DelBugV: Delta-Debugging Neural Network Verifiers

Raya Elsaleh and Guy Katz
The Hebrew University of Jerusalem, Jerusalem, Israel
Email: {rayae,guykatz}@cs.huji.ac.il

***Abstract*—Deep neural networks (DNNs) are becoming a key component in diverse systems across the board. However, despite their success, they often err miserably; and this has triggered significant interest in formally verifying them. Unfortunately, DNN verifiers are intricate tools, and are themselves susceptible to soundness bugs. Due to the complexity of DNN verifiers, as well as the sizes of the DNNs being verified, debugging such errors is a daunting task. Here, we present a novel tool, named DELBUGV, that uses automated *delta debugging* techniques on DNN verifiers. Given a malfunctioning DNN verifier and a correct verifier as a point of reference (or, in some cases, just a single, malfunctioning verifier), DELBUGV can produce much simpler DNN verification instances that still trigger undesired behavior — greatly facilitating the task of debugging the faulty verifier. Our tool is modular and extensible, and can easily be enhanced with additional network simplification methods and strategies. For evaluation purposes, we ran DELBUGV on 4 DNN verification engines, which were observed to produce incorrect results at the 2021 neural network verification competition (VNN-COMP'21). We were able to simplify many of the verification queries that trigger these faulty behaviors, by as much as 99%. We regard our work as a step towards the ultimate goal of producing reliable and trustworthy DNN-based software.**

I. INTRODUCTION

Deep neural networks (DNNs) [21] are software artifacts that are generated automatically, through the generalization of a finite set of examples. These artifacts have been shown to outdo manually crafted software in a variety of key domains, such as natural language processing [19], [25], [37], image recognition [25], [59], protein folding [26], [41], and many others. However, this impressive success comes at a price: unlike traditional software, DNNs are opaque artifacts, and are incomprehensible to humans. This poses a serious challenge when it comes to certifying, modifying, extending, repairing or reasoning about them [22], [27], [32].

In an effort to address these issues, the formal methods community has taken up an interest in *DNN verification* [27], [30], [45]: automated techniques that can determine whether a DNN satisfies a prescribed specification, and provide a counter-example if it does not. DNN verification technology has been making great strides, and its applicability has been demonstrated in various domains [2], [3], [18], [30], [33]. In fact, this technology has progressed to a point where DNN verifiers themselves have become quite complex, and consequently error-prone; especially as they often perform delicate arithmetic operations that can introduce bugs into the verification process [30]. Thus, it is not surprising that various bugs have been observed in these tools [29]. For example, in the VNN-COMP'21 competition [9], various verifiers have been shown to disagree on the result of multiple verification queries (each query is comprised of a neural network and a property to be checked), or produce incorrect counter-examples — indicating bugs in those verifiers. Moreover, many verifiers are still under development, with new and experimental features being introduced, possibly allowing the introduction of new bugs, as well. An inability to trust the results of DNN verifiers could undermine the benefits of DNN verification technology, and clearly needs to be addressed.

Here, we propose to mitigate this issue by adopting known techniques from related fields (e.g., SMT solving [12]) — specifically, that of *delta debugging*. The idea is to leverage the fact that DNN verification is at a point where many verification tools are available, and to allow engineers to readily compare the results produced by their verification tool to those produced by others, in order to identify and correct bugs. When a verification query that triggers some bug in a verifier is detected, we can initiate an automated process that repeatedly and incrementally *simplifies* the verification query. After each simplification step, we can check that the verifier in question still disagrees with the remaining, *oracle* verifiers, until reaching the simplest verification query that we can find. If this final query is much simpler than the original, it will be that much easier for engineers to debug their tools, eventually improving their overall soundness.

We present a new tool, DELBUGV (**Del**ta de**Bug**ging Neural Network **V**erifiers), that takes as input a verification query, a malfunctioning DNN verifier that errs on the given verification query, and an oracle DNN verifier. Within DELBUGV, we implement a set of operations for simplifying the neural network of the given verification query into a network with fewer layers and fewer neurons. We empirically design a strategy that applies these operations sequentially in an order that produces much simpler verification queries. In some cases, when the malfunctioning DNN verifier produces a faulty counter-example, DELBUGV can run in *single solver* mode – without an oracle verifier, where the query is repeatedly simplified as long as the malfunctioning DNN verifier continues to produce incorrect counter-examples.

For evaluation, we tested DELBUGV on 4 DNN verifiers "suspected" of errors, per the results of VNN-COMP'21 [9]: Marabou [32], NNV [50]–[53], [60], NeuralVerification.jl(NV.jl) [36], and nnenum [7], [8], [50], [51]. We ran DELBUGV on queries where pairs of these verifiers

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_11

disagreed. Our evaluation demonstrates that DELBUGV could reduce the size of the error-triggering queries by an average of 96.8%, and by as much as 99% in some cases, resulting in very simple neural networks. We believe that these results highlight the significant potential of our tool and approach.

The rest of the paper is organized as follows. In Sec. II we provide the necessary background on DNNs and their verification. Next, in Sec. III we describe the design of DELBUGV, focusing on its algorithm and network simplification methods and the strategy we use to apply those methods. The implementation and evaluation of DELBUGV are discussed in Sec. IV. This is followed by a discussion of related work in Sec. V, and we conclude in Sec. VI.

II. BACKGROUND

Neural Networks. A *neural network* is a directed acyclic graph in which the nodes, called neurons, are organized in layers $l^0, l^1, \ldots, l^n$. l^0 is called the input layer, l^n the output layer, and layers $l^1, \ldots, l^{n-1}$ are called hidden layers. Each hidden layer has an associated non-linear *activation function*. In feed-forward networks, which are our subject matter here, neurons in layer l^i have edges connecting them only to neurons in the next layer, layer l^{i+1}.

Each neuron in the network (except the ones in the input layer) has a bias value, and each edge has a weight. The biases and weights belonging to neurons in layer l^i are organized into a vector B^i and a matrix W^i, respectively. The j, j'-th entry of W^i is the weight assigned to the edge out-going from the j'-th neuron in layer l^{i-1} and entering the j-th neuron in layer l^i. For a *fully connected* layer, W^i is a full matrix; whereas for a *convolutional* layer, W^i is very sparse, and has a specific structure (discussed later).

An input to neural network $\mathcal{N}$ is a vector I of values of the neurons in the input layer, and it produces an output vector $\mathcal{N}(I)$ which is the values of the neurons in the output layer. We denote the values of neurons in layer l^i, prior to applying the activation function, by $\mathcal{N}^{l^i}(I)$; and the values after applying the activation function by $\mathcal{N}^{a^i}(I)$. The values of the neurons are evaluated according to the rules:

$$\mathcal{N}^{l^0}(I) = I, \qquad \mathcal{N}^{l^i}(I) = W^i \mathcal{N}^{a^{i-1}}(I) + B^i,$$
$$\mathcal{N}^{a^i}(I) = Act^i(\mathcal{N}^{l^i}(I))$$

where Act^i is the activation function associated with layer l_i.

