ReluDiff: Differential Verification of Deep Neural Networks

Brandon Paulsen
University of Southern California
Los Angeles, California, USA

Jingbo Wang
University of Southern California
Los Angeles, California, USA

Chao Wang
University of Southern California
Los Angeles, California, USA

ABSTRACT

As deep neural networks are increasingly being deployed in practice, their efficiency has become an important issue. While there are compression techniques for reducing the network's size, energy consumption and computational requirement, they only demonstrate *empirically* that there is no loss of accuracy, but lack formal guarantees of the compressed network, e.g., in the presence of adversarial examples. Existing verification techniques such as Reluplex, ReluVal, and DeepPoly provide formal guarantees, but they are designed for analyzing a single network instead of the relationship between two networks. To fill the gap, we develop a new method for *differential verification* of two closely related networks. Our method consists of a fast but approximate *forward interval analysis pass* followed by a *backward pass* that iteratively refines the approximation until the desired property is verified. We have two main innovations. During the forward pass, we exploit structural and behavioral similarities of the two networks to more accurately bound the difference between the output neurons of the two networks. Then in the backward pass, we leverage the gradient differences to more accurately compute the most beneficial refinement. Our experiments show that, compared to state-of-the-art verification tools, our method can achieve orders-of-magnitude speedup and prove many more properties than existing tools.

1 INTRODUCTION

As deep neural networks (DNNs) make their way into safety critical systems such as aircraft collision avoidance [17] and autonomous driving [3], where errors may lead to catastrophes, there is a growing need for formal verification. The situation is further exacerbated by *adversarial examples* [11, 42], which are security exploits created specifically to cause erroneous classifications [22, 31, 32, 51]. There is also a growing need for reducing the size of the neural networks deployed on energy- and computation-constrained devices. Consequently, compression techniques [14] have emerged to prune unnecessary edges, quantize the weights of remaining edges, and retrain the networks, but they do not provide any formal guarantee – typically the accuracy of a compressed network is only demonstrated *empirically*.

While empirical evidence or statistical analysis may increase our confidence that a network behaves as expected for most of the inputs, they cannot prove that it does so for all inputs. Similarly, while heuristic search and dynamic analysis techniques, including testing [25, 35, 43] and fuzzing [33, 49, 50], may quickly discover adversarial examples, they cannot prove the absence of such examples. At the same time, while state-of-the-art verification techniques [8–10, 16, 19, 29, 37, 40, 44], including Reluplex [18], ReluVal [45] and DeepPoly [39], can provide formal proofs, they are designed for analyzing a single network as opposed to the relationship between two networks.

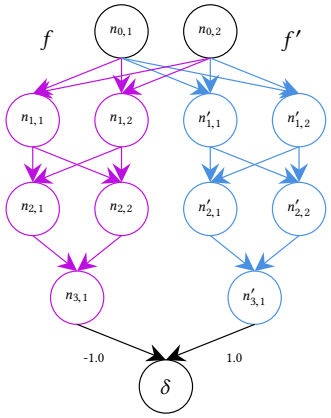

Figure 1: Differential verification of deep neural networks.

In this work, we focus on *differential verification* of two closely related networks. In this problem domain, we assume that f and f' are two neural networks trained for the same task; that is, they accept the same input x and are expected to produce the same output. They are also structurally the same while differing only in the numerical values of edge weights (which allows us to analyze compression techniques such as quantization and edge pruning [14]). In this context, differential verification is concerned with proving $\forall x \in X. |f'(x) - f(x)| < \epsilon$, where X is an input region of interest and ϵ is some reasonably small bound. This problem has not received adequate attention and, as we will show in this work, existing tools are ill-suited for solving this problem.

The key limitation of existing tools is that, since they are designed to analyze the behavior of a single network, they do not have the ability to exploit the structural similarities of two closely related networks. They also have difficulty handling the constraint that the inputs to both f and f' are identical. Typically, these tools work by computing the conservative value ranges of all neurons from input to output in a layer-by-layer style. In the early layers, they may be able to maintain relationships between the inputs, but as the functions become increasingly non-linear in subsequent layers, approximations must be made. This "eager" approximation means relationships between the inputs of f and f' are mostly lost, causing extremely large over-approximations in the output layer.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380337

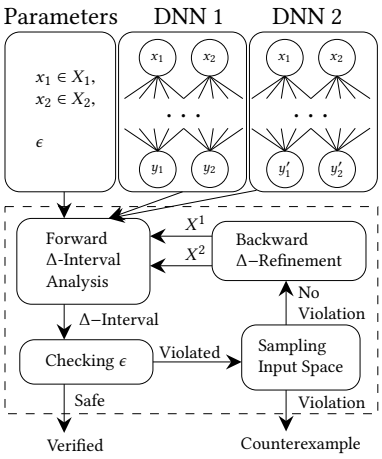

Figure 2: The differential verification flow of ReluDiff.

In fact, state-of-the-art verification tools that we have investigated (ReluVal [45] and DeepPoly [39]) struggle to verify that *two identical networks are the same*. To carry out this *litmus test* without drastically altering these tools, we construct a combined network as shown in Figure 1, where f and f' are actually the same network (i.e. same structure and edge weights). Since they share the same input x, we expect $f(x) - f(x)$ to be 0 regardless of the input region for x. While our method can easily prove that $|f(x) - f(x)| < \epsilon$ for an arbitrarily small ϵ in less than a second, none of the existing tools are able to do so. In fact, DeepPoly cannot verify it no matter how much time is given (it is not a complete method) and ReluVal times out after several hours.

Figure 2 shows the overall flow of our method, ReluDiff, whose input consists of the two networks (DNN1 and DNN2), an input region ($x_1 \in X_1$ and $x_2 \in X_2$), and a bound ϵ on the output difference. There are three possible outcomes: (1) *verified*, meaning that the output difference is proved to be less than ϵ; (2) *falsified*, meaning a counterexample is found; or (3) *unknown*, meaning that verification remains inconclusive due to bounds on the computing resources.

Internally, ReluDiff iterates through two steps: a forward pass and a backward pass. The forward pass computes over-approximated value differences of corresponding neurons in the two networks, and propagates them layer by layer from the input to the output. If the output difference is within the region $[-\epsilon, \epsilon]$, the property is verified. Otherwise, ReluDiff samples a fixed number of concrete examples from the input space and tests if they violate the property. If a violation is found, the property is falsified; otherwise, ReluDiff enters the refinement phase.

The goal of refinement is to identify an input region that should be divided into subregions. By using these subregions to perform the forward pass again, some of the forced over-approximations may be avoided, thus leading to significant accuracy increase. To identify the right input region for refinement, the *backward pass* computes the difference of the gradients of the two networks and uses it to find input regions that, once divided into subregions, are more likely to result in accuracy increase.

While iterative interval analysis has been used in verifying neural networks before [45], the focus has always been on a single network. In this work, we show that, by focusing on both networks simultaneously, we can be more efficient and accurate compared to analyzing each network in isolation. Note that, in differential verification, the two networks have identical structures and similar behaviors; therefore, we can easily develop a correspondence between neurons in f and f', thus allowing a lock-step style verification. Lock-step verification allows us to directly compute the differences in values of neurons and propagate these differences through edges. It also allows symbolic intervals to be used to avoid some of the approximations. Since error caused by approximation grows quickly, sometimes exponentially [44], as it is propagated through edges and neurons, this can significantly increase accuracy.

When approximation must be made, e.g., due to non-linearity of ReLU, we can handle them better by focusing on the value differences instead of the absolute values. For example, in ReluVal [45], if a symbolic expression that represents the ReLU input may be both positive and negative, the symbolic expression must be replaced by an interval with concrete upper and lower bounds, which introduces additional error. In contrast, we can be more accurate: even if the input value of a neuron may be both positive and negative, in many cases we still can avoid introducing error into the difference.

We have implemented ReluDiff in a tool and evaluated it on a number of feed-forward neural network benchmarks, including ACAS Xu for aircraft collision avoidance [17], MNIST for handwritten digit recognition [24], and HAR for human activity recognition [1]. We also experimentally compared ReluDiff with state-of-the-art tools, including ReluVal [45] and DeepPoly [39]. Our experimental results show that, in almost all cases, ReluDiff outperforms these existing tools in both speed and accuracy. In total, we evaluate on 842 properties over our benchmark networks. ReluDiff was often one to two orders-of-magnitude faster, and was able to prove 745 out of the 842 properties whereas none of the other tools can prove more than 413 properties.

To summarize, we make the following contributions:

- We propose the first iterative symbolic interval analysis for differential verification of two neural networks.
- We develop a forward pass algorithm for more accurately computing the value differences for corresponding neurons.
- We develop a backward pass algorithm, based on gradient difference, for computing the refinement.
- We implement the method and demonstrate its advantages over existing tools in terms of both speed and accuracy.

The remainder of the paper is organized as follows. First, we use examples to motivate our method in Section 2. Then, we review the basics of neural networks and interval analysis in Section 3. Next, we present our method for the forward pass in Section 4, followed by our method for the backward pass in Section 5. We present our experimental results in Section 6. We review the related work in Section 7. Finally, we give our conclusions in Section 8.

2 MOTIVATION

We illustrate the problems of existing verification tools using examples and then highlight our main contributions.

