Pruning and Slicing Neural Networks using Formal Verification

Ori Lahav [iD] and Guy Katz [iD]

The Hebrew University of Jerusalem, Jerusalem, Israel

{ori.lahav, guykatz}@cs.huji.ac.il

Abstract—Deep neural networks (DNNs) play an increasingly important role in various computer systems. In order to create these networks, engineers typically specify a desired topology, and then use an automated training algorithm to select the network's weights. While training algorithms have been studied extensively and are well understood, the selection of topology remains a form of art, and can often result in networks that are unnecessarily large — and consequently are incompatible with end devices that have limited memory, battery or computational power. Here, we propose to address this challenge by harnessing recent advances in DNN verification. We present a framework and a methodology for discovering redundancies in DNNs — i.e., for finding neurons that are not needed, and can be removed in order to reduce the size of the DNN. By using sound verification techniques, we can formally guarantee that our simplified network is equivalent to the original, either completely, or up to a prescribed tolerance. Further, we show how to combine our technique with *slicing*, which results in a *family* of very small DNNs, which are together equivalent to the original. Our approach can produce DNNs that are significantly smaller than the original, rendering them suitable for deployment on additional kinds of systems, and even more amenable to subsequent formal verification. We provide a proof-of-concept implementation of our approach, and use it to evaluate our techniques on several real-world DNNs.

I. INTRODUCTION

The wide-spread adoption of *deep learning* [17] has caused a significant leap forward in many domains within computer science. *Deep neural networks* (*DNNs*) have now become the state of the art solution for a myriad of real-world problems, such as game playing [40], image recognition [41], and autonomous vehicles [5], [25]. This trend is likely to continue and intensify, thus creating an urgent need for tools and techniques to analyze and manipulate DNNs.

A part of the appeal of DNNs is that they are produced in a mostly automated way. In order to create a DNN for a particular task at hand, engineers first specify the network architecture — specifically, the number of layers in the network, the size and type of each layer, and the inter-layer connections. Then, they invoke an automated training algorithm for assigning weights to the network's edges [17]. While the automated training process has been extensively studied and is generally well understood [17], the choice of network architecture is still performed according to various rules of thumb, and is considered a form of art. This can often lead to a choice of architecture that is wasteful — i.e., which results in a large DNN, whereas a smaller DNN could have achieved similar accuracy [15], [19], [23]. For DNNs intended to run on devices

with limited resources (e.g., mobile phones, or embedded circuits), excessive DNN size can be a limiting factor [25].

One successful approach for mitigating this difficulty is to first train a large network, and then shrink it by removing *redundant neurons*. Informally, we say that a neuron is redundant if removing it does not change the DNN's output; and thus, removing it from a network N results in a smaller network, N', that is *equivalent* to N. In order to identify redundant neurons within a DNN, prior work has focused primarily on *heuristic pruning*: heuristically identifying neurons and edges that contribute little to the network's output, removing these neurons, and then performing additional training of the network [19], [23]. These methods have been highly successful in reducing DNN sizes, but they provide no formal guarantees; i.e., the removed neurons are not guaranteed to have been redundant, and the simplified network can thus be dramatically different from the original, producing different results for various inputs [35].

Recently, there has been a surge of interest in the formal verification of neural networks (e.g., [2], [14], [20], [26], [28], [32], [46], and many others). These new capabilities have made it possible to identify and remove redundancies in a network, in a way that *guarantees* that the smaller network is completely equivalent to the original [15]. Specifically, Gokulanathan et al. showed how verification could be used to identify and remove "dead" neurons, i.e. neurons whose output is 0 regardless of the network's inputs. This approach was shown to reduce network sizes by up to 10%, which is quite significant, while preserving complete equivalence to the original network.

Here, we propose a new technique, which also attempts to apply formal verification in order to remove neurons from a DNN, but which is significantly stronger. Specifically, our technique: (i) can identify additional kinds of redundant neurons (beyond "dead" neurons), whose removal does not affect the network's outputs at all; and (ii) can identify additional redundant neurons, whose removal *does* affect the network's outputs, but only up to a small, provable bound.