We define the size of a neural network to be the total number of neurons in the graph (including the neurons in the input and output layers) and denote it by $|\mathcal{N}|$. The automated training (i.e., selection of weights and biases) of neural networks is beyond our scope here; see, e.g., [21].

Fig. 1 depicts a neural network, $\mathcal{N}_e$, with a single input, a single output, and 2 hidden layers with 3 neurons in each. It uses the ReLU activation function, $\text{ReLU}(x) = \max(0, x)$. The bias of each neuron is listed above it, and weights are listed over the edges (zero values are omitted). In matrix representation, the weights and biases are:

$$W^1 = \begin{bmatrix} -5 \\ -0.5 \\ -1 \end{bmatrix}, B^1 = \begin{bmatrix} 10 \\ -2.5 \\ 7 \end{bmatrix}, W^2 = \begin{bmatrix} 0.8 & -1 & -2 \\ 0 & 0.5 & 0 \\ 2 & 0.5 & -1 \end{bmatrix},$$
$$B^2 = \begin{bmatrix} 8 \\ 2 \\ 0 \end{bmatrix}, W^3 = \begin{bmatrix} 0.25 \\ 2 \\ 0.5 \end{bmatrix}^T, B^3 = [0]$$

$\mathcal{N}_e$ is of size 8 (every l^i_j and r^i_j pair in the figure are counted as one neuron; we split them only for visualization purposes), and has 4 layers. The figure also demonstrates an evaluation of the network, for the input $x = 5$. The assignment of each node is listed below it; and we can see that the produced output in this case is $y = 5$.

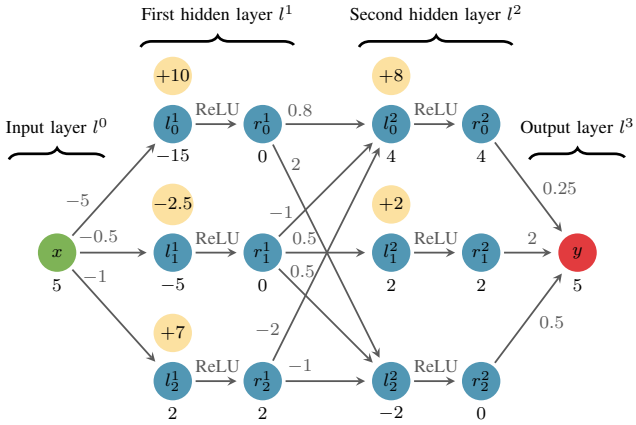

Fig. 1: An example of a neural network, $\mathcal{N}_e$, with ReLU activation functions.

Convolutional Neural Networks. A *convolutional neural network* is a neural network with one or more convolutional layers (typically, these are the first layers of the network). The parameters of a convolutional layer include the height h and width w of images in the input; the kernel size k; the stride size s; the padding size p; the input channels c_i; the output channels c_o; the kernel weights W, given as a tensor of dimensions $(c_o \times c_i \times k \times k)$; and the biases, B, organized in an array of length c_o. We assume for simplicity that the kernel size, padding size, and stride size are equal along all axes, although this is not a limitation of our approach.

The convolutional layer filters its input, which is a $(c_i \times k \times k)$-dimensional matrix, using the above parameters and outputs a multidimensional matrix which represents feature maps. For additional information on how a convolutional layer computes its output, see [21]. Note that convolutional layers are comprised strictly of linear operations.

Neural Network Verification. A *property* $\mathcal{P}$ is a set of constraints on the inputs and outputs of the neural network. These constraints give rise to an input region $I(\mathcal{P})$ and an output region $O(\mathcal{P})$. Verifying $\mathcal{P}$, with respect to some neural

network, entails determining whether there exists an input in $I(\mathcal{P})$ that the neural network maps to an output in $O(\mathcal{P})$ (the SAT case), or not (the UNSAT case). Typically, $\mathcal{P}$ is specified so that $O(\mathcal{P})$ represents *undesirable* behavior, and so an UNSAT result indicates that the system is correct. $\mathcal{P}_e = (5 \leq x \leq 10) \wedge (5 \leq y \leq 10)$ is an example of a property of $\mathcal{N}_e$ in Fig. 1.

A *neural network verifier* takes in a verification query (a neural network and a property) and attempts to automatically verify it. When successful, it returns a SAT or UNSAT answer; otherwise, it can return ERROR, or TIMEOUT. When a neural network verifier returns SAT, it also returns an input that proves the satisfiability of the query. Given a verifier $\mathcal{V}$ and a verification query $Q = (\mathcal{N}, \mathcal{P})$, we denote by $\mathcal{V}(Q) \in \{\texttt{SAT}, \texttt{UNSAT}, \texttt{ERROR}, \texttt{TIMEOUT}\}$ the answer of $\mathcal{V}$ on Q. If $\mathcal{V}(Q) = \texttt{SAT}$, we denote by $\mathcal{V}_w(Q) \in I(\mathcal{P})$ the satisfying assignment (the witness) returned by the verifier.

Continuing with our running example, given a sound neural network verifier $\mathcal{V}_e$ and the verification query $Q_e = (\mathcal{N}_e, \mathcal{P}_e)$, $\mathcal{V}_e(Q_e) = \texttt{SAT}$ and a valid witness is $(\mathcal{V}_e)_w(Q_e) = (5)$, since $\mathcal{N}_e((5)) = (5) \in O(\mathcal{P}_e)$.

Neural network verification is complex, both theoretically and practically [30]; and modern tools apply sophisticated techniques to verify large networks [1]. These techniques are typically theoretically sound, but implementation bugs can cause verifiers to produce incorrect results. These bugs are easier to track and correct if the problem manifests for queries with small networks.

In a situation where two verifiers disagree on the satisfiability of a given query, at least one of them must answer SAT and provide a satisfying assignment. We evaluate the neural network on that assignment, and determine whether it indeed satisfies the property at hand. If so, we conclude that the other verifier, which returned UNSAT, is faulty; otherwise, if the satisfying assignment is incorrect, we determine that the verifier that answered SAT is faulty. The remaining verifier then takes the role of the oracle verifier.

III. DelBugV: Delta-Debugging Verification Queries

A. General Flow

Applying *delta-debugging* techniques means automatically simplifying an input x that triggers a bug in the system into a simpler input, x', that also triggers a bug [40]. x' can often trigger the bug faster, thus reducing overall debugging time; and also trigger fewer code lines that are unrelated to the bug, allowing engineers to more easily identify its root cause. In our setting, given a verification query $Q = (\mathcal{N}, \mathcal{P})$ that triggers a bug in a neural network verifier, we seek to generate another query $Q' = (\mathcal{N}', \mathcal{P})$, with a much smaller (simplified) neural network: $|\mathcal{N}'| < |\mathcal{N}|$. The motivation for focusing on the neural network, and not on the verification conditions, is that common verification conditions are typically already quite simple [56], whereas neural network sizes have a crucial effect on verifier performance [30].