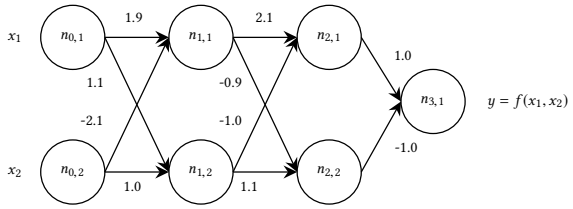

Figure 3: A neural network with two inputs and one output.

2.1 Differential Verification

Figure 3 shows a feed-forward neural network with one input layer, two hidden layers, and one output layer. The input layer has two nodes $n_{0,1}$ and $n_{0,2}$, corresponding to the two input variables x_1 and x_2. Each hidden layer consists of two neurons, $n_{1,1}, n_{1,2}$ in one layer and $n_{2,1}, n_{2,2}$ in the other layer. Each of these neurons has two computation steps: the affine transformation and the ReLU activation. For example, inside $n_{1,1}$, the affine transformation is $1.9x_1 - 2.1x_2$ and the ReLU activation is $max(0, 1.9x_1 - 2.1x_2)$. The output layer has one node, representing the value of $y = f(x_1, x_2)$. In general, f is a non-linear function over x_1 and x_2.

In differential verification, we are concerned with the relationship between $f(x_1, x_2)$ and another network $f'(x_1, x_2)$. For the sake of example, we focus on a network compression technique called *quantiziation* [14] in which the edge weights of f are rounded to the nearest whole number to obtain f'. However, we note that our method can be used on *any* two networks with similar structures, e.g., when f' is created using other techniques including *edge pruning* and *network retraining* [14, 15, 17, 38].

These techniques, in general, raise the concern on how they affect the network's behavior. In particular, we would like to verify that the new network produces outputs within some bound relative to the original network. Formally, let $f' : \mathbb{X} \to \mathbb{Y}$ be the second network and $f : \mathbb{X} \to \mathbb{Y}$ be the first network. We would like to verify that $|f'(x) - f(x)| < \epsilon$ for all $x \in X$, where $X \subseteq \mathbb{X}$ is some region of importance in the input domain $\mathbb{X}$.

2.2 Existing Approaches

Existing tools for verifying neural networks target only a single network at a time, and are often geared toward proving the absence of *adversarial examples*. That is, given an input region of interest, they decide if the output stays in a desired region. For the network in Figure 3, in particular, the input region may be $x_1 \in [4, 6]$ and $x_2 \in [1, 5]$, and the desired output may be $f(x_1, x_2) < 15$. However, these tools are not designed for verifying the relationship between two networks. While we could try and re-use them for our purpose, they lack the ability to exploit the similarities of the two networks.

For example, we could use the existing tool ReluVal [45] on both f and f' to compare the concrete output intervals it computes for an input region of interest, e.g., $[f_{low}, f_{up}]$ and $[f'_{low}, f'_{up}]$. In order to conservatively estimate the difference between f and f', we must assume the maximum difference falls in the interval $[f'_{low} - f_{up}, f'_{up} - f_{low}]$. In Figure 3, the interval difference would be $[-25.76, 22.93]$, which is too large to be useful.

Even though ReluVal could tighten the interval by refining the input intervals, this naive approach cannot even verify that

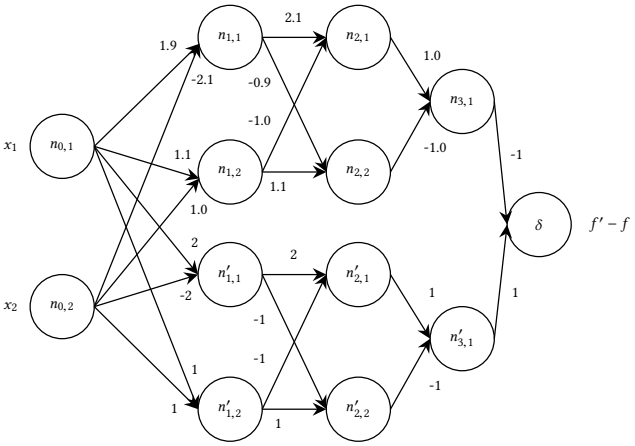

Figure 4: Naive differential verification of the two networks.

two identical networks always produce the same output, since the output intervals do not capture that the corresponding inputs to f and f' (i.e., values of x_1 and x_2) are always the same.

To compensate, we could encode the constraint that values of the corresponding inputs are always the same by composing f and f' into a single feed-forward network equivalent to $f' - f$, as shown in Figure 4. In theory, a sound and complete technique would be able to verify, *eventually*, that the output difference is bounded by an arbitrarily small ϵ, but with a caveat.

That is, to maintain the relationships between the input variables and the difference in the outputs of the two networks, each neuron must remain in a linear state across the entire input region; otherwise, approximation must be made to maintain the soundness of the interval analysis. However, approximation inevitably loses some of the relationships between the inputs and the outputs. Indeed, we constructed some merged networks in the same way as in Figure 4 and then fed them to existing tools. Unfortunately, they all exhibit the "worst-case" value range blowup in the output.

The key reason is that existing tools such as ReluVal are forced to approximate ReLU activations by concretizing, which is then followed by interval subtractions, thus causing error introduced by these approximations to be quickly amplified. The forward pass over f computes an output interval of $[-1.2x_1 - 1.1x_2, -1.1x_1 - x_2 + 19.53]$, and for f' it computes $[-x_1 - x_2, -x_1 - x_2 + 20]$. Although the equations are symbolic, the difference $[-21.36, 19.13]$, computed conservatively by ReluVal, is still too large to be useful.

2.3 Our Method

Existing tools cannot exploit structural and behavioral similarities of the two networks in differential verification. Our insight is to leverage such similarities to drastically improve both the efficiency and the accuracy of the verification tool.

Specifically, in this work, we pair neurons and edges of the first network with those of the second network and then perform a *lock-step* verification. This allows us to focus on the value differences of the corresponding neurons as opposed to their absolute values. The benefit is that doing so results in both fewer and tighter approximations and more error reduction due to the use of symbolic

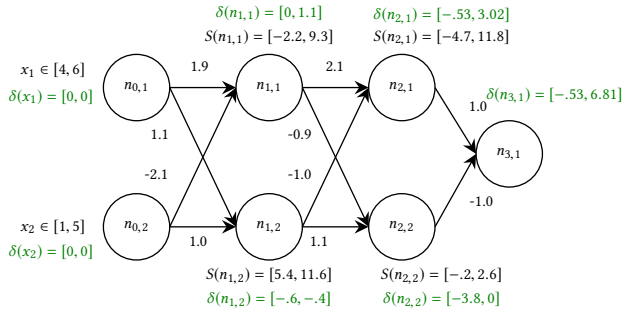

Figure 5: Forward interval analysis of a neural network.

intervals. We also perform better refinement by focusing on inputs that have the greatest influence on the output difference, rather than the absolute output values.

While focusing on the *difference* as opposed to *absolute values seems* to be a straightforward idea, there are many technical challenges. For example, there will be significantly more complex ReLU activation patterns to consider since we have to handle both networks simultaneously, instead of one network at a time. Approximating symbolic intervals when considering the output difference of two ReLU activations (i.e., ReLU(x') - ReLU(x)) has yet to be studied and is non-trivial. Furthermore, how to determine which input neuron to refine when the goal is to reduce error in the output difference between two networks has not been considered either.

In this work, we develop solutions to overcome these challenges. During forward interval analysis, we carefully consider the ReLU activation patterns, and propose a technique for handling each pattern soundly while minimizing the approximation error. During the refinement, we compute the difference between gradients of the two networks, and use it to identify the input neuron most likely to increase the accuracy of the differential verification result.

As a result, our method can solve the differential verification problems much more efficiently. Consider the litmus test of verifying the equivalence of two identical networks. Our method can obtain a formal proof (that $|f' - f| < \epsilon$) after performing the forward interval analysis once; in contrast, all other existing tools have failed to do so. For the example in Figure 5, we can prove the output difference $\delta(n_{3,1})$ is bounded by $[-0.53, 6.81]$ after only the first pass. It also outperforms existing tools on other verification problems where f' is obtained from f through quantization; details of the experimental comparisons are in Section 6.

3 PRELIMINARIES

First, we review the basics of interval analysis for neural networks.

3.1 Neural Networks

We consider a neural network as a non-linear function that takes some value in $\mathbb{R}^n$ as input and returns some value in $\mathbb{R}^m$ as output, where n is the number of input variables and m is the number of output variables. Let the network f be denoted $f : \mathbb{X} \rightarrow \mathbb{Y}$, where $\mathbb{X} \subseteq \mathbb{R}^n$ is the input domain and $\mathbb{Y} \subseteq \mathbb{R}^m$ is the output domain. In image recognition applications, for instance, $\mathbb{X}$ may be a vector of pixels representing an image and $\mathbb{Y}$ may be a vector of probabilities for class labels. In aircraft collision detection, on the other hand, $\mathbb{X}$ may be sensor data and $\mathbb{Y}$ may be a set of actions to take.

In this work, we consider fully-connected feed-forward networks with rectified linear unit (ReLU) activations, which are the most popular in practical hardware/software implementations. Thus, $y = f(x)$ is a series of affine transformations (e.g., $x \cdot W_1 = \Sigma_i x_i w_{1,i}$) followed by point-wise ReLU (e.g., $\text{ReLU}(x \cdot W_1) = max(0, x \cdot W_1)$). Let W_k, where $1 \leq k \leq l$, be the weight matrix associated with the k-th layer, and l be the number of layers; the affine transformation in the k-th layer is a standard matrix multiplication, followed by the point-wise application of ReLU.

