An Abstraction-Based Framework for Neural Network Verification

Yizhak Yisrael Elboher[1], Justin Gottschlich[2], and Guy Katz[1]

[1] The Hebrew University of Jerusalem, Israel
{yizhak.elboher, g.katz}@mail.huji.ac.il
[2] Intel Labs, USA
justin.gottschlich@intel.com

Abstract. Deep neural networks are increasingly being used as controllers for safety-critical systems. Because neural networks are opaque 不透明的, certifying their correctness is a significant challenge. To address this issue, several approaches have recently been proposed to formally verify them. However, network size is often a bottleneck 瓶颈 for such approaches and it can be difficult to apply them to large networks. In this paper, we propose 提出 a framework that can enhance neural network verification techniques by using over-approximation to reduce the size of the network — thus making it more amenable 适合的 to verification. We perform the approximation such that if the property holds for the smaller (abstract) network, it holds for the original as well. The over-approximation may be too coarse, in which case the underlying verification tool might return a spurious counterexample. Under such conditions, we perform counterexample-guided refinement to adjust the approximation, and then repeat the process. Our approach is orthogonal 正交的 to, and can be integrated with, many existing verification techniques. For evaluation purposes 为了评估我们的方法, we integrate it with the recently proposed 提出的 Marabou framework, and observe a significant improvement in Marabou's performance. Our experiments demonstrate 表明 the great potential of our approach for verifying larger neural networks.

1 Introduction

Machine programming (MP), the automatic generation of software, is showing early signs of fundamentally transforming the way software is developed [10]. A key ingredient employed by MP is the *deep neural network* (DNN), which has emerged as an effective means to semi-autonomously implement many complex software systems. DNNs are artifacts produced by *machine learning*: a user provides examples of how a system should behave, and a machine learning algorithm generalizes these examples into a DNN capable of correctly handling inputs that it had not seen before. Systems with DNN components have obtained unprecedented results in fields such as image recognition [18], game playing [27], natural language processing [11], computer networks [22], and many others, often surpassing the results obtained by similar systems that have been carefully handcrafted. It seems evident that this trend will increase and intensify, and that DNN components will be deployed in various safety-critical systems [1,13].

DNNs are appealing in that (in some cases) they are easier to create than handcrafted software, while still achieving excellent results. However, their usage also raises a challenge when it comes to certification. Undesired behavior has been observed in many state-of-the-art DNNs. For example, in many cases slight perturbations to correctly handled inputs can cause severe errors [28,20]. Because many practices for improving the reliability of hand-crafted code have yet to be successfully applied to DNNs (e.g., code reviews, coding guidelines, etc.), it remains unclear how to overcome the opacity of DNNs, which may limit our ability to certify them before they are deployed.

To mitigate this, the formal methods community has begun developing techniques for the formal verification of DNNs (e.g., [7,12,14,30]). These techniques can automatically prove that a DNN always satisfies a prescribed property. Unfortunately, the DNN verification problem is computationally difficult (e.g., NP-complete, even for simple specifications and networks [14]), and becomes exponentially more difficult as network sizes increase. Thus, despite recent advances in DNN verification techniques, network sizes remain a severely limiting factor.

In this work, we propose a technique by which the scalability of many existing verification techniques can be significantly increased. The idea is to apply the well-established notion of *abstraction and refinement* [4]: replace a network N that is to be verified with a much smaller, *abstract* network, $\bar{N}$, and then verify this $\bar{N}$. Because $\bar{N}$ is smaller it can be verified more efficiently; and it is constructed in such a way that if it satisfies the specification, the original network N also satisfies it. In the case that $\bar{N}$ does not specify the specification, the verification procedure provides a counterexample x. This x may be a true counterexample demonstrating that the original network N violates the specification, or it may be *spurious*. If x is spurious, the network $\bar{N}$ is *refined* to make it more accurate (and slightly larger), and then the process is repeated. A particularly useful variant of this approach is to use the spurious x to guide the refinement process, so that the refinement step rules out x as a counterexample. This variant, known as *counterexample guided abstraction refinement* (*CEGAR*) [4], has been successfully applied in many verification contexts.

As part of our technique we propose a method for abstracting and refining neural networks. Our basic abstraction step *merges* two neurons into one, thus reducing the overall number of neurons by one. This basic step can be repeated numerous times, significantly reducing the network size. Conversely, refinement is performed by splitting a previously merged neuron in two, increasing the network size but making it more closely resemble the original. A key point is that not all pairs of neurons can be merged, as this could result in a network that is smaller but is not an over-approximation of the original. We resolve this by first transforming the original network into an equivalent network where each node belongs to one of four classes, determined by its edge weights and its effect on the network's output; merging neurons from the same class can then be done safely. The actual choice of which neurons to merge or split is performed heuristically. We propose and discuss several possible heuristics.

For evaluation purposes, we implemented our approach as a Python framework that wraps the Marabou verification tool [16]. We then used our framework to verify properties of the Airborne Collision Avoidance System (ACAS Xu) set of benchmarks [14]. Our results strongly demonstrate the potential usefulness of abstraction in enhancing existing verification schemes: specifically, in most cases the abstraction-enhanced Marabou significantly outperformed the original. Further, in most cases the properties in question could indeed be shown to hold or not hold for the original DNN by verifying a small, abstract version thereof.

To summarize, our contributions are: (i) we propose a general framework for over-approximating and refining DNNs; (ii) we propose several heuristics for abstraction and refinement, to be used within our general framework; and (iii) we provide an implementation of our technique that integrates with the Marabou verification tool and use it for evaluation.[3]

The rest of this paper is organized as follows. In Section 2, we provide a brief background on neural networks and their verification. In Section 3, we describe our general framework for abstracting an refining DNNs. In Section 4, we discuss how to apply these abstraction and refinement steps as part of a CEGAR procedure, followed by an evaluation in Section 5. In Section 6, we discuss related work, and we conclude in Section 7.

2 Background

2.1 Neural Networks

A neural network consists of an *input layer*, an *output layer*, and one or more intermediate layers called *hidden layers*. Each layer is a collection of nodes, called *neurons*. Each neuron is connected to other neurons by one or more directed edges. In a feedforward neural network, the neurons in the first layer receive input data that sets their initial values. The remaining neurons calculate their values using the weighted values of the neurons that they are connected to through edges from the preceding layer (see Fig. 1). The output layer provides the resulting value of the DNN for a given input.