The general delta debugging framework that our tool follows appears as Alg. 1. The inputs to the process are a faulty verifier $\mathcal{V}$, an oracle verifier $\mathcal{V}_O$, and a verification query $Q = (\mathcal{N}, \mathcal{P})$. The algorithm maintains a candidate result neural network $\mathcal{N}_r$ that triggers a bug in $\mathcal{V}$ and make it produce an incorrect answer, and whose size is iteratively decreased. In each iteration, the algorithm invokes Alg. 2 to attempt simplifying $\mathcal{N}_r$. The process terminates when Alg. 2 states that it cannot simplify $\mathcal{N}_r$ any further, or when a timeout limit is exceeded. Finally, it returns the verification query with the smallest $\mathcal{N}_r$ it achieved.

Algorithm 1 *Reduce Verification Query*

Input: $\mathcal{V}$, $\mathcal{V}_O$, $Q = (\mathcal{N}, \mathcal{P})$
// Faulty Verifier, Oracle Verifier, Verification query
Output: Q_r *// A simplified query*
1: $\mathcal{N}_r \leftarrow \mathcal{N}$
2: progressMade $\leftarrow$ True
3: **while** noTimeout() $\wedge$ progressMade **do**
4: $\quad \mathcal{N}_r \leftarrow \mathcal{N}$
5: $\quad$ progressMade, $\mathcal{N} \leftarrow$ Simplify$(\mathcal{V}, \mathcal{V}_O, Q)$
6: **return** $(\mathcal{N}_r, \mathcal{P})$

Alg. 2 takes in the same arguments as Alg. 1, and its goal is to perform one successful simplification step on $\mathcal{N}$, from a pool of potential steps. The algorithm heuristically chooses a sequence of simplification steps to attempt (Line 1), and then performs them, one by one, until one is successful. We propose several simplification steps in Sec. III-B. Specifying the order according to which these simplification steps are attempted (Line 1) is key, and different strategies may result in different simplified networks — we propose one such strategy in Sec. III-B.

Algorithm 2 *Simplify*

Input: $\mathcal{V}$, $\mathcal{V}_O$, $Q = (\mathcal{N}, \mathcal{P})$
// Faulty Verifier, Oracle Verifier, Verification query
Output: True/False, Q_r *// Whether the query was simplified, and the simplified query*
1: Attempts = $(M_0, M_1, \ldots) \leftarrow$ attemptsBySimplificationStrategy$(\mathcal{N})$
2: **while** Attempts $\neq \emptyset$ **do**
3: $\quad M_i \leftarrow$Attempts.*pop*()
4: $\quad \mathcal{N}_r \leftarrow M_i(\mathcal{N})$
5: $\quad$ **if** successSimplification$(\mathcal{V}, \mathcal{V}_O, (\mathcal{N}_r, \mathcal{P}))$ **then**
6: $\quad\quad$ **return** True, $\mathcal{N}_r$
7: **return** False, $\mathcal{N}$

Line 5 of Alg. 2 invokes Alg. 3 to check whether the simplification step attempted succeeded or not. To do so, Alg. 3 first checks whether $\mathcal{V}$ answers SAT, but returns an incorrect counter-example. If so, this candidate should clearly be kept. Otherwise, the algorithm checks whether $\mathcal{V}$ and $\mathcal{V}_O$ disagree in their verdicts; if so, it returns True. In all other

cases, i.e. where one of the verifiers times out, or when there is no basis for comparison (one of the verifiers returned an error), the algorithm returns False, and an alternative simplification step in Alg. 2 is attempted.

Algorithm 3 *successSimplification*

Input: $\mathcal{V}$, $\mathcal{V}_O$, $Q = (\mathcal{N}, \mathcal{P})$
// Faulty Verifier, Oracle Verifier, Verification query
Output: True/False *// Was the query successfully simplified?*
1: **if** $\mathcal{V}(\mathcal{N}, \mathcal{P}) = \texttt{SAT} \wedge \mathcal{V}_W(Q) \notin I(\mathcal{P})$ **then**
2: **return** True
3: **if** $\mathcal{V}(\mathcal{N}, \mathcal{P}) = \texttt{SAT} \wedge \mathcal{N}(\mathcal{V}_W(Q)) \notin O(\mathcal{P})$ **then**
4: **return** True
5: **if** $\mathcal{V}(\mathcal{N}, \mathcal{P}), \mathcal{V}_O(\mathcal{N}, \mathcal{P}) \in \{\texttt{SAT}, \texttt{UNSAT}\}$
$\wedge\ \mathcal{V}(\mathcal{N}, \mathcal{P}) \neq \mathcal{V}_O(\mathcal{N}, \mathcal{P})$ **then**
6: **return** True
7: **return** False

One possible risk when using Alg. 1 is a "flip" between the two verifiers. This can happen when initially, $\mathcal{V}_O$ produces a correct answer and $\mathcal{V}$ does not; but after a simplification step, $\mathcal{V}$ starts producing the correct answer and $\mathcal{V}_O$ starts producing an incorrect answer. This situation is unlikely: the simplification steps we propose later make local modifications to the network, and are consequently far more likely to continue to trigger the same bug in $\mathcal{V}$ than to trigger a new one in $\mathcal{V}_O$. Still, this concern can be mitigated even further by using multiple oracle verifiers, and ensuring that they all agree amongst themselves while $\mathcal{V}$ dissents. Even though this design does not completely prevent a "flip" scenario, it makes it highly unlikely.

Single Verifier Mode. Our approach could also be applied to delta-debug a single verifier that returns incorrect satisfying assignments, without using an oracle. As we explain in Sec. III-B, the simplification methods we apply require the returned satisfying assignment from either the faulty or the oracle verifier; and thus, if the faulty verifier returns an incorrect satisfying assignment for the query at hand, we can drop the oracle verifier. This is achieved by removing the last "if" condition from Alg. 3 and removing the oracle verifier $\mathcal{V}_O$ from the inputs. Note, however, that if the faulty verifier returns an UNSAT answer, an oracle verifier is always needed.

B. Simplification Methods

A core component of Alg. 1 is the selection of simplification strategy to apply (Line 1 in Alg. 2). We now describe our pool of neural network simplification methods, and the strategy that we suggest for selecting among them. The goal of all the simplification methods we propose here is to reduce neural network sizes, while keeping the network's behavior (i.e., its outputs) similar to that of the original; especially on the counter-example provided by either the faulty verifier or the oracle verifier. Note that a single simplification method can often be applied multiple times, in different ways, using different input parameters.

Method 1: linearizing piecewise-linear activation functions between fully-connected layers. In general, the presence of activation functions is a major source of complexity in the verification process of neural networks: they render the problem NP-complete, require complex mechanisms for linearly approximating them, and often entail case-splitting that slows down the verifiers [30], [39], [57]. Thus, in order to simplify the neural network, we propose to eliminate such activation functions, by *fixing them to a single linear segment*, effectively replacing them with linear constraints. This procedure is performed on an entire layer at a time; which, in turn, creates a sequence of consecutive purely linear layers that can then be merged into a single linear layer, reducing the overall number of layers and neurons in the network.

