AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri*, Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—We present AI^2, the first sound and scalable analyzer for deep neural networks. Based on overapproximation, AI^2 can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks).**

The key insight behind AI^2 is to phrase reasoning about safety and robustness of neural networks in terms of classic abstract interpretation, enabling us to leverage decades of advances in that area. Concretely, we introduce abstract transformers that capture the behavior of fully connected and convolutional neural network layers with rectified linear unit activations (ReLU), as well as max pooling layers. This allows us to handle real-world neural networks, which are often built out of those types of layers.

We present a complete implementation of AI^2 together with an extensive evaluation on 20 neural networks. Our results demonstrate that: (i) AI^2 is precise enough to prove useful specifications (e.g., robustness), (ii) AI^2 can be used to certify the effectiveness of state-of-the-art defenses for neural networks, (iii) AI^2 is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and (iv) AI^2 can handle deep convolutional networks, which are beyond the reach of existing methods.

***Index Terms*—Reliable Machine Learning, Robustness, Neural Networks, Abstract Interpretation**

I. Introduction

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [2], malware detection [44], and aircraft collision avoidance detection [21]. This adoption can be attributed to the near-human accuracy obtained by these models [21], [23].

Despite their success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [40] showing that even highly accurate neural networks are vulnerable to *adversarial examples*. Adversarial examples are typically obtained by slightly perturbing an input that is correctly classified by the network, such that the network misclassifies the perturbed input. Various kinds of perturbations have been shown to successfully generate adversarial examples (e.g., [3], [12], [14], [15], [17], [18], [29], [30], [32], [38], [41]). Fig. 1 illustrates two attacks, FGSM and brightening, against a digit classifier. For each attack, Fig. 1 shows an input in the Original column, the perturbed input in the Perturbed column, and the pixels that were changed in the Diff column. Brightened pixels are marked in yellow and darkened pixels are marked in purple. The FGSM [12] attack perturbs an image by adding to it a particular noise vector multiplied by a small number ϵ (in Fig. 1, $\epsilon = 0.3$). The brightening attack (e.g., [32]) perturbs an image by changing all pixels above the threshold $1 - \delta$ to the brightest possible value (in Fig. 1, $\delta = 0.085$).

Attack	Original	Perturbed	Diff
FGSM [12], $\epsilon = 0.3$			
Brightening, $\delta = 0.085$			

Fig. 1: Attacks applied to MNIST images [25].

Adversarial examples can be especially problematic when safety-critical systems rely on neural networks. For instance, it has been shown that attacks can be executed physically (e.g., [9], [24]) and against neural networks accessible only as a black box (e.g., [12], [40], [43]). To mitigate these issues, recent research has focused on reasoning about neural network robustness, and in particular on *local robustness*. Local robustness (or robustness, for short) requires that all samples in the neighborhood of a given input are classified with the same label [31]. Many works have focused on designing *defenses* that increase robustness by using modified procedures for training the network (e.g., [12], [15], [27], [31], [42]). Others have developed approaches that can show non-robustness by underapproximating neural network behaviors [1] or methods that decide robustness of small fully connected feedforward networks [21]. However, no existing sound analyzer handles convolutional networks, one of the most popular architectures.

Key Challenge: Scalability and Precision. The main challenge facing sound analysis of neural networks is scaling to large classifiers while maintaining a precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. For instance, consider the image of the digit 8 in Fig. 1 and suppose we would like to prove that no matter how we brighten the value of pixels with intensity above $1 - 0.085$, the network will still classify the image as 8 (in this example we have 84 such pixels, shown in yellow). Assuming 64-bit floating

*Rice University, work done while at ETH Zurich.

DOI 10.1109/SP.2018.00058

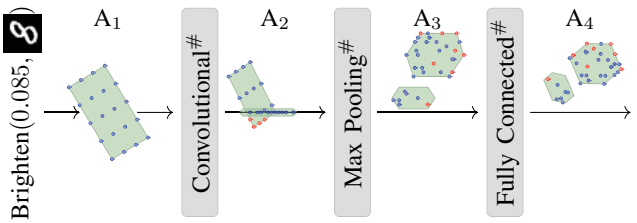


Fig. 2: A high-level illustration of how AI^2 checks that all perturbed inputs are classified the same way. AI^2 first creates an abstract element A_1 capturing all perturbed images. (Here, we use a 2-bounded set of zonotopes.) It then propagates A_1 through the abstract transformer of each layer, obtaining new shapes. Finally, it verifies that all points in A_4 correspond to outputs with the same classification.

point numbers are used to express pixel intensity, we obtain more than 10^{1154} possible perturbed images. Thus, proving the property by running a network exhaustively on all possible input images and checking if all of them are classified as 8 is infeasible. To avoid this state space explosion, current methods (e.g., [18], [21], [34]) symbolically encode the network as a logical formula and then check robustness properties with a constraint solver. However, such solutions do not scale to larger (e.g., convolutional) networks, which usually involve many intermediate computations.

Key Concept: Abstract Interpretation for AI. The key insight of our work is to address the above challenge by leveraging the classic framework of abstract interpretation (e.g., [6], [7]), a theory which dictates how to obtain sound, computable, and precise finite approximations of potentially infinite sets of behaviors. Concretely, we leverage numerical abstract domains – a particularly good match, as AI systems tend to heavily manipulate numerical quantities. By showing how to apply abstract interpretation to reason about AI safety, we enable one to leverage decades of research and any future advancements in that area (e.g., in numerical domains [39]). With abstract interpretation, a neural network computation is overapproximated using an *abstract domain*. An abstract domain consists of logical formulas that capture certain shapes (e.g., zonotopes, a restricted form of polyhedra). For example, in Fig. 2, the green zonotope A_1 overapproximates the set of blue points (each point represents an image). Of course, sometimes, due to abstraction, a shape may also contain points that will not occur in any concrete execution (e.g., the red points in A_2).

The AI^2 Analyzer. Based on this insight, we developed a system called AI^2 (*Abstract Interpretation for Artificial Intelligence*)[1]. AI^2 is the first scalable analyzer that handles common network layer types, including fully connected and convolutional layers with rectified linear unit activations (ReLU) and max pooling layers.

To illustrate the operation of AI^2, consider the example in Fig. 2, where we have a neural network, an image of the digit 8 and a set of perturbations: brightening with parameter 0.085. Our goal is to prove that the neural network classifies all perturbed images as 8. AI^2 takes the image of the digit 8 and the perturbation type and creates an abstract element A_1 that captures all perturbed images. In particular, we can capture the entire set of brightening perturbations exactly with a single zonotope. However, in general, this step may result in an abstract element that contains additional inputs (that is, red points). In the second step, A_1 is automatically propagated through the layers of the network. Since layers work on concrete values and not abstract elements, this propagation requires us to define *abstract layers* (marked with $^\#$) that compute the effects of the layers on abstract elements. The abstract layers are commonly called the *abstract transformers* of the layers. Defining sound and precise, yet scalable abstract transformers is key to the success of an analysis based on abstract interpretation. We define abstract transformers for all three layer types shown in Fig. 2.

At the end of the analysis, the abstract output A_4 is an overapproximation of *all* possible concrete outputs. This enables AI^2 to verify safety properties such as robustness (e.g., are all images classified as 8?) directly on A_4. In fact, with a single abstract run, AI^2 was able to prove that a convolutional neural network classifies all of the considered perturbed images as 8.

We evaluated AI^2 on important tasks such as verifying robustness and comparing neural networks defenses. For example, for the perturbed image of the digit 0 in Fig. 1, we showed that while a non-defended neural network classified the FGSM perturbation with $\epsilon = 0.3$ as 9, this attack is *provably* eliminated when using a neural network trained with the defense of [27]. In fact, AI^2 proved that the FGSM attack is unable to generate adversarial examples from this image for any ϵ between 0 and 0.3.

Main Contributions. Our main contributions are:

- A sound and scalable method for analysis of deep neural networks based on abstract interpretation (Section IV).
- AI^2, an end-to-end analyzer, extensively evaluated on feed-forward and convolutional networks (computing with 53 000 neurons), far exceeding capabilities of current systems (Section VI).
- An application of AI^2 to evaluate *provable robustness* of neural network defenses (Section VII).

II. Representing Neural Networks as Conditional Affine Transformations

In this section, we provide background on feedforward and convolutional neural networks and show how to transform them into a representation amenable to abstract interpretation. This representation helps us simplify the construction and description of our analyzer, which we discuss in later sections. We use the following notation: for a vector $\overline{x} \in \mathbb{R}^n$, x_i denotes its i^{th} entry, and for a matrix $W \in \mathbb{R}^{n \times m}$, W_i denotes its i^{th} row and $W_{i,j}$ denotes the entry in its i^{th} row and j^{th} column.

[1] AI^2 is available at: http://ai2.ethz.ch

$$
\begin{array}{rcl}
f(\overline{x}) & ::= & W \cdot \overline{x} + \overline{b} \\
& \mid & \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{x}) \\
& \mid & f(f'(\overline{x})) \\
E & ::= & E \wedge E \mid x_i \geq x_j \mid x_i \geq 0 \mid x_i < 0
\end{array}
$$

Fig. 3: Definition of CAT functions.

CAT Functions. We express the neural network as a composition of *conditional affine transformations* (CAT), which are affine transformations guarded by logical constraints. The class of CAT functions, shown in Fig. 3, consists of functions $f \colon \mathbb{R}^m \to \mathbb{R}^n$ for $m, n \in \mathbb{N}$ and is defined recursively. Any affine transformation $f(\overline{x}) = W \cdot \overline{x} + \overline{b}$ is a CAT function, for a matrix W and a vector $\overline{b}$. Given sequences of conditions $E_1, \ldots, E_k$ and CAT functions $f_1, \ldots, f_k$, we write:

$$f(\overline{x}) = \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{x}).$$

This is also a CAT function, which returnsx $f_i(\overline{x})$ for the first E_i satisfied by $\overline{x}$. The conditions are conjunctions of constraints of the form $x_i \geq x_j$, $x_i \geq 0$ and $x_i < 0$. Finally, any composition of CAT functions is a CAT function. We often write $f'' \circ f'$ to denote the CAT function $f(\overline{x}) = f''(f'(\overline{x}))$.

Layers. Neural networks are often organized as a sequence of layers, such that the output of one layer is the input of the next layer. Layers consist of *neurons*, performing the same function but with different parameters. The output of a layer is formed by stacking the outputs of the neurons into a vector or three-dimensional array. We will define the functionality in terms of entire layers instead of in terms of individual neurons.