Finally, we propose a method that takes our approach to the extreme, by integrating it with *network slicing*. This method, in which a network is simplified into a family of much smaller sub-networks, is appropriate for cases where fast inference is crucial: an input is checked to identify the appropriate sub-network for handling it, and then only that network needs to be evaluated for that specific input. Slicing is achieved by

partitioning the DNN's input domain into small sub-domains, maintaining a separate DNN for each input sub-domain, and then applying the aforementioned simplification techniques on each of these DNNs. We demonstrate that the use of small input sub-domains causes many neurons to become redundant, and consequently removable.

For evaluation purposes, we implemented our approach in an open-source, publicly available tool [33]. As a backend, our tool uses the Marabou DNN verification tool [29]. We note, however, that our approach is agnostic of the underlying verification engine — indeed, it could be integrated with any other tool, and will consequently benefit from any development in DNN verification technology. We evaluated our approach on a set of airborne collision avoidance networks [25], obtaining highly favorable results. Specifically, we were able to achieve a reduction of up to 71% in overall network sizes, while keeping the outputs identical (up to a prescribed tolerance) to those produced by the original DNN. This reduction in network sizes is a significant improvement over the previous state of the art [15]. Further, while prior techniques were specifically tailored to networks with only a specific activation function (i.e., rectified linear units [15]), our technique is applicable to multiple kinds of DNNs.

The rest of this paper is organized as follows. In Section II, we provide the necessary background on DNNs and their verification. Next, in Section III we present the basic building block of our approach, namely the removal of a single neuron. We then specify multiple kinds of neurons that can be removed in Section IV, and discuss the simultaneous removal of neurons in Section V. Subsequently, in Section VI we present how *input slicing* and simplification can be used to improve network evaluation time. An evaluation appears in Section VII, followed by a discussion of related work in Section VIII. We then conclude in Section IX.

II. BACKGROUND: DNNs AND THEIR VERIFICATION

A deep neural network [17] is a directed, acyclic graph, whose nodes (also referred to as *neurons*) are grouped into layers. The first layer is the *input layer*; the final layer is the *output layer*; and the intermediate layers are the *hidden layers*. When the network is evaluated, the input neurons are assigned some values (e.g., sensor readings), and these values are then propagated through the network, layer by layer, until the output values are computed. In *regression* networks, the numeric value of the output is of interest, while in the case of *classification* networks, the output neurons correspond to possible *labels* that the network can classify the input into; and the label whose neuron obtained the highest score is the one returned by the network.

Each layer in the DNN has a type, which determines how its neuron values are computed. Here, we will focus on two types: *weighted sum* layers, and *piecewise-linear activation* layers. In a weighted-sum layer, the value of a neuron y is computed as $y = b + \sum c_i v_i$ for neurons v_i from preceding layers, where the *weights* c_i are determined when the network

is first trained. In a piecewise-linear activation layer, the value of neuron y is computed as

$$y = \begin{cases} a_1 x + b_1 & \text{if } s_1 \le x < s_2, \\ a_2 x + b_2 & \text{if } s_2 \le x < s_3, \\ \dots \\ a_k x + b_k & \text{if } s_k \le x \le s_{k+1} \end{cases}$$

where x is a neuron from some preceding layer, and the a_i, b_i and s_i parameters determine the piecewise linear function being computed. A common example of a piecewise-linear activation function is the ReLU function, given by

$$y = \max(x, 0) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \ge 0 \end{cases}$$
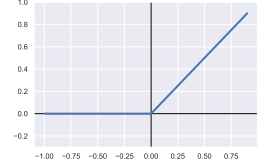

Fig. 1: The ReLU function.