In choosing the linear segment to which each function is fixed, we propose to use the counter-example I provided by either the faulty verifier or the oracle verifier. The output of the new linear segment we choose, with respect to I, will match the output of the activation function on I.

For simplicity, we focus here on the ReLU activation function ($\text{ReLU}(x) = \max(x, 0)$), although the technique is applicable to any piecewise-linear function. Intuitively, in such cases we propose to replace *active* ReLUs ($x \geq 0$) by the identify function, and *inactive* ReLUs ($x < 0$) by zero. More formally, observe two consecutive layers, l^t and l^{t+1}, in the neural network $\mathcal{N}$, where layer l^t has a ReLU activation function. We construct an alternative layer, l^a, to replace both l^t and l^{t+1}. l^a inherits the activation function of l^{t+1}. The weights W^a and the biases B^a of l^a are calculated as:

$$W^a = W^{t+1} W' W^t$$
$$B^a = W^{t+1} W' B^t + B^{t+1}$$

where

$$W'_{i,j} = \begin{cases} 1 & i = j \wedge \left(N_Q^{l^t}(I)\right)_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Here W' is the new linear segment replacing the activation function ReLU. Finally, the obtained simplified network $\mathcal{N}_r$ is the network $\mathcal{N}$ where layers l^t and l^{t+1} are deleted and replaced with l^a.

Fig. 2 depicts the result of applying this method on layers l^2 and l^3 from Fig. 1, using the assignment $I_e = (5)$. Fig. 2a depicts the layers selected for merging; and Fig. 2b depicts the resulting neural network. Notice that $\mathcal{N}_e^{l^2}(I_e) = (4, 2, -2)$, meaning that only the ReLUs in neurons l_0^2 and l_1^2 are active. Thus, these ReLUs are replaced by the identity function, whereas the inactive ReLU of l_2^2 is replaced by 0. After this step, layers l^2 and l^3 perform only linear operations, and are merged into a single layer.

Method 2: linearizing piecewise-linear activation functions between convolutional layers. In this method, a convolutional layer is combined with the layer following it (either a fully connected layer or a convolutional one), and replaced by a single, fully connected layer.

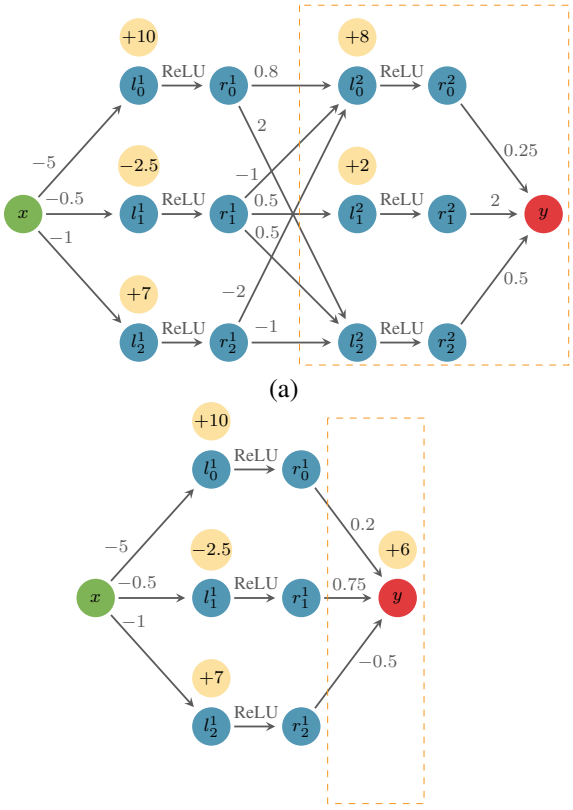

Fig. 2: $\mathcal{N}_e$ with layers l^2 and l^3 selected in orange (a), and then merged (b).

For simplicity, we focus here on the case where the second layer is fully connected. More formally, observe two consecutive layers, l^t and l^{t+1} in $\mathcal{N}$, where l^t is a convolutional layer and l^{t+1} is a fully connected layer. Our goal is to construct an alternative layer, l^a, that will replace l^t and l^{t+1}. Since a convolutional layer is a particular case of fully connected layer, we construct l^a by first converting the convolutional layer l^t into a fully connected one, denoted l^c; then linearizing the activation functions, as in *Method 1*; and finally, combining the two layers into one. This method may add edges to the network, and could potentially cause the network size to temporarily increase. However, when this method is used in conjunction with the remaining methods, it sets the network up for additional simplification, which ultimately results in a much smaller network.

Denote by W^t and W^{t+1} the matrices representing the weights of layer l^t and l^{t+1} respectively, and by B^t and B^{t+1} the vectors representing their respective biases. To transform a convolutional layer into a fully connecting one, we calculate the weights, W^c, and the biases, B^c, of the fully connected layer replacing the convolutional one, according to the conventional layer parameters. First, we turn its input and output from a multidimensional tensors into 1-dimensional vectors. The height and width (dimensions) of the feature maps in the convolutional layer's output are: h_o, w_o where

$$h_o = \left\lfloor \frac{h+2p-k}{s} \right\rfloor + 1, \quad w_o = \left\lfloor \frac{w+2p-k}{s} \right\rfloor + 1.$$

The convolutional layer's output contains c_o feature maps, i.e., the dimensions of the output are $(c_o \times h_o \times w_o)$. Thus, the dimensions of W^c are $(c_o h_o w_o \times c_i h w)$. W^c is a sparse matrix. To calculate the value of the i,j-th entry in W^c, we first compute the following values:

$$c'_i = \left\lfloor \frac{j}{hw} \right\rfloor, \quad c'_o = \left\lfloor \frac{i}{h_o w_o} \right\rfloor,$$
$$i' = \left\lfloor \frac{i - c_i hw}{w} \right\rfloor - \left(\left\lfloor \frac{j - c_o h_o w_o}{w_o} \right\rfloor \cdot s - p \right)$$
$$j' = ((i - c_i hw) \bmod w) - (((j - c_o h_o w_o) \bmod w_o) \cdot s - p)$$

c'_i and c'_o are the input and output channels that the i,j-th entry should be associated with. i' and j' are the indices in the kernel that should match to the i,j-th entry. The weight matrix W^c is given by:

$$W^c_{i,j} = \begin{cases} W^t_{c'_i, c'_o, i', j'} & 0 \le i' \wedge j' < k \\ W^c_{i,j} = 0 & \textit{otherwise} \end{cases}$$

Finally,

$$B^c_i = B^t_{\lfloor \frac{i}{h_o w_o} \rfloor}$$

According to this construction of W^c and B^c, they will have the same functionality as the convolutional operation they replace (assuming no floating-point or numerical errors). This step may temporarily increase the number of edges in the network (the number of neurons remains fixed); but this is required to prepare for the minimization step.