Formally, $f = f_l(f_{l-1}(...f_2(f_1(x \cdot W_1) \cdot W_2))... \cdot W_{l-1})$, where each f_k, $1 \leq k \leq l$, is a point-wise ReLU. For the network in Figure 3, in particular, the input is a vector $x = \{x_1, x_2\}$, the weight matrix $W_3 = \{1.0, -1.0\}^T$, and $x \cdot W_1 = \{1.9x_1 - 2.1x_2, 1.1x_1 + 1.0x_2\}$.

For ease of presentation, we denote the weight of the edge from the i-th neuron of layer $k-1$ to the j-th neuron of layer k as $W_k[i, j]$. We also denote the j-th neuron of layer k as $n_{k,j}$.

3.2 Interval Analysis

To ensure that our analysis is over-approximated, we use interval analysis [30], which can be viewed as a specific instantiation of the general abstract interpretation [5] framework. Interval analysis is well-suited for analyzing ReLU neural networks as it has well-defined transformers over addition, subtraction, and scaling (i.e., multiplication by a constant).

Interval addition as denoted $[a, b] + [c, d] = [a + c, b + d]$ does not lead to loss of accuracy. Scaling as denoted $[a, b] * c = [a * c, b * c]$ when $c \geq 0$, or $[a, b] * c = [b * c, a * c]$ when $c < 0$, does not lead to loss of accuracy either. Interval subtraction as denoted $[a, b] - [c, d] = [a - d, b - c]$, however, may lead to accuracy loss.

To illustrate, consider $f(x) = 2.1x$ and $f'(x) = 2x$, and say we want to approximate their difference for the input region $x \in [-1, 1]$. Using interval arithmetic, we would compute $f([-1, 1]) - f'([-1, 1]) = [-2.1, 2.1] - [-2, 2] = [-4.1, 4.1]$. Clearly this is far from the exact interval of $f(x) - f'(x) = 2.1x - 2x = 0.1x$ over $x \in [-1, 1]$, which is $[-0.1, 0.1]$. The reason for such loss of accuracy is that, during interval arithmetic, the relationship between values of $2.1x$ and $2x$ (i.e., they are for the same value of x) is lost.

3.3 Symbolic Interval

One way to overcome the accuracy loss is using *symbolic intervals* [45], which can encode the constraint that inputs to f and f' are actually related. With this technique, we would use the symbol x with the constraint $x \in [-1, 1]$ to initialize the input intervals. Then, the computation becomes $f([x, x]) - f'([x, x]) = [2.1x, 2.1x] - [2x, 2x] = [0.1x, 0.1x]$. Finally, we would compute the upper and lower bounds for $x \in [-1, 1]$ and return the precise interval $[-0.1, 0.1]$.

Unfortunately, symbolic intervals depend on f and f' being linear in the entire input region in order to be sound. Indeed, if we add ReLU to the functions, i.e., $\text{ReLU}(f(x)) = max(0, 2.1x)$ and $\text{ReLU}(f'(x)) = max(0, 2x)$, where $x \in [-1, 1]$, then the lower and upper bounds are no longer precise nor sound. The reason is because $max(0, 2.1x)$ is non-linear in $x \in [-1, 1]$. Thus, we have to approximate using the concrete interval $[0, 2.1]$. Similarly, $max(0, 2x)$ is approximated using $[0, 2]$. Thus, $[0, 2.1] - [0, 2] = [-2, 2.1]$.

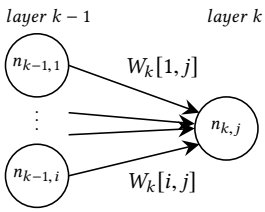

Figure 6: Diagram of weight notations.

3.4 Refinement

To improve the accuracy of the symbolic interval analysis, we need to divide the input region into subregions. The intuition is that, within a smaller subregion, the ReLU is less likely to exhibit non-linear behavior and force the analyzer to over-approximate. Consider $\text{ReLU}(2.1x) = max(0, 2.1x)$, where $x \in [-1, 1]$. After the input region is divided into subregions $x \in [-1, 0] \cup [0, 1]$, we have $max(0, 2.1x) = [0, 0]$ for $x \in [-1, 0]$ and $max(0, 2.1x) = [2.1x, 2.1x]$ for $x \in [0, 1]$. In both cases, the intervals are precise – there is no approximation at all.

When we only have one input variable, we do not have a choice on which variable to refine. However, neural networks have many inputs, and refining some of them will not always yield benefit. Thus, we have to identify the right input to split.

Consider $f(x_1, x_2) = \text{ReLU}(5x_1) + \text{ReLU}(2x_2) - \text{ReLU}(x_2)$, where $x_1 \in [1, 3]$ and $x_2 \in [-1, 1]$. The initial analysis is not very accurate due to approximations caused by the ReLU: $\text{ReLU}(5 * [1, 3]) + \text{ReLU}(2 * [-1, 1]) + \text{ReLU}([-1, 1]) = \text{ReLU}([5, 15]) + \text{ReLU}([-2, 2]) + \text{ReLU}([-1, 1]) = [5, 15] + [0, 2] - [0, 1] = [4, 17]$.

If we split $x_1 \in [1, 3]$ into $x_1 \in [1, 2] \cup [2, 3]$ and perform interval analysis for both subregions, the output would be $[4, 12] \cup [9, 17] = [4, 17]$, which does not improve over the initial result.

In contrast, if we split $x_2 \in [-1, 1]$ into $x_2 \in [-1, 0] \cup [0, 1]$, the accuracy would improve significantly. Since the ReLU is always activated for $x_2 \in [0, 1]$, $\text{ReLU}(2x_2)$ and $\text{ReLU}(x_2)$ can be represented by $[2x_2, 2x_2]$ and $[x_2, x_2]$, respectively, and $\text{ReLU}(2x_2) - \text{ReLU}(x_2) = [x_2, x_2] = [0, 1]$. Since the ReLU is always de-activated for $x_2 \in [-1, 0]$, we have $\text{ReLU}(2x_2) - \text{ReLU}(x_2) = [0, 0]$. Thus, the combined output $[5, 15] \cup [5, 16] = [5, 16]$ is more accurate than the initial approximation $[4, 17]$.

While how to analyze non-linear activation functions such as ReLU has been studied in prior work [19, 39, 44], none of the existing techniques touch upon the complex scenarios arising from differential verification of two closely related networks. Our work fills the gap. Specifically, we propose a more accurate forward pass for the interval analysis (Section 4) and a more accurate backward pass for the refinement (Section 5).

4 FORWARD INTERVAL ANALYSIS

In this section, we describe our forward pass for computing the value differences between neurons in the two networks. Recall that network f has l layers and weight matrices W_k, $1 \le k \le l$, and $n_{k,j}$ is the j-th node in the k-th layer. Furthermore, $W_k[i, j]$ is the weight of the edge from $n_{k-1,i}$ to $n_{k,j}$. We illustrate these notations in Figure 6. Similarly, network f' has weight matrices W'_k, nodes

Input: network f, network f', input region X
Result: difference $\{\delta(n_{l,j})\}$ for output
Initialize $\{S(n_{0,j})\}$, $\{S(n'_{0,j})\}$ to input region X and $\{\delta(n_{0,j})\}$ to 0
for k ***in*** *1..N_{Layer}* **do**
  // Affine transformer
  for j ***in*** *1..layerSize[k]* **do**
    $S^{in}(n_{k,j}) \leftarrow \sum_i S(n_{k-1,i}) \cdot W_k[i, j]$
    $S^{in}(n'_{k,j}) \leftarrow \sum_i S(n'_{k-1,i}) \cdot W'_k[i, j]$
    $\delta^{in}(n_{k,j}) \leftarrow \sum_i (S(n_{k-1,i}) \cdot W^{\Delta}_k[i, j] + \delta(n_{k-1,i}) \cdot W'_k[i, j])$
  end
  if $k = N_{Layer}$ **then**
    return $\{\delta^{in}(n_{k,j})\}$
  end
  // ReLU transformer
  for j ***in*** *1..layerSize[k]* **do**
    $\langle S(n_{k,j}), S(n'_{k,j}), \delta(n_{k,j})\rangle \leftarrow$
      ReluTransform($S^{in}(n_{k,j})$, $S^{in}(n'_{k,j})$, $\delta^{in}(n_{k,j})$)
  end
end

Algorithm 1: Forward symbolic interval analysis.

$n'_{k,j}$, and weights $W'_k[i, j]$. Let $W^{\Delta}_k[i, j]$ be the weight difference, i.e $W^{\Delta}_k[i, j] = W'_k[i, j] - W_k[i, j]$.

We now define notations for the interval values of neurons. Since each neuron $n_{k,j}$ has an affine transformation (multiplying by the incoming weights) and a ReLU, we denote the input interval to the neuron (after applying the affine transform) as $S^{in}(n_{k,j})$, and we denote the output interval of the neuron (after applying the ReLU) as $S(n_{k,j})$. We denote the interval bound on the difference between the inputs to $n'_{k,j}$ and $n_{k,j}$ as $\delta^{in}(n_{k,j})$, and we denote the interval difference between the outputs as $\delta(n_{k,j})$. Finally, we denote the symbolic upper and lower bound of any value using the notation $\text{UB}()$ and $\text{LB}()$. For example, $\text{UB}(S(n_{k,j}))$ and $\text{LB}(S(n_{k,j}))$ denote the symbolic upper and lower bound for the output of neuron $n_{k,j}$.

