word detection for voice assistants, because they allow sensitive user data to stay on the user's own device instead of needing to be sent to a remote computer for processing.

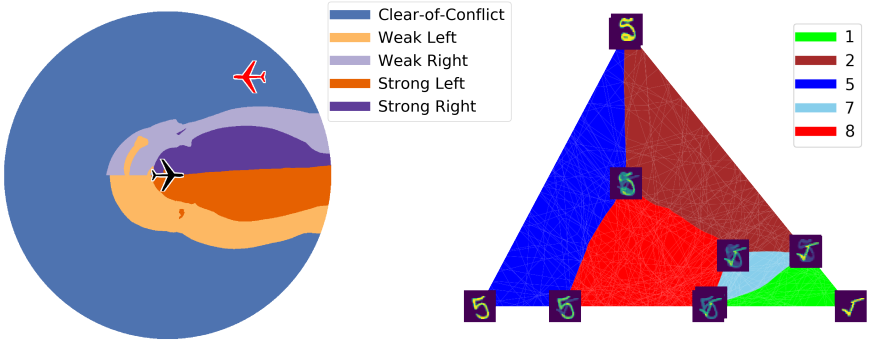
Although DNNs are very popular, they are not perfect. One particularly concerning development is that modern DNNs have been shown to be extremely vulnerable to *adversarial examples,* inputs that are intentionally manipulated to appear unmodified to humans but become misclassified by the DNN [59,21,44,7]. Similarly, *fooling examples* are inputs that look like random noise to humans, but are classified with high confidence by DNNs [46]. Mistakes made by DNNs have led to loss of life [40,19] and wrongful arrests [30,31]. For this reason, it is important to develop techniques for analyzing, understanding, and repairing DNNs.

This paper introduces SyReNN, a tool for understanding and analyzing DNNs. SyReNN implements state-of-the-art algorithms for computing precise symbolic representations of piecewise-linear DNNs (Section 3). Given a bounded polytope subset of the input space of a DNN, SyReNN computes a symbolic representation that decomposes the behavior of the DNN on that infinite subspace into finitely many linear functions. SyReNN implements the one-dimensional analysis algorithm of Sotoudeh and Thakur [55] and extends it to the two-dimensional setting as described in Section 4.

Key insights. There are two key insights enabling this approach, initially identified in Sotoudeh and Thakur [55]. First, most popular DNN architectures today are *piecewise-linear*, meaning they can be precisely decomposed into finitely many linear functions. This allows us to reduce their analysis to equivalent questions in linear algebra, one of the most well-understood fields of modern mathematics. Second, many applications only require analyzing the behavior of the DNN on a *low-dimensional subset* of the input space. Whereas prior work has attempted to give up precision for efficiency while analyzing high-dimensional input regions [53,54,16], our work has focused on algorithms that are *both efficient and precise* while analyzing lower-dimensional regions (Section 4).

Use cases. We demonstrate the utility of SyReNN on three main applications, each highlighting the key advantage of SyReNN; viz., the ability to provide *precise* information about the DNN by restricting the analysis to a *low-dimensional subset* of its input space.

The first application is in *visualizing the decision boundaries* of a DNN. With SyReNN, we can precisely plot the decision boundaries for a two-dimensional subset of the input space. The two-dimensional nature of the plot makes it ideal for a human designer to look at to understand the behavior of the DNN. The precise nature of the information provided by SyReNN ensures that the plot shows the true behavior on the infinite set of points. Figure 1a shows one such 2-dimensional plot for the ACAS Xu network ([35]), which determines what action an aircraft (ownship) should take to avoid a collision with an intruder. Notably, this is *not* the result of plotting the behavior of the DNN on a finite sampling



(a) (b)

Fig. 1: Precise visualization of decision boundaries computed using SyReNN for the (a) ACAS Xu network and (b) MNIST digit recognition network. This is *not* a plot interpolating between finitely-many sampled points, instead SyReNN was used to quickly and precisely compute the *exact* decision boundaries.

of the inputs — such an approach would likely miss key behavior of the DNN, whereas our approach *guarantees* to find the *exact* decision boundaries. The network takes 5 inputs corresponding to the velocity and position of the aircraft; the plot shows the behavior of the DNN for a 2-dimensional subspace of the input space. From this plot, one can already see interesting and potentially dangerous behavior of the network: there is a region behind the plane where an intruder on the left may cause the network to recommend performing a weak left towards the intruder; there are small regions in which the network recommends strong right (or strong left) which should be weak right (or weak left). For safety-critical applications, such as aircraft collision avoidance, such precise visualization can be invaluable to a developer. Visualization of the ACAS Xu network is described in detail in Section 8.1. Section 8.2 describes the visualization of decision boundaries of image recognition networks. For instance, Figure 1b shows the visualization of the decision boundaries of an MNIST handwritten digit recognition DNN. The corners of the triangle are three different drawings of the number 5, while points interior to the triangle correspond to interpolations between those images. The different colors show exactly where the network begins to misclassify fives as a variety of other digits, helping human designers better understand, and eventually improve, the behavior of the DNN.

The second application of SyReNN is the *provable repair* of DNNs (Section 8.3). DNN repair deals with the increasingly important problem of correcting DNN behavior to satisfy a given specification of a DNN after it has been trained. In contrast to heuristic approaches, e.g., based on gradient descent, provable repair guarantees that the repaired DNN does in fact satisfy the repair specification. Prior work can repair the behavior of the DNN for a finite set of

input points [18,56]. With SyReNN one can extend these techniques to repair the behavior for an infinite set of points (represented as a 2D polytope).

The last application of SyReNN is the precise computation of *Integrated Gradients* (IG) [57], a state-of-the-art measure used to determine which input dimensions, e.g., pixels for an image-recognition network, were most important in the final classification produced by the network (Section 8.4). IG is defined in terms of the DNN's behavior on the line connecting the all-zero input and the input in question. Without the precise and efficient low-dimensional analysis of SyReNN, all prior work had only been approximating the IG.

Tool design. The SyReNN tool is designed to be easy to use and extend, as well as efficient (Section 7). The core of SyReNN is written as a highly-optimized, parallel C++ server using Intel TBB for parallelization [50] and Eigen for matrix operations [25]. CPU-based SyReNN uses Intel MKL-DNN for DNN evaluation while GPU-based SyReNN uses Nvidia cuDNN and cuBLAS for DNN evaluation as well as a mix of Intel TBB and Nvidia CUDA for symbolic representation computation. A user-friendly Python front-end interfaces with the PyTorch deep learning framework [49].

Contributions. The contributions of this paper are:

- A definition of symbolic representation of DNNs (Section 3).
- Efficient algorithms for computing symbolic representations for DNNs over low-dimensional input subspaces on CPU (Section 4) and GPU (Section 5).
- A generalization of the algorithm to arbitrary-dimensional input subspaces (Section 6).
- A design of a usable and well-engineered tool implementing these ideas called SyReNN (Section 7).
- Three applications of SyReNN (Section 8).

Section 2 presents preliminaries about DNNs; Section 9 presents related work; Section 10 concludes. SyReNN is available on GitHub at https://github.com/95616ARG/SyReNN_GPU.

2 Preliminaries

We now formally define the notion of *DNN* we will use in this paper.

Definition 1. *A* Deep Neural Network *(DNN) is a function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *which can be written* $f = f_n \circ f_{n-1} \cdots \circ f_1$ *for a sequence of* layer functions f_1, f_2, $\ldots$, f_n.

Our work is primarily concerned with the popular class of *piecewise-linear* DNNs, defined below. In this definition and the rest of this paper, we will use the term "polytope" to mean a *convex and bounded* polytope, i.e., a bounded, finite intersection of linear constraints.

Definition 2. *A function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *is* piecewise-linear (PWL) *if its input domain* $\mathbb{R}^n$ *can be partitioned into finitely many possibly-unbounded polytopes* $X_1, X_2, \ldots, X_k$ *such that* $f_{\restriction X_i}$ *is* linear *for every* X_i.

The most common activation function used today is the ReLU function, a PWL activation function defined below.

Definition 3. *The* rectified linear function *(ReLU) is a function* $\text{ReLU} : \mathbb{R}^n \to \mathbb{R}^m$ *defined component-wise by*

$$\text{ReLU}(\vec{v})_i \coloneqq \begin{cases} 0 & \text{if } v_i < 0 \\ v_i & \text{otherwise,} \end{cases}$$

where $\text{ReLU}(\vec{v})_i$ *is the ith component of the vector* $\text{ReLU}(\vec{v})$ *and* v_i *is the ith component of the vector* $\vec{v}$*.*

To show that ReLU is PWL, we must partition its input domain $\mathbb{R}^n$ so that, in each partition, ReLU is linear. In this case, we can use the orthants of $\mathbb{R}^n$ as our partitioning: within each orthant, the signs of the components do not change hence ReLU is the linear map that just zeros out the negative components.

Although we focus on ReLU due to its popularity and expository power, SyReNN works with a number of other popular PWL layers including MaxPool, Leaky ReLU, Hard Tanh, Fully-Connected, and Convolutional layers, as defined in [20]. PWL layers have become exceedingly common. In fact, nearly all of the state-of-the-art image recognition models bundled with PyTorch [48] are PWL.

Example 1. The DNN $f : \mathbb{R}^1 \to \mathbb{R}^1$ defined by

$$f(x) \coloneqq \begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \text{ReLU} \left(\begin{bmatrix} 1 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} \right)$$

can be broken into layers $f = f_3 \circ f_2 \circ f_1$ where

$$f_1(x) \coloneqq \begin{bmatrix} 1 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}, \quad f_2 = \text{ReLU}, \quad \text{and} \quad f_3(\vec{v}) = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \vec{v}.$$

The DNN's input-output behavior on the domain $[-1, 2]$ is shown in Figure 2.

3 A Symbolic Representation of DNNs

We formalize the symbolic representation according to the following definition:

Definition 4. *Given a PWL function* $f : \mathbb{R}^n \to \mathbb{R}^m$ *and a bounded convex polytope* $X \subseteq \mathbb{R}^n$*, we define the* symbolic representation of f on X*, written* $\widehat{f_{\restriction X}}$*, to be a finite set of polytopes* $\widehat{f_{\restriction X}} = \{P_1, \ldots, P_n\}$*, such that:*

1. *The set* $\{P_1, P_2, \ldots, P_n\}$ *partitions* X*, except possibly for overlapping boundaries.*
2. *Each* P_i *is a bounded convex polytope.*
3. *Within each* P_i*, the function* $f_{\restriction P_i}$ *is linear.*

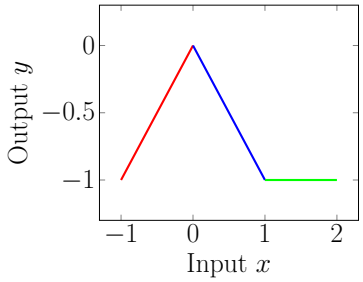


Fig. 2: Input-output behavior of the DNN from Example 1.

Notably, if f is a DNN using only PWL layers, then f is PWL and so we can define $\widehat{f_{\restriction X}}$. This symbolic representation allows one to reduce questions about the DNN f to questions about finitely many linear functions. For example, because linear functions are convex, to verify that $\forall \vec{x} \in X. \quad f(\vec{x}) \in Y$ for some polytope Y, it suffices to verify $\forall P_i \in \widehat{f_{\restriction X}}. \quad \forall \vec{v} \in \texttt{Vert}(P_i). \quad f(\vec{v}) \in Y$, where $\texttt{Vert}(P_i)$ is the (finite) set of vertices for the bounded convex polytope P_i; thus, here both of the quantifiers are over finite sets. The symbolic representation described above can be seen as a generalization of the ExactLine representation [55], which considered only *one-dimensional* restriction domains of interest. ExactLine is now included in SyReNN as an optimization for the case of one-dimensional input polytopes.

Example 2. Consider again the DNN $f : \mathbb{R}^1 \to \mathbb{R}^1$ given by

$$f(x) \coloneqq \begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \text{ReLU} \left(\begin{bmatrix} 1 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} \right)$$

and the region of interest $X = [-1, 2]$. The input-output behavior of f on X is shown in Figure 2. From this, we can see that

$$\widehat{f_{\restriction X}} = \{[-1, 0], [0, 1], [1, 2]\}.$$

Within each of these partitions, the input-output behavior is *linear,* which for $\mathbb{R}^1 \to \mathbb{R}^1$ we can see visually as just a line segment. As this set fully partitions X, then, this is a valid $\widehat{f_{\restriction X}}$.