The next step is to linearize the ReLU. This is done in a similar manner to the linearization in the previous method, from which we get W'. Next, we construct the weights W^a and the biases B^a of the alternative layer l^a:

$$W^a = W^{t+1} W' W^c$$
$$B^a = W^{t+1} W' B^c + B^{t+1}$$

And the activation function assigned to the new layer l_a is the same as the one assigned to layer l_{t+1}. Finally, the simplified neural network $\mathcal{N}_r$ is the network $\mathcal{N}$, where layers l_t and l_{t+1} are deleted and replaced with l_a.

In case l^{t+1} is also a convolutional layer, we convert it to a fully connected layer, as we did with l^t; and the remainder of the process is unchanged.

Method 3: merging neurons. In this method, we seek to merge a pair of neurons in the same layer into a single neuron, thus decreasing the neural network size by one. Of course, this entails selecting the weights of this new neuron's incoming and outgoing edges, as well as its bias. Our motivation is to cause the merged neuron to produce values close to those of the original neurons, and consequently cause little changes in the neural network's eventual output. We present first the technical

process of merging neurons, and later discuss *which* pairs of neurons should be merged.

We focus again on the case where the activation function is ReLU. We first use the counter-example I (returned by either the faulty verifier or the oracle verifier) to check whether the activation functions of the neurons being merged have the same phase — i.e., if they are both active, or both inactive. If they have the same phase, we compute the merged neuron's weights and biases using the original neurons' weights and biases. Specifically, the weight of each edge incoming to the merged neuron is the mean of the original incoming edge weights, and the neuron's bias is the mean of the original neurons' biases; whereas the weights of its outgoing edges are the weighted sum, according to I, of the original outgoing edge weights. We choose a weighted sum, instead of a simple sum, to ensure that the neurons in the following layer obtain values similar to their original ones with respect to I; and also to preserve the network's behavior. In case one of the neurons is active and the other is inactive, we simply delete the inactive one, since it does not contribute to the following layer's neuron values (with respect to I).

Formally, given a neural network, $\mathcal{N}$, two successive layers in it, l^t and l^{t+1}, and two neurons indices $b < c$, we construct two alternative layers l^a and l^{a+1} that will replace l^t and l^{t+1} respectively. l^a and l^{a+1} inherit the activation functions of l^t and l^{t+1} respectively. If the ReLUs of the neurons b and c in layer l^t have the same phases: $\left(\mathcal{N}^{l^t}(I)\right)_b, \left(\mathcal{N}^{l^t}(I)\right)_c > 0$ or $\left(\mathcal{N}^{l^t}(I)\right)_b, \left(\mathcal{N}^{l^t}(I)\right)_c < 0$, the weights and the biases $W^a, W^{a+1}, B^a, B^{a+1}$ of the alternative layers are calculated as follows:

$$B_i^a = \begin{cases} B_i^t & i < b \vee b < i < c \\ \frac{B_b^t + B_c^t}{2} & i = b \\ B_{i+1}^t & c \leq i \end{cases}$$

$$B^{a+1} = B^{t+1}$$

$$W_{i,j}^a = \begin{cases} W_{i,j}^t & i < b \vee b < i < c \\ \frac{W_{b,j}^t + W_{c,j}^t}{2} & i = b \\ W_{i+1,j}^t & c \leq i \end{cases}$$

$$W_{i,j}^{a+1} = \begin{cases} W_{i,j}^{t+1} & j < b \vee b < j < c \\ \frac{2 \cdot \left(W_{i,b}^{t+1} \left(\mathcal{N}^{l^{t+1}}(I)\right)_b + W_{i,c}^{t+1} \left(\mathcal{N}^{l^{t+1}}(I)\right)_c\right)}{\left(\mathcal{N}^{l^{t+1}}(I)\right)_b + \left(\mathcal{N}^{l^{t+1}}(I)\right)_c} & j = b \\ W_{i,j+1}^{t+1} & c \leq j \end{cases}$$

Otherwise, if the ReLUs of the neurons b and c in layer l^t have different phases: $\left(\mathcal{N}^{l^t}(I)\right)_b > 0 \wedge \left(\mathcal{N}^{l^t}(I)\right)_c < 0$ (assume w.l.o.g. that the c-th neuron is the inactive one), the weights and biases $W^a, W^{a+1}, B^a, B^{a+1}$ of the alternative layers are calculated as follows:

$$B_i^a = \begin{cases} B_i^t & i < c \\ B_{i+1}^t & c \leq i \end{cases}, \qquad B^{a+1} = B^{t+1}$$

$$W_{i,j}^a = \begin{cases} W_{i,j}^t & i < c \\ W_{i+1,j}^t & c \leq i \end{cases}, \quad W_{i,j}^{a+1} = \begin{cases} W_{i,j}^{t+1} & j < c \\ W_{i,j+1}^{t+1} & c \leq j \end{cases}$$

Finally, the obtained simplified neural network $\mathcal{N}_r$, is the network $\mathcal{N}$ where layers l^t and l^{t+1} are replaced with l^a and l^{a+1} respectively. This method can be applied repeatedly, to reduce the network size even further.

An example of applying this method on the pair of neurons l_0^2 and l_1^2 in $\mathcal{N}_e$ from Fig. 1 using the assignment $I_e = (5)$ appears in Fig. 3. Fig. 3a shows the neurons selected for merging, and Fig. 3b shows the result of the merge.

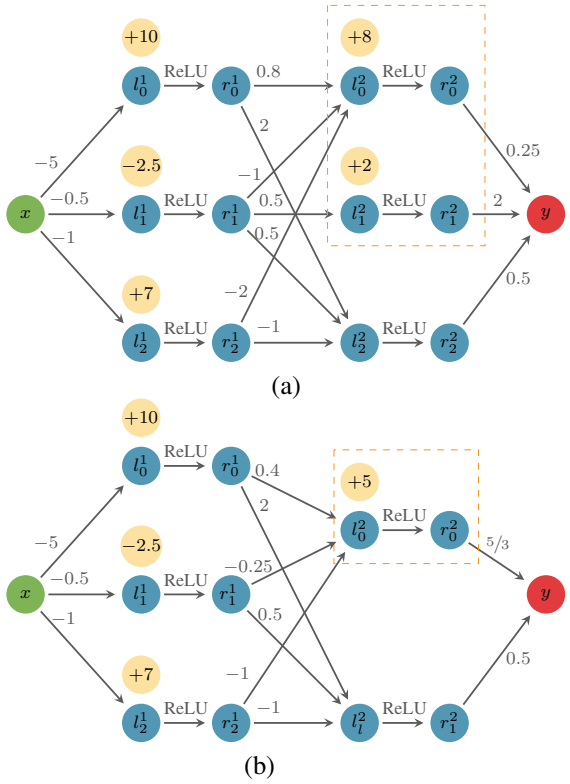

Fig. 3: $\mathcal{N}_e$ with neurons l_0^2 and l_1^2 selected in orange (a), and then merged (b).