(see Fig. 1). Together, weighted-sum layers and piecewise-linear activation functions make up many common DNN architectures [17]. Typically, they are used in alternation (see Fig. 2). Extending our approach to activation functions that are not piecewise-linear remains a work in progress.
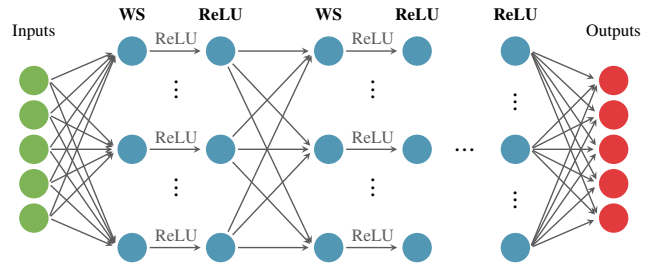

Fig. 2: An illustration of a DNN with alternating weighted-sum (WS) and ReLU layers.

More formally, we regard a DNN N with k inputs and m outputs as a mapping $\mathbb{R}^k \to \mathbb{R}^m$. The DNN is given as a sequence of layers $L_1, \dots, L_n$, where L_1 is the input layer and L_n is the output layer. We use s_i to denote the size of layer L_i, and use $v_i^1, \dots, v_i^{s_i}$ to refer to the individual neurons of L_i. We use V_i to refer to the column vector $[v_i^1, \dots, v_i^{s_i}]^T$. When the network is being evaluated, we assume that the input values V_1 are given, and that $V_2, \dots, V_n$ are computed iteratively. The type of each hidden layer is given via the mapping $T_N : \mathbb{N} \to \mathcal{T}$. For simplicity we set $\mathcal{T} = \{\text{weighted-sum}, \text{ReLU}\}$, although our technique applies to all types of piecewise-linear activation functions.

In a weighted-sum layer L_i, each neuron v_i^j is associated with a linear function $v_i^j = b_i^j + \sum c_{l,t} \cdot v_l^t$; i.e., v_i^j is computed as a weighted-sum of neurons v_l^t from preceding layers $l < i$, plus a bias value b_i^j. In a ReLU layer L_i, each neuron v_i^j is associated with a specific neuron v_l^t from a preceding layer $l < i$, and its value is given by $v_i^j = \text{ReLU}(v_l^t) = \max(v_l^t, 0)$. Note that each neuron's value depends only on neurons from preceding layers.

In recent years, various security and safety issues have been discovered in DNNs [26], [43]. This has led the verification community to study the *DNN verification problem* [36]. Generally, this problem is defined by a set of constraints P on the DNN's inputs, and a set of constraints Q on the DNN's outputs; and solving it entails finding (or proving the non-existence of) an input x such that $P(x) \wedge Q(N(x))$; i.e., an input x that satisfies the input condition, and is mapped by the DNN to a point that satisfies the output condition. When P and Q characterize an unsafe behavior of the DNN, an UNSAT answer to the aforementioned query indicates that the DNN is safe; whereas a SAT answer, accompanied by a satisfying assignment, demonstrates an unsafe behavior. This formalization is sufficiently expressive for capturing many properties of interest [26]. Many approaches for solving the DNN verification problem have been proposed recently (e.g., [14], [20], [26], [46], and many others). The techniques we discuss in this work use a DNN verification engine as a backend, and do not depend on the precise method used — and so we do not elaborate on this topic. We refer the interested reader to [36] for a survey.

III. REMOVING A SINGLE NEURON

The core of our DNN simplification approach is the identification, and then the removal, of *redundant neurons*. Given a DNN N, we seek to identify a redundant neuron v_i^j, and then produce another network, N', which is identical to N except for the redundant neuron that has been removed. Ideally, we would like to ensure that N and N' are equivalent; i.e., that $\forall x.N(x) = N'(x)$. Because N' is obtained from N by removing a neuron, it is smaller; and this process can be repeated iteratively, to eventually obtain a significantly smaller network that is equivalent to N. Of course, the key points that need addressing are: (i) how to technically remove a redundant neuron from the network; and (ii) how to identify redundant neurons. In this section we focus on the first challenge, and describe the mechanics of removing a neuron.

