Scalable Reachability Analysis of Neural Networks with Parallel Computing

ABSTRACT

Artificial neural networks (ANN) have been applied in a wide range of applications from image processing, character recognition to self-driving cars. A trained neural network can perform efficiently complicated tasks; however, it often has a complex nonlinear dynamics which is also the main source of unpredictable behaviors narrowing the application of ANN in safety-critical systems. Thus, there is an urgent need for new methodologies that can either effectively predict or verify the behaviors of ANN. In this paper, we propose a reachability analysis method for feedforward neural networks (FNN) using rectified linear units (ReLUs) as activation functions. The core of our approach are three reachable set computation methods including the *exact*, *lazy-approximate*, and *mixing* schemes. The exact scheme computes the exact reachable set of an FNN while the lazy-approximate and mixing ones compute an over-approximate reachable set. The novelty of our proposed method in comparison with other existing approaches is that the reachability analysis of FNN can be done efficiently by exploiting the power of parallel computing. We implement our method in a Matlab toolbox called NNV and compare its performance with other existing approaches using a benchmark of practical neural networks from tens to thousands of neurons. The experimental results demonstrate that even with a normal computer, our method has successfully computed the exact reachable set, a union of thousands of polyhedra, of the real-world deep neural networks (DNN) for the next-generation airborne collision avoidance system of unmanned aircraft (ACAS Xu). This DNN is a notable case study used in Replulex, a recent scalable SMT solver based verification framework for DNN.

ACM Reference Format:

. 2019. Scalable Reachability Analysis of Neural Networks with Parallel Computing. In *Proceedings of ACM International Conference on Hybrid Systems: Computation and Control (HSCC'19).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Artificial neural networks (ANN) play an essential role in many applications such as image classification [xx], speech and character recognition [xx], market forecasting [xx], autonomous vehicles [xx] and airborne collision avoidance systems for unmanned aircraft (ACAS Xu) [xx]. The ability to learn and to approximate complex nonlinear functions from a large data set makes ANN a preferred choice for many complicated tasks. Although ANN is trained with a finite set of inputs and outputs, it is expected to work appropriately with unseen inputs. However, it has been observed that a well-trained ANN can behave incorrectly to even slight perturbations of their inputs [xx]. The unexpected behaviors of a trained ANN raise a challenging safety verification problem which is crucial in high-assurance applications using neural networks. Since an ANN is non-linear, non-convex and may be large, safety verification for ANN is a difficult problem in which verifying even a simple property of an ANN is NP-complete [xx].

The main focus in this paper is the reachable set computation of a trained feedforward neural networks (FNN) with ReLUs activation functions. From the computed reachable set, ones can verify safety properties of the neural network easily. Besides, the projection of the reachable set of the outputs of the neural network on some specific subspace can also be visualized. Unlike other existing approaches, our method can compute both an exact and over-approximate reachable set of an FNN. More importantly, the algorithms proposed in our approach are designed efficiently to exploit the power of parallel computing. Generally, the reachable set computation time can be decreased linearly by increasing the number of processors (cores) used in the computation.

Method overview. In our framework, convex polyhedra are used to represent the reachable sets. The reachable set of a FNN at the output layer is basically a union of convex polyhedra. We investigate three different procedures for computing reachable set of a FNN including the exact, lazy-approximate, and mixing schemes.

The *exact scheme* computes layer-by-layer the exact reachable set of the FNN. At each layer, the exact scheme performs a sequence of *stepReLU operations*. A *stepReLU* operation computes the *intermediate (incomplete) reachable set* of the layer at a specific neuron by examining the ReLU activation function on an input set feeding to that neuron. It should be noted that the reachable set at a layer's output is only completely obtained after the sequence of stepReLU operations is finished. For example, for a layer with 50 neurons, we generally need to perform 50 stepReLU operations sequentially to obtain the reachable set of the layer. Besides, changing the order of stepReLU operations does not change the final reachable set of the layer. In this paper, we define $ReLU_i(.)$ as the stepReLU operation executed at the i^{th} neuron of a layer. One can see that the reachable set computation time for each layer depends on the number of stepReLU operations that need to be done in that layer. Therefore, to reduce the computation time, our exact scheme preprocesses the input set at each layer to minimize the number of stepReLU operations in the computation. It is worth to emphasized that a stepReLU operation on a single polyhedron input set may produce two polyhedra output set. As a result, in theory, the number of polyhedra of the output set of each layer increases exponentially with the number of neurons of that layer. Therefore, optimizing the

HSCC'19, April 2019, Montreal, Canada
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

stepReLU operations is practically useful in the sense of minimizing the computation cost as well as the number of polyhedra in the output set. The output set of the current layer is the input set for the next layer. Since the input set of a layer is generally a set of individual polyhedra, we can perform the reachable set computation on each input polyhedron independently in parallel.

In real-world applications, it usually requires only the knowledge of the ranges of the outputs of an FNN for safety verification purpose, for example, safety verification of airborne collision avoidance systems for unmanned aircraft [xx]. Therefore, it is useful to have a simple method to over-approximate the ranges of the outputs of an FNN quickly. Our second scheme, i.e., *lazy-approximate scheme*, is a lazy over-approximation procedure explicitly designed for this purpose. Particularly, the output ranges of each layer are bounded by a hyper-rectangle which can be constructed efficiently by solving n linear programming problems, where n is the number of neurons of the layer. One can see that this method does not perform stepReLU operations and thus it is usually much faster than the exact scheme. The main drawback of this lazy-approximate scheme is that the over-approximation error may be accumulated very quickly over layers and thus leads to very conservative ranges of the outputs of an FNN with many layers. Therefore, the approximate scheme is suitable for an FNN with a small number of layers and a large number of neurons at each layer. From our experiments, the lazy-approximate scheme is comparable with the output range analysis approach [xx] for FNN with one or two layers in which each layer has a large number of neurons. It is also worth to point out that, for an FNN with small input space, i.e., one or two inputs, we can partition the input space into a set of smaller polyhedra and then compute in parallel a tight over-approximation reachable set of the FNN using only the lazy-approximate scheme.