Choosing which pair of neurons to merge is crucial for the success of this method. Every two neurons in the same layer are valid candidates; however, some pairs are more likely to succeed than others by resulting in a simplified neural network that behaves similarly to the original. We consider the following possible approaches for prioritizing between the pairs: (1) an arbitrary ordering; (2) prioritizing pairs with neurons that are assigned similar values (prior to the activation function), when the network is evaluated on assignment I. The motivation is that merging such pairs is expected to have smaller effect on the overall functionality of the neural network; (3) prioritizing pairs of neurons whose

ReLUs are inactive when evaluated on I. The motivation is that inactive neurons may have little effect on the bug at hand. This approach can be combined with Approach 2 to prioritize pairs with similar values after categorizing them by the status of the ReLUs; (4) prioritizing pairs of neurons with positive values with respect to I. This approach, too, can be combined with Approach 2; and (5) prioritizing pairs of neurons with negative values, and then pairs with positive values, with respect to I. This approach is a combination of Approaches 3 and 4, and again uses Approach 2 for internal prioritization within each category.

Strategy for applying the simplification rules. Within Alg. 1, the simplification steps mentioned above can be invoked in any order. We propose to attempt methods that significantly reduce the neural network size first, in order to reduce verification times. We empirically observed that this is achieved by the following strategy: first, attempt to linearize and merge convolutional layers (*Method 2*). Second, attempt to linearize and merge fully connected layers (*Method 1*) — starting with the output layer, and working backwards towards the input layer. Finally, merge neurons (*Method 3*) according to Approach 5. However, our implementation is highly customizable, and users can configure it to use any other strategy, according to the task at hand.

To illustrate, applying our proposed strategy to $\mathcal{N}_e$ from Fig. 1, with respect to the assignment $I_e = (5)$ in which $\mathcal{N}_e^{l^1}(I_e) = (-15, -5, 2)$ and $\mathcal{N}_e^{l^2}(I_e) = (4, 2, -2)$, would result in attempting the simplification methods in the following order: (1) merge the layers l^2 and l^3; (2) merge the layers l^1 and l^2; (3) merge the pair of neurons l_0^1, l_1^1; (4) merge the pair of neurons l_1^2, l_2^2; (5) merge the pair of neurons l_0^2, l_2^2; (6) merge the pair of neurons l_1^1, l_2^1; and then, (7) merge the pair of neurons l_0^1, l_2^1. These steps are attempted, in order, until one succeeds; after which the strategy is reapplied to the simplified network, and so on.

IV. Implementation and Evaluation

We designed our tool, DelBugV, to be compatible with the standard input format used in the VNN-COMP competition [9], in which verification queries are encoded using the *VNN-LIB* format [11]; and which, in turn, relies on the *Open Neural Network Exchange* (*ONNX*) format. This facilitated integrating DelBugV with the various verifiers. DelBugV is implemented in Python, and contains classes that wrap objects of these formats. The tool has a modular design that allows applying our proposed minimization methods in any order desired. The source code can be found at https://github.com/Raya5/DelBugV.

VNN-COMP'21 included 12 participating neural network verifiers, and these were tested on a set of verification queries. We began by extracting from the VNN-COMP'21 results pairs of dissenting verifiers, and the verification queries that triggered these discrepancies. Each such triple (two verifiers and a query) constitutes an input to DelBugV. This extraction led us to target the following verifiers: (1) Marabou [32]; (2) NNV [50]–[53], [60]; (3) NeuralVerification.jl (NV.jl) [36]; and (4) nnenum [7], [8], [50], [51]. In the experiments described next, we used the same versions of these verifiers that were used in VNN-COMP'21.

Neuron Merging and Prioritization Approaches. For our first experiment, we set out to determine which of the neuron-pair prioritization schemes described as part of *Method 3* in Sec. III-B is the most successful. We measured success along two parameters: the size of the simplified network obtained, and by the percentage of successful merging steps along the way. We tested our algorithm on 5 input triples, involving networks of size 310 each. Using only *Method 3*, we ran DelBugV with each of the prioritization schemes, and counted for each scheme the number of merging steps performed and the number of the steps that succeeded on all of the 5 input triples. Table. I shows the results of this comparison: the second column indicates, for every approach, the percentage of the successful steps out of all the steps tried, aggregated for all 5 benchmarks.

TABLE I: Comparing neurons merging approaches (Method 3) by size reduction and successful merges.

	Successful merges (%)	Average Reduction (%)
Approach 1	37.2%	96.0%
Approach 2	68.4%	95.9%
Approach 3	71.6%	96.0%
Approach 4	62.9%	95.8%
Approach 5	**75.9%**	96.0%

Looking at the average reduction sizes, the results indicate that all 5 approaches were able to achieve a similar reduction in size, with a slight advantage to approaches 1, 3 and 5. However, the number of successful merges varied significantly — from Approach 1, in which only 37.2% of the merge steps were successful, and up to 75.9% for Approach 5 (in bold). These results thus indicate that Approach 5 is the most efficient among the considered approaches, and so we used it as our default strategy for Method 3 in the subsequent experiments.

Linearizing ReLU Activations. In *Method 1* and *Method 2* in Sec. III-B, we proposed to linearize activation functions, and then merge them with the previous and following layers. These methods can be applied to any piecewise-linear activation function in the network. The order in which they are applied is customizable. In this experiment, we set out to compare linearizing ReLUs in ascending order (from input layer towards output layer), and in descending order (from output towards input). Table II shows the results of this experiment.

Every row in the table corresponds to an input triple to DelBugV (two disagreeing verifiers and a verification query that they disagreed on), and the two simplification approaches that were attempted.

For each such experiment, the second column indicates the number of simplification steps tried, until DelBugV reached saturation (there were no additional steps to try). The third column indicates the number of the successful steps out of all

TABLE II: Comparing linearizing layers approaches by successful steps. * indicates the existence of a convolutional layer.

	Linearizing approach	No. of steps	No. of successful steps	Successful steps %	Neuron reduction %
1.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
2.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
3.	Ascending	6	6	100.0%	96.7%
	Descending	6	6	100.0%	96.7%
4.	Ascending	6	0	0.0%	0.0%
	Descending	6	0	0.0%	0.0%
5.	Ascending	12	5	41.6%	80.6%
	Descending	9	6	**66.6%**	**96.7%**
6.	Ascending	3	2	66.6%	39.2%
	Descending	2	2	**100.0%**	39.2%
7.	Ascending	3*	2*	**66.6%**	**65.8%**
	Descending	2*	1	50.0%	0.0%

the steps. In column four, the percentage of successful steps out of all steps is shown; we use this column to compare the approaches. We mark in bold the leading approach for every triple. The final column shows the reduction percentage in the neural network size. When one of the approaches was clearly superior, the entry appears in bold.