In order to maintain compatibility with the original network, we will refrain from removing neurons from the network's input or output layers; all other neurons are considered candidates for removal. We distinguish between neurons in weighted-sum layers, and neurons in activation function layers. In fact, our proposed approach only supports the removal of weighted-sum neurons that feed only into other weighted-sum neurons; and the removal of activation function neurons will be performed by first transforming them into weighted-sum neurons, as described in later sections.

Consider a neuron v computed as a weighted-sum

$$v = b_v + \sum c_i \cdot x_i,$$

where x_i are neurons from preceding layers. Suppose that v only feeds into other weighted-sum neurons, and let u be such a neuron:

$$u = b_u + c \cdot v + \sum d_i \cdot y_i,$$

where y_i are again neurons from preceding layers. In this case, u's equation can be updated into:

$$u = (b_u + c \cdot b_v) + \sum c \cdot c_i \cdot x_i + \sum d_i \cdot y_i.$$

If this process is repeated for every (weighted-sum) neuron that v feeds into, then afterwards v will have no outgoing edges. Consequently, v could then be eliminated from the network altogether. It is straightforward to show that such an operation will never affect the value of u, and that the modified network will thus be completely equivalent to the original. Also, identifying neurons that can be eliminated is simple, and amounts to searching for weighted-sum neurons that are only connected to other weighted-sum neurons.

In practice, DNN topology usually alternates between weighted-sum and activation function layers, and so consecutive weighted-sum neurons are likely to be scarce. Our strategy will thus be to replace activation function neurons with weighted-sum neurons, in a way that will enable neuron removal while preserving network accuracy. As an example, let us consider a ReLU neuron, $y = \text{ReLU}(x)$. Because of layer-type alternation, it is reasonable to assume that x is a weighted-sum neuron. In this case, if we can express y as a linear function of x, i.e. $y = ax + b$ for some a and b, then the previous case of two consecutive weighted-sum neurons applies: we can remove x entirely, change y's type to weighted-sum, and connect y to x's inputs. Further, if y also feeds into weighted-sum neurons, then we can apply simplification once again, and remove y as well. An illustration appears in Fig. 3.
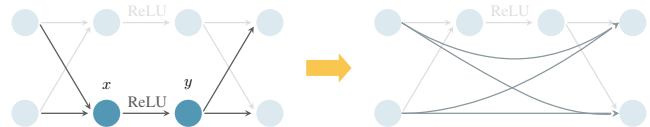

Fig. 3: Illustration: removing a neuron. x is a weighted-sum neuron which feeds into y, a ReLU neuron. After converting y into a weighted-sum neuron, both x and y can be removed.

The aforementioned steps constitute the framework of our approach — to repeat, until saturation, the two steps: (i) identify any weighted-sum neurons that only feed into weighted sum neurons, and remove them; and (ii) identify any activation function neurons that can be changed into weighted-sum neurons, without harming the network's accuracy. The key remaining issue is how to identify those neurons to which step 2 can be applied. We elaborate on this issue in the following sections.

IV. LINEARIZING ACTIVATION FUNCTIONS

We next propose various criteria for determining which activation function neuron can be changed into weighted-sum neurons. Applying these criteria in practice is discussed later, in Section V.

Phase Redundancy. In order to transform an activation function neuron into a weighted-sum neuron without changing the network's outputs, we leverage the properties of

piecewise-linear functions. Let x be a weighted-sum neuron and let $y = f(x)$ be an activation function neuron; then, by definition, the value range of x is divided into segments $[s_1, s_2], [s_2, s_3], \ldots [s_k, s_{k+1}]$, and in each segment y is a linear function (a weighted-sum) of x. If we are able to discover that x is in fact restricted to one of these segments, i.e. $s_i \leq x < s_{i+1}$ for some i, then we can safely discard the constraint $y = f(x)$ and replace it with a linear constraint $y = a_i x + b_i$, thus changing y to be a weighted-sum neuron. We stress that this change does not alter the value of y, and consequently does not alter the network's outputs. When this phenomenon occurs, we say that y is *phase-redundant*. For the ReLU function, this happens if we discover that $x < 0$ (y is *inactive-redundant*), or $x \geq 0$ (y is *active-redundant*). As previously stated, transforming the piecewise-linear constraint into a linear one will often allow us to eliminate two neurons from the network, without changing its outputs.