4 Computing the Symbolic Representation on 2D Regions

This section presents an efficient algorithm for computing $\widehat{f_{\restriction X}}$ for a DNN f composed of PWL layers. To retain both scalability and precision, in this section we will *require the input region X be two-dimensional.* This design choice is relatively unexplored in the neural-network analysis literature (most analyses strike

Algorithm 1: Computing $\widehat{f_{\restriction X}}$ for a function f using the ExtendPWL$(\cdot,\cdot)$. Assumes f can be decomposed into the sequence of piecewise-linear maps (*layers*) of the form $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$.

Input: f, X
Output: $\widehat{f_{\restriction X}}$

```
s ← {X}                          // Holds f_i ∘ ··· ∘ f_1 ↾X after ith iteration.
for i ∈ 1, ··· n do
    if f_i is linear then
        s ← s                    // Linear layers do not impact linear regions
    else
        s ← ExtendPWL(f_i, s)    // PWL layers handled by Algorithm 2
return s
```

a balance between precision and scalability, ignoring dimensionality). We show that, for two-dimensional X, we can use an efficient polytope representation to produce an algorithm that demonstrates good best-case and in-practice efficiency while retaining full precision. This algorithm represents a direct generalization of the approach of [55].

The difficulties our algorithm addresses arise from three areas. First, when computing $\widehat{f_{\restriction X}}$ there may be exponentially many such partitions on all of $\mathbb{R}^n$ but only a small number of them may intersect with X. Consequently, the algorithm needs to be able to find those partitions that intersect with X efficiently without explicitly listing all of the partitions on $\mathbb{R}^n$. Second, it is often more convenient to specify the partitioning via *hyperplanes separating the partitions* than by explicit polytopes. For example, for the one-dimensional ReLU function we may simply state that the point $x = 0$ separates the two partitions, because ReLU is linear both in the region $x \leq 0$ and $x \geq 0$. Finally, neural networks are typically composed of sequences of linear and piecewise-linear layers, where the partitioning imposed by each layer individually may be well-understood but their composition is more complex. For example, identifying the linear partitions of $y = \text{ReLU}(4 \cdot \text{ReLU}(-3x - 1) + 2)$ is non-trivial, even though we know the linear partitions of each composed function individually.

Our algorithm only requires the user to specify the hyperplanes defining the partitioning for the activation function used in each layer, and the current implementation comes with common PWL activation functions built-in. For example, if a ReLU layer is used for an n-dimensional input vector, then the hyperplanes would be defined by the equations $x_1 = 0, x_2 = 0, \ldots, x_n = 0$. SyReNN computes the symbolic representation for a *single layer at a time*, composing them sequentially to compute the symbolic representation across the entire network.

To allow such compositions of layers, instead of directly computing $\widehat{f_{\restriction X}}$, we will define another primitive, denoted by the operator $\otimes$ and sometimes referred

to as ExtendPWL, such that

$$\text{ExtendPWL}(h, \widehat{g_{\restriction X}}) = h \otimes \widehat{g_{\restriction X}} = \widehat{h \circ g_{\restriction X}}. \tag{1}$$

Consider $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$, and let $I : x \mapsto x$ be the identity map. I is linear across its entire input space, and, thus, $\widehat{I_{\restriction X}} = \{X\}$. By the definition of $\text{ExtendPWL}(f_1, \cdot)$, we have $f_1 \otimes \widehat{I_{\restriction X}} = \widehat{(f_1 \circ I)_{\restriction X}} = \widehat{f_{1\restriction X}}$, where the final equality holds by the definition of the identity map I. We can then iteratively apply this procedure to inductively compute $\widehat{(f_i \circ \cdots \circ f_1)_{\restriction X}}$ from $\widehat{(f_{i-1} \circ \cdots \circ f_1)_{\restriction X}}$ like so:

$$f_i \otimes \widehat{(f_{i-1} \circ \cdots \circ f_1)_{\restriction X}} = \widehat{(f_i \circ f_{i-1} \circ \cdots \circ f_1)_{\restriction X}}$$

until we have computed $\widehat{(f_n \circ f_{n-1} \circ \cdots \circ f_1)_{\restriction X}} = \widehat{f_{\restriction X}}$, which is the required symbolic representation. Notably, linear functions do not change linear partitions, hence if f is linear then $f \otimes \widehat{g_{\restriction X}} = \widehat{f \circ g_{\restriction X}} = \widehat{g_{\restriction X}}$. This process is formalized in Algorithm 1.

4.1 Algorithm for ExtendPWL

Algorithm 2 presents an algorithm for computing ExtendPWL for arbitrary PWL functions, where $\text{ExtendPWL}(h, \widehat{g_{\restriction X}}) = h \otimes \widehat{g_{\restriction X}} = \widehat{h \circ g_{\restriction X}}$.

Geometric intuition for the algorithm. Consider the ReLU function (Definition 3). It can be shown that, within any orthant (i.e., when the signs of all coefficients are held constant), $\text{ReLU}(\vec{x})$ is equivalent to some linear function, in particular the element-wise product of $\vec{x}$ with a vector that zeroes out the negative-signed components. However, for our algorithm, all we need to know is that the linear partitions of ReLU (in this case the orthants) are separated by hyperplanes $x_1 = 0, x_2 = 0, \ldots, x_n = 0$.

Given a two-dimensional polytope X, the execution of the algorithm for $f = \text{ReLU}$ can be visualized as follows. We pick some vertex v of X, and begin traversing the boundary of the polytope in counter-clockwise order. If we hit an orthant boundary (corresponding to some hyperplane $x_i = 0$), it implies that the behavior of the function behaves differently at the points of the polytope to one side of the boundary from those at the other side of the boundary. Thus, we *partition X into X_1 and X_2*, where X_1 lies to one side of the hyperplane and X_2 lies to the other side. We recursively apply this procedure to X_1 and X_2 until the resulting polytopes all lie on exactly one side of every hyperplane (orthant boundary). But lying on exactly one side of every hyperplane (orthant boundary) implies each polytope lies entirely within a linear partition of the function (a single orthant), hence the application of the function on that polytope is linear, and hence we have our partitioning.

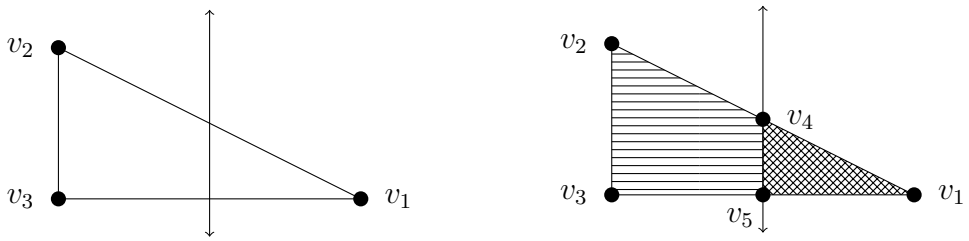
Functions used in algorithms. Given a two-dimensional polytope X, `Vert(`X`)` returns a list of its vertices in counter-clockwise order, repeating the initial vertex at the end. Given a set of points S, `ConvexHull(`S`)` represents their convex hull (the smallest polytope containing every point in S). Given a scalar value x,

`Sign`(x) computes the sign of that value (i.e., -1 if $x < 0$, $+1$ if $x > 0$, and 0 if $x = 0$). Given an n-dimensional polytope P, Facets(P) is the set of $n-1$-dimensional facets that make up the boundary of P.

Algorithm description. The key insight of the algorithm is to recursively partition the polytopes until such a partition lies entirely within a linear region of the function f. Algorithm 2 begins by constructing a queue containing the polytopes of $\widehat{g_{\restriction X}}$. Each iteration either removes a polytope from the queue that lies entirely in one linear region (placing it in Y), or splits (partitions) some polytope into two smaller polytopes that get put back into the queue. When we pop a polytope P from the queue, Line 6 iterates over all hyperplanes $N_k \cdot x = b_k$ defining the piecewise-linear partitioning of f, looking for any for which some vertex V_i lies on the positive side of the hyperplane and another vertex V_j lies on the negative side of the hyperplane. If none exist (Line 7), by convexity we are guaranteed that the entire polytope lies entirely on one side with respect to every hyperplane, meaning it lies entirely within a linear partition of f. Thus, we can add it to Y and continue. If two such vertices are found (starting Line 10), we then call `SplitPlane()` (Algorithm 3) to actually partition the polytope on opposite sides of the hyperplane, adding both to our worklist. `SplitPlane()` works by finding "extreme" i and j indices such that V_i is the last vertex in a counter-clockwise traversal to lie on the same side of the hyperplane as V_1 and V_j is the last vertex lying on the opposite side of the hyperplane, then solving for the new vertex where that edge intersects the hyperplane.

In the best case, each partition is in a single orthant. Then, the algorithm never calls `SplitPlane()` at all — it merely iterates over all of the n input partitions, checks their v vertices, and appends to the resulting set (for a best-case complexity of $O(nv)$). In the worst case, it splits each polytope in the queue on each hyperplane, resulting in exponential time complexity. As we will show in Section 8, this exponential worst-case behavior is not encountered in practice, thus making SyReNN a practical tool for DNN analysis.

Example of the algorithm. Consider the polytope shown in Figure 3a with vertices $\{v_1, v_2, v_3\}$, and suppose our activation function has two piecewise-linear regions separated by the vertical line (1D hyperplane) $Nx + b = 0$ shown.



(a) Before extending (b) After extending

Fig. 3: Diagrams demonstrating the 2D ExtendPWL algorithm

Algorithm 2: Ordered ExtendPWL for computing $f \otimes \widehat{g_{\restriction X}}$ for two-dimensional X. f is a PWL function such that the hyperplanes $N_1 \cdot x = b_1$ through $N_m \cdot x = b_m$ impose a partitioning of the space where f is equivalent to some linear function within any partition. This is a specialization of Algorithm 5 for two-dimensional X, a discussion of differences is available in Section 6.

Input: $\widehat{g_{\restriction X}} = \{P_1, \ldots, P_n\}$, hyperplanes $N_k \cdot x = b_k$ for $k \in [1, m]$.
Output: $\widehat{f \circ g_{\restriction X}}$

```
1  W ← ConstructQueue(ĝ_{↾X})
2  Y ← ∅            // Polytopes that lie entirely in one linear region.
3  while W not empty do
4      P ← Pop(W)
5      V ← Vert(P)
6      K ← {(N_k, b_k) | ∃V_i, V_j ∈ V : Sign(N_k · g(V_i) − b_k) > 0 ∧
                                          Sign(N_k · g(V_j) − b_k) < 0}
7      if K = ∅ then
8          Y ← Y ∪ {P}
9          continue
10     N, b ← any element from K
11     for V' ∈ SplitPlane(V, g, N, b) do
12         W ← Push(W, ConvexHull(V'))
13 return Y
```

Because this hyperplane has some of the vertices of the polytope on one side and some on the other, we will use it as the N, b hyperplane on Line 10. Then SplitPlane is called. We will assume the vertices are ordered so that the extremal vertices found starting at Line 1 are $V_i = v_1$ and $v_j = v_3$. SplitPlane will then add new vertices $p_i = v_4$ (shown in Figure 3b) where the edge $v_1 \to v_2$ intersects the hyperplane, as well as $p_j = v_5$ where the edge $v_3 \to v_1$ intersects the hyperplane. Separating all of the vertices on the left of the hyperplane from those on the right, we find that this has partitioned the original polytope into two sub-polytopes, each on exactly one side of the hyperplane, as desired. If there were more intersecting hyperplanes, we would then recurse on each of the newly-generated polytopes to further subdivide them by the other hyperplanes.

Proofs of correctness. The two theorems below formally argue for correctness of Algorithm 3 and Algorithm 2. They are a special case of the arguments in Section 6.

Theorem 1. *Algorithm 3 correctly splits a 2D polytope* $\texttt{ConvexHull}(V)$ *by the hyperplane* $Nx = b$.