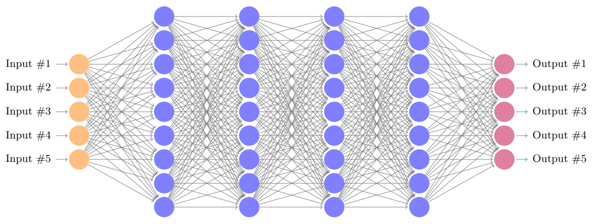

Fig. 1. A fully connected, feedforward DNN with 5 input nodes (in orange), 5 output nodes (in purple), and 4 hidden layers containing a total of 36 hidden nodes (in blue). Each edge is associated with a weight value (not depicted).

There are many types of DNNs, which may differ in the way their neuron values are computed. Typically, a neuron is evaluated by first computing a weighted sum of the preceding layer's neuron values according to the edge weights, and then applying an activation function to this weighted sum [8]. We

[3] We intend to make our code publicly available with the final version of this paper.

focus here on the Rectified Linear Unit (ReLU) activation function [23], given as $\text{ReLU}(x) = \max(0, x)$. Thus, if the weighted sum computation yields a positive value, it is kept; and otherwise, it is replaced by zero.

More formally, given a DNN N, we use n to denote the number of layers of N. We denote the number of nodes of layer i by s_i. Layers 1 and n are the input and output layers, respectively. Layers 2 ... $n-1$ are the hidden layers. We denote the value of the j-th node of layer i by $v_{i,j}$, and denote the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$ as V_i.

Evaluating N is performed by calculating V_n for a given input assignment V_1. This is done by sequentially computing V_i for $i = 2, 3, \ldots, n$, each time using the values of V_{i-1} to compute weighted sums, and then applying the ReLU activation functions. Specifically, layer i (for $i > 1$) is associated with a weight matrix W_i of size $s_i \times s_{i-1}$ and a bias vector B_i of size s_i. If i is a hidden layer, its values are given by $V_i = \text{ReLU}(W_i V_{i-1} + B_i)$, where the ReLUs are applied element-wise; and the output layer is given by $V_n = W_n V_{n-1} + B_n$ (ReLUs are not applied). Without loss of generality, in the rest of the paper we assume that all bias values are 0, and can be ignored. This rule is applied repeatedly once for each layer, until V_n is eventually computed.

We will sometimes use the notation $w(v_{i,j}, v_{i+1,k})$ to refer to the entry of W_{i+1} that represents the weight of the edge between neuron j of layer i and neuron k of layer $i+1$. We will also refer to this edge as an *outgoing edge* for $v_{i,j}$, and as an *incoming edge* for $v_{i+1,k}$.

As part of our abstraction framework, we will sometimes need to consider a *suffix* of a DNN, in which the first layers of the DNN are omitted. For $1 < i < n$, we use $N^{[i]}$ to denote the DNN comprised of layers $i, i+1, \ldots, n$ of the original network. The sizes and weights of the remaining layers are unchanged, and layer i of N is treated as the input layer of $N^{[i]}$.

Fig. 2 depicts a small neural network. The network has $n = 3$ layers, of sizes $s_1 = 1, s_2 = 2$ and $s_3 = 1$. Its weights are $w(v_{1,1}, v_{2,1}) = 1$, $w(v_{1,1}, v_{2,2}) = -1$, $w(v_{2,1}, v_{3,1}) = 1$ and $w(v_{2,2}, v_{3,1}) = 2$. For input $v_{1,1} = 3$, node $v_{2,1}$ evaluates to 3 and node $v_{2,2}$ evaluates to 0, due to the ReLU activation function. The output node $v_{3,1}$ then evaluates to 3.

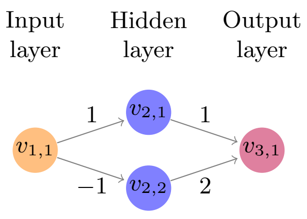

Fig. 2. A simple feedforward neural network.

2.2 Neural Network Verification

DNN verification amounts to answering the following question: given a DNN N, which maps input vector x to output vector y, and predicates P and Q, does there exist an input x_0 such that $P(x_0)$ and $Q(y_0)$ both hold, where $y_0 = N(x_0)$? In other words, the verification process determines whether there exists a particular input that meets the input criterion P, and that is mapped to an output that meets the output criterion Q. We refer to $\langle N, P, Q\rangle$ as the *verification query*. As is usual in verification, Q represents the *negation* of the desired property. Thus, if the query is *unsatisfiable* (`UNSAT`), the property holds; and if it is *satisfiable* (`SAT`), then x_0 constitutes a counterexample to the property in question.

Different verification approaches may differ in *(i)* the kinds of neural networks they allow (specifically, the kinds of activation functions in use); *(ii)* the kinds of input properties; and *(iii)* the kinds of output properties. For simplicity, we focus on networks that employ the ReLU activation function. In addition, our input properties will be conjunctions of linear constraints on the input values. Finally, we will assume that our networks have a single output node y, and that the output property is $y > c$ for a given constant c. We stress that these restrictions are for the sake of simplicity. Many properties of interest, including those with arbitrary Boolean structure, can be reduced into the above single-output setting by adding neurons that encode the Boolean structure [14,26]. In particular, this is true for the ACAS Xu family of benchmarks [14], and also for adversarial robustness queries that use the L_∞ or the L_1 norm as a distance metric [3,9,15]. Additionally, other piecewise-linear activation functions, such as max-pooling layers, can also be encoded using ReLUs [3].

Several techniques have been proposed for solving the aforementioned verification problem in recent years (Section 6 includes a brief overview). Our abstraction technique is designed to be compatible with most of these techniques, by simplifying the network being verified, as we describe next.

3 Network Abstraction and Refinement

Because the complexity of verifying a neural network is strongly connected to its size [14], our goal is to transform a verification query $\varphi_1 = \langle N, P, Q\rangle$ into query $\varphi_2 = \langle \bar{N}, P, Q\rangle$, such that the abstract network $\bar{N}$ is significantly smaller than N (notice that properties P and Q remain unchanged). We will construct $\bar{N}$ so that it is an over-approximation of N, meaning that if φ_2 is `UNSAT` then φ_1 is also `UNSAT`. More specifically, since our DNNs have a single output, we can regard $N(x)$ and $\bar{N}(x)$ as real values for every input x. To guarantee that φ_2 over-approximates φ_1, we will make sure that for every x, $N(x) \leq \bar{N}(x)$; and thus, $\bar{N}(x) \leq c \implies N(x) \leq c$. Because our output properties always have the form $N(x) > c$, it is indeed the case that if φ_2 is `UNSAT`, i.e. $\bar{N}(x) \leq c$ for all x, then $N(x) \leq c$ for all x and so φ_1 is also `UNSAT`. We now propose a framework for generating various $\bar{N}$s with this property.