Forward Redundancy. Phase-redundancy captures the case where an activation function neuron is fixed to a single linear phase, for all possible inputs. However, there actually exist *unstable* activation-function neurons, i.e. neurons not fixed to a particular linear phase, which can still be soundly transformed into weighted-sum neurons computing one of these linear phases. Intuitively, this happens when neuron y's assignment affects its k succeeding layers, for some $k > 0$, but gets "canceled out" in layer $k + 1$. A small, illustrative example appears in Fig. 4. When replacing y with a weighted-sum neuron only affects neurons that are at most k layers away from y, we say that y is *k-forward-redundant*. Much like phase-redundant neurons, k-forward-redundant neurons can be removed from the network without harming its accuracy.
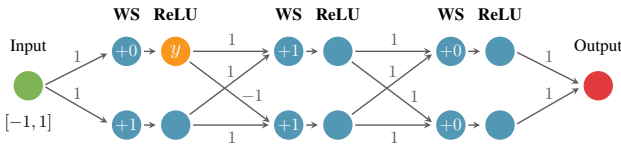
Fig. 4: The **orange** ReLU neuron, marked y, is *2-forward-redundant*. Replacing y with a constant zero affects the following WS and ReLU layers, but it does not affect the last WS layer (and thus the network output). For example, observe that if we input 1 into the network, y evaluates to 1, and the network's output evaluates to 12. This output value is unchanged even if we replace y's value with 0. A careful examination of the network reveals that this will always be the case, regardless of the network's input value.

More formally, let v_i^j be an activation function neuron, and let N' be a network obtained from N replacing v with a weighted-sum neuron $v_i^j = b_i^j + \sum c_k x_k$. Let V_1 denote an input vector, on which both N and N' are evaluated; and let $V_2, \ldots, V_n$ and $V_2', \ldots, V_n'$ denote the layer evaluations of N and N' (respectively) on V_1. If, for every V_1, it holds that $V_{i+k} = V_{i+k}'$, then we say that neuron v_i^j is k-forward-redundant (note that this implies $V_{i+k'} = V_{i+k'}'$ for every $k' > k$). We note that a neuron that is phase-redundant is also k-forward-redundant, for any $k \geq 0$.

Relaxed Redundancy. So far, we discussed replacing a piecewise-linear activation neuron with a weighted-sum neuron that corresponds to one of the activation function's linear segments; e.g., in the case of $y = \text{ReLU}(x)$, neuron y would be changed into a weighted-sum neuron computing either $y = 0$ or $y = x$. We observe that, although these linear functions are natural candidates for replacing the original constraint, in fact any linear function $y = \ell(x)$ could be used. Specifically, given an activation function $y = f(x)$ and some known lower and upper bounds lb and ub for x (computed, e.g., using interval arithmetic [26] or abstract interpretation [14], [46]), we propose to find a linear function $\ell(x)$ that has *minimal error* compared to $f(x)$. We define this error to be

$$\max_{lb \leq x \leq ub} |f(x) - \ell(x)|$$

See Fig. 5 for an illustration of replacing a ReLU constraint, whose phase is not fixed, with three linear constraints. In each illustration, the blue line is the ReLU, the dashed line is the linear replacement, and the red area is the introduced error. In case (c), the maximal introduced error (the height of the red region) is the smallest among the three options.
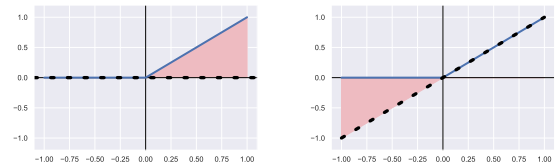
(a) Replacing a ReLU with the zero function

(b) Replacing a ReLU with identity function
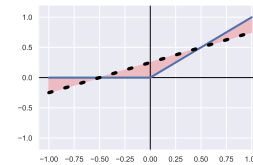
(c) Replacing a ReLU with an arbitrary linear function

Fig. 5: Replacing a ReLU with linear functions.