As mentioned above, the number of polyhedra of the output set of each layer may increases exponentially in the exact scheme. Therefore, even using parallel computing, the reachable set computation time of an FNN with many layers is intractable in general. It is required to have an approach that not only can control the number of polyhedra but also still produces an acceptable over-approximation of the actual reachable set. This requirement can be full-filled with our third scheme, i.e., the mixing scheme which is a regime that combines the benefits of the exact and the lazy-approximate schemes. The mixing scheme works as follows. The exact scheme is used to compute the reachable set layer-by-layer until the number of polyhedra of the reachable set exceeds a user-defined upper bound N_{max}. These polyhedra are then *clustered and merged* into N_{max} hyper-rectangles. It is crucial to have efficient clustering and merging methods such that the union of N_{max} hyper-rectangles is a tight over-approximation of the reachable set. To do that, we cluster N ($N > N_{max}$) polyhedra of the reachable set into N_{max} groups based on the *overlapseness* between the polyhedra. Then, we over-approximate the union of polyhedra in each group by one hyper-rectangle. After clustering and merging steps, the lazy-approximate scheme is invoked to compute in parallel the reachable set of the rest layers with the input set is a union of N_{max} hyper-rectangles. The conservativeness of the reachable set computed using the mixing scheme depends significantly on the user-defined upper bound N_{max}. Generally, when N_{max} increases, the over-approximate reachable set approaches the exact reachable set. The main benefit of this mixing scheme is that the number of polyhedra of the reachable set is controllable by users and the computation time can be reduced significantly.

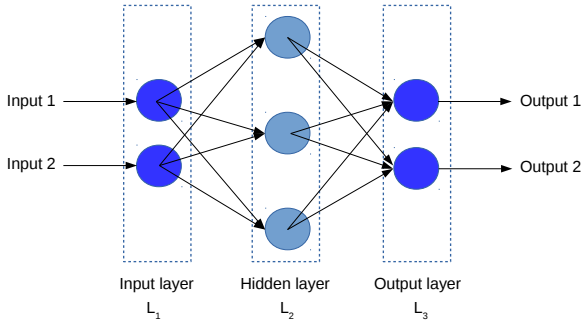

Figure 1: An example of FNN.

Contributions. The main contributions of the paper are summarized as follows.

(1) the three efficient, parallel-computing based schemes to compute the reachable set of an FNN with ReLUs activation functions,
(2) the end-to-end design and implementation of these schemes in a Matlab toolbox that is publicly available for verifying a complex FNN, and
(3) the thorough evaluation and impressive comparison of our proposed methods to other existing approaches via practical case studies of FNNs.

The rest of the paper is organized as follows. Section 2 presents some background on FNNs. Section 3 addresses the exact reachable set computation scheme. Section 4 studies the lazy-approximate scheme, and discusses how to cluster and merge a number of polyhedra of a reachable set into a user-specified number of hyper-rectangles, and then presents the mixing scheme. The experimental results are illustrated in Section 5. The related works are discussed in Section 6, and Section 7 concludes the paper.

2 PRELIMINARIES

An FNN is constituted from an input layer, an output layers, and multiple hidden layers. Each layer consists of some neurons that are connected to the neurons of the preceding layer via a set of weights. An example of FNN is shown in Figure 1 in which the input layer L_1 has two neurons, the hidden layer L_2 has three neurons, and the output layer L_3 has two neurons.

Evaluation of FNN. An evaluation of an FNN on a specific input vector is determined based on three components: the weight matrices $\{W_{k,k-1}\}$ stating the weighted connection between neurons of two consecutive layers $k-1$ and k, the bias vectors $\{b_k\}$ of each layer, and the activation functions $\{f\}$ applied to each layer. Mathematically, an evaluation on a neuron i of a layer is define by:

$$y_i = f(\Sigma_{j=1}^{n} \omega_{ij} x_j + b_i),$$

where x_j is the j^{th} input of the i^{th} neuron, ω_{ij} is the weight from the j^{th} input to the i^{th} neuron, b_i is the bias of the i^{th} neuron. In this paper, we consider FNN with ReLUs activation functions which is defined by $ReLU(x) = max(0, x)$.

Example 2.1 (Evaluation of FNN). The example shown in Figure 1 is assumed to have the following weight matrices and bias vectors.

$$W_{2,1} = \begin{bmatrix} 2 & 0 \\ 1 & -1 \\ 1 & 1 \end{bmatrix},\ b_2 = \begin{bmatrix} 0.5 \\ -1 \\ -0.5 \end{bmatrix},$$
$$W_{3,2} = \begin{bmatrix} -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix},\ b_3 = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}.$$

The weight matrix $W_{2,1}$ expresses the weighted connections between the neurons of the hidden layer L_2 and the neurons of the input layer L_1 while the weight matrix $W_{3,1}$ describes the weighted connections between the neurons of the output layer L_3 and the neurons of the hidden layer L_2.

Assume the input vector of the input layer is $x = [1\ 1]^T$, and ReLUs are the activation functions of the neural network, then the vector containing the values of the hidden layer's neurons is:

$$y_2 = ReLU(W_{2,1} \times x + b_2) = ReLU(\begin{bmatrix} 2 & 0 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.5 \\ -1 \\ -0.5 \end{bmatrix}),$$
$$= ReLU(\begin{bmatrix} 2.5 \\ -1 \\ 1.5 \end{bmatrix}) = \begin{bmatrix} 2.5 \\ 0 \\ 1.5 \end{bmatrix}.$$

The vector y_2 is then fed to the output layer L_3 as an input vector. The final output vector at the output layer is:

$$y_3 = ReLU(W_{3,2} \times y_2 + b_3) = ReLU(\begin{bmatrix} -1 & 0 & 1 \\ 1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 2.5 \\ 0 \\ 1.5 \end{bmatrix} + \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}),$$
$$= ReLU(\begin{bmatrix} -1.5 \\ 3 \end{bmatrix}) = \begin{bmatrix} 0 \\ 3 \end{bmatrix}.$$

Reachability analysis of FNN. The above example describes step-by-step how to compute the output values of an FNN given a specific input vector. In this paper, we are interested in computing the reachable set R of the output of an FNN with a given input set I. Specifically, we consider the reachability analysis of an FNN with a bounded convex polyhedron input set defined by:

$$I = \{x \mid Ax \leq b, x \in \mathbb{R}^n\},$$

where x is the input vector, and n is the dimension of the input space, i.e., the number of inputs of the FNN. Since the ReLU activation function is non-linear, and an FNN may have a large number of layers and neurons, reachability analysis of an FNN is non-trivial and difficult.