With this notation, our forward pass is shown in Algorithms 1 and 2. The input consists of the two networks, f and f', and the input region of interest X, which defines an interval for each input neuron. After initializing the input intervals, the algorithm iteratively computes each $S(n_{k,j}), S(n'_{k,j})$, and $\delta(n_{k,j})$ of the subsequent layer by applying the affine transformation followed by the ReLU transformation. The algorithm iterates until the output layer is reached. In addition, it computes the *gradient masks* for the neurons of f and f', denoted as R and R', which record the state of each neuron in the forward pass ($[0, 0]$ is inactive, $[1, 1]$ is active, and $[0, 1]$ is both). These are used in the refinement phase (Section 5) to determine which input neuron to refine.

We omit discussion of computing $S^{in}(n_{k,j})$ and $S(n_{k,j})$ because it has been studied in previous work [45]. We focus on computing $\delta^{in}(n_{k,j})$ and $\delta(n_{k,j})$ in the following section.

4.1 The Affine Transformer

Computing $\delta^{in}(n_{k,j})$ involves two steps. First, we compute $\delta(W_k[i, j])$ for each incoming edge to $n_{k,j}$. Here, $\delta(W_k[i, j])$ is the difference in values produced by the edges from $n_{k-1,i}$ to $n_{k,j}$ and from $n'_{k-1,i}$ to $n'_{k,j}$. Second, we sum them to obtain $\delta^{in}(n_{k,j})$.

In the first step, there are two components to consider when computing $\delta(W_k[i, j])$. First, there is the "new quantity" introduced by the difference in edge weights, which formally is $S(n_{k-1,i}) \cdot W^{\Delta}_k[i, j]$. In English, this is the interval of neuron $n_{k-1,i}$ in the previous layer

multiplied by the edge weight difference in the current layer. Second, there is the "old quantity" accumulated in previous layers being scaled by the edge weight in the current layer. Formally this is $\delta(n_{k-1,i}) \cdot W'_k[i,j]$. Below we write out the formal derivation:

$$\begin{aligned}
\delta(W_k[i,j]) &= W'_k[i,j] \cdot S(n'_{k-1,i}) - W_k[i,j] \cdot S(n_{k-1,i}) \\
&= W'_k[i,j] \cdot S(n'_{k-1,i}) - W_k[i,j] \cdot S(n_{k-1,i}) + \\
&(W'_k[i,j] \cdot S(n_{k-1,i}) - W'_k[i,j] \cdot S(n_{k-1,i})) \\
&= (W'_k[i,j] \cdot S(n'_{k-1,i}) - W'_k[i,j] \cdot S(n_{k-1,i})) + \\
&(W'_k[i,j] \cdot S(n_{k-1,i}) - W_k[i,j] \cdot S(n_{k-1,i})) \\
&= \delta(n_{k-1,i}) \cdot W'_k[i,j] + S(n_{k-1,i}) \cdot W^{\Delta}_k[i,j] \,.
\end{aligned}$$

In the second step, we sum together each incoming $\delta(W_k[i,j])$ term to obtain $\delta^{in}(n_{k,j})$, which is the difference of the values $S^{in}(n_{k,j})$ and $S^{in}(n'_{k,j})$. That is,

$$\delta^{in}(n_{k,j}) = \sum_i \delta(W_k[i,j]) \,.$$

We demonstrate the computation on the example in Figure 5. First, we compute $\delta(W_1[1,1]) = 0.1 \cdot [4,6] + 2 \cdot [0,0] = [0.4, 0.6]$ and $\delta(W_1[2,1]) = 0.1 \cdot [1,5] + 2 \cdot [0,0] = [0.1, 0.5]$. Then, we compute $\delta(W_1[1,2]) = [-0.6, -0.4]$ and $\delta(W_1[2,2]) = [0,0]$.

Next, we compute $\delta^{in}(n_{1,1}) = \delta(W_1[1,1]) + \delta(W_1[2,1]) = [0.5, 1.1]$ and $\delta^{in}(n_{1,2}) = \delta(W_1[1,2]) + \delta(W_1[2,2]) = [-0.6, -0.4]$.

4.2 The ReLU Transformer

Next, we apply the ReLU activation to $\delta^{in}(n_{k,j})$ to obtain $\delta(n_{k,j})$. We consider nine cases based on whether the ReLUs of $n_{k,j}$ and $n'_{k,j}$ are *always activated*, *always deactivated*, or *non-linear*. In the remainder of the section, we discuss how to soundly over-approximate. Algorithm 2 shows the details.

First, we consider the three cases when the ReLU in $n_{k,j}$ is *always deactivated*. (1) If the ReLU in $n'_{k,j}$ is also *always deactivated*, the outputs of both ReLUs are 0, and the difference is 0. (2) If the ReLU in $n'_{k,j}$ is *always activated*, the output difference will be $S^{in}(n'_{k,j}) - [0,0] = S^{in}(n'_{k,j})$. Note that we can maintain symbolic equations here. (3) If the ReLU in $n'_{k,j}$ is *non-linear*, the difference will be $[0, \overline{\mathrm{UB}}(S^{in}(n'_{k,j}))]$. While $\mathrm{UB}(S^{in}(n'_{k,j}))$ is the symbolic upper bound of $S^{in}(n'_{k,j})$, $\overline{\mathrm{UB}}(S^{in}(n'_{k,j}))$ is the concrete upper bound. Note that since $n'_{k,j}$ is non-linear, we must concretize to be sound.

Next, we consider the three cases when the ReLU in $n_{k,j}$ is *always activated*. (1) If the ReLU in $n'_{k,j}$ is *always deactivated*, the difference is $[0,0] - S^{in}(n_{k,j})$. Again, we can soundly maintain symbolic equations. (2) If the ReLU in $n'_{k,j}$ is *always activated*, then the difference is the same as $\delta^{in}(n_{k,j})$. (3) If the ReLU in $n'_{k,j}$ is *non-linear*, the difference is $[0, \overline{\mathrm{UB}}(S^{in}(n'_{k,j}))] - S^{in}(n_{k,j})$, which is the same as $[-\mathrm{UB}(S^{in}(n_{k,j})), \overline{\mathrm{UB}}(S^{in}(n'_{k,j})) - \mathrm{LB}(S^{in}(n_{k,j}))]$. Again, we concretize to ensure soundness.

Third, we consider the three cases where the ReLU in $n_{k,j}$ is *non-linear*. (1) If the ReLU in $n'_{k,j}$ is *always deactivated*, the difference is $[0,0] - [0, \overline{\mathrm{UB}}(S^{in}(n_{k,j}))]$. (2) If the ReLU in $n'_{k,j}$ is *always activated*, the difference is $S^{in}(n'_{k,j}) - [0, \overline{\mathrm{UB}}(S^{in}(n_{k,j}))]$, which is the same as $[\mathrm{LB}(S^{in}(n'_{k,j})) - \overline{\mathrm{UB}}(S^{in}(n_{k,j})), \mathrm{UB}(S^{in}(n'_{k,j}))]$. (3) If the ReLU

Input: value $S^{in}(n_{k,j})$, value $S^{in}(n'_{k,j})$, difference $\delta^{in}(n_{k,j})$
Result: value $S(n_{k,j})$, value $S(n'_{k,j})$, difference $\delta(n_{k,j})$

```
if UB(S^in(n_{k,j})) ≤ 0 then
    R[k][j] = [0, 0]
    S(n_{k,j}) = [0, 0]
    if UB(S^in(n'_{k,j})) ≤ 0 then
        R'[k][j] = [0, 0]
        S(n'_{k,j}) = [0, 0]
        δ(n_{k,j}) = [0, 0]
    else if LB(S^in(n'_{k,j})) > 0 then
        R'[k][j] = [1, 1]
        S(n'_{k,j}) = S^in(n'_{k,j})
        δ(n_{k,j}) = S^in(n'_{k,j})
    else
        R'[k][j] = [0, 1]
        S(n'_{k,j}) = [0, UB̄(S^in(n'_{k,j}))]
        δ(n'_{k,j}) = [0, UB̄(S^in(n'_{k,j}))]
else if LB(S^in(n_{k,j})) > 0 then
    R[k][j] = [1, 1]
    S(n_{k,j}) = S^in(n_{k,j})
    if UB(S^in(n'_{k,j})) ≤ 0 then
        R'[k][j] = [0, 0]
        S(n'_{k,j}) = [0, 0]
        δ(n_{k,j}) = -S^in(n_{k,j})
    else if LB(S^in(n'_{k,j})) > 0 then
        R'[k][j] = [1, 1]
        S(n'_{k,j}) = S^in(n'_{k,j})
        δ(n_{k,j}) = δ^in(n_{k,j})
    else
        R'[k][j] = [0, 1]
        S(n'_{k,j}) = [0, UB̄(S^in(n'_{k,j}))]
        δ(n_{k,j}) = [-UB(S^in(n_{k,j})), UB̄(S^in(n'_{k,j})) - LB(S^in(n_{k,j}))]
        Opt. 1:
        δ(n_{k,j}) = [max(-UB̄(S^in(n_{k,j})), LB̲(δ^in(n_{k,j}))),
                      max(-LB̲(S^in(n_{k,j})), UB̄(δ^in(n_{k,j})))]
else
    R[k][j] = [0, 1]
    S(n_{k,j}) = [0, UB̄(S^in(n_{k,j}))]
    if UB(S^in(n'_{k,j})) ≤ 0 then
        R'[k][j] = [0, 0]
        S(n'_{k,j}) = [0, 0]
        δ(n_{k,j}) = [-UB̄(S^in(n_{k,j})), 0]
    else if LB(S^in(n'_{k,j})) > 0 then
        R'[k][j] = [1, 1]
        S(n'_{k,j}) = S^in(n'_{k,j})
        δ(n_{k,j}) = [LB(S^in(n'_{k,j})) - UB̄(S^in(n_{k,j})), UB(S^in(n'_{k,j}))]
        Opt. 2:
        δ(n_{k,j}) = [min(LB̲(S^in(n'_{k,j})), LB̲(δ^in(n_{k,j}))),
                      min(UB̄(δ^in(n_{k,j})), UB̄(S^in(n'_{k,j})))]
    else
        R'[k][j] = [0, 1]
        S(n'_{k,j}) = [0, UB̄(S^in(n'_{k,j}))]
        δ(n_{k,j}) = [-UB̄(S^in(n_{k,j})), UB̄(S^in(n'_{k,j}))]
        Opt. 3:
        if LB(δ^in(n_{k,j})) ≥ 0 then
            δ(n_{k,j}) = [0, UB̄(δ^in(n_{k,j}))]
        else if UB(δ^in(n_{k,j})) ≤ 0 then
            δ(n_{k,j}) = [max(LB̲(δ^in(n_{k,j})), -UB̄(S^in(n_{k,j}))), 0]
        else
            δ(n_{k,j}) = [max(LB̲(δ^in(n_{k,j})), -UB̄(S^in(n_{k,j}))),
                          UB̄(S^in(n'_{k,j}))]
```