3.1 Abstraction

We seek to define an abstraction operator that removes a single neuron from the network, by merging it with another neuron. To do this, we will first transform N into an equivalent network, whose neurons have properties that will facilitate their merging. Equivalent here means that for every input vector, both networks produce the exact same output. First, each hidden neuron $v_{i,j}$ of our transformed network will be classified as either a `pos` neuron or a `neg` neuron. A neuron is `pos` if all the weights on its outgoing edges are positive, and is `neg` if all those weights are negative. Second, orthogonally to the `pos`/`neg` classification, each hidden neuron will also be classified as either an `inc` neuron or a `dec` neuron. $v_{i,j}$ is an `inc` neuron of N if, when we look at $N^{[i]}$ (where $v_{i,j}$ is an input neuron), increasing the value of $v_{i,j}$ increases the value of the network's output. Formally, $v_{i,j}$ is `inc` if for every two input vectors x_1 and x_2 where $x_1[k] = x_2[k]$ for $k \neq j$ and $x_1[j] > x_2[j]$, it holds that $N^{[i]}(x_1) > N^{[i]}(x_2)$. A `dec` neuron is defined symmetrically, so that *decreasing* the value of $x[j]$ *increases* the output. We first describe this transformation (an illustration of which appears in Fig. 3), and later we explain how it fits into our abstraction framework.

Our first step is to transform N into a new network, N', in which every hidden neuron is classified as `pos` or `neg`. This transformation is done by replacing each hidden neuron v_{i_j} with two neurons, $v^+_{i,j}$ and $v^-_{i,j}$, which are respectively `pos` and `dec`. Both $v^+_{i,j}$ an $v^-_{i,j}$ retain a copy of all incoming edges of the original $v_{i,j}$; however, $v^+_{i,j}$ retains just the outgoing edges with positive weights, and $v^-_{i,j}$ retains just those with negative weights. Outgoing edges with negative weights are removed from $v^+_{i,j}$ by setting their weights to 0, and the same is done for outgoing edges with positive weights for $v^-_{i,j}$. Formally, for every neuron $v_{i-1,p}$,

$$w'(v_{i-1,p}, v^+_{i,j}) = w(v_{i-1,p}, v_{i,j}), \qquad w'(v_{i-1,p}, v^-_{i,j}) = w(v_{i-1,p}, v_{i,j})$$

where w' represents the weights in the new network N'. Also, for every neuron $v_{i+1,q}$

$$w'(v^+_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$w'(v^-_{i,j}, v_{i+1,q}) = \begin{cases} w(v_{i,j}, v_{i+1,q}) & w(v_{i,j}, v_{i+1,q}) \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

(see Fig. 3). This operation is performed once for every hidden neuron of N, resulting in a network N' that is roughly double the size of N. Observe that N' is indeed equivalent to N, i.e. their outputs are always identical.

Our second step is to alter N' further, into a new network N'', where every hidden neuron is either `inc` or `dec` (in addition to already being `pos` or `neg`). Generating N'' from N' is performed by traversing the layers of N' backwards, each time handling a single layer and possibly doubling its number of neurons:

– Initial step: the output layer has a single neuron, y. This neuron is an `inc` node, because increasing its value will increase the network's output value.
– Iterative step: observe layer i, and suppose the nodes of layer $i+1$ have already been partitioned into `inc` and `dec` nodes. Observe a neuron $v^+_{i,j}$ in layer i which is marked `pos` (the case for `neg` is symmetrical). We replace $v^+_{i,j}$ with two neurons $v^{+,I}_{i,j}$ and $v^{+,D}_{i,j}$, which are `inc` and `dec`, respectively. Both new neurons retain a copy of all incoming edges of $v^+_{i,j}$; however, $v^{+,I}_{i,j}$

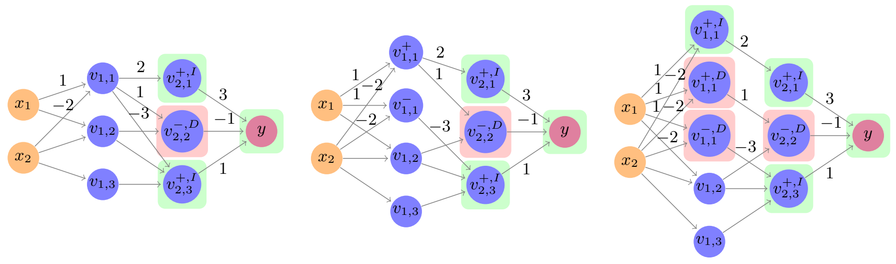

Fig. 3. Classifying neurons as `pos`/`dec` and `inc`/`dec`. In the initial network (left), the neurons of the second hidden layer are already classified: $^+$ and $^-$ superscripts indicate `pos` and `neg` neurons, respectively; the ^{I} superscript and green background indicate `inc`, and the ^{D} superscript and red background indicate `dec`. Classifying node $v_{1,1}$ is done by first splitting it into two nodes $v_{1,1}^+$ and $v_{1,1}^-$ (middle). Both nodes have identical incoming edges, but the outgoing edges of $v_{1,1}$ are partitioned between them, according to the sign of each edge's weight. In the last network (right), $v_{1,1}^+$ is split once more, into an `inc` node with outgoing edges only to other `inc` nodes, and a `dec` node with outgoing edges only to other `dec` nodes. Node $v_{1,1}$ is thus transformed into three nodes, each of which can finally be classified as `inc` or `dec`. Notice that in the worst case, each node is split into four nodes, although for $v_{1,1}$ three nodes were enough.

retains only outgoing edges that lead to `inc` nodes, and $v_{i,j}^{+,D}$ retains only outgoing edges that lead to `dec` nodes. Thus, for every $v_{i-1,p}$ and $v_{i+1,q}$,

$$w''(v_{i-1,p}, v_{i,j}^{+,I}) = w'(v_{i-1,p}, v_{i,j}^{+}), \qquad w''(v_{i-1,p}, v_{i,j}^{+,D}) = w'(v_{i-1,p}, v_{i,j}^{+})$$

$$w''(v_{i,j}^{+,I}, v_{i+1,q}) = \begin{cases} w'(v_{i,j}^{+}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{inc} \\ 0 & \text{otherwise} \end{cases}$$

$$w''(v_{i,j}^{+,D}, v_{i+1,q}) = \begin{cases} w'(v_{i,j}^{+}, v_{i+1,q}) & \text{if } v_{i+1,q} \text{ is } \texttt{dec} \\ 0 & \text{otherwise} \end{cases}$$

where w'' represents the weights in the new network N''. We perform this step for each neuron in layer i, resulting in neurons that are each classified as either `inc` or `dec`.