Safety verification of FNN. The ultimate goal of doing reachability analysis for an FNN is to verify whether or not the outputs of the FNN violate some safety properties S defined by users. In our framework, the safety properties of an FNN is a set of linear constraints on the outputs of the FNN which is defined as:

$$S = \{y \mid Cy \leq d, y \in \mathbb{R}^m\},$$

where y is the output vector, and m is the dimension of the output space, i.e., the number of outputs of the FNN. From the reachable set R of the outputs computed by reachability analysis algorithms, we can verify the safety properties of the FNN by checking the intersection between the output reachable set with the unsafe region, i.e., $\neg S$, where $\neg$ is the symbol for logical negation. Formally, the FNN is called safe if and only if:

$$R \cap \neg S = \emptyset.$$

We have presented some backgrounds on FNN with ReLUs activation functions and introduced the reachability analysis and safety verification problems for FNN. In the next section, we study the first regime in our approach, the *exact scheme* for computing reachable sets of an FNN.

3 EXACT REACHABILITY ANALYSIS

In our approach, the exact reachability analysis is done layer-by-layer. Given an input set I_i, the reachable set of a layer L can be obtained precisely in two steps. First, an affine map $\bar{I}_i$ of the input set I_i is computed with the weight matrix W and bias vector b of the layer. Mathematically, the affine map $\bar{I}_i$ is defined by:

$$\bar{I}_i = \{y \mid y = Wx + b,\ x \in I_i\}.$$

After calculating the affine map of the input set, the reachable set of the layer R_L is obtained by applying the ReLU activation function on the affine-mapped set $\bar{I}_i$,

$$R_L = ReLU(\bar{I}_i).$$

This second step is done by executing a sequence of stepReLU operations $ReLU_j()$ which is clarified in the following.

3.1 stepReLU operation

The basic idea of stepReLU operations is illustrated in Figure 2 for a two-dimensional affine-mapped set $\bar{I}_i$. The stepReLU operation works as follows. First, the affine mapped set $\bar{I}_i$ is decomposed into two sets $\Psi_1 = \bar{I}_i \wedge x_1 \geq 0$ and $\Psi_2 = \bar{I}_i \wedge x_1 < 0$. Since the later set has $x_1 < 0$, applying the ReLU activation function on the first element x_1 of a vector $x = [x_1\ x_2]^T \in \Psi_2$ will lead to a new vector $x' = [0\ x_2]^T$. Also, applying the ReLU activation function on the first element x_1 of a vector $x \in \Psi_1$ does not change the set since we have $x_1 \geq 0$. As a result, the intermediate reachable set after applying the $ReLU_1$ operation on the affine-mapped set $\bar{I}_i$ is $\tilde{R} = \tilde{R}_1 \cup \tilde{R}_2$, where $\tilde{R}_1 = \Psi_1$, $\tilde{R}_2$ is the projection of Ψ_2 on the hyper-plane $x_1 = 0$. The intermediate reachable set $\tilde{R}$ is then fed to the $ReLU_2$ operation as an input. The final reachable set of the layer is a union of three polyhedra $R_L = ReLU(\bar{I}_i) = R_1 \cup R_2 \cup R_3$ as shown in the figure.

For a layer with n neurons, the reachable set of the layer generally can be obtained by executing a sequence of n stepReLU operations as follows.

$$R_L = ReLU_n(ReLU_{n-1}(\cdots ReLU_1(\bar{I}_i))).$$

One can verify that changing the order of the stepReLU operations affects the intermediate reachable set but it does not change the final reachable set of the layer.

Optimize stepReLU operation. From a single polyhedron input set, a stepReLU operation can produce two polyhedra, e.g., $ReLU_1(\bar{I}_i)$ in Figure 2. The number of polyhedra in computation

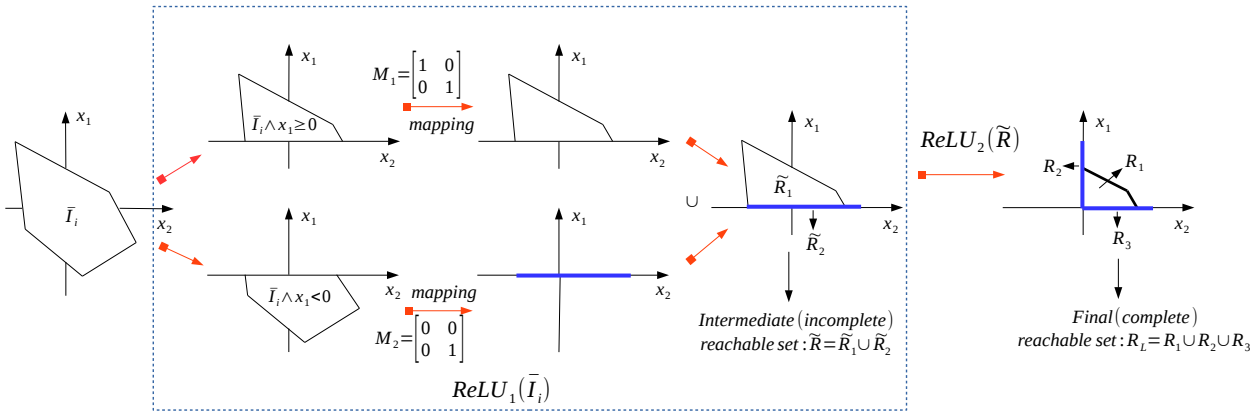

Figure 2: stepReLU operation.

increases along with the number of stepReLU operations. Therefore, to reduce the computation cost, it is crucial to minimize the number of necessary stepReLU operations beforehand and the number of polyhedra in the intermediate reachable sets. The solution for this optimization is, we find the smallest hyper-rectangle, i.e., a box, that bounds the affine-mapped set $\bar{I}_i$, from this hyper-rectangle we determine the ranges of all elements x_i in the vector $x = [x_1\ x_2\ \cdots\ x_n]^T \in \bar{I}_i$. Finding these ranges is easy since it is equivalent to solving n simple linear programming problems. From the ranges of all elements of the vector x, if we know that the minimum value of the element x_i is larger than zero, then we can neglect the stepReLU operation on the i^{th} neuron. To minimize the number of polyhedra in the intermediate reachable sets, in each stepReLU operation, if it produces two polyhedra, we need to check whether or not one polyhedron is a subset of another to delete the subset polyhedron.