Reshaping of Inputs. Layers often take three-dimensional inputs (e.g., colored images). Such inputs are transformed into vectors by reshaping. A three-dimensional array $\overline{x} \in \mathbb{R}^{m \times n \times r}$ can be reshaped to $\overline{x}^v \in \mathbb{R}^{m \cdot n \cdot r}$ in a canonical way, first by depth, then by column, finally by row. That is, given $\overline{x}$:

$$\overline{x}^v = (x_{1,1,1} \ldots x_{1,1,r}\ x_{1,2,1} \ldots x_{1,2,r} \ldots x_{m,n,1} \ldots x_{m,n,r})^T.$$

Activation Function. Typically, layers in a neural network perform a linear transformation followed by a non-linear activation function. We focus on the commonly used rectified linear unit (ReLU) activation function, which for $x \in \mathbb{R}$ is defined as $\text{ReLU}(x) = \max(0, x)$, and for a vector $\overline{x} \in \mathbb{R}^m$ as $\text{ReLU}(\overline{x}) = (\text{ReLU}(x_1), \ldots, \text{ReLU}(x_m))$.

ReLU to CAT. We can express the ReLU activation function as $\text{ReLU} = \text{ReLU}_n \circ \ldots \circ \text{ReLU}_1$ where ReLU_i processes the i^{th} entry of the input $\overline{x}$ and is given by:

$$\begin{array}{rl}
\text{ReLU}_i(\overline{x}) = & \textbf{case } (x_i \geq 0) \colon \overline{x}, \\
& \textbf{case } (x_i < 0) \colon I_{i \leftarrow 0} \cdot \overline{x}.
\end{array}$$

$I_{i \leftarrow 0}$ is the identity matrix with the i^{th} row replaced by zeros.

Fully Connected (FC) Layer. An FC layer takes a vector of size m (the m outputs of the previous layer), and passes it to n neurons, each computing a function based on the neuron's *weights* and *bias*, one weight for each component of the input. Formally, an FC layer with n neurons is a function $\text{FC}_{W,\overline{b}} \colon \mathbb{R}^m \to \mathbb{R}^n$ parameterized by a weight matrix $W \in \mathbb{R}^{n \times m}$ and a bias $\overline{b} \in \mathbb{R}^n$. For $\overline{x} \in \mathbb{R}^m$, we have:

$$\text{FC}_{W,\overline{b}}(\overline{x}) = \text{ReLU}(W \cdot \overline{x} + \overline{b}).$$

Fig. 4a shows an FC layer computation for $\overline{x} = (2, 3, 1)$.

Convolutional Layer. A convolutional layer is defined by a series of t filters $F^{p,q} = (F_1^{p,q}, .., F_t^{p,q})$, parameterized by the same p and q, where $p \leq m$ and $q \leq n$. A filter $F_i^{p,q}$ is a function parameterized by a three-dimensional array of weights $W \in \mathbb{R}^{p \times q \times r}$ and a bias $b \in \mathbb{R}$. A filter takes a three-dimensional array and returns a two-dimensional array:

$$F_i^{p,q} \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{(m-p+1) \times (n-q+1)}.$$

The entries of the output $\overline{y}$ for a given input $\overline{x}$ are given by:

$$y_{i,j} = \text{ReLU}(\sum_{i'=1}^{p} \sum_{j'=1}^{q} \sum_{k'=1}^{r} W_{i',j',k'} \cdot x_{(i+i'-1),(j+j'-1),k'} + b).$$

Intuitively, this matrix is computed by sliding the filter along the height and width of the input three-dimensional array, each time reading a slice of size $p \times q \times r$, computing its dot product with W (resulting in a real number), adding b, and applying ReLU. The function Conv_F, corresponding to a convolutional layer with t filters, has the following type:

$$\text{Conv}_F \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{(m-p+1) \times (n-q+1) \times t}.$$

As expected, the function Conv_F returns a three-dimensional array of depth t, which stacks the outputs produced by each filter. Fig. 4b illustrates a computation of a convolutional layer with a single filter. For example:

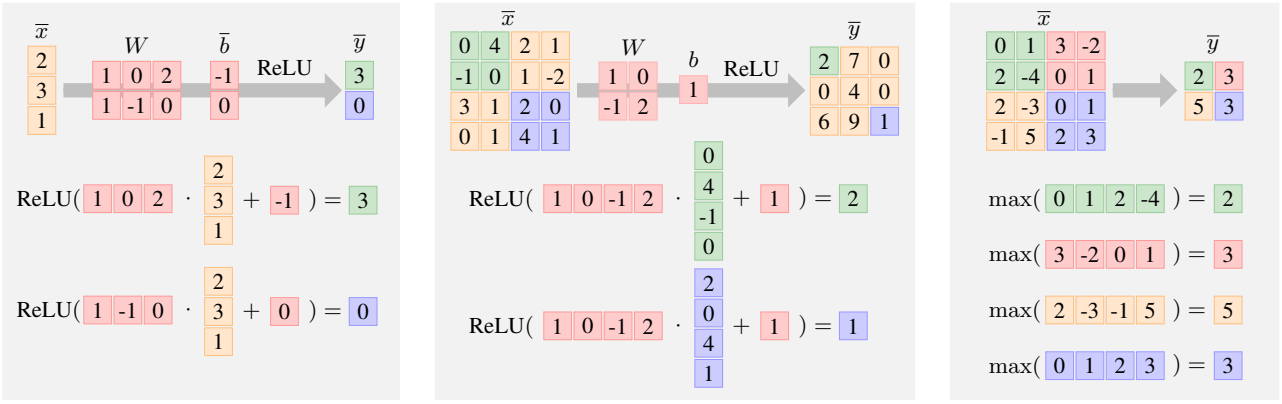
$$y_{1,1,1} = \text{ReLU}((1 \cdot 0 + 0 \cdot 4 + (-1) \cdot (-1) + 2 \cdot 0) + 1) = 2.$$

Here, the input is a three-dimensional array in $\mathbb{R}^{4 \times 4 \times 1}$. As the input depth is 1, the depth of the filter's weights is also 1. The output depth is 1 because the layer has one filter.

Convolutional Layer to CAT. For a convolutional layer $Conv_F$, we define a matrix W^F whose entries are those of the weight matrices for each filter (replicated to simulate sliding), and a bias $\overline{b}^F$ consisting of copies of the filters' biases. We then treat the convolutional layer Conv^F like the equivalent $\text{FC}_{W^F,\overline{b}^F}$. We provide formal definitions of W^F and $\overline{b}^F$ in Appendix A. Here, we provide an intuitive illustration of the translation on the example in Fig. 4b. Consider the first entry $y_{1,1} = 2$ of $\overline{y}$ in Fig. 4b:

$$y_{1,1} = \text{ReLU}(W_{1,1} \cdot x_{1,1} + W_{1,2} \cdot x_{1,2} + W_{2,1} \cdot x_{2,1} + W_{2,2} \cdot x_{2,2} + b).$$

When $\overline{x}$ is reshaped to a vector $\overline{x}^v$, the four entries $x_{1,1}, x_{1,2}, x_{2,1}$ and $x_{2,2}$ will be found in x_1^v, x_2^v, x_5^v and x_6^v, respectively. Similarly, when $\overline{y}$ is reshaped to $\overline{y}^v$, the entry $y_{1,1}$ will be found in y_1^v. Thus, to obtain $y_1^v = y_{1,1}$, we define the first row in W^F such that its 1st, 2nd, 5th, and 6th entries are $W_{1,1}$, $W_{1,2}$, $W_{2,1}$ and $W_{2,2}$. The other entries are zeros.



(a) Fully connected layer $\text{FC}_{W,\overline{b}}$ (b) Convolutional layer $\text{Conv}_{(W,b)}$ (one filter) (c) Max pooling layer $\text{MaxPool}_{2,2}$

Fig. 4: One example computation for each of the three layer types supported by AI^2.

We also define the first entry of the bias to be b. For similar reasons, to obtain $y^v_2 = y_{1,2}$, we define the second row in W^F such that its 2nd, 3rd, 6th, and 7th entries are $W_{1,1}$, $W_{1,2}$, $W_{2,1}$ and $W_{2,2}$ (also $b_2 = b$). By following this transformation, we obtain the matrix $W^F \in \mathbb{R}^9 \times \mathbb{R}^{16}$ and the bias $\overline{b}^F \in \mathbb{R}^9$:

$$W^F = \begin{pmatrix} \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{0} & 0 & 0 & \mathbf{-1} & \mathbf{2} \end{pmatrix} \quad \overline{b}^F = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

To aid understanding, we show the entries from W that appear in the resulting matrix W^F in bold.

Max Pooling (MP) Layer. An MP layer takes a three-dimensional array $\overline{x} \in \mathbb{R}^{m \times n \times r}$ and reduces the height m of $\overline{x}$ by a factor of p and the width n of $\overline{x}$ by a factor of q (for p and q dividing m and n). Depth is kept as-is. Neurons take as input disjoint subrectangles of $\overline{x}$ of size $p \times q$ and return the maximal value in their subrectangle. Formally, the MP layer is a function $\text{MaxPool}_{p,q} \colon \mathbb{R}^{m \times n \times r} \to \mathbb{R}^{\frac{m}{p} \times \frac{n}{q} \times r}$ that for an input $\overline{x}$ returns the three-dimensional array $\overline{y}$ given by:

$$y_{i,j,k} = \max(\{x_{i',j',k} \mid \begin{array}{l} p \cdot (i-1) < i' \leq p \cdot i \\ q \cdot (j-1) < j' \leq q \cdot j\}). \end{array}$$

Fig. 4c illustrates the max pooling computation for $p = 2$, $q = 2$ and $r = 1$. For example, here we have:

$$y_{1,1,1} = \max(\{x_{1,1,1}, x_{1,2,1}, x_{2,1,1}, x_{2,2,1}\}) = 2.$$

Max Pooling to CAT. Let $\text{MaxPool}'_{p,q} \colon \mathbb{R}^{m \cdot n \cdot r} \to \mathbb{R}^{\frac{m}{p} \cdot \frac{n}{q} \cdot r}$ be the function that is obtained from $\text{MaxPool}_{p,q}$ by reshaping its input and output: $\text{MaxPool}'_{p,q}(\overline{x}^v) = \text{MaxPool}_{p,q}(\overline{x})^v$. To represent max pooling as a CAT function, we define a series of CAT functions whose composition is $\text{MaxPool}'_{p,q}$:

$$\text{MaxPool}'_{p,q} = f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r} \circ \ldots \circ f_1 \circ f^{\text{MP}}.$$

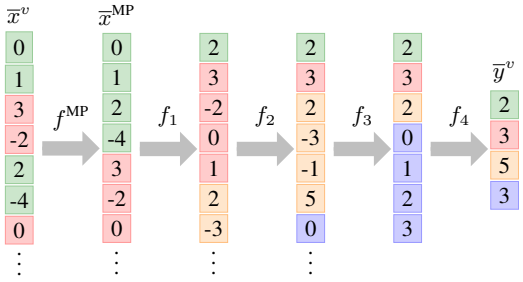


Fig. 5: The operation of the transformed max pooling layer.