To understand the intuition behind this classification, recall that by our assumption all hidden nodes, including $v_{i,j}^+$, use the ReLU activation function and so take on only non-negative values. Because $v_{i,j}^+$ is `pos`, all its outgoing edges have positive weights, and so if its assignment was to increase (decrease), the assignments of all nodes to which it is connected in the following layer would also increase (decrease). Thus, we split $v_{i,j}^+$ in two, and make sure one copy, $v_{i,j}^{+,I}$, is only connected to nodes that need to increase (`inc` nodes), and that the other copy, $v_{i,j}^{+,D}$, is only connected to nodes that need to decrease (`dec` nodes). This ensures that $v_{i,j}^{+,I}$ is itself `inc`, and that $v_{i,j}^{+,D}$ is `dec`. Also, both $v_{i,j}^{+,I}$ and $v_{i,j}^{+,D}$ remain `pos` nodes, because their outgoing edges all have positive weights.

When this procedure terminates, N'' is equivalent to N', and so also to N; and N'' is double the size of N', and four times the size of N. Both transformation steps are only performed for hidden neurons, whereas the input and output neurons remain unchanged. This is summarized by the following lemma:

Lemma 1. *Any DNN N can be transformed into an equivalent network N'' where each hidden neuron is `pos` or `dec`, and also `inc` or `dec`, by increasing its number of neurons by a factor of at most* 4*.*

Using Lemma 1, we can assume without loss of generality that the DNN nodes in our input query φ_1 are each marked as `pos`/`neg` and as `inc`/`dec`. We are now ready to construct the over-approximation network $\bar{N}$. We do this by specifying an `abstract` operator that merges a pair of neurons in the network (thus reducing network size by one), and can be applied multiple times. The only restrictions are that the two neurons being merged need to be from the same hidden layer, and must share the same `pos`/`neg` and `inc`/`dec` attributes. Consequently, applying `abstract` to saturation will result in a network with at most 4 neurons in each hidden layer, which over-approximates the original network. This, of course, would be an immense reduction in the number of neurons for most reasonable input networks.

The `abstract` operator's behavior depends on the attributes of the neurons being merged. For simplicity, we will focus on the $\langle$`pos`,`inc`$\rangle$ case. Let $v_{i,j}$, $v_{i,k}$ be two hidden neurons of layer i, both classified as $\langle$`pos`,`inc`$\rangle$. Because layer i is hidden, we know that layers $i+1$ and $i-1$ are defined. Let $v_{i-1,p}$ and $v_{i+1,q}$ denote arbitrary neurons in the preceding and succeeding layer, respectively. We construct a network $\bar{N}$ that is identical to N, except that: (i) nodes $v_{i,j}$ and $v_{i,k}$ are removed and replaced with a new single node, $v_{i,t}$; and (ii) all edges that touched nodes $v_{i,j}$ or $v_{i,k}$ are removed, and other edges are untouched. Finally, we add new incoming and outgoing edges for the new node $v_{i,t}$ as follows:

- Incoming edges: $\bar{w}(v_{i-1,p}, v_{i,t}) = \max\{w(v_{i-1,p}, v_{i,j}), w(v_{i-1,p}, v_{i,k})\}$
- Outgoing edges: $\bar{w}(v_{i,t}, v_{i+1,q}) = w(v_{i,j}, v_{i+1,q}) + w(v_{i,k}, v_{i+1,q})$

where $\bar{w}$ represents the weights in the new network $\bar{N}$. An illustrative example appears in Fig. 4. Intuitively, this definition of `abstract` seeks to ensure that the new node $v_{i,t}$ always contributes more to the network's output than the two original nodes $v_{i,j}$ and $v_{i,k}$ — so that the new network produces a larger output than the original for every input. By the way we defined the incoming edges of the new neuron $v_{i,t}$, we are guaranteed that for every input x passed into both N and $\bar{N}$, the value assigned to $v_{i,t}$ in $\bar{N}$ is greater than the values assigned to both $v_{i,j}$ and $v_{i,k}$ in the original network. This works to our advantage, because $v_{i,j}$ and $v_{i,k}$ were both `inc` — so increasing their values increases the output value. By our definition of the outgoing edges, the values of any `inc` nodes in layer $i+1$ increase in $\bar{N}$ compared to N, and those of any `dec` nodes decrease. By definition, this means that the network's overall output increases.

The abstraction operation for the $\langle$`neg`,`inc`$\rangle$ case is identical to the one described above. For the remaining two cases, i.e. $\langle$`pos`,`dec`$\rangle$ and $\langle$`neg`,`dec`$\rangle$, the max operator in the definition is replaced with a min.

The next lemma (proof omitted due to lack of space) justifies the use of our abstraction step, and can be applied once per each application of `abstract`:

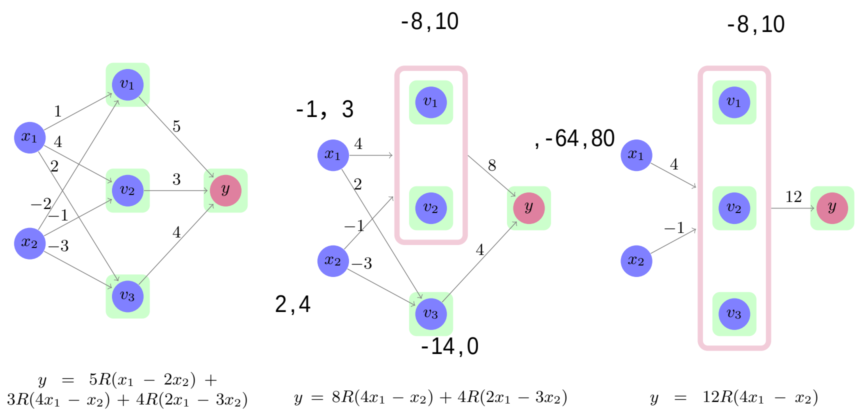

Fig. 4. Using `abstract` to merge ⟨`pos`,`inc`⟩ nodes. Initially (left), the three nodes v_1, v_2 and v_3 are separate. Next (middle), `abstract` merges v_1 and v_2 into a single node. For the edge between x_1 and the new abstract node we pick the weight 4, which is the maximal weight among edges from x_1 to v_1 and v_2. Likewise, the edge between x_2 and the abstract node has weight -1. The outgoing edge from the abstract node to y has weight 8, which is the sum of the weights of edges from v_1 and v_2 to y. Next, `abstract` is applied again to merge v_3 with the abstract node, and the weights are adjusted accordingly (right). With every abstraction, the value of y (given as a formula at the bottom of each DNN, where R represents the ReLU operator) increases. For example, to see that $12R(4x_1 - x_2) \geq 8R(4x_1 - x_2) + 4R(2x_1 - 3x_2)$, it is enough to see that $4R(4x_1 - x_2) \geq 4R(2x_1 - 3x_2)$, which holds because ReLU is a monotonically increasing function and x_1 and x_2 are non-negative (being, themselves, the output of ReLU nodes).

Lemma 2. *Let $\bar{N}$ be derived from N by a single application of* `abstract`*. For every x, it holds that $\bar{N}(x) \geq N(x)$.*

3.2 Refinement

The aforementioned `abstract` operator reduces network size by merging neurons, but at the cost of accuracy: whereas for some input x_0 the original network returns $N(x_0) = 3$, the over-approximation network $\bar{N}$ created by `abstract` might return $\bar{N}(x_0) = 5$. If our goal is prove that it is never the case that $N(x) > 10$, this over-approximation may be adequate: we can prove that always $\bar{N}(x) \leq 10$, and this will be enough. However, if our goal is to prove that it is never the case that $N(x) > 4$, the over-approximation is inadequate: it is possible that the property holds for N, but because $\bar{N}(x_0) = 5 > 4$, our verification procedure will return x_0 as a *spurious counterexample* (a counterexample for $\bar{N}$ that is not a counterexample for N). In order to handle this situation, we define a *refinement operator*, `refine`, that is the inverse of `abstract`: it transforms $\bar{N}$ into yet another over-approximation, $\bar{N}'$, with the property that for every x, $N(x) \leq \bar{N}'(x) \leq \bar{N}(x)$. If $\bar{N}'(x_0) = 3.5$, it might be a suitable over-approximation for showing that never $N(x) > 4$. In this section we define the `refine` operator, and in Section 4 we explain how to use `abstract` and `refine` as part of a CEGAR-based verification scheme.

Recall that `abstract` merges together a couple of neurons that share the same attributes. After a series of applications of `abstract`, each hidden layer i

of the resulting network can be regarded as a partitioning of hidden layer i of the original network, where each partition contains original, *concrete* neurons that share the same attributes. In the abstract network, each partition is represented by a single, *abstract* neuron. The weights on the incoming and outgoing edges of this abstract neuron are determined according to the definition of the `abstract` operator. For example, in the case of an abstract neuron $\bar{v}$ that represents a set of concrete neurons $\{v_1, \ldots, v_n\}$ all with attributes $\langle$`pos`,`inc`$\rangle$, the weight of each incoming edge to $\bar{v}$ is given by

$$\bar{w}(u, v) = \max(w(u, v_1), \ldots, w(u, v_n))$$

where u represents a neuron that has not been abstracted yet, and w is the weight function of the original network. The key point here is that the order of `abstract` operations that merged $v_1, \ldots, v_n$ does not matter — but rather, only the fact that they are now grouped together determines the abstract network's weights. The following corollary, which is a direct result of Lemma 2, establishes this connection between sequences of `abstract` applications and partitions:

Corollary 1. *Let N be a DNN where each hidden neuron is labeled as* `pos/neg` *and* `inc/dec`*, and let $\mathcal{P}$ be a partitioning of the hidden neurons of N, that only groups together hidden neurons from the same layer that share the same labels. Then N and $\mathcal{P}$ give rise to an abstract neural network $\bar{N}$, which is obtained by performing a series of* `abstract` *operations that group together neurons according to the partitions of $\mathcal{P}$. This $\bar{N}$ is an over-approximation of N.*

We now define a `refine` operation that is, in a sense, the inverse of `abstract`. `refine` takes as input a DNN $\bar{N}$ that was generated from N via a sequence of `abstract` operations, and splits a neuron from $\bar{N}$ in two. Formally, the operator receives the original network N, the partitioning $\mathcal{P}$, and a finer partition $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by splitting a single class in two. The operator then returns a new abstract network, $\bar{N}'$, that is the abstraction of N according to $\mathcal{P}'$.

Due to Corollary 1, and because $\bar{N}$ returned by `refine` corresponds to a partition $\mathcal{P}'$ of the hidden neurons of N, it is straightforward to show that $\bar{N}$ is indeed an over-approximation of N. The other useful property that we require is the following:

Lemma 3. *Let $\bar{N}$ be an abstraction of N, and let $\bar{N}'$ be a network obtained from $\bar{N}$ by applying a single* `refine` *step. Then for every input x it holds that $\bar{N}(x) \geq \bar{N}'(x) \geq N(x)$.*

The second part of the inequality, $\bar{N}'(x) \geq N(x)$ holds because $\bar{N}'$ is an over-approximation of N (Corollary 1). The first part of the inequality, $\bar{N}(x) \geq \bar{N}'(x)$, follows from the fact that $\bar{N}(x)$ can be obtained from $\bar{N}'(x)$ by a single application of `abstract`.

Of course, in practice, starting from the original N and applying a sequence of `abstract` operations is a wasteful way of implementing `refine`. Instead, it is possible to split an abstract neuron in two in a single step and examine just the concrete neurons that are mapped to the two new abstract neurons to determine the new edge weights.

4 A CEGAR-Based Approach

In Section 3 we defined the `abstract` operator that reduces network size at the cost of reducing network accuracy, and its inverse `refine` operator that increases network size and restores accuracy. Together with a black-box verification procedure *Verify* that can dispatch queries of the form $\varphi = \langle N, P, Q \rangle$, these components now allow us to design an abstraction-refinement algorithm for DNN verification, given as Alg. 1 (we assume that all hidden neurons in the input network have already been marked `pos`/`neg` and `inc`/`dec`).