Algorithm 2: Over-approximating ReLU activation function.

in $n'_{k,j}$ is also *non-linear*, then the difference is $[0, \overline{\text{UB}}(S^{in}(n'_{k,j}))] - [0, \overline{\text{UB}}(S^{in}(n_{k,j}))]$, which is $[-\overline{\text{UB}}(S^{in}(n_{k,j})), \overline{\text{UB}}(S^{in}(n'_{k,j}))]$.

4.3 Optimization

The most important optimization we make in the forward pass when computing $\delta(n_{k,j})$ is in shifting from bounding the equation

$$\text{ReLU}\big(S^{in}(n'_{k,j})\big) - \text{ReLU}\big(S^{in}(n_{k,j})\big)$$

to bounding one the following *equivalent* equations

$$\text{ReLU}\big(S^{in}(n_{k,j}) + \delta^{in}(n_{k,j})\big) - \text{ReLU}\big(S^{in}(n_{k,j})\big) \quad (1)$$

$$\text{ReLU}\big(S^{in}(n'_{k,j})\big) - \text{ReLU}\big(S^{in}(n'_{k,j}) - \delta^{in}(n_{k,j})\big) \quad (2)$$

Equation 1 says that for any concrete $n_{k,j} \in S^{in}(n_{k,j})$, the most $n'_{k,j}$ can change is bounded by $\delta^{in}(n_{k,j})$, and similarly for Equation 2. As shown in Algorithm 2, we have identified three optimization opportunities, marked as *Opt.1-3*. We note that, even though we widen the difference interval in some of these cases, the interval is almost always tighter than if we subtract the bounds of $n'_{k,j}$ and $n_{k,j}$, even when they are symbolic. Below, we give formal proofs for most of the bounds, and the remaining proofs can be found in the appendix of our arXiv paper [34].

4.3.1 Opt. 1: $n_{k,j}$ is active, $n'_{k,j}$ is non-linear. Using Equation 1, we can potentially tighten the lower bound. To reduce the notation complexity, we rewrite Equation 1 as the function:

$$\Delta(n, d) = \text{ReLU}\,(n + d) - \text{ReLU}\,(n) \quad (3)$$

$$= \text{ReLU}\,(n + d) - n \quad (4)$$

where $n \in S^{in}(n_{k,j})$ and $d \in \delta^{in}(n_{k,j})$, and we can simplify from Equation 3 to 4 because $n_{k,j}$ is active. Now, computing $\delta(n_{k,j})$ amounts to finding the upper and lower bounds on $\Delta(n, d)$.

Observe that if $n + d \geq 0$ then $\Delta(n, d) = d$ because $\text{ReLU}(n + d)$ simplifies to $n+d$, and the like terms cancel. Otherwise $\Delta(n, d) = -n$ because $\text{ReLU}(n + d) = 0$. Observing that $n + d \geq 0 = d \geq -n$, this means $\Delta(n, d)$ is equivalent to:

$$\Delta(n, d) = \begin{cases} d & d \geq -n \\ -n & d < -n \end{cases} = max(-n, d)$$

$max()$ is well-defined for intervals. Specifically, for two intervals $[a, b], [c, d]$, we have:

$$max([a, b], [c, d]) = [max(a, c), max(b, d)].$$

Now plugging in the the bounds of n and d we get:

$$\Delta(n, d) = max\Big([-\overline{\text{UB}}(S^{in}(n_{k,j})), -\underline{\text{LB}}(S^{in}(n_{k,j}))],$$
$$[\underline{\text{LB}}(\delta^{in}(n_{k,j})), \overline{\text{UB}}(\delta^{in}(n_{k,j}))]\Big)$$
$$= [max\Big(-\overline{\text{UB}}(S^{in}(n_{k,j})), \underline{\text{LB}}(\delta^{in}(n_{k,j}))\Big),$$
$$max\Big(-\underline{\text{LB}}(S^{in}(n_{k,j})), \overline{\text{UB}}(\delta^{in}(n_{k,j}))\Big)]$$

4.3.2 Opt. 2: $n_{k,j}$ is non-linear, $n'_{k,j}$ is active. Using Equation 2, we can tighten the upper bound. We first rewrite Equation 2 as:

$$\Delta'(n', d) = n' - \text{ReLU}\,(n' - d)\,. \quad (5)$$

Just like Equation 4, Equation 5 can be broken into two cases based on the inequality $n' - d \geq 0 = n' \geq d$, which gives us the piece-wise equation:

$$\Delta'(n', d) = \begin{cases} d & n' \geq d \\ n' & n' < d \end{cases} = min(n', d)$$

For two intervals $[a, b], [c, d]$, we have

$$min([a, b], [c, d]) = [min(a, c), min(b, d)].$$

Replacing n' and d with the proper bounds gives us the $min()$ function in Algorithm 2.

4.3.3 Opt. 3: both $n_{k,j}$ and $n'_{k,j}$ are non-linear. We consider three cases. First, let $\underline{\text{LB}}(\delta^{in}(n_{k,j})) \geq 0$. This means that $n'_{k,j} \geq n_{k,j}$ before applying ReLU, and then we can derive 0 as a lower bound as follows:

$$n'_{k,j} \geq n_{k,j} \implies \text{ReLU}(n'_{k,j}) \geq \text{ReLU}(n_{k,j})$$
$$= \text{ReLU}(n'_{k,j}) - \text{ReLU}(n_{k,j}) \geq 0$$

In addition, $\overline{\text{UB}}(\delta^{in}(n_{k,j}))$ can be derived as an upper bound[1] from Equation 2 [34]. This is the case in our motivating example, so $\delta(n_{1,1}) = [0, 1.1]$. Second, we consider $\overline{\text{UB}}(\delta^{in}(n_{k,j})) \leq 0$. This means $n'_{k,j} \leq n_{k,j}$ before ReLU, which allows us to derive an upper bound of 0 in a symmetric manner to the first case. The lower bound shown in Algorithm 2 can be derived from Equation 1 [34]. In the third case where $\underline{\text{LB}}(\delta^{in}(n_{k,j})) < 0 < \overline{\text{UB}}(\delta^{in}(n_{k,j}))$, the lower bound and the upper bound shown in Algorithm 2 can be derived from Equations 1 and 2, respectively [34] (also see Footnote 1).

4.4 On the Correctness

The operations of the affine transformation are soundly defined for intervals as described in Section 3.2. For the ReLU transformation, we give formal explanations to show that they over-approximate (see also [34] for proofs). Since composing over-approximations also results in an over-approximation, the forward analysis is itself a sound over-approximation.

5 GRADIENT BASED REFINEMENT

After performing the forward pass, the computed difference may not be tight enough to prove the desired property. In this section, we discuss how we can improve the analysis result.

5.1 Splitting Input Intervals

As mentioned in Section 3, a common way to improve the result of interval analysis is dividing an input interval into disjoint sub-intervals, and then performing interval analysis on the sub-intervals. After unioning the output intervals, the result will be *at least* as good as the original result [30]. Prior work [45] also shows a nice property of ReLU networks: after a finite number of such splits, the result of the interval analysis can be arbitrarily accurate.

However, determining the optimal order of refinement is difficult and, so far, the best algorithms are all heuristic based. For example, the method used in ReluVal chooses to split the input interval that has *the most influence* on the output value. The intuition is that

[1] In fact, the tighter upper bound $min(\overline{\text{UB}}(\delta^{in}(n_{k,j})), \overline{\text{UB}}(S^{in}(n'_{k,j})))$ can be derived, however we had not yet proved this at the time of submission.