The first function is $f^{\text{MP}}(\overline{x}^v) = W^{\text{MP}} \cdot \overline{x}^v$, which reorders its input vector $\overline{x}^v$ to a vector $\overline{x}^{\text{MP}}$ in which the values of each max pooling subrectangle of $\overline{x}$ are adjacent. The remaining functions execute standard max pooling. Concretely, the function $f_i \in \{f_1, \ldots, f_{\frac{m}{p} \cdot \frac{n}{q} \cdot r}\}$ executes max pooling on the i^{th} subrectangle by selecting the maximal value and removing the other values. We provide formal definitions of the CAT functions f^{MP} and f_i in Appendix A. Here, we illustrate them on the example from Fig. 4c, where $r = 1$. The CAT computation for this example is shown in Fig. 5. The computation begins from the input vector $\overline{x}^v$, which is the reshaping of $\overline{x}$ from Fig. 4c. The values of the first 2×2 subrectangle in $\overline{x}$ (namely, 0, 1, 2 and -4) are separated in $\overline{x}^v$ by values from another subrectangle (3 and -2). To make them contiguous, we reorder $\overline{x}^v$ using a permutation matrix W^{MP}, yielding $\overline{x}^{\text{MP}}$. In our example, W^{MP} is:

$$W^{\text{MP}} = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix}$$

One entry in each row of W^{MP} is 1, all other entries are zeros.

If row i has entry j set to 1, then the j^{th} value of $\overline{x}^v$ is moved to the i^{th} entry of $\overline{x}^{\text{MP}}$. For example, we placed a one in the fifth column of the third row of W^{MP} to move the value x_5^v to entry 3 of the output vector.

Next, for each $i \in \{1, \ldots, \frac{m}{p} \cdot \frac{n}{q}\}$, the function f_i takes as input a vector whose values at the indices between i and $i + p \cdot q - 1$ are those of the i^{th} subrectangle of $\bar{x}$ in Fig. 4c. It then replaces those $p \cdot q$ values by their maximum:

$$f_i(\overline{x}) = (x_1, \ldots, x_{i-1}, x_k, x_{i+p\cdot q}, \ldots, x_{m\cdot n-(p\cdot q-1)\cdot(i-1)}),$$

where the index $k \in \{i, \ldots, i + p \cdot q - 1\}$ is such that x_k is maximal. For k given, f_i can be written as a CAT function: $f_i(\overline{x}) = W^{(i,k)} \cdot \overline{x}$, where the rows of the matrix $W^{(i,k)} \in \mathbb{R}^{(m\cdot n-(p\cdot q-1)\cdot i)\times(m\cdot n-(p\cdot q-1)\cdot(i-1))}$ are given by the following sequence of standard basis vectors:

$$\overline{e}_1, \ldots, \overline{e}_{i-1}, \overline{e}_k, \overline{e}_{i+p\cdot q}, \ldots, \overline{e}_{m\cdot n-(p\cdot q-1)\cdot(i-1)}.$$

For example, in Fig. 5, $f_1(\overline{x}^{\text{MP}}) = W^{(1,3)} \cdot \overline{x}^{\text{MP}}$ deletes 0, 1 and -4. Then it moves the value 2 to the first component, and the values at indices $5, \ldots, 16$ to components $2, \ldots, 13$. Overall, $W^{(1,3)}$ is given by:

$$W^{(1,3)} = \begin{pmatrix}
0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&\mathbf{1}
\end{pmatrix}$$

As, in general, k is not known in advance, we need to write f_i as a CAT function with a different case for each possible index k of the maximal value in $\overline{x}$. For example, in Fig. 5:

$$f_1(\overline{x}) = \begin{array}{ll}
\textbf{case } (x_1 \geq x_2) \wedge (x_1 \geq x_3) \wedge (x_1 \geq x_4): & W^{(1,1)} \cdot \overline{x},\\
\textbf{case } (x_2 \geq x_1) \wedge (x_2 \geq x_3) \wedge (x_2 \geq x_4): & W^{(1,2)} \cdot \overline{x},\\
\textbf{case } (x_3 \geq x_1) \wedge (x_3 \geq x_2) \wedge (x_3 \geq x_4): & W^{(1,3)} \cdot \overline{x},\\
\textbf{case } (x_4 \geq x_1) \wedge (x_4 \geq x_2) \wedge (x_4 \geq x_3): & W^{(1,4)} \cdot \overline{x}.
\end{array}$$

In our example, the vector $\overline{x}^{\text{MP}}$ in Fig. 5 satisfies the third condition, and therefore $f_1(\overline{x}^{\text{MP}}) = W^{(1,3)} \cdot \overline{x}^{\text{MP}}$. Taking into account all four subrectangles, we obtain:

$$\text{MaxPool}'_{2,2} = f_4 \circ f_3 \circ f_2 \circ f_1 \circ f^{\text{MP}}.$$

In summary, each function f_i replaces $p \cdot q$ components of their input by the maximum value among them, suitably moving other values. For $\overline{x}^v$ in Fig. 5:

$$\text{MaxPool}'_{2,2}(\overline{x}^v) = W^{(4,7)} \cdot W^{(3,6)} \cdot W^{(2,2)} \cdot W^{(1,3)} \cdot W^{\text{MP}} \cdot \overline{x}^v.$$

Network Architectures. Two popular architectures of neural networks are fully connected feedforward (FNN) and convolutional (CNN). An FNN is a sequence of fully connected layers, while a CNN [19] consists of all previously described layer types: convolutional, max pooling, and fully connected.

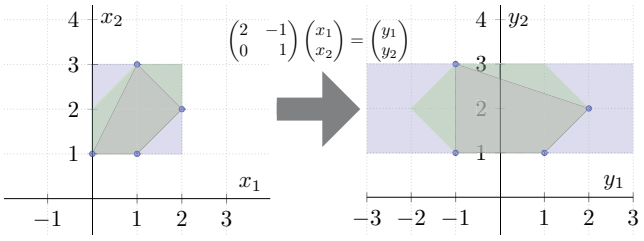


Fig. 6: (a) Abstracting four points with a polyhedron (gray), zonotope (green), and box (blue). (b) The points and abstractions resulting from the affine transformer.

III. Background: Abstract Interpretation

We now provide a short introduction to Abstract Interpretation (AI). AI enables one to prove program properties on a set of inputs *without* actually running the program. Formally, given a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$, a set of inputs $X \subseteq \mathbb{R}^m$, and a property $C \subseteq \mathbb{R}^n$, the goal is to determine whether the property holds, that is, whether $\forall \overline{x} \in X.\, f(\overline{x}) \in C$.

Fig. 6 shows a CAT function $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ that is defined as $f(\overline{x}) = \left(\begin{smallmatrix}2 & -1\\ 0 & 1\end{smallmatrix}\right) \cdot \overline{x}$ and four input points for the function f, given as $X = \{(0,1), (1,1), (1,3), (2,2)\}$. Let the property be $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$, which holds in this example. To reason about all inputs simultaneously, we lift the definition of f to be over a set of inputs X rather than a single input:

$$T_f\colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n), \qquad T_f(X) = \{f(\overline{x}) \mid \overline{x} \in X\}.$$

The function T_f is called the *concrete transformer* of f. With T_f, our goal is to determine whether $T_f(X) \subseteq C$ for a given input set X. Because the set X can be very large (or infinite), we cannot enumerate all points in X to compute $T_f(X)$. Instead, AI overapproximates sets with abstract elements (drawn from some abstract domain $\mathcal{A}$) and then defines a function, called an *abstract transformer* of f, which works with these abstract elements and overapproximates the effect of T_f. Then, the property C can be checked on the resulting abstract element returned by the abstract transformer. Naturally, because AI employs overapproximation, it is sound, but may be imprecise (i.e., may fail to prove the property when it holds). Next, we explain the ingredients of AI in more detail.

Abstract Domains. Abstract domains consist of shapes expressible as a set of logical constraints. A few popular numerical abstract domains are: Box (i.e., Interval), Zonotope, and Polyhedra. These domains provide different precision versus scalability trade-offs (e.g., Box's abstract transformers are significantly faster than Polyhedra's abstract transformers, but polyhedra are significantly more precise than boxes). The Box domain consists of boxes, captured by a set of constraints of the form $a \leq x_i \leq b$, for $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$ and $a \leq b$. A box B contains all points which satisfy all constraints in B. In our example, X can be abstracted by the following box:

$$B = \{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}.$$

Note that B is not very precise since it includes 9 integer points (along with other points), whereas X has only 4 points.

The Zonotope domain [10] consists of *zonotopes*. A zonotope is a center-symmetric convex closed polyhedron $Z \subseteq \mathbb{R}^n$ that can be represented as an affine function:

$$z\colon [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_m, b_m] \to \mathbb{R}^n.$$

In other words, z has the form $z(\overline{\epsilon}) = M \cdot \overline{\epsilon} + \overline{b}$ where $\overline{\epsilon}$ is a vector of error terms satisfying interval constraints $\overline{\epsilon}_i \in [a_i, b_i]$ for $1 \leq i \leq m$. The bias vector $\overline{b}$ captures the center of the zonotope, while the matrix M captures the boundaries of the zonotope around $\overline{b}$. A zonotope z represents all vectors in the image of z (i.e., $z[[a_{1,1}] \times \cdots \times [a_m, b_m]]$). In our example, X can be abstracted by the zonotope $z\colon [-1, 1]^3 \to \mathbb{R}^2$:

$$z(\epsilon_1, \epsilon_2, \epsilon_3) = (1 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_2, 2 + 0.5 \cdot \epsilon_1 + 0.5 \cdot \epsilon_3).$$

Zonotope is a more precise domain than Box: for our example, z includes only 7 integer points.

The Polyhedra [8] domain consists of convex closed polyhedra, where a polyhedron is captured by a set of linear constraints of the form $A \cdot \overline{x} \leq \overline{b}$, for some matrix A and a vector $\overline{b}$. A polyhedron C contains all points which satisfy the constraints in C. In our example, X can be abstracted by the following polyhedron:

$$C = \{x_2 \leq 2 \cdot x_1 + 1, x_2 \leq 4 - x_1, x_2 \geq 1, x_2 \geq x_1\}.$$

Polyhedra is a more precise domain than Zonotope: for our example, C includes only 5 integer points.

To conclude, abstract elements (from an abstract domain) represent large, potentially infinite sets. There are various abstract domains, providing different levels of precision and scalability.

Abstract Transformers. To compute the effect of a function on an abstract element, AI uses the concept of an *abstract transformer*. Given the (lifted) concrete transformer $T_f\colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n)$ of a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$, an abstract transformer of T_f is a function over abstract domains, denoted by $T_f^{\#}\colon \mathcal{A}^m \to \mathcal{A}^n$. The superscripts denote the number of components of the represented vectors. For example, elements in $\mathcal{A}^m$ represent sets of vectors of dimension m. This also determines which variables can appear in the constraints associated with an abstract element. For example, elements in $\mathcal{A}^m$ constrain the values of the variables $x_1, \ldots, x_m$.