Algorithm 1 Abstraction-based DNN Verification(N, P, Q)

```
1: Use abstract to generate an initial over-approximation $\bar{N}$ of $N$
2: if $Verify(\bar{N}, P, Q)$ is UNSAT then
3:    return UNSAT
4: else
5:    Extract counterexample $c$
6:    if $c$ is a counterexample for $N$ then
7:       return SAT
8:    else
9:       Use refine to refine $\bar{N}$ into $\bar{N}'$
10:      $\bar{N} \leftarrow \bar{N}'$
11:      Goto step 2
12:   end if
13: end if
```

Because our over-approximation network $\bar{N}$ is obtained via applications of `abstract` and `refine`, and because we assume the underlying *Verify* procedure is sound, Lemmas 2 and 3 guarantee the soundness of Alg. 1. Further, the algorithm always terminates: this is the case because all the `abstract` steps are performed first, followed by a sequence of `refine` steps. Because no additional `abstract` operations are performed beyond Step 1, after finitely many `refine` steps $\bar{N}$ will become identical to N, at which point no spurious counterexample will be found, and the algorithm will terminate with either `SAT` or `UNSAT`. Of course, termination is only guaranteed when the underlying *Verify* procedure is guaranteed to terminate.

There are two steps in the algorithm that we intentionally left ambiguous: Step 1, where the initial over-approximation is computed, and Step 9, where the current abstraction is refined due to the discovery of a spurious counterexample. The motivation was to make Alg. 1 general, and allow it to be customized by plugging in different heuristics for performing Steps 1 and 9, that may depend on the problem at hand. Below we propose a few such heuristics.

4.1 Generating an Initial Abstraction

The most naïve way to generate the initial abstraction is to apply the `abstract` operator to saturation. As previously discussed, `abstract` can merge together any pair of hidden neurons from a given layer that share the same attributes. Since there are four possible attribute combinations, this will result in each hidden layer of the network having four neurons or fewer. However, for a reasonably large DNN, we expect this abstraction to be very coarse, and so it might lead to multiple rounds of refinement before a `SAT` or `UNSAT` answer can be reached.

A different heuristic for producing abstractions that may require fewer refinement steps is as follows. First, we select a finite set of input points, $X = \{x_1, \ldots, x_n\}$, all of which satisfy the input property P. These points can be generated randomly, or according to some coverage criterion of the input space. The points of X are then used as indicators in estimating when the abstraction has become too coarse: after every abstraction step, we check whether the property still holds for $x_1, \ldots, x_n$, and stop abstracting if this is not the case. The exact technique appears in Alg. 2, which is used to perform Step 1 of Alg. 1.

Algorithm 2 Create Initial Abstraction(N, P, Q)

```
 1: $\bar{N} \leftarrow N$
 2: while $\forall x \in X.\ \bar{N}(x)$ satisfies $Q$ and there are still neurons that can be merged do
 3:     $\Delta \leftarrow \infty$, bestPair $\leftarrow \bot$
 4:     for every pair of hidden neurons $v_{i,j}, v_{i,k}$ with identical attributes do
 5:         m $\leftarrow 0$
 6:         for every node $v_{i-1,p}$ do
 7:             a $\leftarrow \bar{w}(v_{i-1,p}, v_{i,j})$, b $\leftarrow \bar{w}(v_{i-1,p}, v_{i,k})$
 8:             if $|a - b| >$ m then
 9:                 m $\leftarrow |a - b|$
10:             end if
11:         end for
12:         if m $< \Delta$ then
13:             $\Delta \leftarrow$ m, bestPair $\leftarrow \langle v_{i,j}, v_{i,k} \rangle$
14:         end if
15:     end for
16:     Use abstract to merge the nodes of bestPair, store the result in $\bar{N}$
17: end while
18: return $\bar{N}$
```

Another point that is address by Alg. 2, besides how many rounds of abstraction should be performed, is which pair of neurons should be merged in every application of `abstract`. This, too, is determined heuristically. Since any pair of neurons that we pick will result in the same reduction in network size, our strategy is to prefer neurons that will result in a a more accurate approximation. Inaccuracies are caused by the max and min operators within the `abstract` operator: e.g., in the case of max, every pair of incoming edges with weights a, b are replaced by a single edge with weight $\max(a, b)$. Our strategy here is to merge the pair of neurons for which the *maximal* value of $|a - b|$ (over all incoming edges with weights a and b) is *minimal*. Intuitively, this leads to $\max(a, b)$ being close to both a and b — which, in turn, leads to an over-approximation network that is smaller than the original, but is close to it weight-wise.

As a small example, consider the network depicted on the left hand side of Fig. 4. This network has three pairs of neurons that can be merged using `abstract` (any subset of $\{v_1, v_2, v_3\}$). Consider the pair v_1, v_2: the maximal value of $|a - b|$ for these neurons is $\max(|1 - 4)|, |(-2) - (-1)|) = 3$. For pair v_1, v_3, the maximal value is 1; and for pair v_2, v_3 the maximal value is 2. According to the strategy described in Alg. 2, we would first choose to apply `abstract` on the pair with the minimal maximal value, i.e. on the pair v_1, v_3.

4.2 Performing the Refinement Step

A refinement step is performed when a spurious counterexample x has been found, indicating that the abstract network is too coarse. In other words, our abstraction steps, and specifically the max and min operators that were used to select edge weights for the abstract neurons, have resulted in the abstract network's output being too great for input x, and we now need to reduce it. Thus, our refinement strategies are aimed at applying `refine` in a way that will result in a significant reduction to the abstract network's output.

One heuristic approach, which we refer to as *weight-based refinement*, is to look for a concrete neuron v, currently mapped into an abstract neuron $\bar{v}$, such that the incoming weights of v and $\bar{v}$ differ significantly. This indicates that by mapping v into $\bar{v}$ we have performed a coarse approximation, and that splitting v away from $\bar{v}$ may restore accuracy. This heuristic is formally defined in Alg. 3. The algorithm simply traverses the original neurons, looks for the edge weight that has changed the most as a result of the current abstraction, and then performs refinement on the neuron at the end of that edge.