Input: network f, mask matrix R
Result: gradient ∇
// Initialize to edge weights in the output layer
$\underline{UB}(\nabla_1) = \underline{LB}(\nabla_1) = 1$
for $k = l - 1..2$ **do**
  $\nabla^{New} = \{[0, 0], \ldots, [0, 0]\}$
  for $j = 1..layerSize[k]$ **do**
    // Perform ReLU for node $n_{k,j}$
    if $R[k][j] == [0, 0]$ **then**
      $\underline{LB}(\nabla_j) = \underline{UB}(\nabla_j) = 0$
    else if $R[k][j] = [0, 1]$ **then**
      $\underline{LB}(\nabla_j) = min(0, \underline{LB}(\nabla_j))$
      $\underline{UB}(\nabla_j) = max(0, \underline{UB}(\nabla_j))$
    // Multiply by weights of incoming edges to node $n_{k,j}$
    for $i = 1..layerSize[k-1]$ **do**
      if $W_k[i, j] \geq 0$ **then**
        $\underline{UB}(\nabla_i^{New})$ += $W_k[i, j] * \underline{UB}(\nabla_j)$
        $\underline{LB}(\nabla_i^{New})$ += $W_k[i, j] * \underline{LB}(\nabla_j)$
      else
        $\underline{UB}(\nabla_i^{New})$ += $W_k[i, j] * \underline{LB}(\nabla_j)$
        $\underline{LB}(\nabla_i^{New})$ += $W_k[i, j] * \underline{UB}(\nabla_j)$
    end
  end
  $\nabla = \nabla^{NEW}$
end
return ∇

Algorithm 3: Computing the gradient of a network.

splitting such an input interval reduces the approximation error of the output interval.

However, the approach is not suitable in our case because we focus on the difference between the two networks: the input interval with the most influence on the absolute value of the output may not have the most influence on the output difference. To account for this difference, we develop a method for determining which input interval to split.

5.2 The Refinement Algorithm

Our idea is to compute the difference of the gradients for the two networks, denoted ∇^{δ}. Toward this end, we compute the gradient of the first network (∇) and the gradient of the second network (∇'). Then, we use them to compute the difference ∇^{δ}.

Formally, $\nabla = \{\partial f/\partial x_1, \ldots, \partial f/\partial x_n\}$ is a vector whose i-th element, $\nabla_i = \partial f/\partial x_i$, is the partial derivative of the output f with respect to the input x_i. Similarly, $\nabla' = \{\partial f'/\partial x_1, \ldots, \partial f'/\partial x_n\}$. The difference is $\nabla^{\delta} = \{\partial(f'-f)/\partial x_1, \ldots, \partial(f'-f)/\partial x_n\} = \nabla' - \nabla$. That is, the derivative of a difference of functions is the difference of their derivatives.

During interval analysis, the accurate gradient is difficult to compute. Therefore, we compute the approximated gradient, where each element ∇_i is represented by a concrete interval.

Algorithm 3 shows our gradient computation procedure. In addition to the network, which may be either f or f', it also takes the mask matrix R as input. Recall that both $R[k][j]$ and $R'[k][j]$ have been computed by Algorithm 2 during the forward pass. $R[k][i]$ may be $[0, 0]$, $[1, 1]$, or $[0, 1]$, indicating if the ReLU in $n_{k,j}$ is *always de-activated*, *always activated*, or *non-linear*, respectively. It can be understood as the gradient interval of the ReLU.

The gradient computation is performed backwardly beginning at the output layer and then moving through the previous layers. In each layer, the computation has two steps. First we apply ReLU to the current gradient and update the upper and lower bounds of the gradient if needed. Then, we scale the gradient interval by the weights of the previous layer.

After computing ∇ and ∇' by invoking Algorithm 3 on f and f', respectively, we compute the gradient difference ∇^{δ}.

Then, we use the gradient difference to determine which input has the most influence on the output difference. Note that the gradient itself is not sufficient to act as an indicator of influence. For example, while an input's gradient may be large, but the width of its input interval is small, splitting it will not have much impact on the output interval. Thus, we split the input interval with the maximum *smear* value [20, 21]. The smear value of an input x_i is defined as the width of its input interval $|\overline{UB}(x_i) - \underline{LB}(x_i)|$ scaled by the upper bound of its corresponding gradient difference $\overline{UB}(\nabla^{\delta})_i$.

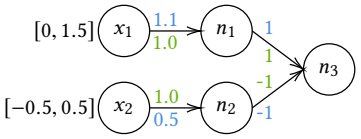

Figure 7: Example for backward refinement.

5.3 An Example

We now walk through the gradient computation in Algorithm 3 for the example in Figure 7, where blue weights are for network f, and green weights are for network f'. We focus on the gradient of f first. After performing the forward pass, we know that n_1 is in a linear state, i.e., $R[1][0] = [1, 1]$, and n_2 is in a non-linear state, i.e., $R[1][1] = [0, 1]$.

We initialize the gradient to the weights of the final layer; that is, $\underline{UB}(\nabla_1) = \underline{LB}(\nabla_1) = 1$ and $\underline{UB}(\nabla_2) = \underline{LB}(\nabla_2) = -1$. Next, we apply ReLU. Since n_1 is in the *always activated* mode, we leave its gradient unchanged. However, n_2 is in the *non-linear* mode, meaning the gradient could be 0, and hence we must ensure that 0 is in the gradient interval. We update $\underline{LB}(\nabla_2) = 0$. Then, we scale the gradient interval by weights of the incoming edges, which gives us the gradient intervals for input variables: $\nabla_1 = [1.1, 1.1]$ for x_1 and $\nabla_2 = [-0.5, 0]$ for x_2.

Here, we point out a problem with ReluVal's refinement method. It would compute the smear value of x_1 and x_2 to be $|(1.5-0)*1.1| = 1.65$ and $|(0.5 - (-0.5)) * -0.5| = 0.5$, respectively, which means it would split on x_1. However, this is not appropriate for differential verification, since the two networks differ the most in the weights of the outgoing edge of x_2.

Our method, instead, would compute the gradient difference $\nabla^{\delta} = \nabla - \nabla'$. Therefore, we have $\nabla_1^{\delta} = [-0.1, -0.1]$ for x_1 and $\nabla_2^{\delta} = [-0.5, 1]$ for x_2. Based on the new smear values, we would choose to split the input interval of x_2.

6 EXPERIMENTS

We have implemented ReluDiff and compared it experimentally with state-of-the-art neural network verification tools. Like ReluVal, ReluDiff is written in C using OpenBLAS [52] as the library for matrix multiplications. We also note that we implement outward-rounding to soundly handle floating point arithmetic. Symbolic interval arithmetic is implemented using matrix multiplication.

ReluDiff takes two networks f and f' together with a small ϵ and input region X as input, and then decides whether $\forall x \in X. |f'(x) - f(x)| < \epsilon$ for the target label's value. Since ReluDiff is the only tool currently available for differential verification of neural networks, to facilitate the experimental comparison with existing tools, we developed a tool to merge f and f' into a combined network f'', as shown in Figure 4, before feeding f'' to these existing tools as input.

6.1 Benchmarks

Our benchmarks are 49 feed-forward neural networks from three applications: aircraft collision detection, image recognition, and human activity recognition. We produce f' by truncating each network's weights from 32-bit floats to 16-bit floats.

6.1.1 ACAS Xu [17]. ACAS Xu is a set of 45 neural networks commonly used in evaluating neural network verification tools. They are designed to be used in an aircraft to advise the pilot of what action to take in the presence of an intruder aircraft. They each take five inputs: distance between self and the intruder, angle of self relative to the intruder, angle of intruder relative to self, speed of self, and speed of intruder. They output a score in the range $[-0.5, 0.5]$ for five different actions: clear-of-conflict, weak left, weak right, strong left, and strong right. The action with the minimum score is the action advised. In addition to the input and output layers, each network has 6 hidden layers of 50 neurons each, for a total of 300 neurons. For differential verification, we use the same input ranges as in [18, 45] as our regions of interest.

6.1.2 MNIST [24]. MNIST is a data set of labeled images of handwritten digits that are often used as a benchmark to test image classifiers. The images are 28x28 pixels, and each pixel has a grayscale value in the range $[0, 255]$. Neural networks trained on this data set take in 784 inputs (one per pixel) each in the range $[0, 255]$, and output 10 scores, typically in the range of $[-10, 10]$ for our networks, for each of the 10 digits. The digit with the highest score is the chosen classification. We use three neural networks trained on the MNIST data set with architectures of 3x100, 2x512, and 4x1024; that is, the networks have 3, 2, and 4 hidden layers, with layer size 100, 512, and 1024 neurons, respectively. Thus, in addition to the input and output layers, these networks have 300, 1024, and 4096 hidden neurons, respectively. Empirical analysis shows that each network has > 95% accuracy on hold-out test data.

6.1.3 Human Activity Recognition (HAR) [1]. HAR is a labeled data set used to train models to recognize specific human activities based on input from a smartphone's accelerometer and gyroscope. Input examples in this data set are labeled with one of six activities: walking, walking upstairs, walking downstairs, sitting, standing, and laying down. The input data for the model are statistics computed from a smartphone's accelerometer and gyroscope sensor, such as mean, median, min, max, etc. In total, 561 input statistics are computed from these two sensors. Inputs to the network are normalized to be in the range of $[-1, 1]$. We use a network trained on this data set with an architecture of 1x500, meaning there is a hidden layer with 500 neurons. The network takes the 561 inputs, and produces a score in the range of $[-20, 20]$ for each of the 6 outputs, one per activity. The output with the maximum score is the classification.

Table 1: Statistics of the benchmarks: The total number of verification problems is 842.