Abstract transformers have to be *sound*. To define soundness, we introduce two functions: the *abstraction* function α and the *concretization* function γ. An abstraction function $\alpha^m\colon \mathcal{P}(\mathbb{R}^m) \to \mathcal{A}^m$ maps a set of vectors to an abstract element in $\mathcal{A}^m$ that overapproximates it. For example, in the Box domain:

$$\alpha^2(\{(0,1),(1,1),(1,3),(2,2)\}) = \{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}.$$

A concretization function $\gamma^m\colon \mathcal{A}^m \to \mathcal{P}(\mathbb{R}^m)$ does the opposite: it maps an abstract element to the set of concrete vectors that it represents. For example, for Box:

$$\gamma^2(\{0 \leq x_1 \leq 2, 1 \leq x_2 \leq 3\}) = \{(0,1),(0,2),(0,3),\\ (1,1),(1,2),(1,3),\\ (2,1),(2,2),(2,3),\ldots\}.$$

This only shows the 9 vectors with integer components. We can now define soundness. An abstract transformer $T_f^{\#}$ is *sound* if for all $a \in \mathcal{A}^m$, we have $T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^{\#}(a))$, where T_f is the concrete transformer. That is, an abstract transformer has to overapproximate the effect of a concrete transformer. For example, the transformers illustrated in Fig. 6 are sound. For instance, if we apply the Box transformer on the box in Fig. 6a, it will produce the box in Fig. 6b. The box in Fig. 6b includes all points that f could compute in principle when given any point included in the concretization of the box in Fig. 6a. Analogous properties hold for the Zonotope and Polyhedra transformers. It is also important that abstract transformers are *precise*. That is, the abstract output should include as few points as possible. For example, as we can see in Fig. 6b, the output produced by the Box transformer is less precise (it is larger) than the output produced by the Zonotope transformer, which in turn is less precise than the output produced by the Polyhedra transformer.

Property Verification. After obtaining the (abstract) output, we can check various properties of interest on the result. In general, an abstract output $a = T_f^{\#}(X)$ proves a property $T_f(X) \subseteq C$ if $\gamma^n(a) \subseteq C$. If the abstract output proves a property, we know that the property holds for all possible concrete values. However, the property may hold even if it cannot be proven with a given abstract domain. For example, in Fig. 6b, for all concrete points, the property $C = \{(y_1, y_2) \in \mathbb{R}^2 \mid y_1 \geq -2\}$ holds. However, with the Box domain, the abstract output violates C, which means that the Box domain is not precise enough to prove the property. In contrast, the Zonotope and Polyhedra domains are precise enough to prove the property.

In summary, to apply AI successfully, we need to: (a) find a suitable abstract domain, and (b) define abstract transformers that are sound and as precise as possible. In the next section, we introduce abstract transformers for neural networks that are parameterized by the numerical abstract domain. This means that we can explore the precision-scalability trade-off by plugging in different abstract domains.

IV. AI^2: AI for Neural Networks

In this section we present AI^2, an abstract interpretation framework for sound analysis of neural networks. We begin by defining abstract transformers for the different kinds of neural network layers. Using these transformers, we then show how to prove robustness properties of neural networks.

A. Abstract Interpretation for CAT Functions

In this section, we show how to overapproximate CAT functions with AI. We illustrate the method on the example in

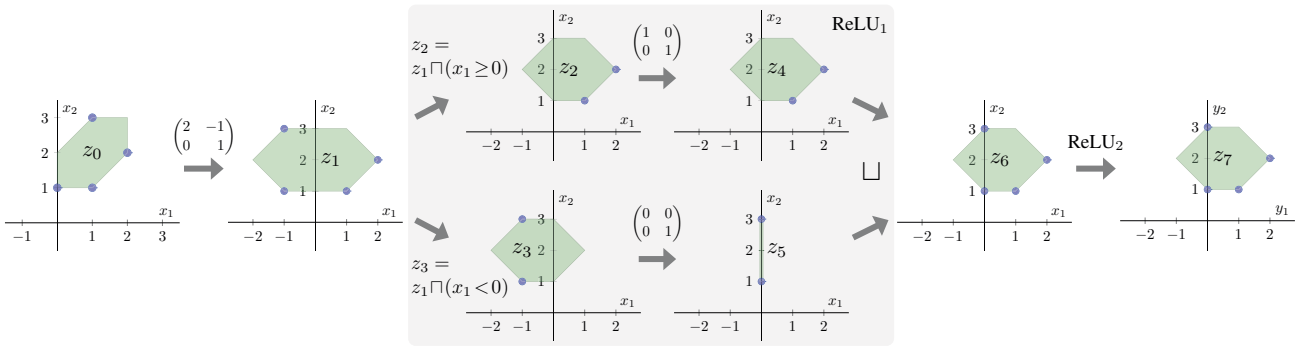


Fig. 7: Illustration of how AI^2 overapproximates neural network states. Blue circles show the concrete values, while green zonotopes show the abstract elements. The gray box shows the steps in one application of the ReLU transformer (ReLU$_1$).

Fig. 7, which shows a simple network that manipulates two-dimensional vectors using a single fully connected layer of the form $f(\overline{x}) = \text{ReLU}_2\left(\text{ReLU}_1\left(\left(\begin{smallmatrix}2 & -1\\ 0 & 1\end{smallmatrix}\right) \cdot \overline{x}\right)\right)$. Recall that

$$\begin{aligned}\text{ReLU}_i(\overline{x}) \quad = \quad & \textbf{case } (x_i \geq 0)\colon \overline{x},\\ & \textbf{case } (x_i < 0)\colon I_{i\leftarrow 0} \cdot \overline{x},\end{aligned}$$

where $I_{i\leftarrow 0}$ is the identity matrix with the i^{th} row replaced by the zero vector. We overapproximate the network behavior on an abstract input. The input can be obtained directly (see Sec. IV-B) or by abstracting a set of concrete inputs to an abstract element (using the abstraction function α). For our example, we use the concrete inputs (the blue points) from Fig 6. Those concrete inputs are abstracted to the green zonotope $z_0\colon [-1,1]^3 \to \mathbb{R}^2$, given as:

$$z_0(\epsilon_1,\epsilon_2,\epsilon_3) = (1 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_2, 2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3).$$

Due to abstraction, more (spurious) points may be added. In this example, except the blue points, the entire area of the zonotope is spurious. We then apply abstract transformers to the abstract input. Note that, if a function f can be written as $f = f'' \circ f'$, the concrete transformer for f is $T_f = T_{f''} \circ T_{f'}$. Similarly, given abstract transformers $T^{\#}_{f'}$ and $T^{\#}_{f''}$, an abstract transformer for f is $T^{\#}_{f''} \circ T^{\#}_{f'}$. When a neural network $N = f'_\ell \circ \cdots \circ f'_1$ is a composition of multiple CAT functions f'_i of the shape $f'_i(\overline{x}) = W \cdot \overline{x} + \overline{b}$ or $f_i(\overline{x}) = \textbf{case } E_1\colon f_1(\overline{x}), \ldots, \textbf{case } E_k\colon f_k(\overline{x})$, we only have to define abstract transformers for these two kinds of functions. We then obtain the abstract transformer $T^{\#}_N = T^{\#}_{f'_\ell} \circ \cdots \circ T^{\#}_{f'_1}$.

Abstracting Affine Functions. To abstract functions of the form $f(\overline{x}) = W \cdot \overline{x} + \overline{b}$, we assume that the underlying abstract domain supports the operator Aff that overapproximates such functions. We note that for Zonotope and Polyhedra, this operation is supported and exact. Fig. 7 demonstrates Aff as the first step taken for overapproximating the effect of the fully connected layer. Here, the resulting zonotope $z_1 : [-1,1]^3 \to \mathbb{R}^2$ is:

$$\begin{aligned}z_1(\epsilon_1,\epsilon_2,\epsilon_3) = \\ (2\cdot(1 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_2) - (2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3),\\ 2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3) = \\ (0.5\cdot\epsilon_1 + \epsilon_2 - 0.5\cdot\epsilon_3, 2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3).\end{aligned}$$

Abstracting Case Functions. To abstract functions of the form $f(\overline{x}) = \textbf{case } E_1\colon f_1(\overline{x}), \ldots, \textbf{case } E_k\colon f_k(\overline{x})$, we first split the abstract element a into the different cases (each defined by one of the expressions E_i), resulting in k abstract elements $a_1, \ldots, a_k$. We then compute the result of $T^{\#}_{f_i}(a_i)$ for each a_i. Finally, we unify the results to a single abstract element. To split and unify, we assume two standard operators for abstract domains: (1) *meet* with a conjunction of linear constraints and (2) *join*. The meet ($\sqcap$) operator is an abstract transformer for set intersection: for an inequality expression E from Fig. 3, $\gamma^n(a) \cap \{x \in \mathbb{R}^n \mid x \models E\} \subseteq \gamma^n(a \sqcap E)$. The *join* ($\sqcup$) operator is an abstract transformer for set union: $\gamma^n(a_1) \cup \gamma^n(a_2) \subseteq \gamma^n(a_1 \sqcup a_2)$. We further assume that the abstract domain contains an element $\bot$, which satisfies $\gamma^n(\bot) = \{\}$, $\bot \sqcap E = \bot$ and $a \sqcup \bot = a$ for $a \in \mathcal{A}$.

For our example in Fig. 7, abstract interpretation continues on z_1 using the meet and join operators. To compute the effect of ReLU$_1$, z_1 is split into two zonotopes $z_2 = z_1 \sqcap (x_1 \geq 0)$ and $z_3 = z_1 \sqcap (x_1 < 0)$. One way to compute a meet between a zonotope and a linear constraint is to modify the intervals of the error terms (see [11]). In our example, the resulting zonotopes are $z_2 : [-1,1] \times [0,1] \times [-1,1] \to \mathbb{R}^2$ such that $z_2(\overline{\epsilon}) = z_1(\overline{\epsilon})$ and $z_3 : [-1,1] \times [-1,0] \times [-1,1] \to \mathbb{R}^2$ such that $z_3(\overline{\epsilon}) = z_1(\overline{\epsilon})$ for $\overline{\epsilon}$ common to their respective domains. Note that both z_2 and z_3 contain small spurious areas, because the intersections of the respective linear constraints with z_1 are not zonotopes. Therefore, they cannot be captured exactly by the domain. Here, the meet operator $\sqcap$ overapproximates set intersection $\cap$ to get a sound, but not perfectly precise, result.

Then, the two cases of ReLU$_1$ are processed separately. We apply the abstract transformer of $f_1(\overline{x}) = \overline{x}$ to z_2 and we apply the abstract transformer of $f_2(\overline{x}) = I_{0\leftarrow 0} \cdot \overline{x}$ to z_3. The resulting zonotopes are $z_4 = z_2$ and $z_5 : [-1,1]^2 \to \mathbb{R}^2$ such that $z_5(\epsilon_1,\epsilon_3) = (0, 2 + 0.5\cdot\epsilon_1 + 0.5\cdot\epsilon_3)$. These are then joined to obtain a single zonotope z_6. Since z_5 is contained in z_4, we get $z_6 = z_4$ (of course, this need not always be the case). Then, z_6 is passed to ReLU$_2$. Because $z_6 \sqcap (x_1 < 0) = \bot$, this results in $z_7 = z_6$. Finally, $\gamma^2(z_7)$ is our overapproximation of the network outputs for our initial set of points. The abstract element z_7 is a finite representation of this infinite set.

$$\text{For } f(\overline{x}) = W \cdot \overline{x} + \overline{b},\ T_f^{\#}(a) = \text{Aff}(a, W, \overline{b}).$$

$$\text{For } f(\overline{x}) = \textbf{case } E_1 \colon f_1(\overline{x}), \ldots, \textbf{case } E_k \colon f_k(\overline{y}),$$
$$T_f^{\#}(a) = \bigsqcup_{1 \le i \le k} {f_i}^{\#}(a \sqcap E_i).$$

$$\text{For } f(\overline{x}) = f_2(f_1(\overline{x})),\ T_f^{\#}(a) = T_{f_2}^{\#}(T_{f_1}^{\#}(a)).$$

Fig. 8: Abstract transformers for CAT functions.