Algorithm 3 Weight-Based Refinement($N, \bar{N}$)

```
1: bestNeuron ← ⊥, $m \leftarrow 0$
2: for each concrete neuron $v_{i,j}$ of $N$ mapped into abstract neuron $\bar{v}_{i,j'}$ of $\bar{N}$ do
3:   for each concrete neuron $v_{i-1,k}$ of $N$ mapped into abstract neuron $\bar{v}_{i-1,k'}$ of $\bar{N}$
     do
4:     if $|w(v_{i,j}, v_{i-1,k}) - \bar{w}(\bar{v}_{i,j'}, \bar{v}_{i-1,k'})| > m$ then
5:       $m \leftarrow |w(v_{i,j}, v_{i-1,k}) - \bar{w}(\bar{v}_{i,j'}, \bar{v}_{i-1,k'})|$
6:       bestNeuron ← $v_{i,j}$
7:     end if
8:   end for
9: end for
10: Use refine to split bestNeuron from its abstract neuron
```

As an example, let us use Alg. 3 to choose a refinement step for the right hand side network of Fig. 4. Suppose v_1 is considered first. In the abstract network, $\bar{w}(x_1, \bar{v_1}) = 4$ and $\bar{w}(x_2, \bar{v_1}) = -1$; whereas in the original network, $w(x_1, v_1) = 1$ and $w(x_2, v_1) = -2$. Thus, the largest value m computed for v_1 is $|w(x_1, v_1) - \bar{w}(x_1, \bar{v_1})| = 3$. This value of m is larger than the one computed for v_2 (0) and for v_3 (2), and so v_1 is selected for the refinement step. After this step is performed, v_2 and v_3 are still mapped to a single abstract neuron, whereas v_1 is mapped to a separate neuron in the abstract network.

Weight-based refinement attempts to reduce the output value y by as much as possible, but does not take into account the counterexample x that was discovered by the verification procedure. Consider a case where abstract neuron $\bar{v}$ is selected for refinement based on its incoming weights, but where $\bar{v}$ is actually assigned value 0 when the network is evaluated on counterexample x. By performing this refinement step, we might not change the network's output at all for x. Intuitively, by focusing our refinements efforts on neurons that take on small assignments in our input region of interest, we may be ignoring other choices that could decrease the network's output more significantly.

To address this situation, we propose to change Alg. 3 in a way that would make it counterexample-guided. Specifically, we suggest to adjust lines 4 and 5 of Alg. 3, by multiplying the term $|w(v_{i,j}, v_{i-1,k}) - \bar{w}(\bar{v}_{i,j'}, \bar{v}_{i-1,k'})|$ that appears therein by the value of neuron $\bar{v}$ when the abstract network is evaluated for counterexample x. This heuristic, which we refer to as *counterexample-guided refinement*, takes into account both edge weights and the discovered counterexample, and may ignore neurons that weight-based refinement might select if these neurons are assigned small values.

5 Implementation and Evaluation

The implementation of our abstraction-refinement framework includes scripts that read a DNN in the NNet format [13] and a property to be verified, create an initial abstract DNN as described in Section 4, invoke a black-box verification engine, and perform refinement as described in Section 4. The process terminates when the underlying engine returns either `UNSAT`, or an assignment that is a true counterexample for the original network. For experimentation purposes, we integrated our framework with the Marabou DNN verification engine [16].

Our experiments included verifying several properties of the 45 ACAS Xu DNNs for airborne collision avoidance [13,14]. Each of these networks has 300 hidden nodes spread across 6 layers, leading to 1200 neurons when the transformation from Section 3.1 is applied. Fig. 5 depicts the results of verifying these 45 networks using our approach, once with the naïve abstraction scheme in which each hidden layer initially has 4 neurons (x axis), and once when creating the initial abstraction according to Alg. 2 (y axis). The left plot depicts the number of refinement steps required before the procedure terminated. It shows that properties can indeed be proved on abstract networks that are significantly smaller than the original (i.e., without requiring many refinement steps that effectively restore the original network); and also that using Alg. 2 generally leads to fewer refinement steps than using the naïve scheme. The right plot indicates the total time (log-scale, in seconds, with a 20-hour timeout) spent by Marabou solving verification queries as part of the abstraction-refinement procedure. Interestingly, it shows that the naïve approach sometimes leads to faster query solving times — i.e., although the number of queries is greater, they are apparently easier for Marabou to solve.

Next, we compared the weight-based and counterexample-guided approaches for performing refinement steps discussed in Section 4.2. The results appear in Fig. 6. As before, we compared the number of required refinement steps (left plot) and the time required by Marabou to solve the generated queries (right plot). The results indicate the superiority of the counterexample-guided approach (x axis), especially in that it leads to far fewer timeouts.

Finally, we compared our abstraction-enhanced Marabou to the vanilla version, using the optimal configuration discovered in the previous experiments. The plot in Fig. 7 compares the total query solving time of vanilla Marabou

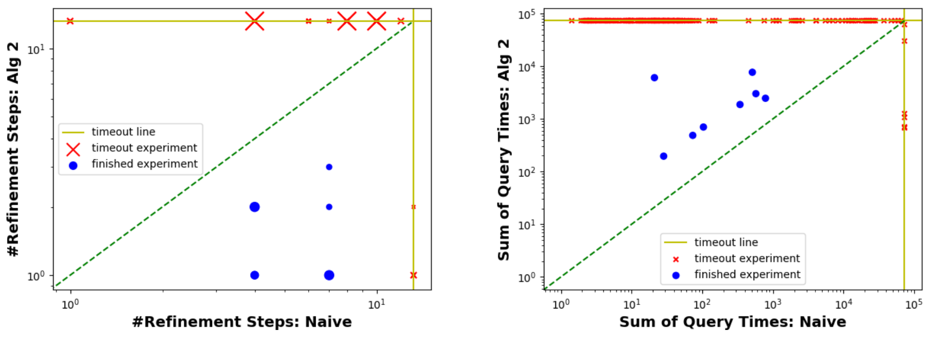

Fig. 5. Generating initial abstractions naïvely and using Alg. 2.