To analyze the results, observe, e.g., the 5th experiment in Table II. The results imply that when using the ascending approach, 12 linearizing and merging steps were made, until the network could not be simplified further with either *Method 1* or *Method 2*. Of these 12 steps, 5 were successful — and consequently, the simplified network has 5 fewer layers than the original. In contrast, with the descending approach only 9 steps were made until the network could not be simplified further, 6 of which were successful. Consequently, the simplified network in this case has 6 fewer layers compared to the original.

The results indicate that linearizing in descending order slightly outperforms linearizing in ascending order, although the gap is not very significant. We believe that the results depend also on both the functionality of the verifier and the values of the network as well. Meaning, they can vary between the benchmarks. The neural network in the last row included a convolutional layer, and, according to the results, linearizing it in ascending order performed better. After investigating this query further, we noticed that in the ascending order approach, the convolutional layer was merged into a fully connected one; whereas the descending approach did not succeed in removing or merging any convolutional layers. We thus conclude that, for a convolutional network, it is advisable to apply *Method 2* before applying *Method 1*.

An interesting phenomenon in both our methods is that they have an overall high reduction percentage for most of the strategies. The strategies mainly differ in the number of steps taken to reach this reduction. This phenomenon implies that the simplification methods are overall effective, but may be time-consuming. Thus, using strategies that entail dispatching fewer verification queries using the verifiers is more productive.

Delta Debugging Discrepancies from VNN-COMP'21. For our final experiment, we considered 13 triples of verifiers, oracle verifiers, and verification queries. Of these triples, 11 contained DNNs from the ACAS-Xu family [30], 1 was a DNN from the MNIST DNNs [35], and 1 was a DNN from the Oval21 benchmark [9]. The DNNs from the ACAS-Xu family had 8 layers: 6 inner fully-connected layers with 50 neurons in each and 5 neurons in both the input and output layer — 310 neurons in total in each network. The MNIST DNN contained 4 fully-connected layers with 784 neurons in the input layer, 256 neurons in each of the hidden layers and 10 neurons in the output layer — 1306 neurons in total. The Oval21 DNN is a convolutional neural network with 5 layers. Its input layer contains 3072 neurons (which represent $3 \times 32 \times 32$ images). Those neurons are processed by the first two convolutional hidden layers to 4096 neurons and then to 2048 neurons. The following hidden layer is a fully-connected one with 100 neurons followed by the output layer which contains 10 neurons — in total, the Oval21 DNN has 9326 neurons. It is worth mentioning that the properties we used (taken from VNN-COMP) separately state the input specifications and output specifications; however, our methods do not require such a distinction. Further, although the VNN-COMP specifications contains primarily linear constraints, our method can also handle relational properties.

Using the optimal configuration of our tool as previously discussed, we applied the full-blown delta-debugging algorithm to all of our 13 benchmarks. The results appear in Table. III. Every row in the table represents a triple, and the first two columns indicate the number of neurons in the original network, and the number of remaining neurons after delta debugging was applied. The next two columns indicate the number of layers in the original and reduced networks; and the final column indicates the percentage of neurons that were removed.

TABLE III: Delta-debugging using our algorithm. * indicates the existence of a convolutional layer.

Neurons		Layers		Reduction
In Original	In reduced	In original	In reduced	percentage
310	6	8	2	98%
310	7	8	2	97%
310	6	8	2	98%
310	12	8	8	96%
310	6	8	2	98%
9326	12	5*	3	99%
1306	11	4	2	99%
310	10	8	3	96%
310	6	8	2	98%
310	10	8	4	96%
310	10	8	4	96%
310	9	8	4	97%
310	13	8	6	95%

Overall, the algorithm performed exceedingly well, reducing the network sizes by an average of 96.8% (!); and, in some cases, causing a size decrease of 99%, from a neural network with 1306 neurons and 4 layers to just 11 neurons and 2 layers (an input layer and an output layer, without any activation

functions). The minimal decrease observed was 95%, from 310 neurons to 13. We regard these results as a very strong indication of the usefulness of delta debugging in the context of DNN verification. Further analyzing the results, we observe that the ReLU linearization simplification rule was responsible for an average of 66% of the size reduction, whereas the remaining two rules were responsible for an average of 34% — indicating that the ReLU linearization simplification rule is the main workhorse of our approach at its current configuration.

V. Related Work

With the increasing pervasiveness of DNNs, the verification community has been devoting growing efforts to verifying them. Numerous approaches have been proposed, including SMT-based approaches [23], [30]–[32], [48], [58], approaches based on LP or MILP solvers [14], [16], [49], reachability-based approaches [38], [60], abstraction and abstract-interpretation based approaches [5], [18], [24], [27], [39], [44], [46], [57], synthesis-based approaches [33], [42], run-time optimization [4], [6], quantitative verification [10], verification of recurrent networks [28], [62], and many others. These approaches, in turn, have been used in numerous application domains [15], [17], [20], [47], [54], [55], [61]. Given the scope of these efforts, and the number of available tools, it is not surprising that bugs are abundant, and that engineers are in need of efficient debugging tools.

To the best of our knowledge, no previous work has applied delta debugging in the context of DNN verification, although similar approaches have been shown successful in the related domains of SMT [12], [40] and SAT [13] solving. Related efforts have attempted to reduce DNN sizes, with the purpose of producing smaller-but-equivalent networks, or networks smaller with respect to a particular verification property of interest [5], [34], [43], [44]. In the future, principles from these approaches could be integrated as simplification strategies within our delta-debugging approach.

VI. Conclusion

In this paper, we presented the DelBugV tool for automatically reducing the size of a verification query with respect to an erroneous neural network verifier. We focused on delta-debugging techniques, and proposed multiple minimization methods for reducing neural network sizes. These techniques attempt to simplify the neural network in question, while modifying it as little as possible. We also suggested a strategy for the order in which to apply those methods. We demonstrated the effectiveness of DelBugV on actual benchmarks from the VNN-COMP'21 competition, and were able to significantly simplify them. We regard this work as another step towards more sound tools for DNN verification.

Moving forward, we aim to continue this line of work in several directions. One direction we plan to pursue is to extend our evaluation, by considering more diverse sets of benchmarks and comparing our approach also to existing, general-purpose delta debuggers. Another direction is to extend our pool of neural network simplification methods, for example by supporting also activation functions that are not piecewise linear (e.g., Sigmoids). Additionally, we want to theoretically analyze our simplification methods. Such analysis will potentially benefit us in reducing the need for oracle verifiers.

Acknowledgements. This work was partially supported by the Binational Science Foundation (grant number 2021769).

References

[1] A. Albarghouthi. *Introduction to Neural Network Verification*. verified-deeplearning.com, 2021.

[2] G. Amir, Z. Freund, G. Katz, E. Mandelbaum, and I. Refaeli. veriFIRE: Verifying an Industrial, Learning-Based Wildfire Detection System. In *Proc. 25th Int. Symposium on Formal Methods (FM)*, 2023.

[3] G. Amir, T. Zelazny, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 27–37, 2022.