The stepReLU operation is stated in Algorithm 1 in which $\tilde{I}$ is the input set (i.e., the output of the preceding stepReLU operation), i is the index of the current neuron we want to perform the stepReLU operation, x_{min} and $x_{\max}$ are the lower and upper bounds of the element $x[i]$.

3.2 Parallel-exact reachability analysis

Algorithm 1 computes the intermediate reachable set at each neuron. To compute the reachable set of a layer, we need to perform step-by-step a sequence of stepReLU operations. The reachable set for the last step is the reachable set of a layer. Algorithm 2 is the reachable set computation of one layer using the stepReLU operation. The algorithm first calculates the affine map of the input set using the layer's weight matrix and bias vector. Then, it performs the *reachReLU* procedure which basically executes a sequence of stepReLU operations. To optimize the stepReLU operations, the *reachReLU* procedure finds the lower and upper bounded of a vector x in the affine-mapped set. Then, it constructs a computation map which minimizes the number of stepReLU operations needs to be executed. Since a layer can have multiple polyhedra as input

Algorithm 1 stepReLU operation with multiple inputs

1: **procedure** $\tilde{R}$ = stepReLU($\tilde{I}, i, x_{min}, x_{max}$)
2: % $\tilde{I}$: intermediate input set
3: % i: index of current neuron
4: % x_{min}: lower-bound of $x[i]$
5: % x_{max}: upper-bound of $x[i]$
6: % $\tilde{R}$: intermediate output set
7:
8: $n = length(\tilde{I})$
9: $\tilde{R} = \emptyset$
10: **for** $i = 1 : n$ **do**
11: $I_1 = I(i)$
12: $R_1 = \emptyset$
13: **if** $x_{min} \geq 0$ **then**
14: $R_1 = I_1$
15: **if** $x_{max} < 0$ **then**
16: $R_1 = projection\ of\ I_1\ on\ x[i] = 0$
17: **if** $x_{min} < 0\ \&\ x_{max} \leq 0$ **then**
18: $Z_1 = I_1 \wedge x[i] \geq 0$
19: $Z_2 = I_1 \wedge x[i] < 0$
20: $Z_2' = projection\ of\ Z_2\ on\ x[i] = 0$
21: **if** Z_2' *is a subset of* Z_1 **then**
22: $R_1 = Z_1$
23: **else**
24: $R_1 = Z_1 \cup Z_2'$
25: $\tilde{R} = \tilde{R} \cup R_1$

sets, we can perform the reachable set computation for each input set in parallel. To derive the reachable set for an FNN, we perform the reachable set computation layer-by-layer. The reachable set of the last layer is the output reachable set of the FNN.

Example 3.1 (Exact reachability analysis of FNN). In this example, we perform the reachability analysis for the FNN in Example 2.1 with the input set $I = \{0 \leq x[1] \leq 2 \wedge 0 \leq x[2] \leq 2$. Note that, the

Algorithm 2 Parallel-exact reachable set computation for one layer

Input: I, W, b % input set, weight matrix, bias vector
Output: R % reachable set

procedure R_1 = reachReLU(I_1)
  $lb \leftarrow$lower-bound of x $\in I_1$
  $ub \leftarrow$upper-bound of x $\in I_1$
  $map = find(lb < 0)$ % computation map
  $m = length(map)$ % number of stepReach operations
  $In = I_1$
  for $i = 1 : m$ **do**
    $In = stepReLU(In, map(i), lb(map(i)), ub(map(i)))$;
  $R_1 = In$;
procedure R = layerReach(I, W, b)
  $n = length(I)$
  $R = \emptyset$
  parfor $i = 1 : n$ **do** % parallel for loop
    $I_1 = I(i).affineMap(W) + b$
    $R_1 = reachReLU(I_1)$
    $R = R \cup R_1$
  end parfor

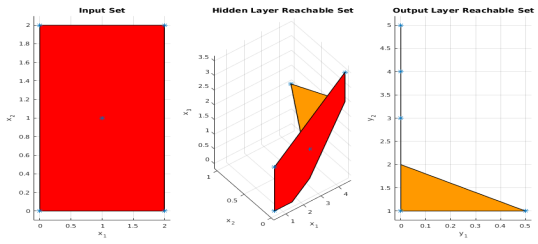

Figure 3: Exact reachable set of the FFN in Example 2.1.

vector $x = [1\ 1]^T$ is the center of the input set. Using Algorithm 2, the exact reachable sets of the hidden layer $L2$ and the output layer $L3$ are computed in which each layer's exact reachable set consists of two polyhedra as depicted in Figure 3. It can be seen that the output $y = [0\ 3]^T$ corresponding to the input $x = [1\ 1]^T$ lies in the output reachable set.

4 APPROXIMATE REACHABILITY ANALYSIS

4.1 Lazy-approximate reachability analysis

We have investigated the exact scheme for reachability analysis. In this section, we discuss a lazy approximation approach for reachability analysis of FNN with ReLU activation functions. This lazy-approximate scheme does not perform any stepReLU operations. Instead, it finds the smallest hyper-rectangle that bounds an affine-mapped set $\bar{I}_i$. Since it is well-known at the beginning that for all value of x, we have $ReLU(x) \geq 0$, the output ranges of a layer can be derived quickly by querying the lower-bound and upper-bound information in each dimension of the state vector x in the obtained hyper-rectangle. Figure 4 illustrates the idea of the lazy-approximate scheme. Algorithm 3 is the pseudo-code of the parallel, lazy-approximate scheme.