In summary, we define abstract transformers for every kind of CAT function (summarized in Fig. 8). These definitions are general and are compatible with any abstract domain $\mathcal{A}$ which has a minimum element $\bot$ and supports (1) a meet operator between an abstract element and a conjunction of linear constraints, (2) a join operator between two abstract elements, and (3) an affine transformer. We assume that the operations are sound. We note that these operations are standard or definable with standard operations. By definition of the abstract transformers, we get soundness:

Theorem 1. *For any CAT function f with transformer $T_f : \mathcal{P}(\mathbb{R}^m) \to \mathcal{P}(\mathbb{R}^n)$ and any abstract input $a \in \mathcal{A}^m$,*

$$T_f(\gamma^m(a)) \subseteq \gamma^n(T_f^{\#}(a)).$$

Theorem 1 is the key to sound neural network analysis with our abstract transformers, as we explain in the next section.

B. Neural Network Analysis with AI

In this section, we explain how to leverage AI with our abstract transformers to prove properties of neural networks. We focus on robustness properties below, however, the framework can be applied to reason about any safety property.

For robustness, we aim to determine if for a (possibly unbounded) set of inputs, the outputs of a neural network satisfy a given condition. A robustness property for a neural network $N \colon \mathbb{R}^m \to \mathbb{R}^n$ is a pair $(X, C) \in \mathcal{P}(\mathbb{R}^m) \times \mathcal{P}(\mathbb{R}^n)$ consisting of a robustness region X and a robustness condition C. We say that the neural network N satisfies a robustness property (X, C) if $N(\overline{x}) \in C$ for all $\overline{x} \in X$.

Local Robustness. This is a property (X, C_L) where X is a robustness region and C_L contains the outputs that describe the *same* label L:

$$C_L = \left\{ \overline{y} \in \mathbb{R}^n \;\middle|\; \underset{i \in \{1,\ldots,n\}}{\arg\max}\,(y_i) = L \right\}.$$

For example, Fig. 7 shows a neural network and a robustness property (X, C_2) for $X = \{(0,1),(1,1),(1,3),(2,2)\}$ and $C_2 = \{\overline{y} \mid \arg\max(y_1, y_2) = 2\}$. In this example, (X, C_2) holds. Typically, we will want to check that there is *some* label L for which (X, C_L) holds.

We now explain how our abstract transformers can be used to prove a given robustness property (X, C).

Robustness Proofs using AI. Assume we are given a neural network $N \colon \mathbb{R}^m \to \mathbb{R}^n$, a robustness property (X, C) and an abstract domain $\mathcal{A}$ (supporting $\sqcup$, $\sqcap$ with a conjunction of linear constraints, Aff, and $\bot$) with an abstraction function α and a concretization function γ. Further assume that N can be written as a CAT function. Denote by $T_N^{\#}$ the abstract transformer of N, as defined in Fig. 8. Then, the following condition is sufficient to prove that N satisfies (X, C):

$$\gamma^n(T_N^{\#}(\alpha^m(X))) \subseteq C.$$

This follows from Theorem 1 and the properties of α and γ. Note that there may be abstract domains $\mathcal{A}$ that are not precise enough to prove that N satisfies (X, C), even if N in fact satisfies (X, C). On the other hand, if we are able to show that some abstract domain $\mathcal{A}$ proves that N satisfies (X, C), we know that it holds.

Proving Containment. To prove the property (X, C) given the result $a = T_N^{\#}(\alpha^m(X))$ of abstract interpretation, we need to be able to show $\gamma^n(a) \subseteq C$. There is a general method if C is given by a CNF formula $\bigwedge_i \bigvee_j l_{i,j}$ where all literals $l_{i,j}$ are linear constraints: we show that the negated formula $\bigvee_i \bigwedge_j \neg l_{i,j}$ is inconsistent with the abstract element a by checking that $a \sqcap \left(\bigwedge_j \neg l_{i,j}\right) = \bot$ for all i.

For our example in Fig. 7, the goal is to check that all inputs are classified as 2. We can represent C using the formula $y_2 \ge y_1$. Its negation is $y_2 < y_1$, and it suffices to show that no point in the concretization of the abstract output satisfies this negated constraint. As indeed $z_7 \sqcap (y_2 < y_1) = \bot$, the property is successfully verified. However, note that we would not be able to prove some other true properties, such as $y_1 \ge 0$. This property holds for all concrete outputs, but some points in the concretization of the output z_7 do not satisfy it.

V. Implementation of AI^2

The AI^2 framework is implemented in the D programming language and supports any neural network composed of fully connected, convolutional, and max pooling layers.

Properties. AI^2 supports properties (X, C) where X is specified by a zonotope and C by a conjunction of linear constraints over the output vector's components. In our experiments, we illustrate the specification of local robustness properties where the region X is defined by a box or a line, both of which are precisely captured by a zonotope.

Abstract Domains. The AI^2 system is fully integrated with all abstract domains supported by Apron [20], a popular library for numerical abstract domains, including: Box [7], Zonotope [10], and Polyhedra [8].

Bounded Powerset. We also implemented bounded powerset domains (disjunctive abstractions [33], [36]), which can be instantiated with any of the above abstract domains. An abstract element in the powerset domain $\mathcal{P}(\mathcal{A})$ of an underlying abstract domain $\mathcal{A}$ is a set of abstract elements from $\mathcal{A}$, concretizing to the union of the concretizations of the individual elements (i.e., $\gamma(A) = \bigcup_{a \in A} \gamma(a)$ for $A \in \mathcal{P}(\mathcal{A})$).

The powerset domain can implement a precise join operator using standard set union (potentially pruning redundant elements). However, since the increased precision can become

prohibitively costly if many join operations are performed, the bounded powerset domain limits the number of abstract elements in a set to N (for some constant N).

We implemented bounded powerset domains on top of standard powerset domains using a greedy heuristic that repeatedly replaces two abstract elements in a set by their join, until the number of abstract elements in the set is below the bound N. For joining, AI^2 heuristically selects two abstract elements that minimize the distance between the centers of their bounding boxes. In our experiments, we denote by Zonotope$N$ or ZN the bounded powerset domain with bound $N \geq 2$ and underlying abstract domain Zonotope.

VI. Evaluation of AI^2

In this section, we present our empirical evaluation of AI^2. Before discussing the results in detail, we summarize our three most important findings:

- AI^2 can prove useful robustness properties for convolutional networks with $53\,000$ neurons and large fully connected feedforward networks with $1\,800$ neurons.
- AI^2 benefits from more precise abstract domains: Zonotope enables AI^2 to prove substantially more properties over Box. Further, Zonotope$N$, with $N \geq 2$, can prove stronger robustness properties than Zonotope alone.
- AI^2 scales better than the SMT-based Reluplex [21]: AI^2 is able to verify robustness properties on large networks with ≥ 1200 neurons within few minutes, while Reluplex takes hours to verify the same properties.

In the following, we first describe our experimental setup. Then, we present and discuss our results.

A. Experimental Setup

We now describe the datasets, neural networks, and robustness properties used in our experiments.

Datasets. We used two popular datasets: MNIST [25] and CIFAR-10 [22] (referred to as CIFAR from now on). MNIST consists of $60\,000$ grayscale images of handwritten digits, whose resolution is 28×28 pixels. The images show white digits on a black background.

CIFAR consists of $60\,000$ colored photographs with 3 color channels, whose resolution is 32×32 pixels. The images are partitioned into 10 different classes (e.g., airplane or bird). Each photograph has a different background (unlike MNIST).

Neural Networks. We trained convolutional and fully connected feedforward networks on both datasets. All networks were trained using accelerated gradient descent with at least 50 epochs of batch size 128. The training completed when each network had a test set accuracy of at least 0.9.

For the convolutional networks, we used the LeNet architecture [26], which consists of the following sequence of layers: 2 convolutional, 1 max pooling, 2 convolutional, 1 max pooling, and 3 fully connected layers. We write $n_{p\times q}$ to denote a convolutional layer with n filters of size $p \times q$, and m to denote a fully connected layer with m neurons. The hidden layers of the MNIST network are $8_{3\times3}, 8_{3\times3}, 14_{3\times3}, 14_{3\times3}, 50, 50, 10$, and those of the CIFAR network are $24_{3\times3}, 24_{3\times3}, 32_{3\times3}, 32_{3\times3}, 100, 100, 10$. The max pooling layers of both networks have a size of 2×2. We trained our networks using an open-source implementation [37].

We used 7 different architectures of fully connected feedforward networks (FNNs). We write $l \times n$ to denote the FNN architecture with l layers, each consisting of n neurons. Note that this determines the network's size; e.g., a 4×50 network has 200 neurons. For each dataset, MNIST and CIFAR, we trained FNNs with the following architectures: 3×20, 6×20, 3×50, 3×100, 6×100, 6×200, and 9×200.

Robustness Properties. In our experiments, we consider local robustness properties (X, C_L) where the region X captures changes to lighting conditions. This type of property is inspired by the work of [32], where adversarial examples were found by brightening the pixels of an image.