Unlike in the phase-redundancy and k-forward-redundancy cases, setting $y = \ell(x)$ will introduce some imprecision to the network's output. The motivation is that by replacing $y = f(x)$ with $y = \ell(x)$ that has minimal error, we would be introducing only a small imprecision, while enabling the removal of y. Let e_t be some user-defined error threshold; when replacing $y = f(x)$ with $\ell(x)$ introduces an error e such that $e \leq e_t$, we say that neuron y is *relaxed-redundant*.

Let us focus on the $y = \text{ReLU}(x)$ function as an example, and suppose we know that $x \in [lb, ub]$. If $lb < 0$ and $ub > 0$, the neuron is not phase-redundant. In this case, a linear function $y = l_m(x)$ with minimal error can be easily

computed, and is given by:

$$l_m(x) = \frac{ub}{ub - lb} \cdot x + \frac{-lb \cdot ub}{2(ub - lb)}.$$

It is straightforward to check that the maximum error is obtained when $x = 0$, and it is given by $\frac{-lb \cdot ub}{2(ub - lb)}$ (a proof appears in Appendix 1 in the extended version of this paper [34]). Unsurprisingly, when lb or ub are close to 0, the error becomes very small — indicating that such ReLUs, which are "almost phase-redundant", could be removed at a small cost to precision. It should be noted, however, that minimizing the maximum error introduced by the removal of a single neuron does not necessarily minimize the overall imprecision introduced to the network's outputs.

Result-Preserving Redundancy. In classification networks, it may be acceptable to give up some precision, as long as the output label for each input is unchanged; i.e., if the original network classified input x as label l with 80% confidence, it may be acceptable to remove neurons in a way that reduces this confidence to 60%, as long as x is still classified as l.

More formally, let $y = f(x)$ be an activation neuron in a network N, and let N' denote the same network with y replaced by a weighted sum neuron, $y = \ell(x)$. If, for every input vector V_1, it holds that $\text{argmax}(V_n) = \text{argmax}(V_n')$, i.e. if both networks classify each input vector in the same way (regardless of the actual output neuron values computed), then we say that neuron y is *result-preserving redundant*. See Fig. 6 for an example.
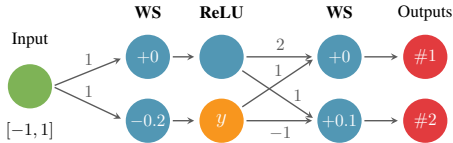

Fig. 6: The **orange** ReLU, marked y, is result-preserving redundant and can be replaced with a constant zero. Observe that any input in range $(0.1, 1]$ is classified as label $\#1$, while any input in range $[-1, 0.1)$ is classified as label $\#2$. The ReLU in **orange** is active only for inputs in $(0.2, 1]$, and it only increases the confidence in label $\#1$. For example, the network output for input 0.5 is $[1.3, 0.3]^T$, and after replacing y with 0 the output becomes $[1.0, 0.6]^T$. Label $\#1$ still wins, but with a lower confidence. Thus, y is result-preserving redundant — replacing it with a constant zero does not change the winning class, for the entire input domain.

Note that result-preserving redundancy is, in a way, more permissive than the previous categories: we do not directly try to bound the imprecision introduced, but rather only try to maintain the same output *label* for every input. Clearly, any neuron that is phase-redundant or k-forward-redundant is also result-preserving; and it is reasonable to assume that relaxed-redundant neurons with a small error would also be result-preserving redundant. The motivation for considering this kind of redundancy is that, due to its more permissive nature, it can identify additional redundant neurons.