Name	# NN's	in each network				# input region	out ϵ
		# in	# out	# hidden	# neurons		
ACAS-ϕ_1-ϕ_2	45	5	5	6 * 50	300	1	0.05
ACAS-ϕ_3	42	5	5	6 * 50	300	1	0.05
ACAS-ϕ_4	42	5	5	6 * 50	300	1	0.05
ACAS-ϕ_5-ϕ_{13}	1	5	5	6 * 50	300	1	0.05
ACAS-ϕ_{14}	2	5	5	6 * 50	300	1	0.05
ACAS-ϕ_{15}	2	5	5	6 * 50	300	1	0.05
MNIST 3x100	1	784	10	3 * 100	300	200	1
MNIST 2x512	1	784	10	2 * 512	1,024	200	1
MNIST 4x1024	1	784	10	4 * 1024	4,096	200	1
HAR 1x500	1	561	6	1 * 500	500	100	0.25

Table 1 shows the statistics of these benchmarks, including the number of input neurons, the number of output neurons, the number of hidden layers, as well as the total number of neurons in these hidden layers. The last two columns list the experimental parameters we used, namely the number of "regions of interest" in the verification problems and the output ϵ we attempt to verify.

6.2 Experimental Evaluation

We want to answer the following research questions:

(1) Is ReluDiff more efficient than existing methods in differential verification of neural networks in that it can both verify properties faster and verify more properties in general?
(2) Is ReluDiff more accurate than existing methods in the forward pass?

Toward this end, we directly compared ReluDiff to two state-of-the-art verification tools: ReluVal [45] and DeepPoly [39]. Both are designed to formally verify the absence of adversarial examples.

A comparison with Reluplex [18] was not possible since it does not support affine hidden layers, which are necessary for analyzing the combined network $f''(x)$ as shown in Figure 4, however we note that ReluVal previously has been shown to significantly outperform Reluplex on all ACAS Xu benchmarks [45]. DeepPoly, a followup of AI2 [10], has also been shown to outperform AI2.

We ran all experiments on a Linux server running Ubuntu 16.04, an Intel Xeon CPU E5-2620, and 124 GB memory. Timeout is set at 30 minutes for each verification problem. When available, we enable parallelization in all tools and configure them to allow up to 10 threads at a time.

6.3 Results

To evaluate efficiency and accuracy, we truncate each network's weights from 32-bit floats to 16-bit floats, and attempt to verify the ϵ shown in Table 1. We measure the number of properties verified and the execution time to verify each property.

6.3.1 ACAS Xu. The results for ACAS Xu are shown in Tables 2 and 3. In Table 2, columns 1 and 2 show the input property used, and the number of networks we verified the property on, which are taken from [18, 45]. Columns 3-5 show the number of neural networks for which the property was verified and undetermined for each tool. Undetermined means that either the tool reported it could not verify the problem (due to over-approximation), or the timeout of 30 minutes was reached.

Table 2: Accuracy of comparison of the three tools on ACAS.

Benchmark	Verif. problems	ReluDiff (new)		ReluVal		DeepPoly	
		proved	undet.	proved	undet.	proved	undet.
ACAS ϕ_1-ϕ_2	45	28	17	7	38	0	45
ACAS ϕ_3	42	42	0	24	18	6	36
ACAS ϕ_4	42	42	0	34	8	2	40
ACAS ϕ_5	1	1	0	0	1	0	1
ACAS ϕ_6	1	1	0	1	0	0	1
ACAS ϕ_7	1	0	1	0	1	0	1
ACAS ϕ_8	1	0	1	0	1	0	1
ACAS ϕ_9	1	1	0	0	1	0	1
ACAS ϕ_{10}	1	1	0	1	0	0	1
ACAS ϕ_{11}	1	1	0	0	1	0	1
ACAS ϕ_{12}	1	0	1	0	1	0	1
ACAS ϕ_{13}	1	1	0	1	0	0	1
ACAS ϕ_{14}	2	2	0	0	2	0	2
ACAS ϕ_{15}	2	2	0	0	2	0	2
Total	142	123	19	69	73	8	134

Table 3: Efficiency of ReluDiff vs. ReluVal on ACAS Xu.

Benchmark	Verif. problems	Total Time (s)		
		ReluDiff (new)	ReluVal	Avg. Speedup
ACAS ϕ_{1--2}	45	40595.6	69167.5	1.7
ACAS ϕ_3	42	175.4	38414.2	$\geq$ 219
ACAS ϕ_4	42	46.8	22159.2	$\geq$ 473.4
ACAS ϕ_5	1	9.6	1800.0	$\geq$ 187.5
ACAS ϕ_6	1	11.0	50.8	4.6
ACAS ϕ_7	1	1800.0	1800.0	1.00
ACAS ϕ_8	1	1800.0	1800.0	1.00
ACAS ϕ_9	1	52.3	1800.0	$\geq$ 34.4
ACAS ϕ_{10}	1	31.0	53.3	1.6
ACAS ϕ_{11}	1	10.2	1800.0	$\geq$ 177.3
ACAS ϕ_{12}	1	1800.0	1800.0	1.0
ACAS ϕ_{13}	1	157.9	999.2	6.3
ACAS ϕ_{14}	2	859.0	3600.0	$\geq$ 4.2
ACAS ϕ_{15}	2	453.8	3600.0	$\geq$ 7.9

In Table 3, Columns 3-4 show the time taken by ReluDiff and ReluVal for all problems verified. The last column shows the time of ReluVal divided by the time of ReluDiff. For timeouts, we add 30 minutes to the total, which is why we display that the speedup is *greater than or equal to X* for some properties. We omit the timing data for DeepPoly since it cannot verify most properties.

These results emphasize the improvement that ReluDiff can obtain in both speed and accuracy. It achieves orders of magnitude speedups over ReluVal. For example, ReluDiff finishes the 42 networks for ϕ_4 in 48.6 seconds, whereas ReluVal takes *at least* more than 6 hours. Overall ReluDiff verifies 54 more problems for which ReluVal times out, and 115 more problems for which DeepPoly is too inaccurate to verify.

To understand why ReluDiff performs better, we plot the distribution of the depth at which each sub-interval was finally able to be verified for ϕ_4, shown in Figure 8. We can see that ReluDiff consistently verifies sub-intervals at much shallower split depths. We point out that the number of sub problems grows exponentially as the split depth increases. Indeed, even though the difference between the average depths does not seem large (about 14 for ReluDiff and 29 for ReluVal), ReluVal had to verify > 66 million sub-intervals for ϕ_4, whereas ReluDiff only had to verify 66K.

6.3.2 MNIST. While in ACAS Xu the input region to verify is defined by the property, for MNIST, we must generate the input region ourselves. We generate 200 input regions for MNIST using two methods. The first method is based on global perturbation [39]. We take 100 test images, and for each one, we allow each of the pixels to be perturbed by +/-3 gray scale units. The second method is based on targeted pixel perturbation [12, 13]. We take the same 100 test images, and for each one, we set the range of 3 random pixels to $[0, 255]$, while the remaining 781 remain fixed.

We can again see in Tables 4 and 5 that ReluDiff is significantly more accurate and efficient than both ReluVal and DeepPoly. Both competing techniques struggle to handle global perturbations even on the small 3x100 network, let alone the larger 2x512 and 4x1024 networks. On the other hand, ReluDiff can easily handle both the 3x100 and 2x512 networks, achieving at least 3 orders of magnitude speedup on these networks. We also see a three orders of magnitude speedup on the two largest networks for our targeted-pixel perturbation experiments.

Even though ReluDiff begins to reach its limit in the global perturbation experiment on the largest 4x1024 network, we point out that ReluDiff is significantly outperforming both DeepPoly and ReluVal in the accuracy of their forward passes. Figure 9 compares the output bound verified on the *first, single* forward pass of each technique. The comparison is presented as a scatter plot, where the x-axis is the bound verified by ReluDiff, and the y-axis is that of the competing technique.

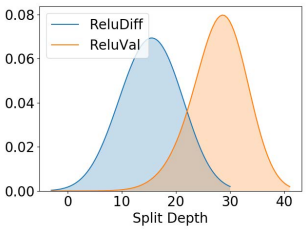

Figure 8: ϕ_4 max depth distribution.

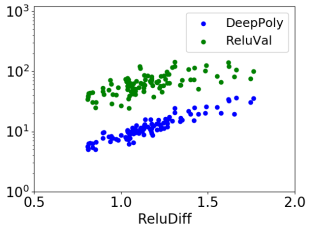

Figure 9: Δ-interval on MNIST 4x1024.

Table 4: Accuracy comparison of the three tools on MNIST.

Benchmark	Verif. problems	ReluDiff (new)		ReluVal		DeepPoly	
		proved	undet.	proved	undet.	proved	undet.
3x100-global	100	100	0	47	53	34	66
2x512-global	100	100	0	0	100	0	100
4x1024-global	100	22	78	0	100	0	100
3x100-3-pixel	100	100	0	100	0	100	0
2x512-3-pixel	100	100	0	100	0	80	20
4x1024-3-pixel	100	100	0	97	3	100	0

Table 5: Efficiency comparison of the three tools on MNIST.

Benchmark	Verif. problems	Total Time (s)		
		ReluDiff (new)	ReluVal	DeepPoly
3x100-global	100	29.47	95458.32	118823.09
2x512-global	100	77.83	180000.00	180000.0
4x1024-global	100	141604.53	180000.00	180000.0
3x100-3-pixel	100	23.90	32.60	163.75
2x512-3-pixel	100	79.24	715.16	37674.40
4x1024-3-pixel	100	296.59	92100.10	49042.98

Table 6: Accuracy comparison of the three tools on HAR.