[4] G. Anderson, S. Pailoor, I. Dillig, and S. Chaudhuri. Optimization and Abstraction: a Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. 40th ACM SIGPLAN Conf. on Programming Languages Design and Implementations (PLDI)*, pages 731–744, 2019.

[5] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 92–107, 2020.

[6] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Könighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.

[7] B. Bak. nnenum: Verification of Relu Neural Networks with Optimized Abstraction Refinement. In *Proc. 13th NASA Formal Methods Symposium (NFM)*, pages 19–36, 2021.

[8] S. Bak. Execution-Guided Overapproximation (EGO) for Improving Scalability of Neural Network Verification. In *Proc. 3rd Int. Workshop on Verification of Neural Networks (VNN)*, 2020.

[9] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. http://arxiv.org/abs/2109.00498.

[10] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks and its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.

[11] C. Barrett, G. Katz, D. Guidotti, L. Pulina, N. Narodytska, and A. Tacchella. The Verification of Neural Networks Library (VNN-LIB), 2019. www.vnnlib.org.

[12] R. Brummayer and A. Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proc. 7th Int. Workshop on Satisfiability Modulo Theories (SMT)*, 2009.

[13] R. Brummayer, F. Lonsing, and A. Biere. Automated Testing and Debugging of SAT and QBF Solvers. In *Proc. 13th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, pages 44–57, 2010.

[14] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.

[15] G. Dong, J. Sun, J. Wang, X. Wang, and T. Dai. Towards Repairing Neural Networks Correctly, 2020. Technical Report. http://arxiv.org/abs/2012.01872.

[16] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.

[17] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 305–318, 2021.

[18] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[19] Y. Goldberg. A Primer on Neural Network Models for Natural Language Processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[20] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.

[21] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[22] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples, 2014. Technical Report. http://arxiv.org/abs/1412.6572.

[23] D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks. In *Proc. 16th. Int. Symp. on on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.

[24] E. Goubault, S. Palumby, S. Putot, L. Rustenholz, and S. Sankaranarayanan. Static Analysis of ReLU Neural Networks with Tropical Polyhedra. In *Proc. 28th Int. Symposium on Static Analysis (SAS)*, pages 166–190, 2021.

[25] J. Guo, H. He, T. He, L. Lausen, M. Li, H. Lin, X. Shi, C. Wang, J. Xie, S. Zha, et al. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research*, 21(23):1–7, 2020.

[26] J. Hou, B. Adhikari, and J. Cheng. DeepSF: Deep Convolutional Neural Network for Mapping Protein Sequences to Folds. *Bioinformatics*, 34(8):1295–1303, 2018.

[27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.

[28] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.

[29] K. Jia and M. Rinard. Exploiting Verified Neural Networks via Floating Point Numerical Error, 2020. Technical Report. http://arxiv.org/abs/2003.03021.

[30] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.

[31] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks, 2021.

[32] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

[33] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.

[34] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.

[35] Y. LeCun. The MNIST Database of Handwritten Digits, 1998. http://yann.lecun.com/exdb/mnist/.

[36] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2020. Technical Report. http://arxiv.org/abs/1903.06758.

[37] X. Liu, P. He, W. Chen, and J. Gao. Multi-Task Deep Neural Networks for Natural Language Understanding, 2019. Technical Report. http://arxiv.org/abs/1901.11504.

[38] A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1706.07351.

[39] M. Müller, G. Makarchuk, G. Singh, M. Püschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.

[40] A. Niemetz, M. Preiner, and C. Barrett. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 92–106, 2022.

[41] F. Noé, G. De Fabritiis, and C. Clementi. Machine Learning for Protein Folding and Dynamics. *Current Opinion in Structural Biology*, 60:77–84, 2020.

[42] E. Polgreen, R. Abboud, and D. Kroening. Counterexample Guided Neural Synthesis, 2020. Technical Report. https://arxiv.org/abs/2001.09245.

[43] P. Prabhakar. Bisimulations for Neural Network Reduction. In *Proc. 23rd Int. Conf. Verification on Model Checking, and Abstract Interpretation (VMCAI)*, pages 285–300, 2022.

[44] P. Prabhakar and Z. Afzal. Abstraction Based Output Range Analysis for Neural Networks, 2020. Technical Report. https://arxiv.org/abs/2007.09527.

[45] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.

[46] G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[47] M. Sotoudeh and A. Thakur. Correcting Deep Neural Networks with Small, Generalizing Patches. In *Workshop on Safety and Robustness in Decision Making*, 2019.

[48] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.

[49] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. http://arxiv.org/abs/1711.07356.

[50] H.-D. Tran, S. Bak, W. Xiang, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.

[51] H.-D. Tran, D. Manzanas Lopez, P. Musau, X. Yang, L. Nguyen, W. Xiang, and T. Johnson. Star-Based Reachability Analysis of Deep Neural Networks. In *Proc. Int. Symposium on Formal Methods (FM)*, pages 670–686, 2019.

[52] H.-D. Tran, P. Musau, D. Lopez, X. Yang, L. Nguyen, W. Xiang, and T. Johnson. Parallelizable Reachability Analysis Algorithms for Feed-Forward Neural Networks. In *Proc. 7th Int. Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 31–40, 2019.

[53] H.-D. Tran, X. Yang, D. Lopez, P. Musau, L. Nguyen, W. Xiang, S. Bak, and T. Johnson. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems, 2020. Technical Report. http://arxiv.org/abs/2004.05519.

[54] C. Urban, M. Christakis, V. Wüstholz, and F. Zhang. Perfectly Parallel Fairness Certification of Neural Networks. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–30, 2020.

[55] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. Păsăreanu. NNrepair: Constraint-based Repair of Neural Network Classifiers, 2021. Technical Report. http://arxiv.org/abs/2103.12535.

[56] International Verification of Neural Networks Competition (VNN-COMP), 2020. https://sites.google.com/view/vnn20/vnncomp.

[57] S. Wang, H. Zhang, K. Xu, X. Lin, S. Jana, C.-J. Hsieh, and Z. Kolter. Beta-CROWN: Efficient Bound Propagation with Per-Neuron Split Constraints for Complete and Incomplete Neural Network Verification. In *Proc. 35th Conf. on Neural Information Processing Systems (NeurIPS)*, 2021.

[58] H. Wu, A. Ozdemir, A. Zeljić, A. Irfan, K. Julian, D. Gopinath, S. Fouladi, G. Katz, C. Păsăreanu, and C. Barrett. Parallelization Techniques for Verifying Neural Networks. In *Proc. 20th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 128–137, 2020.

[59] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep Image: Scaling up Image Recognition. Technical Report. http://arxiv.org/abs/1501.02876.

[60] W. Xiang, H. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multi-Layer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2018.

[61] X. Yang, T. Yamaguchi, H.-D. Tran, B. Hoxha, T. Johnson, and D. Prokhorov. Neural Network Repair with Reachability Analysis, 2021. Technical Report. https://arxiv.org/abs/2108.04214.

[62] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.