Our definition of result-preserving redundancy can also be slightly relaxed, to exclude inputs whose classification was

borderline; i.e., inputs whose highest-scored label and the second-highest label received very similar scores. Intuitively, with this alteration, a neuron is considered result-preserving redundant if it does not change the classification of any inputs which were previously classified with a high degree of confidence, but may flip the classification of inputs about which the DNN was not sure to begin with. The motivation for this change is to allow the removal of additional neurons.

V. Neuron Removal Strategies

In Section III we laid the theoretical foundations of our DNN simplification approach, by defining four kinds of redundant neurons that could be removed to reduce network size. There exist many strategies for applying these definitions in practice, in order to reduce network sizes. Intuitively, a good strategy is one that identifies large sets of neurons that can be removed simultaneously, in a way that is computationally efficient. In this Section, we propose one such strategy, which we have empirically observed to perform well.

Step 1: Bound Estimation using MILP. Let v be an activation function neuron which we are considering for removal. In this context, it is useful to deduce lower and upper bounds for v that are as tight as possible. Such bounds could lead, for example, to the classification of v as phase-redundant, or enable us to compute $l_m(v)$ and declare v to be relaxed-redundant.

Mixed-Integer Linear Programming (MILP) [9] is a well-studied method for solving a system of linear constraints with real and integer variables. In the context of DNN verification, MILP can be used to derive lower and upper bounds on the values that the various neurons in the DNN can obtain [10], [44]. This is done by encoding a linear over-approximation of the neural network into the MILP solver, and then using the solver's objective function to maximize/minimize each of the individual neurons. For example, after encoding a network N, we could set the solver's objective function to $1 \cdot v$, where v is some neuron in N; and the optimal solution discovered would then constitute v's upper bound.

As a first step in the simplification process, we propose to run such MILP queries for every neuron that is candidate for removal. The number of resulting queries can be large — two queries per neuron, one for each bound — but the gains are significant, as the discovered bounds can often be quite tight [44]. At the end of this step, we immediately remove all phase-redundant neurons.

In practice, it is useful to run the MILP solver with a short timeout (e.g., 10 second) for each neuron. In case a timeout occurs, modern solvers are able to provide a sound approximation of the optimal solution [38]. In our experiments, we observed that this initial step already detects a large number of phase-redundant neurons.

Step 2: Simulations. After the MILP phase is concluded, we are left with multiple activation-function neurons whose phases are not yet fixed. It is possible that some of these neurons are also phase-redundant, but that the bounds discovered

in the MILP pass were too loose to indicate this. It is also possible that they are k-forward-redundant or result-preserving redundant. At this point we wish to quickly *rule out* as many of these candidates as possible, before applying computationally expensive steps to dispatch the remaining candidates.

To do this, we follow in the footsteps of Gokulanathan et al. [15], and apply *simulations*; i.e., we evaluate the network on a large number of random inputs, and for each input record the values assigned to the network's neurons. Simulations can easily show that a neuron is not phase-redundant, by demonstrating two different inputs for which the neuron is in two different linear phases. Similarly, they can show that a neuron is not k-forward-redundant or result-preserving redundant.

Step 3: Formal Verification. After the MILP and simulation phases, we are left with activation-function neurons that are candidates for removal, if we can prove them redundant. We now apply formal verification to classify these remaining neurons. Specifically, for each candidate neuron v, we: (i) apply verification to check whether v is fixed to one if its linear phases, and is hence phase-redundant; and if not, (ii) if N is a classification network, apply verification to check whether v is result-preserving redundant; else, if N is a regression network, apply verification to check whether v is k-forward-redundant, for a value of k that corresponds to the output layer. Each of these conditions can be posed as a DNN verification query, as described next. As soon as a neuron is marked redundant, it is removed, and the process continues.