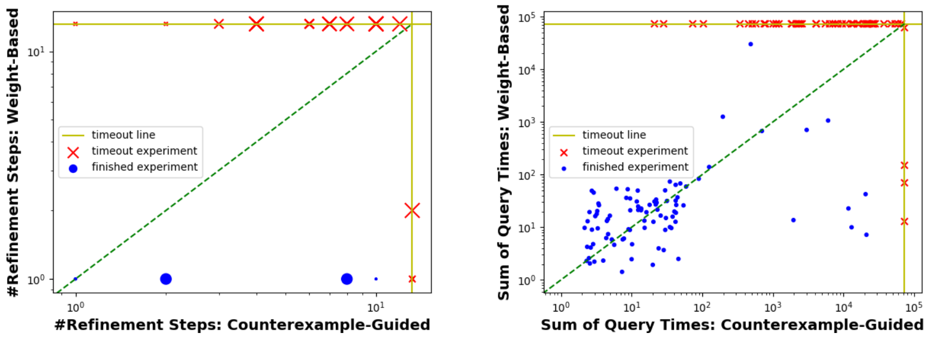

Fig. 6. Weight-based and counterexample-guided refinement steps.

(y axis) to that of our approach (x axis). We observe that despite having to verify networks that are approximately four times larger (due to preprocessing), the abstraction-enhanced version significantly outperforms vanilla Marabou on average — often solving queries orders-of-magnitude more quickly. These results clearly indicate the usefulness of combining our technique with an existing verification engine, in order to boost its performance.

6 Related Work

In recent years, multiple schemes have been proposed for the verification of neural networks. Besides Marabou [16,17], these include Marabou's predecessor Reluplex [14,19], the Planet solver [6], the BaB solver [2], the Sherlock solver [5], the ReluVal solver [30], and others (e.g., [21,24,29,31]). Other approaches use sound but incomplete strategies, such as input space discretization [12] or abstract interpretation [7]. Our approach can be integrated with any sound and complete solver as its engine; incomplete approaches could also be used and might afford better performance, but could result in non-termination.

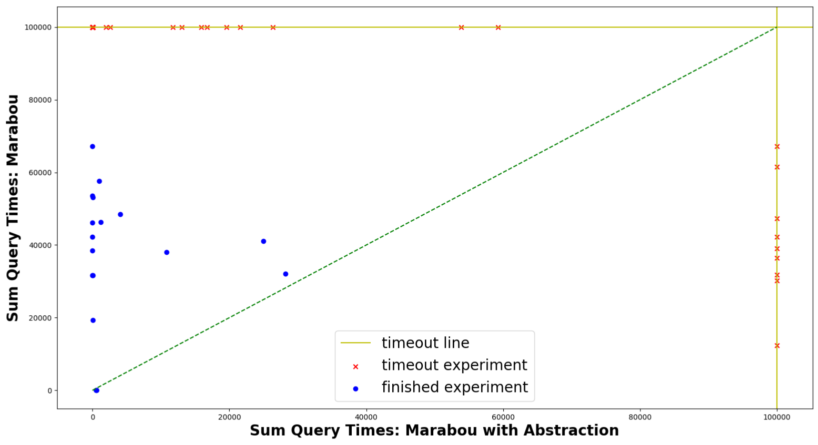

Fig. 7. Comparing the run time (in seconds) of vanilla Marabou and the abstraction-enhanced version.

Some existing DNN verification techniques incorporate abstraction elements. In [25], the authors use abstraction to over-approximate the Sigmoid activation function with a collection of rectangles. If the abstract verification query they produce is `UNSAT`, then so is the original. When a spurious counterexample is found, an arbitrary refinement step is performed. The authors report limited scalability, tackling only networks with a few dozen neurons. Abstraction techniques also appear in the AI2 approach [7], but there it is the input property and reachable regions that are over-approximated, as opposed to the DNN itself.

7 Conclusion

With deep neural networks becoming widespread and with their forthcoming integration into safety-critical systems, there is an urgent need for scalable techniques to verify and reason about them. However, the size of these networks poses a serious challenge. Abstraction-based techniques can mitigate this difficulty, by replacing networks with smaller versions thereof to be verified, without compromising the soundness of the verification procedure. The abstraction-based approach we have proposed here can provide a significant reduction in network size, thus boosting the performance of existing verification technology.

In the future, we plan to continue this work along several axes. First, we intend to investigate refinement heuristics that can split an abstract neuron into two arbitrary sized neurons. In addition, we will investigate abstraction schemes for networks that use additional activation functions, beyond ReLUs. Finally, we plan to make our abstraction scheme parallelizable, allowing users to use multiple worker nodes to explore different combinations of abstraction and refinement steps, hopefully leading to faster convergence.

Acknowledgements. This project was partially supported by grants from the Binational Science Foundation (2017662) and the Israel Science Foundation (683/18).

References

1. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. `http://arxiv.org/abs/1604.07316`.
2. R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and M. Kumar. Piecewise Linear Neural Network Verification: A Comparative Study, 2017. Technical Report. `https://arxiv.org/abs/1711.00455v1`.
3. N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. `https://arxiv.org/abs/1709.10207`.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2010.
5. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari. Output Range Analysis for Deep Neural Networks. In *Proc. 10th NASA Formal Methods Symposium (NFM)*, pages 121–138, 2018.
6. 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.
7. 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.
8. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
9. 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.
10. J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. Tenenbaum, and T. Mattson. The Three Pillars of Machine Programming. In *Proc. 2nd ACM SIGPLAN Int. Workshop on Machine Learning and Programming Languages (MALP)*, pages 69–80, 2018.
11. G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
12. 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.
13. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.
14. 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.

15. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Towards Proving the Adversarial Robustness of Deep Neural Networks. In *Proc. 1st Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, pages 19–26, 2017.
16. 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)*, 2019. To appear.
17. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, 2019.
18. A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.
19. L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. https://arxiv.org/abs/1801.05950.
20. A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial Examples in the Physical World, 2016. Technical Report. http://arxiv.org/abs/1607.02533.
21. A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. https://arxiv.org/abs/1706.07351.
22. H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proc. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 197–210, 2017.
23. V. Nair and G. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML)*, pages 807–814, 2010.
24. N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh. Verifying Properties of Binarized Deep Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1709.06662.
25. 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.
26. W. Ruan, X. Huang, and M. Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proc. 27th Int. Joing Conf. on Artificial Intelligence (IJACI)*, pages 2651–2659, 2018.
27. D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.
28. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.
29. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *Proc. 7th Int. Conf. on Learning Representations (ICLR)*, 2019.
30. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.
31. W. Xiang, H.-D. Tran, and T. Johnson. Output Reachable Set Estimation and Verification for Multilayer Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 99:1–7, 2018.