Formally, we consider robustness regions $S_{\overline{x},\delta}$ that are parameterized by an input $\overline{x} \in \mathbb{R}^m$ and a robustness bound $\delta \in [0, 1]$. The robustness region is defined as:

$$S_{\overline{x},\delta} = \{\overline{x}' \in \mathbb{R}^m \mid \forall i \in [1, m].\ 1-\delta \leq x_i \leq x'_i \leq 1 \vee x'_i = x_i\}.$$

For example, the robustness region for $\overline{x} = (0.6, 0.85, 0.9)$ and bound $\delta = 0.2$ is given by the set:

$$\{(0.6, x, x') \in \mathbb{R}^3 \mid x \in [0.85, 1], x' \in [0.9, 1]\}.$$

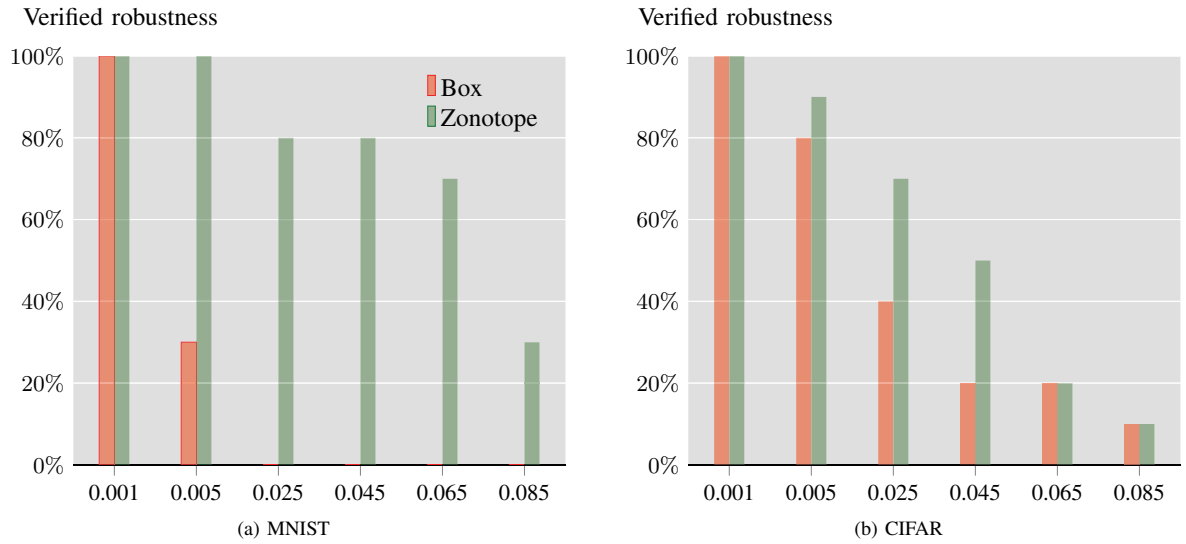
Note that increasing the bound δ increases the region's size.

In our experiments, we used AI^2 to check whether all inputs in a given region $S_{\overline{x},\delta}$ are classified to the label assigned to $\overline{x}$. We consider 6 different robustness bounds δ, which are drawn from the set $\Delta = \{0.001, 0.005, 0.025, 0.045, 0.065, 0.085\}$.

We remark that our robustness properties are stronger than those considered in [32]. This is because, in a given robustness region $S_{\overline{x},\delta}$, each pixel of the image $\overline{x}$ is brightened independently of the other pixels. We remark that this is useful to capture scenarios where only part of the image is brightened (e.g., due to shadowing).

Other perturbations. Note that AI^2 is not limited to certifying robustness against such brightening perturbations. In general, AI^2 can be used whenever the set of perturbed inputs can be overapproximated with a set of zonotopes in a precise way (i.e., without adding too many inputs that do not capture actual perturbations to the robustness region). For example, the inputs perturbed by an ℓ_∞ attack [3] are captured exactly by a single zonotope. Further, rotations and translations have low-dimensional parameter spaces, and therefore can be overapproximated by a set of zonotopes in a precise way.

Benchmarks. We selected 10 images from each dataset. Then, we specified a robustness property for each image and each robustness bound in Δ, resulting in 60 properties per dataset. We ran AI^2 to check whether each neural network satisfies the robustness properties for the respective dataset. We compared the results using different abstract domains,



(a) MNIST (b) CIFAR

Fig. 9: Verified properties by AI^2 on the MNIST and CIFAR convolutional networks for each bound $\delta \in \Delta$ (x-axis).

including Box, Zonotope, and Zonotope$N$ with N ranging between 2 and 128.

We ran all experiments on an Ubuntu 16.04.3 LTS server with two Intel Xeon E5-2690 processors and 512GB of memory. To compare AI^2 to existing solutions, we also ran the FNN benchmarks with Reluplex [21]. We did not run convolutional benchmarks with Reluplex as it currently does not support convolutional networks.

B. Discussion of Results

In the following, we first present our results for convolutional networks. Then, we present experiments with different abstract domains and discuss how the domain's precision affects AI^2's ability to verify robustness. We also plot AI^2's running times for different abstract domains to investigate the trade-off between precision and scalability. Finally, we compare AI^2 to Reluplex.

Proving Robustness of Convolutional Networks. We start with our results for convolutional networks. AI^2 terminated within 1.5 minutes when verifying properties on the MNIST network and within 1 hour when verifying the CIFAR network.

In Fig. 9, we show the fraction of robustness properties verified by AI^2 for each robustness bound. We plot separate bars for Box and Zonotope to illustrate the effect of the domain's precision on AI^2's ability to verify robustness.

For both networks, AI^2 verified all robustness properties for the smallest bound 0.001 and it verified at least one property for the largest bound 0.085. This demonstrates that AI^2 can verify properties of convolutional networks with rather wide robustness regions. Further, the number of verified properties converges to zero as the robustness bound increases. This is expected, as larger robustness regions are more likely to contain adversarial examples.

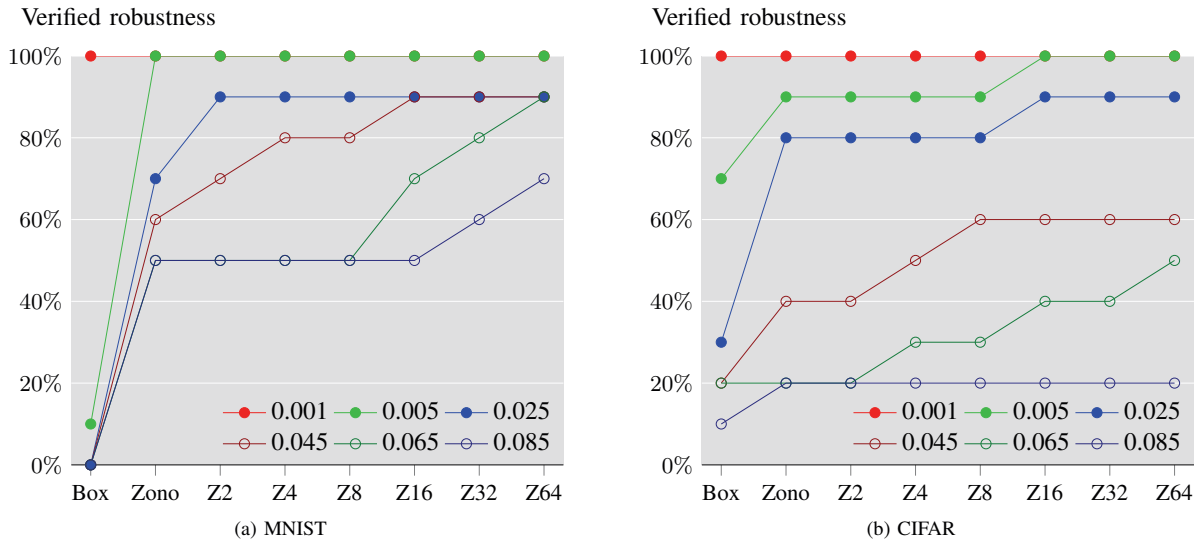
In Fig. 9a, we observe that Zonotope proves significantly more properties than Box. For example, Box fails to prove any robustness properties with bounds at least 0.025, while Zonotope proves 80% of the properties with bounds 0.025 and 0.045. This indicates that Box is often imprecise and fails to prove properties that the network satisfies.

Similarly, Fig. 9b shows that Zonotope proves more robustness properties than Box also for the CIFAR convolutional network. The difference between these two domains is, however, less significant than that observed for the MNIST network. For example, both Box and Zonotope prove the same properties for bounds 0.065 and 0.085.

Precision of Different Abstract Domains. Next, we demonstrate that more precise abstract domains enable AI^2 to prove stronger robustness properties. In this experiment, we consider our 9×200 MNIST and CIFAR networks, which are our largest fully connected feedforward networks. We evaluate the Box, Zonotope, and the Zonotope$N$ domains for exponentially increasing bounds of N between 2 and 64. We do not report results for the Polyhedra domain, which takes several days to terminate for our smallest networks.

In Fig. 10, we show the fraction of verified robustness properties as a function of the abstract domain used by AI^2. We plot a separate line for each robustness bound. All runs of AI^2 in this experiment completed within 1 hour.

The graphs show that Zonotope proves more robustness properties than Box. For the MNIST network, Box proves 11 out of all 60 robustness properties (across all 6 bounds), failing to prove any robustness properties with bounds above 0.005. In contrast, Zonotope proves 43 out of the 60 properties and proves at least 50% of the properties across the 6 robustness bounds. For the CIFAR network, Box proves 25 out of the 60 properties while Zonotope proves 35.



(a) MNIST (b) CIFAR

Fig. 10: Verified properties as a function of the abstract domain used by AI^2 for the 9×200 network. Each point represents the fraction of robustness properties for a given bound (as specified in the legend) verified by a given abstract domain (x-axis).

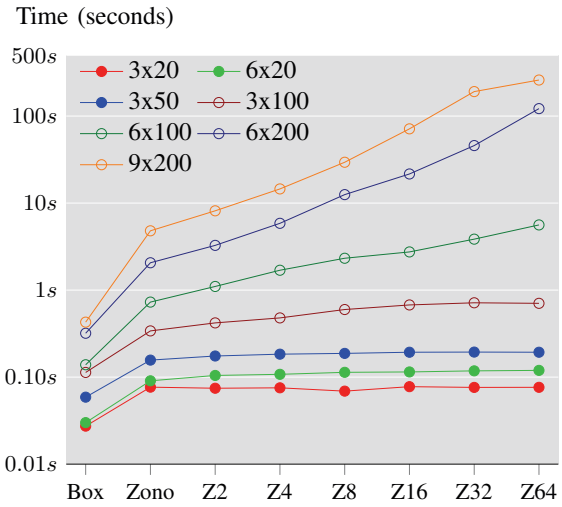


Fig. 11: Average running time of AI^2 when proving robustness properties on MNIST networks as a function of the abstract domain used by AI^2 (x-axis). Axes are scaled logarithmically.

The data also demonstrates that bounded sets of zonotopes further improve AI^2's ability to prove robustness properties. For the MNIST network, Zonotope64 proves more robustness properties than Zonotope for all 4 bounds for which Zonotope fails to prove at least one property (i.e., for bounds $\delta \geq 0.025$). For the CIFAR network, Zonotope64 proves more properties than Zonotope for 4 out of the 5 the bounds. The only exception is the bound 0.085, where Zonotope64 and Zonotope prove the same set of properties.

Trade-off between Precision and Scalability. In Fig. 11, we plot the running time of AI^2 as a function of the abstract domain. Each point in the graph represents the average running time of AI^2 when proving a robustness property for a given MNIST network (as indicated in the legend). We use a log-log plot to better visualize the trade-off in time.

The data shows that AI^2 can efficiently verify robustness of large networks. AI^2 terminates within a few minutes for all MNIST FNNs and all considered domains. Further, we observe that AI^2 takes less than 10 seconds on average to verify a property with the Zonotope domain.

As expected, the graph demonstrates that more precise domains increase AI^2's running time. More importantly, AI^2's running time is polynomial in the bound N of Zonotope$N$, which allows one to adjust AI^2's precision by increasing N.