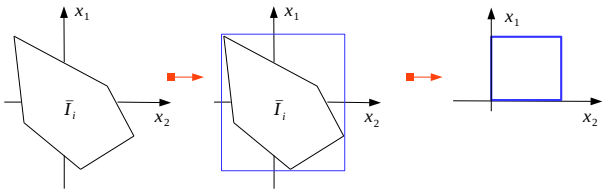

Figure 4: Lazy-approximate scheme.

Algorithm 3 Parallel-lazy reachable set approximation for one layer

Input: I, W, b % input set, weight matrix, bias vector
Output: R % an over-approximate reachable set

procedure B = lazyReLU(I_1)
  $lb \leftarrow$lower-bound of x $\in I_1$
  $ub \leftarrow$upper-bound of x $\in I_1$
  $m = length(lb)$
  for $i = 1 : m$ **do**
    if $lb(i) \geq 0$ **then**
      $lb(i) = 0$
    if $ub(i) \geq 0$ **then**
      $ub(i) = 0$
procedure R = layerLazyReach(I, W, b)
  $n = length(I)$
  $R = \emptyset$
  parfor $i = 1 : n$ **do** % parallel for loop
    $I_1 = I(i).affineMap(W) + b$
    $R_1 = lazyReLU(I_1)$
    $R = R \cup R_1$
  end parfor

Example 4.1 (Lazy output range analysis of FNN). In this example, we perform a lazy reachability analysis for the FNN in Example 2.1 with the same input set as in the Example 3.1. The resulted over-approximate reachable sets of the hidden and output layers are depicted in Figure 5. It can be seen from the figure that the output ranges of the hidden layer can be derived precisely from its lazy, over-approximate reachable set. However, the output ranges of the output layer are conservatively approximated by its corresponding lazy, over-approximate reachable set. One can see that the error in over-approximation is accumulated quickly over layers. Therefore, the lazy-scheme is only useful for FNN with a small number of layers.

Lazy-approximate reachability analysis with input partition. As shown in Figure 5, the ranges of the first and second outputs are $[0,\ 2.5]$ and $[0,\ 5]$ which are much coarser then the actual ranges of these outputs computed by the exact reachability analysis algorithm, i.e., $[0,\ 0.5]$ and $[1,\ 5]$ as depicted in Figure 3. Interestingly, we can estimate tighter ranges for the outputs by partitioning the input set into smaller sets, and then performing the lazy-approximate scheme on these partitioned input sets.

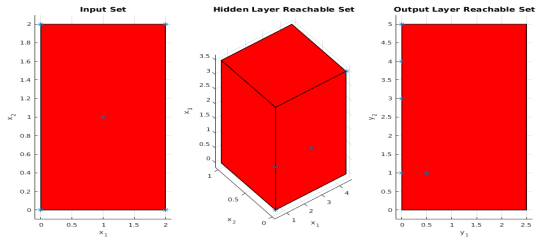

Figure 5: Reachable set of the FFN in Example 3.1 using lazy-approximate scheme.

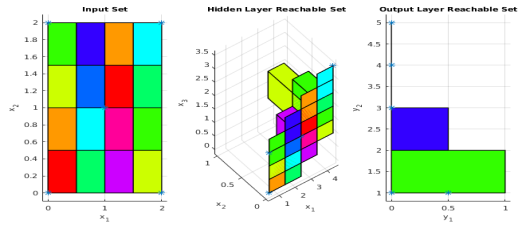

Figure 6: Reachable set of the FFN in Example 3.1 using lazy-approximate scheme and input partition.

Example 4.2 (Output range analysis of FNN with lazy-approximate scheme and input partition). In this example, we partition uniformly the input set $I = \{0 \leq x[1] \leq 2 \wedge 0 \leq x[2] \leq 2$ into 16 smaller sets. We perform Algorithm 3 on these partitioned sets. The over-approximate reachable sets of the hidden and output layers are given in Figure 6. As shown in the figure, the output ranges of the FNN are precisely match the one obtained by the exact scheme.

We have discussed in detail the lazy-approximate scheme. Next, we consider a combination of the exact and lazy-approximate schemes to control the number of polyhedra in the reachable set while still obtain a over-approximate reachable set with an acceptable conservativeness.

4.2 Mixing reachability analysis

The mixing reachability analysis scheme is proposed with three main objectives: 1) controlling the number of polyhedra in the reachable set; 2) reducing the reachable set computation time, and 3) producing an acceptable over-approximation of the actual reachable set. For the first objective, the mixing scheme lets users choose a maximum, allowable number of polyhedra N_{max} to represent the reachable set. The mixing scheme computes the exact reachable set layer-by-layer and observes the number of polyhedra N in the reachable set. When $N > N_{max}$, the mixing scheme performs clustering and merging algorithm to merge N polyhedra into N_{max} hyper-rectangles. Then, it continues computing reachable set for the rest layers using the lazy-approximate scheme with the N_{max} hyper-rectangles as input sets. The conservativeness of the resulted reachable set comes from two sources. The first one is the error caused by merging N polyhedra into N_{max} hyper-rectangles. The second one is the error from using the lazy-approximate scheme to compute the reachable set. To improve the conservativeness of the result, methods are needed to reduce these errors. In this paper, we propose an efficient clustering and merging algorithm using the *overlapness* between polyhedra to reduce the first source of error. The second source of error remains a challenging problem and should be investigated deeper in future work.

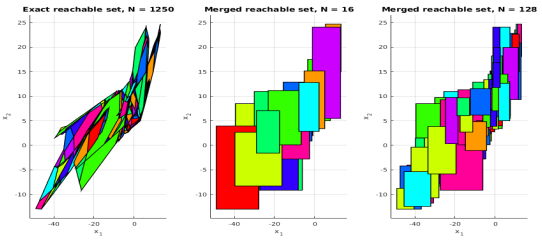

Figure 7: Clustering and merging reachable set.