Proof. The intuition was described earlier in this section. Formally, Algorithm 3 is the special case of Algorithm 6 for $k = 2$. See Theorem 4 for the proof of its correctness.

Algorithm 3: SplitPlane(V, g, N, b)

Input: V, the vertices of the polytope in the input space of g. A function g. N and b define the hyperplane $N \cdot x = b$ to split on.

Output: $\{P_1, P_2\}$, two sets of vertices whose convex hulls form a partitioning of V such that each lies on only one side of the $N \cdot x = b$ hyperplane.

$i \leftarrow \arg\max_i \{\texttt{Sign}(N \cdot g(V_i) - b) = \texttt{Sign}(N \cdot g(V_1) - b)\}$

$j \leftarrow \arg\max_j \{\texttt{Sign}(N \cdot g(V_j) - b) \neq \texttt{Sign}(N \cdot g(V_i) - b)\}$

$p_i \leftarrow V_i + \frac{b - N \cdot g(V_i)}{N \cdot (g(V_{i+1}) - g(V_i))}(V_{i+1} - V_i)$

$p_j \leftarrow V_j + \frac{b - N \cdot g(V_j)}{N \cdot (g(V_{j+1}) - g(V_j))}(V_{j+1} - V_j)$

$A \leftarrow \{p_i, p_j\} \cup \{v \in V \mid \texttt{Sign}(N \cdot v - b) = \texttt{Sign}(N \cdot V_i - b)\}$

$B \leftarrow \{p_i, p_j\} \cup \{v \in V \mid \texttt{Sign}(N \cdot v - b) = \texttt{Sign}(N \cdot V_j - b)\}$

return $\{A, B\}$

This special-casing relies on one key observation, which is that, in Algorithm 6 for $k = 2$, exactly two of the facets (edges) will cross the hyperplane. To see why, first consider the restriction of the hyperplane $Nx = b$ to the polytope $\texttt{ConvexHull}(V)$. Notice that, assuming the hyperplane actually crosses $\texttt{ConvexHull}(V)$ (i.e., does not contain it entirely), then this restriction will correspond to a line crossing a polytope. But, if this line crossed three distinct edges, then it would have at least three distinct intersection points with the polytope. But these three points must make a line, meaning one of those points must be between the other two, meaning one of those edges must have points that can be written as a convex combination of those on the other two edges, i.e., one of the edges must be on the *interior* of the polytope, a contradiction. □

Theorem 2. *Algorithm 2 correctly computes* $f \otimes \widehat{g_{\restriction X}}$.

Proof. The intuition was described earlier in this section. For a formal correctness proof, see the proof of the generalized Theorem 5, from which this claim follows as a special case. The only difference in the algorithms is that we call the 2D-specialized SplitPlane instead of SplitHyperPlane[2]. □

4.2 Representing Polytopes

We close this section with a discussion of implementation concerns when representing the polytopes that make up the partitioning of $\widehat{f_{\restriction X}}$. In standard computational geometry, polytopes can be represented in two equivalent forms:

1. The *half-space* or *H-representation*, which encodes the polytope as an intersection of finitely many half-spaces. (Each half-space being defined by a linear inequality $ax \leq b$.)
2. The *vertex* or *V-representation*, which encodes the polytope as a set of finitely many points; the polytope is then taken to be the convex hull of the points (i.e., smallest convex shape containing all of the points).

Certain operations are more efficient when using one representation compared to the other. For example, finding the intersection of two polytopes in an H-representation can be done in linear time by conjoining their representative linear constraints, but the same is not possible in V-representation. On the other hand, checking if a polytope in V-representation is empty is as simple as checking whether it has any vertices at all, while the same query for a polytope in H-representation requires solving a considerably more expensive linear programming problem.

There are three main operations on polytopes we need perform in our algorithms: (i) identifying which hyperplanes a polytope intersects, (ii) splitting a polytope with a hyperplane, and (iii) applying a linear map to all points in the polytope. In general, the first and third are significantly more efficient in a V-representation, while the second is often more efficient in an H-representation. In particular, (i) is linear time on a V-representation but requires solving a complicated linear programming problem on an H-representation. Similarly, unless the linear map happens to be invertible, which almost never happens for DNN weight matrices which are usually learned and between different-dimensional spaces, (iii) is only as expensive as a matrix multiplication on a V-representation, but requires a doubly-exponential variable elimination algorithm such as Fourier-Motzkin [68] for polytopes in the H-representation.

While it is true in general that splitting a V-representation polytope with a hyperplane is difficult, when restricted to two-dimensional polytopes, it is actually efficient in a V-representation, as demonstrated by Algorithm 3, helping to motivate our use of the V-representation in our algorithm. Our algorithm shows how to do polytope splitting efficiently for two-dimensional polytopes embedded in *any* dimensional space, i.e., even though the polytopes are two-dimensional, they (and their vertices) live in a much higher-dimensional space.

Furthermore, even though it is easy to split an H-representation polytope, determining when such a split leads to a non-empty polytope requires solving a relatively expensive LP problem. This is exacerbated by the fact that, although the polytopes in question are two-dimensional, they lie within a much higher-dimensional space. This means that the corresponding LP problem will have thousands of variables, even if the actual polytope it describes has at most two dimensions. This significantly increases overhead in the H-representation approach, a problem neatly avoided by specifying the vertices directly in the V-representation.

Comparing performance. To better quantify the performance characteristics of both representations for our particular application area, we evaluated two different operations in the H-representation and compare to our results using the V-representation in Section 8.

First, we used an off-the-shelf Fourier-Motzkin implementation [17] to transform a single two-dimensional input polytope through the first layer of the ACAS Xu network as evaluated in Section 8.1, except using the H-representation. Transforming even this single polytope requires over 0.1 seconds on modern hardware. Such transformation operations would have to be performed *at least* once for

each of the approximately 30,000 linear regions identified by SyReNN in Section 8.1 and Table 1, leading to almost an hour of solving time compared to 0.1 seconds *for the entire run* with SyReNN using a V-representation.

Alternatively, a different way to implement SyReNN with H-representation polytopes but avoiding the expensive projection step would be to perform the entire analysis in a space with as many dimensions as DNN nodes. This would avoid projection, but still relies on an LP solver to determine which linear regions, i.e., assignment of activated nodes, are feasible. We performed a similar experiment, using the Gurobi LP solver [26] to identify a linear region of the DNN by iteratively asking the LP solver if there exists an input that causes the ith DNN node to be activated. If so, we modify the LP to assert it is activated and continue to the $(i+1)$th node. Otherwise, we assert it is not activated and again continue to the $(i+1)$th node. Even with a state-of-the-art LP solver, Gurobi solving time for even a single linear region of the network is over 0.3 seconds, which is larger than it takes our V-representation SyReNN to identify over 30,000 linear regions.

While it is likely that optimizations may reduce the runtime for an H-representation based approach, it is unlikely to make up the four-orders-of-magnitude difference in runtime demonstrated by these experiments. Furthermore, such approaches require the use of complicated polytope projection and feasibility solvers, compared to the relatively simple SplitPlane algorithm used by SyReNN.

Numerical precision. Furthermore, the two polytope representations have different resiliency to floating-point operations. In particular, H-representations for polytopes in $\mathbb{R}^n$ are notoriously difficult to achieve high-precision with, because the error introduced from using floating point numbers gets arbitrarily large as one goes in a particular direction along any hyperplane facet. Ideally, we would like the hyperplane to be most accurate in the region of the polytope itself, which corresponds to choosing the magnitude of the norm vector correctly. Unfortunately, to our knowledge, there is no efficient algorithm for computing the ideal floating point H-representation of a polytope, although libraries such as APRON [34] are able to provide reasonable results for low-dimensional spaces. However, because neural networks utilize extremely high-dimensional spaces (often hundreds or thousands of dimensions) and we wish to iteratively apply our analysis, we have found that errors from using floating-point H-representations can quickly multiply and compound to become infeasible. By contrast, floating-point inaccuracies in a V-representation are directly interpretable as slightly misplacing the vertices of the polytope; no "localization" process is necessary to penalize inaccuracies close to the polytope more than those far away from it.

Another difference is in the space complexity of the representation. In general, H-representations can be more space-efficient for common shapes than V-representations. However, when the polytope lies in a low-dimensional subspace of a larger space, the V-representation is usually significantly more efficient.

Thus, V-representations are a good choice for low-dimensionality polytopes embedded in high-dimensional space, which is exactly what we need for analyzing

neural networks with two-dimensional restriction domains of interest. This is why we designed our algorithms to rely on $\texttt{Vert}(X)$, so that they could be directly computed on a V-representation. Importantly, our algorithm operates entirely within the V-representation: we never need to convert between them. This is particularly nice when the polytope lies in a two-dimensional subspace, as storing the vertices in counter-clockwise order allows us to recover the edges from the vertices. Meanwhile, converting to a full H-representation would likely incur significant overhead due to the very high-dimensional space the polytope is embedded in.

5 Batched ExtendPWL Algorithm

This section presents Batched ExtendPWL algorithm for computing $f \otimes \widehat{g_{\restriction X}}$ for two-dimensional X. f is a PWL function such that the hyperplanes impose a partitioning of the space where f is equivalent to some linear function within any partition. The algorithm is presented in Algorithm 4. The batched nature of the algorithm allows it to exploit GPU-style parallelism. In this algorithm, we introduce the notion of edges of polytopes (1-dimensional faces). $\texttt{Edges}(P)$ returns the set of edges of the polytope P. The algorithm begins by initializing the result Y with $\widehat{g_{\restriction X}}$ (Line 1). Each of the following iterations over all $k \in [1, m]$ hyperplanes (Line 2-10) splits each polytope in Y by the kth hyperplane and adds the resulting partitions back to Y. This ensures that, after the kth iteration, none of the polytopes in Y cross the first k hyperplanes. During iteration k, the algorithm first collects all vertices (Line 3) and edges (Line 6) of all polytopes in Y as V and E, respectively. The algorithm then computes a map $\texttt{S}$ from any vertex $v \in V$ to its sign regarding the working hyperplane k (Line 5). With the sign map $\texttt{S}$, the algorithm then identifies edges that cross the kth hyperplane (Line 6) as $E^{\times}$. $\texttt{CrossPlane}$ checks if the two endpoints of edge e has different signs. Then the algorithm computes a map from each such edge e in $E^{\times}$ to its intersection vertex with the hyperplane k as $\texttt{I}$ by interpolation (Line 7). The algorithm then collects polytopes p in Y that intersect with the kth hyperplane as $P^{\times}$ (Line 8), splits each of them using SplitPlane (presented later) and collects all partitions as P^{Δ} (Line 9). The kth iteration ends by replacing intersected polytopes in Y with their partitions regarding the hyperplane k.

The procedure SplitPlane (Line 12–29) splits the polytope p into two halves, one on either side of the kth hyperplane, using the sign map $\texttt{S}$ and intersection map $\texttt{I}$. It begins by initializing the sets *pos* and *neg* to contain edges of polytope p which lie on the positive and negative side of the hyperplane k, with empty sets (Line 13), as well as the set $V^{\times}$, which will contain the vertices lying on the kth hyperplane itself (shared by both partitions), with empty set (Line 14). The procedure then iterates over edges of polytope k (Line 15-26). The iteration for edge e begins by checking if it crosses the hyperplane k (Line 16). If so, the procedure first adds the intersection vertex $\texttt{I}(e)$ to $V^{\times}$ (Line 17). Then the procedure splits the intersected edge e by the intersection vertex $\texttt{I}(e)$ as new edges $(e.u, \texttt{I}(e))$ and $(e.v, \texttt{I}(e))$ (Line 18), where $e.u$ and $e.v$ are two endpoints

Algorithm 4: Batched ExtendPWL for computing $f \otimes \widehat{g_{\restriction X}}$ for two-dimensional X. f is a PWL function such that the hyperplanes $N_1 \cdot x = b_1$ through $N_m \cdot x = b_m$ impose a partitioning of the space where f is equivalent to some linear function within any partition.