Benchmark	Verif. problems	ReluDiff (new)		ReluVal		DeepPoly	
		proved	undet.	proved	undet.	proved	undet.
1x500	100	100	0	0	100	0	100

Table 7: Efficiency comparison of the three tools on HAR.

Benchmark	Verif. problems	Total Time (s)		
		ReluDiff (new)	ReluVal	DeepPoly
1x500	100	28.79	180000.00	180000.00

The graph shows that ReluDiff is nearly two orders of magnitude more accurate than ReluVal and one order of magnitude more than DeepPoly. The improvement over DeepPoly especially emphasizes the promise of ReluDiff's approach. This is because ReluDiff is already outperforming DeepPoly, yet it uses a simpler *concretization* approach during the forward pass, whereas DeepPoly uses a more sophisticated *linear relaxation*. We believe that ReluDiff can be extended to use more accurate techniques such as linear relaxation which would further improve the accuracy, however we leave this as future work.

6.3.3 HAR. For HAR, we also created our verification problems using input perturbation. We take 100 concrete test inputs, and for each one, we allow a global perturbation of +/-0.1. The results are summarized in Tables 6 and 7. Again, the experimental comparison shows that ReluDiff is significantly more accurate and efficient.

6.4 Threats to Validity

Our method is designed for verifying neural networks typically found in control applications, where the number of input signals is not large. In this context, dividing the input region turns out to be a very effective way of increasing the accuracy of interval analysis. However, neural networks in different application domains may have different characteristics. Therefore, it remains an open problem whether bi-section of individual input intervals is always an effective way of performing refinement.

Our method is designed for feed-forward ReLU networks. Although there is no significant technical hurdle for it to be extended to convolutional neural networks or other activation functions, such as sigmoid, tanh and max-pool as shown recently by Singh et al. [39], we have not evaluated the effectiveness. Specifically, linear relaxation can be used to handle these features when it comes to approximating non-linear behavior. While we use concretization in ReluDiff, extending it with linear relaxation is possible [44]. However, we leave these extensions for future work.

7 RELATED WORK

While there is a large and growing body of work on detecting adversarial examples for neural networks, they are typically based on heuristic search or other dynamic analysis techniques such as testing [4, 26, 35, 41, 43, 47]. Although they are effective in finding security vulnerabilities and violations of other critical properties, we consider them as being orthogonal to formal verification. The reason is because these techniques are geared toward finding violations, as opposed to proving the absence of violations.

Early work on formal verification of deep neural networks relies on using SMT solvers [8, 16], or SMT solving algorithms [18, 19] designed for efficiently reasoning about constraints from the ReLU activation function. Along this line, a state-of-the-art tool is Reluplex [18]. In theory, these SMT solver based techniques can solve the neural network verification problem in a sound and complete fashion, i.e., returning a proof if and only if the network satisfies the property. In practice, however, their scalability is often limited and they may run out of time for larger networks.

Another line of work on verification of deep neural networks is based on interval analysis, which can be more scalable than SMT solver based techniques [45]. They compute conservative bounds on the value ranges of the neurons and output signals for an input region of interest. They also exploit the fact that neural networks are Lipschitz continuous [37] to ensure that the interval analysis results are sound. ReluVal [45] and DeepPoly [39] are two representatives, among other similar tools [9, 10, 29, 40, 44].

In addition to formal verification, there are techniques for evaluating and certifying the robustness of neural networks [2, 4, 7, 46] or certified defense against adversarial examples [36, 48]. However, neither they nor the existing verification techniques were designed for *differential verification* of two closely related neural networks, which is the focus of this paper. As shown by the examples in Section 2 and the experimental results in Section 6, directly applying these techniques to differential verification is often extremely inefficient. In contrast, our method is designed specifically for solving the differential verification problem efficiently.

At a higher level, our method relies on symbolic interval analysis, which can be viewed as a specific form of abstract interpretation [5]. While the abstract interpretation framework allows approximations to be performed in a more general way, e.g., using relational abstract domains [28] such as the octagon [27] and polyhedral [6] domains, so far, it has not be adequately explored. We plan to explore the use of these abstract domains as part of the future work.

Finally, the term *differential verification* has been used in the context of verifying a new version of a program with respect to a previous version, which is treated as an "oracle" [23]. In a sense, the truncated network is a "new version" of the original network, and the original network can be thought of as an oracle.

8 CONCLUSION

We have presented a new method, named ReluDiff, for differential verification of two closely related neural networks. It is capable of formally proving the accuracy of a compressed network with respect to the original network. Internally, ReluDiff relies on symbolic interval analysis to more accurately compute and propagate differences in the values of neurons of the two networks from the input to the output, and then relies on the gradient difference to more accurately compute the refinement. Our experimental comparison of ReluDiff with state-of-the-art formal verification techniques shows that it can often achieve two orders of magnitude speedup and produce many more proofs.

ACKNOWLEDGMENTS

This work was partially funded by the U.S. Office of Naval Research (ONR) under the grant N00014-17-1-2896.

REFERENCES

[1] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge L. Reyes-Ortiz. 2013. A Public Domain Dataset for Human Activity Recognition Using Smartphones. *21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning* (2013).

[2] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Annual Conference on Neural Information Processing Systems*. 2613–2621.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[4] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy*. 39–57.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 238–252.

[6] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 84–96.

[7] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A. Mann, and Pushmeet Kohli. 2018. A Dual Approach to Scalable Verification of Deep Networks. In *International Conference on Uncertainty in Artificial Intelligence*. 550–559.

[8] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*. 269–286.

[9] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. 2019. DL2: Training and Querying Neural Networks with Logic. In *International Conference on Machine Learning*. 1931–1941.

[10] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *IEEE Symposium on Security and Privacy*. 3–18.

[11] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

[12] Divya Gopinath, Guy Katz, Corina S. Pasareanu, and Clark W. Barrett. 2018. DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. 3–19.

[13] Divya Gopinath, Corina S. Pasareanu, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. 2019. Symbolic execution for attribution and attack synthesis in neural networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 282–283.

[14] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations*.

[15] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conference on Computer Vision*. 815–832.

[16] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *International Conference on Computer Aided Verification*. 3–29.

[17] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2018. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *CoRR* abs/1810.04240 (2018). arXiv:1810.04240 http://arxiv.org/abs/1810.04240

[18] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *International Conference on Computer Aided Verification*. 97–117.

[19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *International Conference on Computer Aided Verification*. 443–452.

[20] R Baker Kearfott. 2013. *Rigorous global search: continuous problems*. Vol. 13. Springer Science & Business Media.

[21] R Baker Kearfott and Manuel Novoa III. 1990. Algorithm 681: INTBIS, a portable interval Newton/bisection package. *ACM Transactions on Mathematical Software (TOMS)* 16, 2 (1990), 152–157.

[22] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *International Conference on Learning Representations*.

[23] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 345âĂŞ355. https://doi.org/10.1145/2491411.2491452

[24] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[25] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *IEEE/ACM International Conference On Automated Software Engineering*. ACM, 120–131.

[26] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 175–186.

[27] A. Miné. 2001. The Octagon Abstract Domain. In *Analysis, Slicing, and Transformation*. 310–319.

[28] A. Miné. 2004. *Weakly Relational Numerical Abstract Domains*. Ph.D. Thesis. Computer Science Department, ENS, France.

[29] Matthew Mirman, Timon Gehr, and Martin T. Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning*. 3575–3583.

[30] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. 2009. *Introduction to interval analysis*. Vol. 110. Siam.

[31] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition*. 2574–2582.

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

[33] Augustus Odena and Ian Goodfellow. 2018. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875* (2018).

[34] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: Differential Verification of Deep Neural Networks. arXiv:cs.LG/2001.03662 https://arxiv.org/abs/2001.03662

[35] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *ACM symposium on Operating Systems Principles*. 1–18.

[36] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Certified Defenses against Adversarial Examples. In *International Conference on Learning Representations*.

[37] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *International Joint Conference on Artificial Intelligence*. 2651–2659.

[38] Vikash Sehwag, Shiqi Wang, Prateek Mittal, and Suman Jana. 2019. Towards Compact and Robust Deep Neural Networks. *CoRR* abs/1906.06110 (2019). arXiv:1906.06110 http://arxiv.org/abs/1906.06110

[39] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (2019), 41:1–41:30.

[40] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. Boosting Robustness Certification of Neural Networks. In *International Conference on Learning Representations*.

[41] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 109–119.

[42] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[43] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering*. 303–314.

[44] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient Formal Safety Analysis of Neural Networks. In *Annual Conference on Neural Information Processing Systems*. 6369–6379.

[45] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *USENIX Security Symposium*. 1599–1614.

[46] Tsui-Wei Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane S. Boning, and Inderjit S. Dhillon. 2018. Towards Fast Computation of Certified Robustness for ReLU Networks. In *International Conference on Machine Learning*. 5273–5282.

[47] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *International Conference*

on Tools and Algorithms for Construction and Analysis of Systems. 408–426.

[48] Eric Wong and J. Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *International Conference on Machine Learning*. 5283–5292.

[49] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[50] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. Diffchaser: Detecting disagreements for deep neural networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 5772–5778.

[51] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. In *Network and Distributed System Security Symposium*.

[52] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012*. 684–691.