Assume that we want to merge N polyhedra into N_{max} hyper-rectangles such that the union of these hyper-rectangles tightly over-approximates the union of the N polyhedra. Our clustering and merging algorithm work as follows. First, we obtain N smallest hyper-rectangles that bounds the N polyhedra. Then, based on the lower-bound lb and upper-bound ub vectors of these hyper-rectangles, we compute the overlapness OLN between the N polyhedra which is defined by:

$$OLN_{i,j} = \sqrt{(lb_i - lb_j)^2 + (ub_i - ub_j)^2}, \tag{1}$$

where $lb_i(ub_i), lb_j(ub_j) \in \mathbb{R}^n$ are lower-bound (upper-bound) vectors of polyhedron i and j respectively, and n is the number of neurons of considering layer.

Using the overlapness data and the well-known clustering algorithm *kmeans* [**?**], we cluster N polyhedra into N groups and merge all polyhedra in each group into one hyper-rectangle by interfering the lower-bound and upper-bound information of all hyper-rectangles bounding the polyhedra in the group.

Example 4.3 (Clustering and merging polyhedra). In this example, we perform the clustering and merging algorithm on a reachable set with 1250 polyhedra [**?**]. These polyhedra are clustered and merged into 16 and 128 hyper-rectangles. Figure 7 shows the result of our algorithm. It can be observed from the figure that even using a small number of hyper-rectangles, we can over-approximate the actual reachable set tightly with more than one thousand polyhedra. Another benefit of the clustering and merging algorithm is that it is fast. The conservativeness in the merging process can be reduced by increasing the number of hyper-rectangles representing the reachable set.

Conservativeness and computation time reduction. Although the clustering and merging algorithm can reduce the first source of error significantly, the second source of error in the mixing scheme can still lead to a very conservative reachable set. This is due to the over-approximation error of the lazy-approximate scheme is accumulated over layers. In general, whatever approximation scheme

we use, e.g., zonotope-based approximation scheme [**?**], the accumulation of the over-approximation error is inevitable. Therefore, choosing an appropriate number of polyhedra N_{max} to represent the reachable set is crucial. This number should be chosen in the way that the lazy-approximate scheme is only invoked in the last one or two layers which will guarantees that the result reachable set is acceptable. It is worth to point out that from our experiments, in the exact scheme, the reachable set computation time of the last two layers of an FNN usually constitutes $> 50\%$ the total reachable set computation time. This is because: 1) the last two layers take a vast number of polyhedra as input sets, and 2) the constraints of each polyhedron in the input sets is large. Note that the number of constraints of a reachable set increases over layers due to the intersection operations in the computation process, e.g., see Algorithm 1. Since the lazy-approximate scheme is usually much faster than the exact scheme, choosing N_{max} is a trade-off problem between conservativeness and computation time reduction. If we use the lazy-approximate scheme for the last layer, we usually reduce $\approx 30\%$ total reachable set computation time while still derive precisely the output ranges of a FNN.

Algorithm 4 summarizes the main steps of the mixing scheme.

Algorithm 4 Mixing scheme for reachable set approximation for one layer

Input: I, W, b, N_{max} % Input set, weight matrix, bias vector, maximum number of polyhedra
Output: R % an reachable set
1: **procedure** R = `mixingReach`(I, W, b, N_{max})
2: $n = length(I)$
3: **if** $n < N_{max}$ **then**
4: $R = layerReach(I, W, b)$ % Algorithm 2
5: **else**
6: $I_1 = cluster\_merge(I, N_{max})$ % clustering and merging
7: $R = layerLazyReach(I_1, W, b)$ % Algorithm 3

Example 4.4 (Mixing reachability analysis of FNN). In this example, we reuse the FNN in Example 3.1 in which the input set is partitioned into 256 smaller sets. We want to compute the reachable set of this FNN with these partitioned input sets using the mixing scheme in which the maximum allowable number of polyhedra is chosen by $N_{max} = 100$. In this case, if we use the exact scheme to compute the reachable set, the numbers of polyhedra in the reachable set of the hidden and output layers respectively are 276 and 281. Using the mixing scheme with $N_{max} = 100$, the number of polyhedra of the reachable sets of the hidden and output layers is reduced to 100. The final result is displayed in Figure 8 with noticing that some polyhedra are overlapped by others.

4.3 Reachability analysis with conservativeness guarantee

Although several, over-approximation approaches have been proposed to compute an over-approximate reachable set of an FNN [**?**], it is unclear how good they are in term of conservativeness since there is no exact reachable set available for comparison. It is required that the over-approximate reachable set should be as

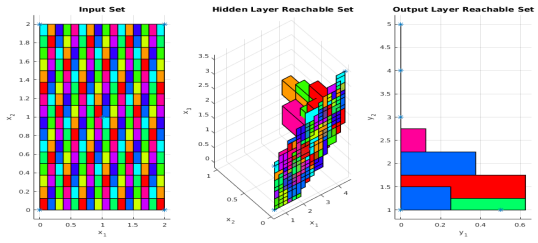

Figure 8: Mixing reachability analysis of the FNN in Example 3.1.

tight as possible in comparison with the actual reachable set. If the over-approximate reachable set is too conservative, it may be useless in safety verification. With the reachable set computed from the exact scheme, we can evaluate the conservativeness of existing over-approximation approaches. The conservativeness of a reachable set of an FNN is defined below.

Definition 4.5 (Conservativeness). Assume that the exact range of an output y of an FNN is $[a,\ b]$, $a < b$, and the approximate range of that output is $[\tilde{a},\ \tilde{b}]$, $\tilde{a} < \tilde{b}$, the conservativeness CSV on the output y is defined by:

$$CSV(y) = \frac{max(|\tilde{a} - a|, |\tilde{b} - b|)}{(b - a)} \times 100\%.$$

The range approximation is called an under-approximation (UA) if we have $\tilde{a} > a$ or $\tilde{b} < b$. If $\tilde{a} < a$ and $\tilde{b} > b$, it is called an over-approximation (OA). The conservativeness illustrates in percentage the maximum ratio of the distance between the approximate and the actual reachable sets and the length of the actual range. Generally, a large conservativeness implies that the over-approximation approach produces a coarse approximation of the actual reachable set and vice versa. Because the exact reachable set computation is expensive in general for a large FNN, there may be the case that the exact scheme does not work. In this case, we still can estimate the conservativeness of different approaches by evaluating an FNN on a randomly sampled input vectors on the input set. Then based on the evaluated outputs data, we determine approximately the ranges of the outputs by $[a \approx \bar{a},\ b \approx \bar{b}]$. Although this method is not sound, we still can estimate how conservative of the computed output ranges are. Besides computation time and scalability, the conservativeness defined above is utilized in the later section as a metric to compare different approaches.