Input: $\widehat{g_{\restriction X}} = \{P_1, \ldots, P_n\}$.
Output: $\widehat{f \circ g_{\restriction X}}$
`/* Interatively split` $\widehat{g_{\restriction X}}$ `by hyperplanes */`
$Y \leftarrow \widehat{g_{\restriction X}}$
for $k \in [1, m]$ **do**
  $V \leftarrow \bigcup_{P_i \in Y} \texttt{Vert}(P_i)$
  $E \leftarrow \bigcup_{P_i \in Y} \texttt{Edges}(P_i)$
⋆ $\texttt{S} \leftarrow \left\{v \mapsto \texttt{Sign}(N_k \cdot g(v) - b_k) \mid v \in V\right\}$ `// Sign map for vertices.`
⋆ $E^\times \leftarrow \left\{e \mid e \in E, \texttt{CrossPlane}(e, \texttt{S})\right\}$ `// Intersected edges.`
⋆ $\texttt{I} \leftarrow \left\{e \mapsto \texttt{Interp}(e, N_k, b_k) \mid e \in E^\times\right\}$ `// Intersection vertex map.`
⋆ $P^\times \leftarrow \left\{p \mid p \in Y, \texttt{Edges}(p) \cap E^\times \neq \emptyset\right\}$ `// Intersected polytopes.`
◇ $P^\Delta \leftarrow \bigcup_{p \in P^\times} \text{SplitPlane}(p, \texttt{S}, \texttt{I})$
  $Y \leftarrow (Y \backslash P^\times) \cup P^\Delta$
return Y
def SplitPlane(p, S, I)**:**
  $pos, neg \leftarrow \emptyset, \emptyset$ `// Edges lie in the positive/negative side.`
  $V^\times \leftarrow \emptyset$ `// Intersection vertices on the face.`
  for $e \in \texttt{Edges}(P)$ **do**
    if $\texttt{CrossPlane}(e, \texttt{S})$ **then**
      $V^\times \leftarrow V^\times \cup \texttt{I}(e)$
      for $e' \in \{(e.u, \texttt{I}(e)), (e.v, \texttt{I}(e))\}$ **do**
        if $\texttt{OnPosSide}(e, \texttt{S})$ **then**
          $pos \leftarrow pos \cup e'$
        else
          $neg \leftarrow neg \cup e'$
    else if $\texttt{OnPosSide}(e, \texttt{S})$ **then**
      $pos \leftarrow pos \cup e'$
    else
      $neg \leftarrow neg \cup e'$
  $pos \leftarrow pos \cup \texttt{ConvexHull}(V^\times)$
  $neg \leftarrow neg \cup \texttt{ConvexHull}(V^\times)$
  return $\{\text{FaceHull}(pos), \text{FaceHull}(neg)\}$

of edge e. For each new edge e', the procedure adds e' to the *pos* set if it is on the positive side of the hyperplane k (Line 19-20), otherwise to the *neg* set (Line 21-22). If the iterating edge e does not cross hyperplane k, then the procedure adds e to the *pos* set if it is on the positive side of the hyperplane k (Line 23-24), otherwise to the *neg* set (Line 25-26). After the iterations, the procedure ends by adding the convex hull of intersected vertices $V^\times$ to both *pos* and *neg* (Line 27-28), and returns the set of face hulls of *pos* and *neg* (Line 29).

Theorem 3. *Algorithm 4 correctly computes* $f \otimes \widehat{g_{\restriction X}}$.

Proof. It suffices to show that the algorithm correctly partitions each input polytope P such that the signs within a partition are constant. Notably, because of convexity, it suffices to show that the signs of the vertices of each partition are constant.

In iteration k, Algorithm 4 splits all polytopes which cross the hyperplane k and maintains the invariant that each SplitPlane splits an intersected polytope into two new ones such that the signs within each partition regarding hyperplanes $[1, k]$ are constant. This invariant ensures that the algorithm is correct. □

Algorithm 4 is optimized to utilize GPU. Specifically, lines marked by $\star$ (Line 5-8) can be parallelized on GPU and lines marked by $\diamond$ (Line 9) can be parallelized on CPU. Thus, one would expect that this algorithm when executed using a GPU would perform significantly better than the same algorithm on a CPU; we empirically confirm this in Section 8.2. Furthermore, we also see that this Batched ExtendPWL algorithm when executed using a GPU outperforms the Ordered ExtendPWL algorithm (2) on a CPU. Two major overheads when using GPU are data movement and kernel launches. To avoid redundant data movement between GPU memory and CPU memory, we maintain vertices as a matrix on GPU memory and only access it from GPU. We also maintain the endpoints of all edges in a contiguous GPU memory. There are only two floating-point computations; viz., computing the signs of vertices and the intersection vertices of the intersected edges. To avoid redundant launches of the GPU kernel, we perform these computations in batches, and cache the results in maps for later use (Line 5 and 7).

6 Extending to Higher-Dimensional Regions

The 2D algorithm described in Section 4.1 can be seen as implementing the recursive case of a more general, n-dimensional version of the algorithm that recurses on each of the $(n-1)$-dimensional facets. In 2D, we trace the edges (1D faces) and use the 1D algorithm from [55] to subdivide them based on intersections with the hyperplanes defining the function.

In this section, we will describe the more general n-dimensional recursive algorithm. This generalized algorithm is given in Algorithm 5 and Algorithm 6.

The overall operation of Algorithm 5 is identical to that of Algorithm 2, as we repeatedly split the input polytopes by the hyperplanes until each resulting

Algorithm 5: $f \otimes \widehat{g_{\restriction X}}$ for n-dimensional X. The hyperplanes $N_1 \cdot x = b_1$ through $N_m \cdot x = b_m$ are such that f is equivalent to some linear function within any partition imposed by these hyperplanes.

Input: $\widehat{g_{\restriction X}} = \{P_1, \ldots, P_n\}$, hyperplanes $N_k \cdot x = b_k$ for $k \in [1, m]$.
Output: $\widehat{f \circ g_{\restriction X}}$

```
W ← ConstructQueue(ĝ|X)
Y ← ∅            // Polytopes that lie entirely in one linear region.
while W not empty do
    P ← Pop(W)
    V ← Vert(P)
    K ← {(N_k, b_k) | ∃V_i, V_j ∈ V : Sign(N_k · g(V_i) − b_k) > 0 ∧
                                      Sign(N_k · g(V_j) − b_k) < 0}
    if K = ∅ then
        Y ← Y ∪ {P}
        continue
    N, b ← any element from K
    for V' ∈ SplitHyperPlane[n](P, g, N, b) do
        W ← Push(W, ConvexHull(V'))
return Y
```

partition lies in exactly one linear region, i.e., on exactly one side of each hyperplane. The key difference is that now the polytopes are no longer 2D, hence, we can no longer use the 2D-optimized `SplitPlane()` and instead call the more general SplitHyperPlane$[k](P, g, N, b)$ defined in Algorithm 6.

The rest of the changes occur in Algorithm 6. The goal of this algorithm is to take a polytope P and split it into (at most) two polytopes P_1 and P_2, such that each P_i lies entirely on one side of the hyperplane defined by $Nx \leq b$.

In the one-dimensional base case, this is relatively simple, as we can directly compute the point m at which the line interpolating between the endpoints s, e of the one-dimensional polytope (line segment) P crosses the hyperplane.

In the n-dimensional recursive case, we recurse on each facet of P. We then partition each of those *facets* into sub-facets F_1, F_2 such that each sub-facet lies on entirely one side of the hyperplane. We collect all of the sub-facets that lie on one side into the set l and those that lie on the other side into the set g. Then B collects the vertices that lie on the border, i.e., $\texttt{ConvexHull}(B) = P \cap \{x \mid Nx = b\}$. Now, consider the surfaces of our desired partitions P_1 and P_2. Every point on the surface of P_1 is either a point on the surface of P that lies on one side of the hyperplane (i.e., in one of the l facets), or it is a point on the intersection $\texttt{ConvexHull}(B)$. Therefore, FaceHull(l) gives a satisfying P_1 and FaceHull(u) a satisfying P_2.

We have experimented with such approaches, but found that the overhead of keeping track of all $(n-k)$-dimensional faces (commonly known as the *face poset* or *combinatorial structure* [15] of a polytope) was too large in higher dimensions. The two-dimensional algorithm addresses this concern by storing the

Algorithm 6: SplitHyperPlane$[k](P, g, N, b)$

Input: P, the polytope in the input space of g. A function g. The dimensionality k of the polytope P. N and b define the hyperplane $N \cdot x = b$ to split on.

Output: $\{P_1, P_2\}$, two polytopes forming a partitioning of P such that each lies on only one side of the $N \cdot x = b$ hyperplane.

if $k = 1$ **then**
  `/* Base case */`
  $s, e \leftarrow \text{Vert}(P)$
  if $(Ng(s) - b)\,(Ng(e) - b) < 0$ **then**
    $m \leftarrow s + \frac{b - N \cdot g(s)}{N \cdot (g(e) - g(s))}(e - s)$
    return $\{\texttt{ConvexHull}(\{s, m\}), \texttt{ConvexHull}(\{m, e\})\}$
  return $\{P\}$
`/* Recursive case */`
$l, u \leftarrow \emptyset, \emptyset$
$B \leftarrow \emptyset$
for $F \in \text{Facets}(P)$ **do**
  for $S \in \text{SplitHyperPlane}[k-1](F, g, N, b)$ **do**
    $B \leftarrow B \cup \{v \in \text{Vert}(S) \mid Nv = b\}$
    if $\texttt{OnPosSide}(S, N, b)$ **then**
      $u \leftarrow u \cup \{S\}$
    else
      $l \leftarrow l \cup \{S\}$
$l \leftarrow l \cup \{\texttt{ConvexHull}(B)\}$
$u \leftarrow u \cup \{\texttt{ConvexHull}(B)\}$
return $\{\text{FaceHull}(l), \text{FaceHull}(u)\} \setminus \{\emptyset\}$

combinatorial structure *implicitly*, representing 2D polytopes by their vertices in counter-clockwise order, from which edges correspond exactly to sequential vertices. To our knowledge, such a compact representation allowing arbitrary $(n-k)$-dimensional faces to be read off is not known for higher-dimensional polytopes.

Theorem 4. *Algorithm 6 correctly splits a polytope* $\texttt{ConvexHull}(V)$ *by the hyperplane* $Nx = b$.

Proof. In the $k = 1$ case, we have two options:

- If both endpoints are on the same side of the hyperplane, then the splitting is just the polytope (line segment) itself, so we return $\{P\}$ correctly.
- Otherwise, we wish to split the line segment P defined by $s + \alpha(e - s)$ by the intersecting, non-parallel line defined by $Nx = b$. This intersection point can be computed algebraically to be the point m as shown, leaving the two partition segments $\texttt{ConvexHull}(\{s, m\})$ and $\texttt{ConvexHull}(\{m, e\})$ returned.

Next, the recursive case. Notice that the algorithm only ever partitions polytopes (never adding new points outside the input polytope), so FaceHull(l) $\cup$ FaceHull(u) $\subseteq P$.

Therefore, it suffices to show that the facets in l form the surface of $L = P \cap \{x \mid Nx \leq b\}$ and the facets in u form the surface of $U = P \cap \{x \mid Nx \geq b\}$. We will argue for l here, u is analogous.

Consider any point x on the surface of L. Either (i) x is also on the surface of P, or (ii) x is interior to P, and thus must be on the facet $P \cap \{x \mid Nx = b\}$.

Case (i) implies x will be on some facet F of P, and thus x will be in the sub-facet F_l of F intersected with the lower-half L. This facet is included in l by line 13.

Case (ii) implies x will be on $P \cap \{x \mid Nx = b\}$. Notice that this is a single facet, hence its vertices will all be adjacent to other facets. Hence, each of the vertices will be added to set B on line 11. So these points will be in `ConvexHull`(B), and thus in l as of line 16.

Therefore, all of the surface points are accounted for in the facet sets and therefore the returned polytopes are supersets of the desired P_1, P_2. We already saw that they are also subsets, hence they are the desired partitions P_1 and P_2. □

Theorem 5. *Algorithm 5 correctly computes* $f \otimes \widehat{g_{\restriction X}}$.

Proof. It suffices to show that the algorithm correctly partitions each input polytope P such that the signs within a partition are constant. Notably, because of convexity, it suffices to show that the signs of the vertices of each partition are constant.