Comparison to Reluplex. The current state-of-the-art system for verifying properties of neural networks is Reluplex [21]. Reluplex supports FNNs with ReLU activation functions, and its analysis is sound and complete. Reluplex would eventually either verify a given property or return a counterexample.

To compare the performance of Reluplex and AI^2, we ran both systems on all MNIST FNN benchmarks. We ran AI^2 using Zonotope and Zonotope64. For both Reluplex and AI^2, we set a 1 hour timeout for verifying a single property.

Fig. 12 presents our results: Fig. 12a plots the average running time of Reluplex and AI^2 and Fig. 12b shows the fraction of robustness properties verified by the systems. The data shows that Reluplex can analyze FNNs with at most 600 neurons efficiently, typically within a few minutes. Overall, both system verified roughly the same set of properties. However, Reluplex crashed during verification of some of the properties. This explains why AI^2 was able to prove slightly

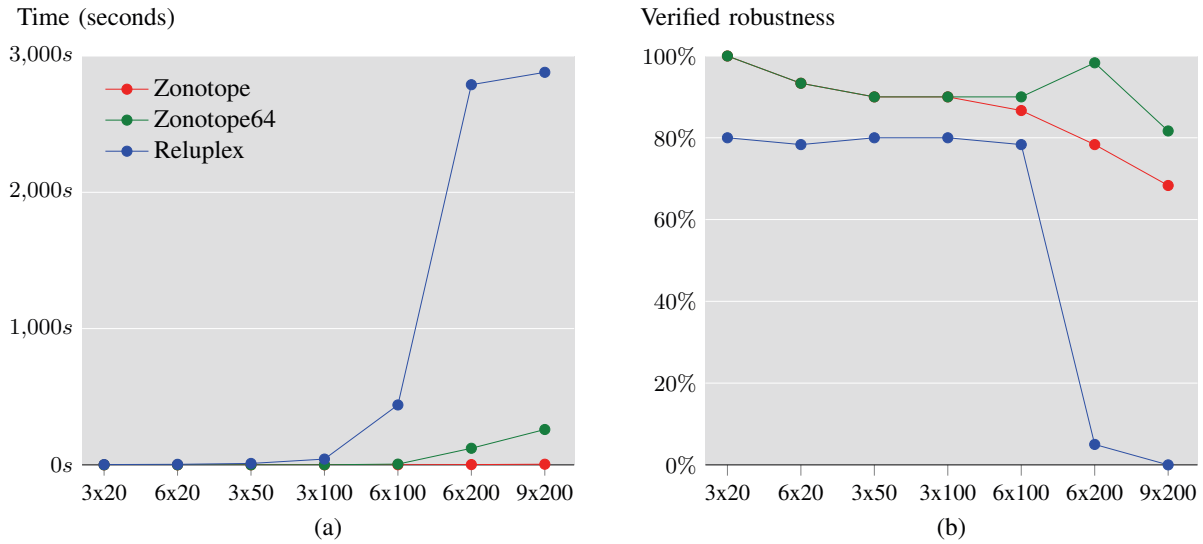


Fig. 12: Comparing the performance of AI^2 to Reluplex. Each point is an average of the results for all 60 robustness properties for the MNIST networks. Each point in (a) represents the average time to completion, regardless of the result of the computation. While not shown, the result of the computation could be a failure to verify, timeout, crash, or discovery of a counterexample. Each point in (b) represents the fraction of the 60 robustness properties that were verified.

more properties than Reluplex on the smaller FNNs.

For large networks with more than 600 neurons, the running time of Reluplex increases significantly and its analysis often times out. In contrast, AI^2 analyzes the large networks within a few minutes and verifies substantially more robustness properties than Reluplex. For example, Zonotope64 proves 57 out of the 60 properties on the 6×200 network, while Reluplex proves 3. Further, Zonotope64 proves 45 out of the 60 properties on the largest 9×200 network, while Reluplex proves none. We remark that while Reluplex did not verify any property on the largest 9×200 network, it did disprove some of the properties and returned counterexamples.

We also ran Reluplex without a predefined timeout to investigate how long it would take to verify properties on the large networks. To this end, we ran Reluplex on properties that AI^2 successfully verified. We observed that Reluplex often took more than 24 hours to terminate. Overall, our results indicate that Reluplex does not scale to larger FNNs whereas AI^2 succeeds on these networks.

VII. Comparing Defenses with AI^2

In this section, we illustrate a practical application of AI^2: evaluating and comparing neural network defenses. A defense is an algorithm whose goal is to reduce the effectiveness of a certain attack against a specific network, for example, by retraining the network with an altered loss function. Since the discovery of adversarial examples, many works have suggested different kinds of defenses to mitigate this phenomenon (e.g., [12], [27], [42]). A natural metric to compare defenses is the average "size" of the robustness region on some test set. Intuitively, the greater this size is, the more robust the defense.

We compared three state-of-the-art defenses:

- **GSS** [12] extends the loss with a regularization term encoding the fast gradient sign method (FGSM) attack.
- **Ensemble** [42] is similar to GSS, but includes regularization terms from attacks on other models.
- **MMSTV** [27] adds, during training, a perturbation layer before the input layer which applies the FGSM^k attack. FGSM^k is a multi-step variant of FGSM, also known as projected gradient descent.

All these defenses attempt to reduce the effectiveness of the FGSM attack [12]. This attack consists of taking a network N and an input $\overline{x}$ and computing a vector $\overline{\rho}_{N,\overline{x}}$ in the input space along which an adversarial example is likely to be found. An adversarial input $\overline{a}$ is then generated by taking a step ϵ along this vector: $\overline{a} = \overline{x} + \epsilon \cdot \overline{\rho}_{N,\overline{x}}$.

We define a new kind of robustness region, called *line*, that captures resilience with respect to the FGSM attack. The line robustness region captures all points from $\overline{x}$ to $\overline{x} + \delta \cdot \overline{\rho}_{N,\overline{x}}$ for some robustness bound δ:

$$L_{N,\overline{x},\delta} = \{\overline{x} + \epsilon \cdot \overline{\rho}_{N,\overline{x}} \mid \epsilon \in [0, \delta]\}.$$

This robustness region is a zonotope and can thus be precisely captured by AI^2.

We compared the three state-of-the-art defenses on the MNIST convolutional network described in Section VI; we call this the Original network. We trained the Original network with each of the defenses, which resulted in 3 additional networks: GSS, Ensemble, and MMSTV. We used 40 epochs for GSS, 12 epochs for Ensemble, and 10 000 training steps for MMSTV using their published frameworks.

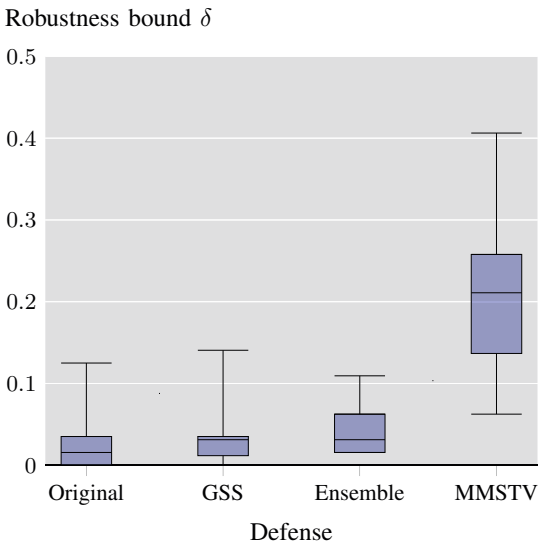


Fig. 13: Box-and-whisker plot of the verified bounds for the Original, GSS, Ensemble, and MMSTV networks. The boxes represent the δ for the middle 50% of the images, whereas the whiskers represent the minimum and maximum δ. The inner-lines are the averages.

We conducted 20 experiments. In each experiment, we randomly selected an image $\overline{x}$ and computed $\overline{\rho}_{N,\overline{x}}$. Then, for each network, our goal was to find the largest bound δ for which AI^2 proves the region $L_{N,\overline{x},\delta}$ robust. To approximate the largest robustness bound, we ran binary search to depth 6 and ran AI^2 with the Zonotope domain for each candidate bound δ. We refer to the largest robustness bound verified by AI^2 as the *verified bound*.

The average verified bounds for the Original, GSS, Ensemble, and MMSTV networks are 0.026, 0.031, 0.042, and 0.209, respectively. Fig. 13 shows a box-and-whisker plot which demonstrates the distribution of the verified bounds for the four networks. The bottom and top of each whisker show the minimum and maximum verified bounds discovered during the 20 experiments. The bottom and top of each whisker's box show the first and third quartiles of the verified bounds.

Our results indicate that MMSTV provides a significant increase in provable robustness against the FGSM attack. In all 20 experiments, the verified bound for the MMSTV network was larger than those found for the Original, GSS, and Ensemble networks. We observe that GSS and Ensemble defend the network in a way that makes it only slightly more provably robust, consistent with observations that these styles of defense are insufficient [16], [27].

VIII. Related Work

In this section, we survey the works closely related to ours.

Adversarial Examples. [40] showed that neural networks are vulnerable to small perturbations on inputs. Since then, many works have focused on constructing adversarial examples. For example, [30] showed how to find adversarial examples without starting from a test point, [41] generated adversarial examples using random perturbations, [35] demonstrated that even intermediate layers are not robust, and [14] generated adversarial examples for malware classification. Other works presented ways to construct adversarial examples during the training phase, thereby increasing the network robustness (see [3], [12], [15], [17], [29], [38]). [1] formalized the notion of robustness in neural networks and defined metrics to evaluate the robustness of a neural network. [32] illustrated how to systematically generate adversarial examples that cover all neurons in the network.

Neural Network Analysis. Many works have studied the robustness of networks. [34] presented an abstraction-refinement approach for FNNs. However, this was shown successful for a network with only 6 neurons. [37] introduced a bounded model checking technique to verify safety of a neural network for the Cart Pole system. [18] showed a verification framework, based on an SMT solver, which verified the robustness with respect to a certain set of functions that can manipulate the input and are *minimal* (a notion which they define). However, it is unclear how one can obtain such a set. [21] extended the simplex algorithm to verify properties of FNNs with ReLU.

Robustness Analysis of Programs. Many works deal with robustness analysis of programs (e.g., [4], [5], [13], [28]). [28] considered a definition of robustness that is similar to the one in our work, and [5] used a combination of abstract interpretation and SMT-based methods to prove robustness of programs. The programs considered in this literature tend to be small but have complex constructs such as loops and array operations. In contrast, neural networks (which are our focus) are closer to circuits, in that they lack high-level language features but are potentially massive in size.

IX. Conclusion and Future Work

We presented AI^2, the first system able to certify convolutional and large fully connected networks. The key insight behind AI^2 is to phrase the problem of analyzing neural networks in the classic framework of abstract interpretation. To this end, we defined abstract transformers that capture the behavior of common neural network layers and presented a bounded powerset domain that enables a trade-off between precision and scalability. Our experimental results showed that AI^2 can effectively handle neural networks that are beyond the reach of existing methods.