Reachable set refinement with conservativeness. The conservativeness defined above can also be used to refine the output reachable set of an FNN automatically. Given a desired conservativeness, after computing an over-approximate reachable set, one can estimate the conservativeness of the reachable set quickly. If this conservativeness is larger than the desired one, the algorithm can recompute a new reachable set by 1) partitioning the input set using a large number of partitions (if we use lazy-approximate + input partition scheme), or 2) increasing the maximum allowable number of polyhedra for reachable set representation, i.e., N_{max}.

5 IMPLEMENTATION

We implement our approach in a Matlab toolbox called *nnv*[1] using the set library in MPT toolbox [**?**]. There are five main classes in our toolbox including *ReLU*, *Partition*, *Reduction*, *Layer*, and *FFNN*.

The *ReLU* class contains core static methods for the stepReLU operations. Methods for partitioning or over-approximating an input set are implemented in the *Partition* class. The *Reduction* class is responsible for clustering and merging polyhedra. It also contains an angle-based method for reducing the number of constraints of a polyhedron [**?**] and *recursiveMerge* method for merging a set of polyhedra into one polyhedron with neglecting the expensive convex-hull operation. These methods are beyond the purpose of this paper and used for our future work.

A layer of an FNN is constructed using the constructor of the *Layer* class. The inputs for the constructor are a weight matrix W, a bias vector b and an activation function, for example, $L = Layer(W, b, \text{“}ReLU\text{”})$. Currently, *nnv* supports linear and ReLU activation functions. An FNN is constructed from a set of layers using the *FFNN* class's constructor, for example, $F = FFNN([L_1\ L_2])$. The core method in the $FFNN$ class is the *reach* method that computes the reachable set of the constructed FNN with different schemes. The inputs of the *reach* method are: 1) an input set I (can be a polyhedron or a union of several polyhedra); 2) a reachability analysis scheme which may be “$exact$”, or “$approx$”, or “$mixing$”; 3) a number of cores utilized for computation N_{cores}, and 4) a maximum number of polyhedra N_{max} for the reachable set representation. Note that N_{max} is only used for the lazy-approximate and mixing schemes. For example, $F.reach(I, \text{“}exact\text{”}, 4, [])$ computes the *exact* reachable set of F with the input set I using the exact scheme in parallel with four cores.

6 EVALUATION

We evaluate the advantages and disadvantages of our approach on three aspects: conservativeness, computation time, and scalability. We run our proposed schemes on a variety of real-world FNN with several to thousand of neurons, and compare our approach with Sherlock [**?**], ReLuVal [**?**], and Reluplex [**?**] on these aspects.

6.1 Conservativeness on output ranges analysis

In this section, we evaluate the conservativeness of our approach in comparison with Sherlock [**?**], an optimization-based output ranges analysis framework for FNN.

Table 1 presents the ranges computed by different approaches on a set of FNN. We can see that the lazy-approximate scheme can estimate the output ranges of an FNN with one or two layers, e.g., *Abalone*_1 or *Housing*_1, tightly. For an FNN with more than two layers, the lazy-approximate scheme produces relatively conservative results as in the cases of *Abalone*_2, *Pollution*, and *Housing*_2. Combining the lazy-approximate scheme with input partition reduces the conservativeness of the results significantly, and the mixing scheme outperforms these two schemes in most cases. Notably, for the case of *Sherlock* N_4 network with 1000 neurons, the exact scheme reaches the time limitation for calculation (we use 4 cores in the computation and set time limitation of 2 hours). To estimate the range of the output, we get 5000 samples

[1] https://github.com/verivital/nnv

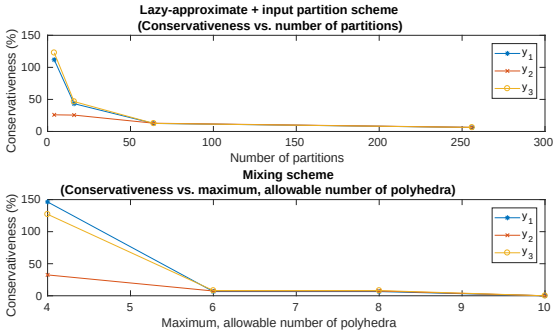

Figure 9: Conservativeness reduction for Pollution FNN.

of input vector from the input space and evaluate the neuron network on these samples. The lazy-approximate scheme produces a very conservative output ($\approx$ 227.27%) while its combination with input partition still gives a tight estimation for the output range ($\approx$ 15.79%). All approximate ranges produced by our scheme are either *exact* or *over-approximation (OA)*.

From thorough experiments, we note that Sherlock produces unexpected results. First, since Sherlock only supports ReLU activation functions, it is not applicable for the *Housing*_1 that has a linear activation function at the output layer, i.e., f(x) = x. Second, Sherlock produces precise ranges for the output of the *Pollution* network as our exact scheme does. However, for others networks, the output ranges obtained by Sherlock are neither an under-approximation nor an over-approximation of the actual ranges. Notably, we run the analysis on Sherlock benchmarks N_0 and N_4 and get incorrect output ranges. For example, in the case of N_4, we use their input set $0 \leq x_1 \leq 10, 0 \leq x_2 \leq 10$, and Sherlock estimates the output range is $[12.2373,\ 30.6204]$ while the range we estimate from rigorously sampling the network is $[8.9412,\ 128.3266]$. In the case of N_0, the input set is, $-0.5 \leq x_1 \leq 0, 0 \leq x_2 \leq 0.5$, and the exact output ranges computed by the exact scheme is $[2.3097,\ 8.7908]$ while Sherlock produces $[8.43312,\ 10.7476]$. The codes for evaluating conservativeness are available online for readers to reproduce the results. The input sets for all networks are presented in Appendix **??**.