We maintain two invariants every time we process some polytopes from the queue. The first is that the corresponding polytope will only be added to Y if the signs of all vertices are constant (or zero). The second is that at each step, we partition the polytope into two new ones (using `SplitPlane`) such that fewer sign switches happen in each than the original polytope. This follows from the correctness of the SplitPlane algorithm.

The first invariant ensures that, if it halts, the algorithm is correct. The second ensures that it will halt, as there are finitely many dimensions to consider. □

7 SyReNN tool

This section provides more details about the design and implementation of our tool, SyReNN (Symbolic Representations of Neural Networks), which computes $\widehat{f_{\restriction X}}$, where f is a DNN using only piecewise-linear layers and X is a union of one- or two-dimensional polytopes. The tool is available under the MIT license at https://github.com/95616ARG/SyReNN_GPU.

Input and output format. SyReNN supports reading DNNs from two standard formats: ERAN (a textual format used by the ERAN project [1]) as well as

ONNX (an industry-standard format supporting a wide variety of different models) [47]. Internally, the input DNN is described as an instance of the `Network` class, which is itself a list of sequential `Layer`s. A number of layer types are provided by SyReNN, including `FullyConnectedLayer`, `ConvolutionalLayer`, and `ReLULayer`. To support more complicated DNN architectures, we have implemented a `ConcatLayer`, which represents a concatenation of the output of two different layers. The input region of interest, X, is defined as a polytope described by a list of its vertices in counter-clockwise order. The output of the tool is the symbolic representation $\widehat{f_{\restriction X}}$.

Overall architecture. We designed SyReNN in a client-server architecture using gRPC [22] and protocol buffers [23] as a standard method of communication between the two. This architecture allows the bulk of the heavy computation to be done in efficient C++ code, while allowing user-friendly interfaces in a variety of languages. It also allows practitioners to run the server remotely on a more powerful machine if necessary. The C++ server implementation uses the Intel TBB library for parallelization. Our front-end library `pysyrenn` is written in Python. The entire project can be built using the Bazel build system.

Server architecture. The major algorithms are implemented as a gRPC server written in C++. When a connection is first made, the server initializes the state with an empty DNN $f(x) = x$. During the session, three operations are permitted: (i) append a layer g so that the current session's DNN is updated from f_0 to $f_1(x) \coloneqq g(f_0(x))$, (ii) compute $\widehat{f_{\restriction X}}$ for a one-dimensional X, or (iii) compute $\widehat{f_{\restriction X}}$ for a two-dimensional X. We have separate methods for one- and two-dimensional X, because the one-dimensional case has specific optimizations for controlling memory usage. The `SegmentedLine` and `UPolytope` types are used to represent one- and two-dimensional partitions of X, respectively. When operation (i) is performed, a new instance of the `LayerTransformer` class is initialized with the relevant parameters and added to a running `vector` of the current layers. When operation (ii) is performed, a new queue of `SegmentedLine`s is constructed, corresponding to X, and the before-allocated `LayerTransformer`s are applied sequentially to compute $\widehat{f_{\restriction X}}$. In this case, extra control is provided to automatically gauge memory usage and pause computation for portions of X until more memory is made available. Finally, when operation (iii) is performed, a new instance of `UPolytope` is initialized with the vertices of X and the `LayerTransformer`s are again applied sequentially to compute $\widehat{f_{\restriction X}}$. In this case, SyReNN can optimize the memory layout of `UPolytope`s and `LayerTransformer`s for GPU as well as perform the GPU-based algorithm if the client asks for it.

Client architecture. Our Python client exposes an interface for defining DNNs similar to the popular Sequential-Network Keras API [10]. Objects represent individual layers in the network, and they can be combined sequentially into a `Network` instance. The key addition of our library is that this `Network` exposes methods for computing $\widehat{f_{\restriction X}}$ given a V-representation description of X. To do

this, it invokes the server and passes a layer-by-layer description of f followed by the polytope X, then parses the response $\widehat{f_{\restriction X}}$.

Extending to support different layer types. Different layer types are supported by sub-classing the `LayerTransformer` class. Instances of this class expose a method for computing ExtendPWL$(h, \cdot)$ for the corresponding layer h. To simplify implementation, two sub-classes of `LayerTransformer` are provided: one for linear layers (such as fully-connected and convolutional layers), and one for piecewise-linear layers. For linear layers, all that needs to be provided is a method computing the layer function itself. For piecewise-linear layers, two methods need to be provided: (i) computing the layer function itself, and (ii) describing the hyperplanes which separate the linear regions. The base class then directly implements Algorithm 2 for that layer. At least one CPU and one GPU implementation should be provided. This architecture makes supporting new layers a straight-forward process.

Float safety. Like Reluplex [36], SyReNN uses floating-point arithmetic to compute $\widehat{f_{\restriction X}}$ efficiently. Unfortunately, this means that in some cases its results will not be entirely precise when compared to a real-valued or multiple-precision version of the algorithm. If a perfectly precise solution is required, the server code can be modified to use multiple-precision rationals instead of floats. Alternatively, a confirmation pass can be run using multiple-precision numbers after the initial float computation to confirm the accuracy of its results. The use of over-approximations may also be explored for ensuring correctness with floating-point evaluation, like in DeepPoly [54]. Unfortunately, our algorithm does not directly lift to using such approximations, since they may blow the originally-2D region into a higher-dimensional (but very "flat") over-approximate polytope, preventing us from applying the 2D algorithm for the next layer.

Usage examples. The tool user will begin by loading a model from disk, like

```
from pysyrenn import Network
network = Network.from_file("model.eran")
```

We can use this network to compute the network on finite input points like so

```
import numpy as np
in_1 = np.array([1., -1., 0.5])
in_2 = np.array([2., -1.5, 2.5])
out_1, out_2 = network.compute([in_1, in_2])
```

Or we can compute the behavior of the network on *all* points between in_1 and in_2 like so:

```
syrenn_endpoints = network.exactline(in_1, in_2,
                                     compute_preimages=True,
                                     include_post=False)
```

Here, syrenn_endpoints is a Numpy array where each element is an endpoint between in_1 and in_2, defining the partitioning $\widehat{f_{\restriction X}}$. The include_post option

allows us to avoid re-computing the DNN on those endpoints, if its output is important:

```
pre, post = network.exactline(in_1, in_2,
                              compute_preimages=True,
                              include_post=True)
# is equivalent to...
pre = network.exactline(in_1, in_2,
                        compute_preimages=True,
                        include_post=False)
post = network.compute(pre)
```

Meanwhile, compute_preimages controls whether the preimages will be relative (a ratio between 0 and 1) or absolute (a point between in_1 and in_2).

Similarly, given the vertices of a 2D polytope, we can compute the $\widehat{f_{\restriction X}}$ like so

```
in_3 = np.array([4., 1., 2.])
polytope = [in_1, in_2, in_3]
syrenn = network.transform_plane(polytope,
                                 compute_preimages=True,
                                 include_post=False)
```

The resulting variable syrenn will be a list of partitions, which each partition being an array of its vertices.

8 Applications of SyReNN

This section presents the use of SyReNN in four example case studies.

8.1 Visualization of Decision Boundaries for ACAS Xu Networks

Our first major task is *visualizing* the decision boundaries of a DNN on *infinitely many* input points. Figure 4 shows a visualization of an ACAS Xu DNN [35] which takes as input the position of an airplane and an approaching attacker, then produces as output one of five advisories instructing the plane, such as "clear of conflict" or to move "weak left." Every point in the diagram represents the relative position of the approaching plane, while the color indicates the advisory.

One approach to such visualizations is to simply sample finitely many points and extrapolate the behavior on the entire domain from those finitely many points. However, this approach is imprecise and risks missing vital information because there is no way to know the correct sampling density to use to identify all important features.

Another approach is to use a tool such as DeepPoly [54] to over-approximate the output range of the DNN. However, because DeepPoly is a coarse over-approximation, there may be regions of the input space for which it cannot state with confidence the decision made by the network. In fact, the approximations

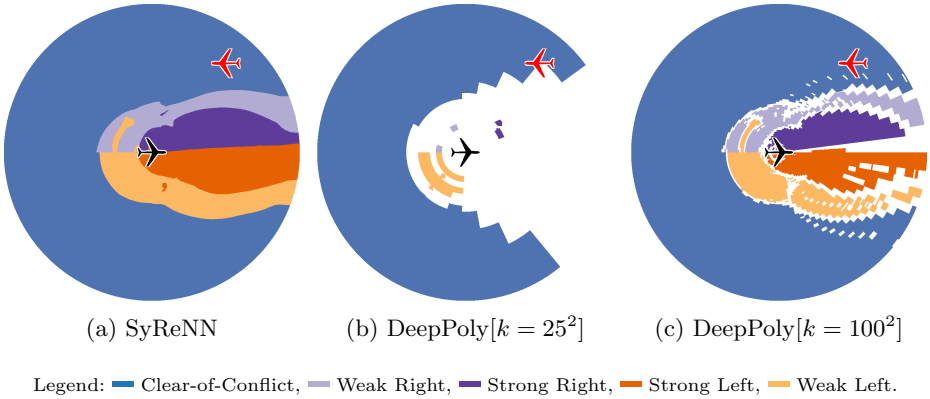


Fig. 4: Visualization of decision boundaries for the ACAS Xu network using three different approaches. Using SyReNN (left) quickly produces the exact decision boundaries. Using abstract interpretation-based tools like DeepPoly (middle and right) is slower and produces only imprecise approximations of the decision boundaries. k gives the number of partitions used by the DeepPoly algorithm, impacting the coarseness of the resulting approximation.

used by DeepPoly are extremely coarse. A naïve application of DeepPoly to this problem results in it being unable to make claims about *any* of the input space of interest. In order to utilize it, we must *partition* the space and run DeepPoly within each partition, which significantly slows down the analysis. Even when using 25^2 partitions, Figure 4b shows that most of the interesting region is still unclassifiable with DeepPoly (shown in white). Only with 100^2 partitions can DeepPoly effectively approximate the decision boundaries, although it is still quite imprecise.

By contrast, SyReNN can be used to *exactly* determine the decision boundaries on any 2D polytope subset of the input space, which can then be plotted. This is shown in Figure 4a. Furthermore, as shown in Table 1, the approach using SyReNN is *significantly* faster than that using DeepPoly, even as we get the *precise* answer instead of an approximation. Such visualizations can be particularly helpful in identifying issues to be fixed using techniques such as those in Section 8.3.

Implementation. The helper class `PlanesClassifier` is provided by our Python client library. It takes as input a DNN f and an input region X, then computes the decision boundaries of f on X.

The MNIST and CIFAR-10 DNNs used are from the ERAN project [1], and more details about the models are available on the ERAN repository. For example, the MNIST 3×100 model has three layers and 210 nodes, while the MNIST 9×200 model has nine layers with 1610 nodes. The ACAS Xu networks have 300 nodes each [36].

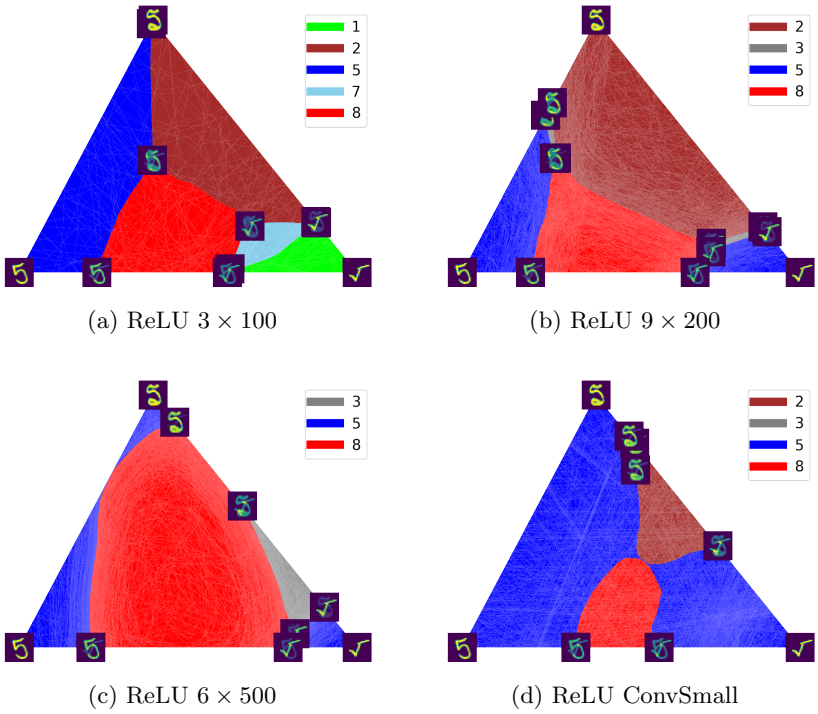
Table 1: Comparing the performance of DNN visualization using SyReNN versus DeepPoly for the ACAS Xu network [35]. $\widehat{f_{\restriction X}}$ size is the number of partitions in the symbolic representation. SyReNN time is the time taken to compute $\widehat{f_{\restriction X}}$ using SyReNN. DeepPoly[k] time is the time taken to compute DeepPoly for approximating decision boundaries with k partitions. Each scenario represents a different two-dimensional slice of the input space; within each slice, the heading of the intruder relative to the ownship along with the speed of each involved plane is fixed.