We believe AI^2 and the approach behind it is a promising step towards ensuring the safety and robustness of AI systems. Currently, we are extending AI^2 with additional abstract transformers to support more neural network features. We are also building a library for modeling common perturbations, such as rotations, smoothing, and erosion. We believe these extensions would further improve AI^2's applicability and foster future research in AI safety.

REFERENCES

[1] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS)*, pages 2621–2629, 2016.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[3] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.

[4] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *Proceedings of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 57–70, 2010.

[5] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 102–112. ACM, 2011.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[9] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017.

[10] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 627–633, 2009.

[11] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV)*, 2010.

[12] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014.

[13] Eric Goubault and Sylvie Putot. Robustness analysis of finite precision implementations. In *Programming Languages and Systems - 11th Asian Symposium (APLAS)*, pages 50–57, 2013.

[14] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, abs/1606.04435, 2016.

[15] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *CoRR*, abs/1412.5068, 2014.

[16] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *USENIX (WOOT 17)*. USENIX Association, 2017.

[17] Ruitong Huang, Bing Xu, Dale Schuurmans, and Csaba Szepesvári. Learning with a strong adversary. *CoRR*, abs/1511.03034, 2015.

[18] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification, 29th International Conference (CAV)*, pages 3–29, 2017.

[19] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[20] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference (CAV)*, pages 661–667, 2009.

[21] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification, 29th International Conference (CAV)*, pages 97–117, 2017.

[22] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[24] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[25] Yann Lecun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[26] Yann Lecun, Larry Jackel, Bernhard E. Boser, John Denker, H.P. Graf, Isabelle Guyon, Don Henderson, R. E. Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11), 1989.

[27] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*, 2018.

[28] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 355–363, 2009.

[29] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016.

[30] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, 2015.

[31] Nicolas Papernot, Patrick D. McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 582–597, 2016.

[32] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017.

[33] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues (ASIAN)*, pages 331–345, 2007.

[34] Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification, 22nd International Conference (CAV)*, 2010.

[35] Sara Sabour, Yanshuai Cao, Fartash Faghri, and David J. Fleet. Adversarial manipulation of deep representations. *CoRR*, abs/1511.05122, 2015.

[36] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis, 13th International Symposium, SAS)*, pages 3–17, 2006.

[37] Karsten Scheibler, Leonore Winterer, Ralf Wimmer, and Bernd Becker. Towards verification of artificial neural networks. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV) 2015*, pages 30–40, 2015.

[38] Uri Shaham, Yutaro Yamada, and Sahand Negahban. Understanding adversarial training: Increasing local stability of neural nets through robust optimization. *CoRR*, abs/1511.05432, 2015.

[39] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 46–59, 2017.

[40] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[41] Pedro Tabacof and Eduardo Valle. Exploring the space of adversarial images. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 426–433, 2016.

[42] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[43] Florian Tramèr, Nicolas Papernot, Ian J. Goodfellow, Dan Boneh, and Patrick D. McDaniel. The space of transferable adversarial examples. *CoRR*, abs/1704.03453, 2017.

[44] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM 2014 Conference*, pages 371–372, 2014.

APPENDIX

A. CAT function representations of the Convolutional Layer and the Max Pooling Layer

In this section, we provide the formal definitions of the matrices and vectors used to represent the convolutional layer and the max pooling layer as CAT functions.

Convolutional Layer. Recall that for filters $W^k \in \mathbb{R}^{p\times q\times r}$, $b^k \in \mathbb{R}$ for $1 \le k \le t$, we have

$$\begin{aligned}&\mathrm{Conv}_F(\overline{x})\colon \mathbb{R}^{n\times m\times r} \to \mathbb{R}^{(m-p+1)\times(n-q+1)\times t}\\ &\mathrm{Conv}_F(\overline{x})_{i,j,k} = \mathrm{ReLU}(\sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'}\cdot x_{(i+i'-1),(j+j'-1),k'} + b^k),\end{aligned}$$

for $1 \le i \le m-p+1$, $1 \le j \le n-q+1$ and $1 \le k \le t$. Reshaping both the input and the output vector such that they have only one index, we obtain

$$\begin{aligned}&\mathrm{Conv}'_F(\overline{x})\colon \mathbb{R}^{n\cdot m\cdot r} \to \mathbb{R}^{(m-p+1)\cdot(n-q+1)\cdot t}\\ &\mathrm{Conv}'_F(\overline{x})_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} = \mathrm{ReLU}(\sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'}\cdot x_{n\cdot r\cdot(i+i'-2)+r\cdot(j+j'-2)+k'} + b^k),\end{aligned}$$

for $1 \le i \le m-p+1$, $1 \le j \le n-q+1$ and $1 \le k \le t$. The function Conv'_F is ReLU after an affine transformation, therefore there is a matrix $W^F \in \mathbb{R}^{((m-p+1)\cdot(n-q+1)\cdot t)\times(n\cdot m\cdot r)}$ and a vector $\overline{b}^F \in \mathbb{R}^{(m-p+1)\cdot(n-q+1)\cdot t}$ such that

$$\mathrm{Conv}_F(\overline{x})^v = \mathrm{Conv}'_F(\overline{x}^v) = \mathrm{ReLU}(W^F\cdot\overline{x}^v + b^F) = \mathrm{FC}_{W^F,\overline{b}^F}(\overline{x}).$$

The entries of W^F and $\overline{b}^F$ are obtained by equating

$$\begin{aligned}\mathrm{FC}(\overline{e}_l)_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} &= \mathrm{ReLU}(W^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k,l} + b^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k}) \text{ with}\\ \mathrm{Conv}'_F(\overline{e}_l)_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} &= \mathrm{ReLU}(\sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'}\cdot[l = n\cdot r\cdot(i+i'-2)+r\cdot(j+j'-2)+k'] + b^k),\end{aligned}$$

for standard basis vectors $\overline{e}_l$ with $(\overline{e}_l)_i = [l = i]$ for $1 \le l \le n$ and $1 \le i \le n\cdot m\cdot r$. This way, we obtain

$$\begin{aligned}W^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k,l} &= \sum_{i'=1}^{p}\sum_{j'=1}^{q}\sum_{k'=1}^{r} W^k_{i',j',k'}\cdot[l = n\cdot r\cdot(i+i'-2)+r\cdot(j+j'-2)+k'] \text{ and}\\ b^F_{(n-q+1)\cdot t\cdot(i-1)+t\cdot(j-1)+k} &= b^k,\end{aligned}$$

for $1 \le i \le m-p+1$, $1 \le j \le n-q+1$ and $1 \le k \le t$. Note that here, $[\varphi]$ is an Iverson bracket, which is equal to 1 if φ holds and equal to 0 otherwise.

Max Pooling Layer. Recall that $\mathrm{MaxPool}_{p,q}\colon \mathbb{R}^{m\times n\times r} \to \mathbb{R}^{\frac{m}{p}\times\frac{n}{q}\times r}$ partitions the input vector into disjoint blocks of size $p\times q\times 1$ and replaces each block by its maximum value. Furthermore, $\mathrm{MaxPool}'_{p,q}\colon \mathbb{R}^{m\cdot n\cdot r} \to \mathbb{R}^{\frac{m}{p}\cdot\frac{n}{q}\cdot r}$ is obtained from $\mathrm{MaxPool}_{p,q}$ by reshaping both the input and output: $\mathrm{MaxPool}'_{p,q}(\overline{x}^v) = \mathrm{MaxPool}_{p,q}(\overline{x})^v$. We will represent $\mathrm{MaxPool}'_{p,q}$ as a composition of CAT functions,

$$\mathrm{MaxPool}'_{p,q} = f_{\frac{m}{p}\cdot\frac{n}{q}\cdot r}\circ\ldots\circ f_1\circ f^{\mathrm{MP}}.$$

Here, f^{MP} rearranges the input vector such that values from the same block are adjacent. Values from different blocks are brought into the same order as the output from each block appears in the output vector.

Note that $((i-1) \bmod p)+1, (j-1) \bmod q)+1, 1)$ are the indices of input value $x_{i,j,k}$ within its respective block and $\left(\left\lfloor\frac{i-1}{p}\right\rfloor+1, \left\lfloor\frac{j-1}{q}\right\rfloor+1, k\right)$ are the indices of the unique value in the output vector whose value depends on $x_{i,j,k}$. Recall that the permutation matrix M representing a permutation π is given by $M_{\pi(i)} = \overline{e}_i$.

The CAT function f^{MP} is a linear transformation $f^{\mathrm{MP}}(\overline{x}^v) = W^{\mathrm{MP}}\cdot\overline{x}^v$ where the permutation matrix W^{MP} is given by

$$W^{\mathrm{MP}}_{r\cdot p\cdot q\cdot\left(\frac{n}{q}\left\lfloor\frac{i-1}{p}\right\rfloor+\left\lfloor\frac{j-1}{q}\right\rfloor\right)+p\cdot q\cdot(k-1)+q\cdot((i-1) \bmod p)+((j-1) \bmod q)+1} = \overline{e}_{n\cdot r\cdot(i-1)+r\cdot(j-1)+k},$$

for $1 \le i \le m$, $1 \le j \le n$ and $1 \le k \le r$.

For each $1 \leq i \leq \frac{m}{p} \cdot \frac{n}{p} \cdot r$, the CAT function f_i selects the maximum value from a $(p \cdot q)$-segment starting from the i^{th} component of the input vector. The function f_i consists of a sequence of cases, one for each of the $p \cdot q$ possible indices of the maximal value in the segment:

$$\begin{aligned} f_i(\overline{x}) = \textbf{case } & (x_i \geq x_{i+1}) \wedge \ldots \wedge (x_i \geq x_{i+p\cdot q-1})\colon W^{(i,i)} \cdot \overline{x}, \\ \textbf{case } & (x_{i+1} \geq x_i) \wedge \ldots \wedge (x_{i+1} \geq x_{i+p\cdot q-1})\colon W^{(i,i+1)} \cdot \overline{x}, \\ & \vdots \\ \textbf{case } & (x_{i+p\cdot q-1} \geq x_i) \wedge \ldots \wedge (x_{i+p\cdot q-1} \geq x_{i+p\cdot q-2})\colon W^{(i,i+p\cdot q-1)} \cdot \overline{x}. \end{aligned}$$

The matrix $W^{(i,k)} \in \mathbb{R}^{(m\cdot n\cdot r-(p\cdot q-1)\cdot i)\times(m\cdot n\cdot r-(p\cdot q-1)\cdot(i-1))}$ replaces the segment $x_i, \ldots, x_{i+p\cdot q-1}$ of the input vector $\overline{x}$ by the value x_k and is given by

$$W^{(i,k)}_j = \begin{cases} \overline{e}_j, & \text{if } 1 \leq j \leq i-1 \\ \overline{e}_k, & \text{if } j = i \\ \overline{e}_{j+p\cdot q-1}, & \text{if } i+1 \leq j \leq m\cdot n\cdot r-(p\cdot q-1)\cdot i \end{cases}.$$