The conservativeness of the lazy-approximation and input partition scheme depends on the number of partitions of the input set. The conservativeness decreases as the number of partitions increases. The cost we have to pay is the growth of the reachable set computation time. Similarly, in the mixing scheme, when we increase the maximum allowable number of polyhedra for reachable set representation, i.e., N_{max}, the conservativeness decreases while the computation time grows. Figure 9 describes the conservativeness reduction in the combined lazy-approximate and input partition scheme and the mixing scheme for the *Pollution* neural network.

FNN		Exact	Approximate	Approximate & partition	Mixing	Sherlock
Abalone$_1$[?]	Range	[2.1867, 9.0659]	[2.1867, 9.0659]	[2.1867, 9.0659]	[2.1867, 9.0659]	[0, 0]
$i = 8, o = 1, l = 2, n = 16$ [a]	CSV	***0%***	***0%***	***0%***	***0%***	***UN***
Abalone$_2$[?]	Range	[3.8949, 9.2042]	[0, 27.9824]	[0.1269 12.7738]	[1.964913.0393]	[6.5491, 13.184]
$i = 8, o = 1, l = 3, n = 19$	CSV	***0%***	***353.68% - OA*** [b]	***70.96% - OA***	***72.23% - OA***	***UN***
Pollution [?]	Range	[122.7841, 206.6794] [2.8291, 13.9060] [65.2020, 116.5141]	[0, 236.4108] [0, 18.0421] [0, 138.5269]	[86.4262, 222.3954] [0, 16.1268] [41.2877, 128.2185]	[122.6851, 212.1629] [2.8080, 14.7285] [65.1140, 120.6925]	[122.7841, 206.6794] [2.8291, 13.9060] [65.2020, 116.5141]
$i = 24, o = 3, l = 3, n = 16$	CSV	***[0%]*** ***[0%]*** ***[0%]***	***[146.4%] - OA*** ***[37.34.9%] - OA*** ***[127%] - OA***	***[43.337%] - OA*** ***[25.54%] - OA*** ***[46.6056%] - OA***	***[6.536%] - OA*** ***[7.426%] - OA*** ***[8.14%] - OA***	***[0%]*** ***[0%]*** ***[0%]***
Housing$_1$[?]	Range	[0.4302, 37.9303]	[0.4288, 41.2690]	[0.4288, 41.2690]	[0.4288, 41.2690]	Not applicable
$i = 12, o = 1, l = 2, n = 26$	CSV	***0%***	***8.9% - OA***	***8.9% - OA***	***8.9% - OA***	Not applicable
Housing$_2$[?]	Range	[2.2988, 48.7528]	[0.5183, 105.4048]	[0.5183, 66.3668]	[2.1951, 52.9881]	[1.732, 11.0073]
$i = 12, o = 1, l = 3, n = 31$	CSV	***0%***	***121.953% - OA***	***37.917% - OA***	***9.117% - OA***	***UN***
Sherlock N$_0$[?]	Range	[2.3097, 8.7908]	[0, 15.462]	[1.8203, 9.0728]	[0, 9.6517]	[8.43312, 10.7476]
$i = 2, o = 1, l = 1, n = 100$	CSV	***0%***	***102.93% - OA***	***7.55% - OA***	***35.63% - OA***	***UN***
Sherlock N$_4$[?]	Range	≈ [8.9412, 128.3266]	[0, 399.6550]	[0, 147.1845]	Timeout	[12.2373, 30.6204]
$i = 2, o = 1, l = 1, n = 1000$	CSV	≈ ***0%***	≈ ***227.27% -OA***	≈ ***15.79% - OA***	Timeout	***UN***

[a] i is the number of inputs, o is the number of outputs, l is the number of layers, n is the total number of neurons.
[b] *OA*: over-approximation, *UA*: under-approximation, *UN*: neither an over-approximation nor an under-approximation.

Table 1: Conservativeness (CSV) of different approaches.

FNN	Cores	Exact		Approximate		Approximate & Partition		Mixing	
		T(sec)	*R(%)*	*T(sec)*	*R(%)*	*T(sec)*	*R(%)*	*T(sec)*	*R(%)*
MNIST_1 []	1	58.85	0%	0.28	0%	4.54	0%	58.25	0%
i = 784	2	29.47	49.9%	0.24	14.5%	0.43	90.5%	28.06	51.8%
o = 1, l = 6	3	29.58	49.7%	0.23	17.8%	0.47	89.6%	27.5	52.8%
n = 141	4	20.94	64.4%	0.23	16.4%	0.46	89.9%	17.95	69.2%

*T is the reachable set computation time, R is the time reduction in percentage.

Table 2: Time reduction with parallel computing.

6.2 Time reduction with parallel computing

To evaluate the reduction in computation time with parallel computing, we train an image classification FNN called $MNIST_1$, using the well-known MNIST data set [**?**] with the accuracy of 90%. The MNIST data set consists of 60000 images of handwritten digits with resolution of 28×28 pixels. The trained $MNIST_1$ has $28 \times 28 = 784$ inputs, one output, 6 layers and 141 neurons in total. The normalized input set for this study is constructed as $I = \{x \in \mathbb{R}^{784} | \ x[i] = 0, 1 \leq i \leq 782, 0 \leq x[i] \leq 0.2, i = 783, 784\}$. This input set is obtained by using a specific image whose the last two pixels are affected by bounded perturbation. The improvement in reachable set computation time using parallel computing is demonstrated in Table 2. It can be seen from the table that the computation times of the exact, lazy-approximate and input partition and mixing schemes are reduced significantly when we perform these schemes with multiple cores. For the lazy-approximate scheme, parallel computing does not help much since there is only one input set (a hyper-rectangle) for all layers in computation. Notably, the lazy-approximate scheme and its combination with input partition are much faster than the exact and mixing schemes.

6.3 Scalability evaluation on DNN

Safety verification for ACAS Xu DNNs.

Adversarial robustness verification of DNN.

REFERENCES