			DeepPoly time (secs)		
Scenario	$\widehat{f_{\restriction X}}$ size	SyReNN time (secs)	k = 25^2	k = 55^2	k = 100^2
Head-On, Slow	33200	0.09	26.7	127.9	421.7
Head-On, Fast	30769	0.09	25.2	119.7	394.1
Perpendicular, Slow	37251	0.12	26.7	127.0	418.9
Perpendicular, Fast	33931	0.09	24.9	118.3	389.7
Opposite, Slow	36743	0.10	27.8	132.4	436.9
Opposite, Fast	38965	0.11	27.1	129.1	425.1
-Perpendicular, Slow	36037	0.09	27.4	130.5	430.0
-Perpendicular, Fast	33208	0.09	25.1	119.9	394.6

Performance evaluation. Timing comparisons are given in Table 1. We see that SyReNN is quite performant, and the time taken to compute the exact SyReNN is negligible even comparing with the coarsest approximation from DeepPoly using 25^2 partitions. Experiments were performed on an Intel Core i9-9960X with 32 cores at 4.4GHz and 128GB of memory.

8.2 Visualization of Decision Boundaries for Image Recognition Networks

This section compares the performance of the ordered ExtendPWL (Algorithm 2) implemented on CPU against the batched ExtendPWL (Algorithm 4) implemented on both CPU and GPU, in visualization tasks for larger image recognition networks including MNIST and CIFAR-10. Table 2 shows the pre-trained models we used from ERAN [1]. All models only use the ReLU activation function. The models labeled with $m \times n$ are fully-connected feed-forward neural networks with m layers in total, n neurons at each hidden layer. MNIST ConvSmall is a four-layer convolutional network with 3,604 neurons. The input plane we chose to visualize for all MNIST networks or all CIFAR-10 networks are the same respectively. The MNIST networks take as input a 28×28 image of a handwritten digit embedded in a 784-dimensional input space, and predict the digit. The CIFAR-10 DNNs take as input a 32×32 color image embedded in a 3,072-dimensional input space, and predict one of ten labels including "airplane", "bird", "dog", etc. Figure 5 visualizes the decision boundaries of four MNIST networks over the same input plane (which correspond to the first four



(a) ReLU 3×100 (b) ReLU 9×200

(c) ReLU 6×500 (d) ReLU ConvSmall

Fig. 5: Visualization of decision boundaries for the MNIST networks.

rows in Table 3). The diagrams present the classified input planes computed using SyReNN in its two-dimensional subspace. Every point inside the triangle formed by the three input images represents an image interpolated from the three vertex images, while the color indicates the classification.

Performance evaluation. All experiments were performed on an Intel Core i9-9960X with 32 cores at 4.4GHz with 128GB of memory and Titan RTX with 24GB of GPU memory.

Table 3 shows the total time taken to compute SyReNN using different implementations of ExtendPWL. The Ordered-CPU column uses the Ordered ExtendPWL algorithm (Algorithm 2) on a CPU; the Batched-CPU and Batched-GPU columns use the Batched ExtendPWL algorithm (Algorithm 4) on a CPU and a GPU, respectively. All three implementations exploit multi-threading on CPU; additionally Batched-GPU exploits CUDA on GPU.

For all but the smallest two networks (MNIST 3×100 and CIFAR-10 4×100), Batched-GPU is significantly faster than Ordered-CPU. For those two small networks the total SyReNN time is negligible; the slowdown of Batched-GPU could be attributed to the overhead of using the GPU (e.g., kernel launch time).

Table 2: Pretrained MNIST and CIFAR-10 models used to evaluate the performance of DNN visualization using different implementations. $\widehat{f_{\restriction X}}$ is the number of partitions in the symbolic representation. #Neurons is the number of neurons. #Split is the number (in millons) of calls to SplitPlane in batched ExtendPWL Algorithm 4.

Model		#Neurons	$\widehat{f_{\restriction X}}$	#Split
MNIST	3×100	210	7,852	0.5M
	9×200	1,610	460,236	227M
	6×500	3,000	665,990	607M
	ConvSmall	3,604	430,502	530M
CIFAR-10	4×100	410	37,682	4M
	9×200	1,810	1,523,301	814M
	6×500	3,000	295,888	248M

Except for MNIST 9×200 network, Ordered-CPU outperforms Batched-CPU. Batched-CPU and Ordered-CPU should have the same performance in computing linear layers. Thus, one can infer that the slowdown of Batched-CPU can be attributed to the poor performance of Batched ExtendPWL (Algorithm 4) compared to that of Ordered ExtendPWL (Algorithm 2). Specifically, the implementation of Ordered ExtendPWL uses pipeline parallelism, which is better suited for the CPU. In contrast, the batched parallelism in Batched ExtendPWL is ill suited for the CPU. Consequently, we see that the Batched-GPU is significantly faster than Batched-CPU.

Table 4 shows the time spent in ExtendPWL when computing SyReNN. The Ordered-CPU column uses the Ordered ExtendPWL algorithm (Algorithm 2) on a CPU; the Batched-CPU and Batched-GPU columns use the Batched ExtendPWL algorithm (Algorithm 4) on a CPU and a GPU, respectively. For all but the smallest two networks (MNIST 3×100 and CIFAR-10 4×100), Batched-GPU is significantly faster than Ordered-CPU. Thus, we can conclude that the speedup of SyReNN when using Batched-GPU compared to Ordered-CPU (as seen in Table 3) cannot only be attributed to the fact that the computation of the linear layers is significantly faster on a GPU owing to faster matrix multiplications; the use of the Batched ExtendPWL plays an important role in achieving the speedup.

8.3 Provable Repair of DNNs

We have so far seen how SyReNN can be used to analyze trained DNNs and better understand their behavior. A natural next step is to *repair DNNs* to remove buggy behavior. In this section, we briefly describe Provable Repair of DNNs [56], and show how SyReNN forms a key component of Provable Polytope Repair of DNNs.

Table 3: Total time for computing $\widehat{f_{\restriction X}}$ of X using SyReNN for image recognition networks. The parenthesized number is speed up comparing to Ordered-CPU.

Model		SyReNN time (sec)		
		Ordered-CPU	Batched-CPU	Batched-GPU
MNIST	3×100	0.03	0.05 (0.60x)	0.07 (0.43x)
	9×200	5.40	2.12 (2.55x)	1.76 (3.07x)
	6×500	78.27	122.86 (0.64x)	3.70 (21.15x)
	ConvSmall	9.12	94.69 (0.10x)	4.04 (2.26x)
CIFAR-10	4×100	0.13	0.20 (0.65x)	0.17 (0.76x)
	9×200	25.28	43.12 (0.59x)	4.70 (5.37x)
	6×500	19.60	41.04 (0.48x)	1.92 (10.20x)

Table 4: Total time spent in ExtendPWL when computing $\widehat{f_{\restriction X}}$ of X using SyReNN for image recognition networks. The parenthesized number is speed up comparing to Ordered-CPU.

Model		ExtendPWL time (sec)		
		Ordered-CPU	Batched-CPU	Batched-GPU
MNIST	3×100	0.02	0.05 (0.40x)	0.07 (0.29x)
	9×200	3.12	1.90 (1.64x)	1.76 (1.77x)
	6×500	10.80	63.47 (0.17x)	3.70 (2.92x)
	ConvSmall	7.80	93.61 (0.08x)	3.99 (1.95x)
CIFAR-10	4×100	0.10	0.18 (0.56x)	0.17 (0.59x)
	9×200	17.43	35.83 (0.49x)	4.70 (3.71x)
	6×500	2.30	26.43 (0.09x)	1.92 (1.20x)

Traditional methods for fixing DNNs involve simply re-training the DNN while focusing on the identified-buggy points. However, this approach has a number of issues. First, it does not provide any guarantees that a repair will be found. This is exacerbated by the many hyperparameters involved with DNN re-training, so the user must try many combinations hoping that one works. Second, even if a repair is found, it does not provide any guarantee that this is the *smallest* repair — indeed, DNNs often over-correct, forgetting things they learned earlier, causing *drawdown,* or a degradation of accuracy on the original dataset after re-training. Finally, re-training operates inherently on finitely many input points at a time, whereas we often want to guarantee a DNN's behavior on infinitely many inputs, as described below.

Pointwise Repair. The simplest setting for repair is what we call *pointwise repair*. In the image recognition case, pointwise repair consists of correcting the DNN so it assigns the correct classification to every one of a set of finitely many input images. For example, [29] describes a dataset of *Natural Adversarial*

Fig. 6: Natural adversarial example [29]

Fig. 7: Fog-corrupted digit [45]

Examples, i.e., challenging images that many state-of-the-art image recognition models fail to correctly classify. An example of such an image is given in Figure 6. We can use provable pointwise repair to find a minimal modification to a given DNN that causes it to correctly classify those images.

The key theory behind pointwise DNN repair we will use was developed in [56]. In that work, we consider the satisfiability problem corresponding to the single-layer pointwise DNN repair. In essence, we can describe the DNN's output as an equation in terms of the input points and the weights of the DNN. We then want to solve these equations for the weights that produce the desired outputs on the given inputs. The key source of non-linearity is that, even if the input is fixed, changing a weight in the DNN can change which nodes are activated or inactivated, i.e., more generally, which linear region each of the intermediate vectors in the DNN's evaluation falls into.

To address this, [56] introduced a new DNN architecture called *Decoupled DNNs* (DDNNs), where there are two sets of weights: the *activation weights* and the *value weights*. The former have sole control over which nodes are activated, while the latter have sole control over what the nodes output if they are activated. Once the weights are decoupled in this manner, we see that changing the *value* weights alone will never change which nodes are activated or inactivated, and this turns out to be enough to make the single-layer repair problem linear (thus solvable in polynomial time) for DDNNs. This is summarized by the following theorem, reproduced from [56]:

Theorem 6. *Let N be a DDNN with layers $(W^{(a,i)}, W^{(v,i)}, \sigma^{(i)})$ and fix an index j. Then, for any $\vec{v}$, $N(\vec{v})$ varies linearly* as a function of $W^{(v,j)}$.

Fortunately, there is a simple, syntactic procedure to convert any DNN into an equivalent DDNN. Thus, we can take the user-provided DNN, convert it to a DDNN, and then repair the DDNN as a linear programming problem using off-the-shelf solvers such as Gurobi [26].

Polytope Repair. A more challenging setting for repair is *polytope repair*. For this, we consider an image classifier that has been trained to correctly recognize digits in clear images, and want to modify this classifier so it *also* correctly classifies foggy variants of those images (Figure 7). The key is that we want to ensure it works no matter the level of fog corruption applied. In pixel space, a single image forms a point and the set of all foggy variants of that image forms a line segment, with each point on the line segment corresponding to the image with some percent of fog applied. We can use provable polytope repair to find a

minimal modification to a given DNN that causes it to classify all of the points on those lines correctly.

The key result we need for polytope repair of DDNNs was presented in [56]:

Theorem 7. *Let N be a PWL DNN with layers $(W^{(i)}, \sigma^{(i)})$ and define a DDNN M with layers $(W^{(i)}, W^{(v,i)}, \sigma^{(i)})$. Then, within any linear region of the DNN N, the DDNN M is also linear.*

This theorem states that modifications to the value weights in a DDNN *do not change* the partitioning of the SyReNN for the DDNN. Recall further that, by definition, the DDNN behaves linearly within each SyReNN partition, and due to properties of linear maps, we can say that the entire partition satisfies some linear constraint if and only if its finitely many vertex points do. Thus, as these partitions do not change during repair for a DDNN, *provable polytope repair reduces to provable pointwise repair on the finitely many vertices of the linear regions.* The key is that one must compute those vertex points, but this is exactly what SyReNN allows us to do.

Case Studies. Here we report on three case studies from [56]. In the first, Provable Repair was used to repair an image-recognition network that struggled to classify a set of challenging images. In the second, Provable Repair was used to repair a digit-recognition network that struggled to classify images with varying amounts of fog applied. Finally, the third uses Provable Repair to enforce a safety specification on the ACAS Xu DNN seen in Section 8.1.

Implementation. The DNN Repair code is available as a separate Python library called PRDNN that interfaces with our Python client. It takes as input a DNN f and pairs of input region, output label X_i, Y_i, then computes a new DNN f' which maps all points in each X_i into Y_i.

Baseline. We compared against a baseline using fine-tuning, i.e., re-training the DNN with the buggy inputs until they are all correctly classified. We discuss a variety of alternative repair methods in Section 9; to our knowledge, almost none have publicly-available implementations, none support the guarantees of time complexity and completeness that PRDNN does, and none support provable *polytope* repair.

Empirical results. Results for the challenging-images task are summarized in Table 5. Results for the foggy-digits task are summarized in Table 6. Drawdown is the drop in accuracy on the original dataset, while generalization is the increase in accuracy on points that are misclassified by the buggy network in a similar way to those repaired on.

For the aircraft collision-avoidance task, Provable Polytope Repair succeeded in under 22 seconds, with zero drawdown and 95% generalization. Meanwhile, the fine-tuning baseline timed out after over 1 hour, and had *negative* efficacy; while the original network misclassified only 3 points in the sampled repair set, the FT-repaired network misclassified 181 points; 12% drawdown; and 96% generalization.

Table 5: Summary of experimental results for task 1. D: Drawdown (%), T: Time, BD: Best Drawdown, PR: Provable Repair, FT: Fine-Tuning baseline.

	PR (BD)		FT[1]		FT[2]	
Points	D	T	D	T	D	T
100	3.6	1m39.0s	10.2	4m31.8s	8.2	9m24.0s
200	1.1	2m50.8s	9.6	12m19.5s	9.6	26m35.0s
400	5.1	4m45.3s	13.8	34m2.6s	11.1	1h9m26.8s
752	5.3	8m28.1s	15.4	1h22m18.7s	13.4	2h33m8.2s

Table 6: Summary of experimental results for task 2. D: Drawdown (%), G: Generalization (%), T: Time, PR: provable repair, FT: fine-tuning baseline. $*$ means fine-tuning diverged and timed out after 1000 epochs, the results shown are from the last iteration of fine-tuning before the timeout.

		PR (Layer 2)			PR (Layer 3)			FT[1]			FT[2]		
Lines	Points	D	G	T	D	G	T	D	G	T	D	G	T
10	1730	1.3	30.7	2m	5.7	32.1	2s	56.0	4.2	0.4s	8.3	27.5	0.6s
25	4314	1.8	35.5	3m	5.5	38.3	4s	36.5	22.4	1s	3.8	51.0	0.4s
50	8354	2.6	38.3	4m	5.9	44.5	8s	85.2*	-8.2*	30m*	4.7	55.8	0.8s
100	16024	2.4	42.9	11m	5.9	46.0	18s	31.4	37.7	3s	3.2	60.0	2s

8.4 Integrated Gradients

A common problem in the field of *explainable machine learning* is understanding *why* a DNN made the prediction it did. For example, given an image classified by a DNN as a 'cat,' why did the DNN decide it was a cat instead of, say, a dog? Were there particular pixels which were particularly important in deciding this? Integrated Gradients (IG) [57] is the state-of-the-art method for computing such *model attributions.*

Definition 5. *Given a DNN f, the* integrated gradients along dimension i for input x and baseline x' *is defined to be:*

$$IG_i(x) \stackrel{def}{=} (x_i - x'_i) \times \int_{\alpha=0}^{1} \frac{\partial f(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha. \tag{2}$$

The computed value $IG_i(x)$ determines relatively how important the ith input (e.g., pixel) was to the classification.

However, exactly computing this integral requires a symbolic, closed form for the gradient of the network. Until [55], it was not known how to compute such a closed-form and so IGs were always only *approximated* using a sampling-based approach. Unfortunately, because it was unknown how to compute the true value, there was no way for practitioners to determine how accurate their approximations were. This is particularly concerning in fairness applications where an accurate attribution is exceedingly important.

In [55], it was recognized that, when $X = \texttt{ConvexHull}(\{x, x'\})$, $\widehat{f_{\restriction X}}$ can be used to *exactly* compute $IG_i(x)$. This is because within each partition of $\widehat{f_{\restriction X}}$ the gradient of the network is *constant* because it behaves as a linear function, and hence the integral can be written as the weighted sum of such finitely many gradients.[1] Using our symbolic representation, the exact IG can thus be computed as follows:

$$\sum_{\texttt{ConvexHull}(\{y_i, y_i'\}) \in \widehat{f_{\restriction \texttt{ConvexHull}(\{x, x'\})}}} (y_i' - y_i) \times \frac{\partial f(0.5 \times (y_i + y_i'))}{\partial x_i} \tag{3}$$

Where here y_i, y_i' are the endpoints of the segment with y_i closer to x and y_i' closest to x'.

Implementation. The helper class `IntegratedGradientsHelper` is provided by our Python client library. It takes as input a DNN f and a set of (x, x') input-baseline pairs and then computes IG for each pair.

Empirical results. In [55] SyReNN was used to show conclusively that existing sampling-based methods were insufficient to adequately approximate the true IG. This realization led to changes in the official IG implementation to use the more-precise trapezoidal sampling method we argued for.

Timing numbers. In those experiments, we used SyReNN to compute $\widehat{f_{\restriction X}}$ for three different DNNs f, namely the small, medium, and large convolutional models from [1]. For each DNN, we ran SyReNN on 100 one-dimensional lines. The 100 calls to SyReNN completed in 20.8 seconds for the small model, 183.3 for the medium model, and 615.5 for the big model. Tests were performed on an Intel Core i7-7820X CPU at 3.60GHz with 32GB of memory.

9 Related Work

SyReNN Primitive. The related problem of exact reach set analysis for DNNs was investigated in [64]. However, the authors use an algorithm that appears to suggest explicitly enumerating all exponentially-many (2^n) possible signs at each ReLU layer. By contrast, our algorithm adapts to the actual input polytopes, efficiently restricting its consideration to activations that are actually possible. Furthermore, they focus on the particular problem of reach set analysis for full-dimensional subsets of the DNN's input domain, whereas the key focus of SyReNN is enabling analyses such as IG, visualization, and repair that rely on precise and efficient analysis of low-dimensional subsets of the input domain. These differences in focus become clear in the evaluation, where the approach of [64] takes multiple minutes to identify 1250 linear regions, while SyReNN identifies tens of thousands of lower-dimensional linear regions in less than a second (Section 8). Newer algorithms and representations [61] may further improve the performance of higher-dimensional exact analysis in the future,

[1] As noted in [55], this technically requires a slight strengthening of the definition of $\widehat{f_{\restriction X}}$ which is satisfied by our algorithms as defined above.

however we expect techniques specialized to the low-dimensional case to continue to be significantly faster. Thus, the choice of which approach to use is mostly driven by application domain constraints, e.g., the applications in Section 8 were inherently low-dimensional.

Hanin and Rolnick [28] prove theoretical properties about the cardinality of $\widehat{f_{\restriction X}}$ for ReLU networks, showing that $|\widehat{f_{\restriction X}}|$ is expected to grow polynomially with the number of nodes in the network for randomly-initialized networks.

Thrun [60] and Bastani et al.[4] extract symbolic rules meant to approximate DNNs, which can approximate the symbolic representation $\widehat{f_{\restriction X}}$.

DNN Visualization and Understanding. Integrated Gradients was first proposed by [57], however they used a Riemann sum approximation instead of computing it exactly. The one-dimensional version of SyReNN [55] was the first approach able to exactly compute IG.

The ERAN [1] tool and underlying DeepPoly [54] domain were designed to verify the non-existence of adversarial examples. Breutel et al. [5] give an iterative refinement algorithm for an overapproximation of the weakest precondition as a polytope where the required output is also a polytope.

Scheibler et al. [51] verify the safety of a machine-learning controller using the SMT-solver iSAT3, but support small unrolling depths and basic safety properties. Zhu et al. [67] use a synthesis procedure to generate a safe deterministic program that can enforce safety conditions by monitoring the deployed DNN and preventing potentially unsafe actions. The presence of adversarial and fooling inputs for DNNs as well as applications of DNNs in safety-critical systems has led to efforts to verify and certify DNNs [3,36,13,33,16,6,63,54,2]. *Approximate reachability analysis* for neural networks safely overapproximates the set of possible outputs [16,64,65,63,12,62,61].

Given polytopes in the output space, Yang et al. [66] compute the exact set of corresponding input polytopes. Their technique is restricted to DNNs with ReLU but supports polytopes in arbitrary-dimensional subspaces. They utilize the facet-vertex incidence matrix (FVIM), which is a compact representation of convex polytopes.

Provable Repair. Prior work in the area of network repair focuses on enforcing constraints on the network during training. DiffAI [43] is an approach to train neural networks that are certifiably robust to adversarial perturbations. DL2 [14] allows for training and querying neural networks with logical constraints.

The layer-wise repair process described in [18] is most similar to the one described in this paper. The key advantage of our approach is the use of Decoupled DNNs. Decoupled DNNs allow for both guaranteed polytope repair as well as guaranteed polynomial-time pointwise repair, neither of which is guaranteed by any other system that we are aware of. Furthermore, to our knowledge, the repair system in [18] is not publicly available for comparison. In theory, any pointwise repair algorithm (such as the one in [18]) could be used along with Decoupled DNNs in order to enable polytope repair.

10 Conclusion

We presented SyReNN, a tool for understanding and analyzing DNNs. Given a piecewise-linear network and a low-dimensional polytope subspace of the input space, SyReNN computes a symbolic representation that decomposes the behavior of the DNN into finitely many linear functions. We showed how to efficiently compute this representation, and presented the design of the corresponding tool. We illustrated the utility of SyReNN on three application domains: visualizing the behavior of DNNs, repairing DNNs, and computing exact IG.

In contrast to prior work, SyReNN explores a unique point in the design space of DNN analysis tools. Instead of trading off precision of the analysis for efficiency, SyReNN focuses on analyzing DNN behavior on *low-dimensional subspaces* of the domain, for which we can provide *both* efficiency and precision.

Acknowledgements. We thank the reviewers for their comments, which greatly improved the quality of the paper. Matthew Sotoudeh is supported by NSF grant DGE-1656518. This work is supported in part by NSF grant CCF-2048123 and DOE Award DE-SC0022285.

References

1. ETH robustness analyzer for neural networks (ERAN). https://github.com/eth-sri/eran (2019), accessed: 2019-05-01
2. Anderson, G., Pailoor, S., Dillig, I., Chaudhuri, S.: Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 731–744. ACM (2019). https://doi.org/10.1145/3314221.3314614, https://doi.org/10.1145/3314221.3314614
3. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Lee, D.D., Sugiyama, M., von Luxburg, U., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. pp. 2613–2621 (2016), http://papers.nips.cc/paper/6339-measuring-neural-net-robustness-with-constraints
4. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 2499–2509 (2018), http://papers.nips.cc/paper/7516-verifiable-reinforcement-learning-via-policy-extraction
5. Breutel, S., Maire, F., Hayward, R.: Extracting interface assertions from neural networks in polyhedral format. In: ESANN 2003, 11th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 23-25, 2003, Proceedings. pp. 463–468 (2003), https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2003-72.pdf
6. Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.)

Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 4795–4804 (2018), http://papers.nips.cc/paper/7728-a-unified-view-of-piecewise-linear-neural-network-verification

7. Carlini, N., Wagner, D.A.: Audio adversarial examples: Targeted attacks on speech-to-text. In: 2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018. pp. 1–7. IEEE Computer Society (2018). https://doi.org/10.1109/SPW.2018.00009, https://doi.org/10.1109/SPW.2018.00009
8. Chen, J., Ran, X.: Deep learning with edge computing: A review. Proc. IEEE **107**(8), 1655–1674 (2019). https://doi.org/10.1109/JPROC.2019.2921977, https://doi.org/10.1109/JPROC.2019.2921977
9. Ching, T., Himmelstein, D.S., Beaulieu-Jones, B.K., Kalinin, A.A., Do, B.T., Way, G.P., Ferrero, E., Agapow, P.M., Zietz, M., Hoffman, M.M., Xie, W., Rosen, G.L., Lengerich, B.J., Israeli, J., Lanchantin, J., Woloszynek, S., Carpenter, A.E., Shrikumar, A., Xu, J., Cofer, E.M., Lavender, C.A., Turaga, S.C., Alexandari, A.M., Lu, Z., Harris, D.J., DeCaprio, D., Qi, Y., Kundaje, A., Peng, Y., Wiley, L.K., Segler, M.H.S., Boca, S.M., Swamidass, S.J., Huang, A., Gitter, A., Greene, C.S.: Opportunities and obstacles for deep learning in biology and medicine. Journal of The Royal Society Interface **15**(141), 20170387 (2018). https://doi.org/10.1098/rsif.2017.0387
10. Chollet, F., et al.: Keras. https://keras.io (2015)
11. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). pp. 4171–4186. Association for Computational Linguistics (2019). https://doi.org/10.18653/v1/n19-1423, https://doi.org/10.18653/v1/n19-1423
12. Dutta, S., Jha, S., Sankaranarayanan, S., Tiwari, A.: Output range analysis for deep feedforward neural networks. In: Dutle, A., Muñoz, C.A., Narkawicz, A. (eds.) NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10811, pp. 121–138. Springer (2018). https://doi.org/10.1007/978-3-319-77935-5_9, https://doi.org/10.1007/978-3-319-77935-5_9
13. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10482, pp. 269–286. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_19, https://doi.org/10.1007/978-3-319-68167-2_19
14. Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.T.: DL2: training and querying neural networks with logic. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97, pp. 1931–1941. PMLR (2019), http://proceedings.mlr.press/v97/fischer19a.html
15. Fukuda, K., et al.: Frequently asked questions in polyhedral computation. ETH, Zurich, Switzerland (2004)
16. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: 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. pp. 3–18. IEEE Computer Society (2018). https://doi.org/10.1109/SP.2018.00058, https://doi.org/10.1109/SP.2018.00058
17. Gläßle, T.: C++11 fourier motzkin elimination utilities. https://github.com/coldfix/cfme (2016)
18. Goldberger, B., Katz, G., Adi, Y., Keshet, J.: Minimal modifications of deep neural networks using verification. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020. EPiC Series in Computing, vol. 73, pp. 260–278. EasyChair (2020). https://doi.org/10.29007/699q, https://doi.org/10.29007/699q
19. Gonzales, R.: Feds say self-driving Uber SUV did not recognize jaywalking pedestrian in fatal crash. NPR https://www.npr.org/2019/11/07/777438412/feds-say-self-driving-uber-suv-did-not-recognize-jaywalking-pedestrian-in-fatal- (Nov 2019), accessed: 2020-06-06
20. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org
21. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6572
22. Google: grpc: A high-performance, open source universal rpc framework. "https://grpc.io/ (2020)
23. Google: Protocol buffers - google's data interchange format. https://developers.google.com/protocol-buffers/ (2020)
24. Gopinath, S., Ghanathe, N., Seshadri, V., Sharma, R.: Compiling kb-sized machine learning models to tiny iot devices. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 79–95. ACM (2019). https://doi.org/10.1145/3314221.3314597, https://doi.org/10.1145/3314221.3314597
25. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)
26. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022), https://www.gurobi.com
27. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In: Bengio, Y., LeCun, Y. (eds.) 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (2016), http://arxiv.org/abs/1510.00149
28. Hanin, B., Rolnick, D.: Complexity of linear regions in deep networks. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97, pp. 2596–2604. PMLR (2019), http://proceedings.mlr.press/v97/hanin19a.html
29. Hendrycks, D., Zhao, K., Basart, S., Steinhardt, J., Song, D.: Natural adversarial examples. arXiv preprint arXiv:1907.07174 (2019)
30. Hern, A.: Facebook translates 'good morning' into 'attack them', leading to arrest. https://www.theguardian.com/technology/2017/oct/24/facebook-palestine-israel-translates-good-morning-attack-them-arrest (Jun 2017), accessed: 2020-06-06

31. Hill, K.: Wrongfully accused by an algorithm. New York Times. https://www.nytimes.com/2020/06/24/technology/facial-recognition-arrest.html (Jun 2020), accessed: 2020-06-06
32. Hosny, A., Parmar, C., Quackenbush, J., Schwartz, L.H., Aerts, H.J.: Artificial intelligence in radiology. Nature Reviews Cancer p. 1 (2018)
33. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 3–29. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_1, https://doi.org/10.1007/978-3-319-63387-9_1
34. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
35. Julian, K.D., Kochenderfer, M.J., Owen, M.P.: Deep neural network compression for aircraft collision avoidance systems. CoRR **abs/1810.04240** (2018), http://arxiv.org/abs/1810.04240
36. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 97–117. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_5, https://doi.org/10.1007/978-3-319-63387-9_5
37. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Bartlett, P.L., Pereira, F.C.N., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States. pp. 1106–1114 (2012), http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks
38. Kumar, A., Seshadri, V., Sharma, R.: Shiftry: RNN inference in 2kb of RAM. Proc. ACM Program. Lang. **4**(OOPSLA), 182:1–182:30 (2020). https://doi.org/10.1145/3428250, https://doi.org/10.1145/3428250
39. Kusupati, A., Singh, M., Bhatia, K., Kumar, A., Jain, P., Varma, M.: Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 9031–9042 (2018)
40. Lee, D.: US opens investigation into Tesla after fatal crash. BBC. https://www.bbc.co.uk/news/technology-36680043 (Jul 2016), accessed: 2020-06-06
41. Mendelson, E.B.: Artificial intelligence in breast imaging: potentials and limitations. American Journal of Roentgenology **212**(2), 293–299 (2019)
42. Miotto, R., Wang, F., Wang, S., Jiang, X., Dudley, J.T.: Deep learning for healthcare: review, opportunities and challenges. Briefings Bioinform. **19**(6), 1236–1246 (2018). https://doi.org/10.1093/bib/bbx044, https://doi.org/10.1093/bib/bbx044

43. Mirman, M., Gehr, T., Vechev, M.T.: Differentiable abstract interpretation for provably robust neural networks. In: Dy, J.G., Krause, A. (eds.) 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, pp. 3575–3583. PMLR (2018), http://proceedings.mlr.press/v80/mirman18b.html
44. Moosavi-Dezfooli, S., Fawzi, A., Frossard, P.: Deepfool: A simple and accurate method to fool deep neural networks. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. pp. 2574–2582. IEEE Computer Society (2016). https://doi.org/10.1109/CVPR.2016.282, https://doi.org/10.1109/CVPR.2016.282
45. Mu, N., Gilmer, J.: MNIST-C: A robustness benchmark for computer vision. CoRR **abs/1906.02337** (2019), http://arxiv.org/abs/1906.02337
46. Nguyen, A.M., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. pp. 427–436. IEEE Computer Society (2015). https://doi.org/10.1109/CVPR.2015.7298640, https://doi.org/10.1109/CVPR.2015.7298640
47. ONNX: Open neural network exchange. https://onnx.ai/ (2020)
48. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch (2017)
49. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 8024–8035 (2019), http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library
50. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. " O'Reilly Media, Inc." (2007)
51. Scheibler, K., Winterer, L., Wimmer, R., Becker, B.: Towards verification of artificial neural networks. In: Heinkel, U., Kriesten, D., Rößler, M. (eds.) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2015, Chemnitz, Germany, March 3-4, 2015. pp. 30–40. Sächsische Landesbibliothek (2015)
52. Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J.K., Shao, C., Mishra, A., Esmaeilzadeh, H.: From high-level deep neural models to fpgas. In: 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016. pp. 17:1–17:12. IEEE Computer Society (2016). https://doi.org/10.1109/MICRO.2016.7783720, https://doi.org/10.1109/MICRO.2016.7783720
53. Singh, G., Gehr, T., Mirman, M., Püschel, M., Vechev, M.T.: Fast and effective robustness certification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018,

NeurIPS 2018, December 3-8, 2018, Montréal, Canada. pp. 10825–10836 (2018), http://papers.nips.cc/paper/8278-fast-and-effective-robustness-certification
54. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. Proc. ACM Program. Lang. **3**(POPL), 41:1–41:30 (2019). https://doi.org/10.1145/3290354, https://doi.org/10.1145/3290354
55. Sotoudeh, M., Thakur, A.V.: Computing linear restrictions of neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 14132–14143 (2019), http://papers.nips.cc/paper/9562-computing-linear-restrictions-of-neural-networks
56. Sotoudeh, M., Thakur, A.V.: Provable repair of deep neural networks. In: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI). ACM (2021)
57. Sundararajan, M., Taly, A., Yan, Q.: Axiomatic attribution for deep networks. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. Proceedings of Machine Learning Research, vol. 70, pp. 3319–3328. PMLR (2017), http://proceedings.mlr.press/v70/sundararajan17a.html
58. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. pp. 2818–2826. IEEE Computer Society (2016). https://doi.org/10.1109/CVPR.2016.308, https://doi.org/10.1109/CVPR.2016.308
59. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: Bengio, Y., LeCun, Y. (eds.) 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings (2014), http://arxiv.org/abs/1312.6199
60. Thrun, S.: Extracting rules from artifical neural networks with distributed representations. In: Tesauro, G., Touretzky, D.S., Leen, T.K. (eds.) Advances in Neural Information Processing Systems 7, [NIPS Conference, Denver, Colorado, USA, 1994]. pp. 505–512. MIT Press (1994)
61. Tran, H., Lopez, D.M., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-based reachability analysis of deep neural networks. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 670–686. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_39, https://doi.org/10.1007/978-3-030-30942-8_39
62. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1599–1614. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi
63. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: Dy, J.G., Krause, A. (eds.) 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, pp. 5273–5282. PMLR (2018), http://proceedings.mlr.press/v80/weng18a.html
64. Xiang, W., Tran, H., Johnson, T.T.: Reachable set computation and safety verification for neural networks with relu activations. CoRR **abs/1712.08163** (2017), http://arxiv.org/abs/1712.08163
65. Xiang, W., Tran, H., Rosenfeld, J.A., Johnson, T.T.: Reachable set estimation and safety verification for piecewise linear systems with neural network controllers. In: 2018 Annual American Control Conference, ACC 2018, Milwaukee, WI, USA, June 27-29, 2018. pp. 1574–1579. IEEE (2018). https://doi.org/10.23919/ACC.2018.8431048, https://doi.org/10.23919/ACC.2018.8431048
66. Yang, X., Johnson, T.T., Tran, H., Yamaguchi, T., Hoxha, B., Prokhorov, D.V.: Reachability analysis of deep relu neural networks using facet-vertex incidence. In: HSCC '21: 24th ACM International Conference on Hybrid Systems: Computation and Control, Nashville, Tennessee, May 19-21, 2021. pp. 18:1–18:7. ACM (2021). https://doi.org/10.1145/3447928.3456650, https://doi.org/10.1145/3447928.3456650
67. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 686–701. ACM (2019). https://doi.org/10.1145/3314221.3314638, https://doi.org/10.1145/3314221.3314638
68. Ziegler, G.M.: Lectures on polytopes, vol. 152. Springer Science & Business Media (2012)