In order to determine whether $v = f(x)$ is phase-redundant, we must check whether x is restricted to a certain linear segment. Let $[s_1, s_2], [s_2, s_3], \ldots [s_k, s_{k+1}]$ be the set of possible segments. For each such segment $[s_i, s_{i+1}]$, we can encode the DNN into the verifier, and pose the query: $\exists V_1.(x < s_i) \lor (x > s_{i+1})$. If the answer is UNSAT, we know that x is indeed fixed into segment $[s_i, s_{i+1}]$. An illustration appears in Fig. 7.
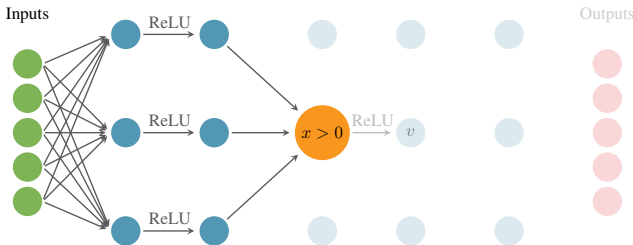

Fig. 7: A query for determining whether ReLU node $v = \text{ReLU}(x)$ is *phase-redundant*: we check whether it is possible that $x > 0$, and if not, we conclude that v is inactive-redundant. To facilitate the verification process, the neurons in subsequent layers, as well as all other neurons in layer 2 (grayed out), are not encoded.

Determining whether $v = f(x)$ is k-forward-redundant is done by creating a query where the part of the network starting from the neuron in question is duplicated. One copy of the network is the unmodified one, and in the other copy $v = f(x)$ is replaced with a weighted-sum neuron, $v' = \ell(x)$. We query the verifier whether it is possible that a neuron k layers away from v is assigned different values in the original and

modified copies. If the answer is UNSAT, the neuron is k-forward-redundant. See Fig. 8 for an illustration.
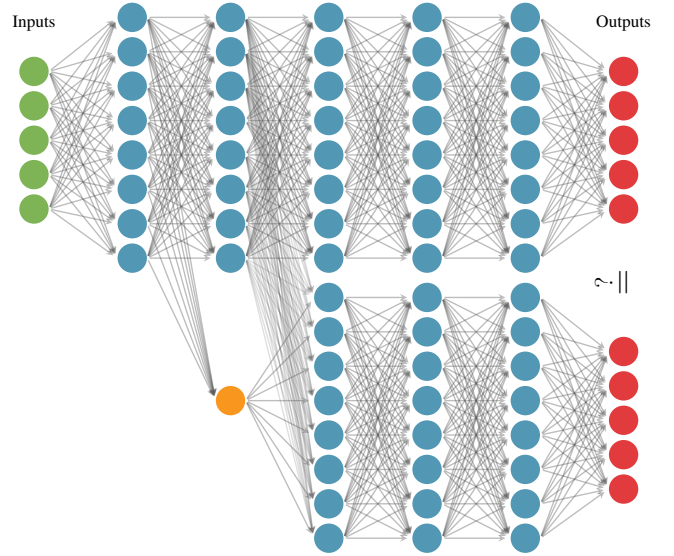

Fig. 8: **4-Forward-Redundancy** query illustration. The neuron in **orange** is the neuron being checked for forward-redundancy. In this case the layer being checked is at distance 4, which happens to be the output layer.

Determining whether $v = f(x)$ is result-preserving redundant is done by creating a query similar to the k-forward-redundant case, only this time we ask the verifier whether there exists an input that the two networks classify differently. If the answer is UNSAT, we know that the neuron is indeed result-preserving redundant.

Step 4: Relaxed Redundancy and Accumulative Error. The aforementioned steps were aimed at identifying and removing redundant neurons, without introducing any imprecision into the simplified network. Last but not least, we discuss the removal of relaxed-redundant neurons. Recall that relaxed-redundant neurons are determined by a user-specified error threshold e_t. Identifying these neurons is thus a local operation, that does not require verification; for every neuron we can compute the maximum error introduced by replacing it with l_m, and see whether it exceeds the threshold.

While each relaxed-redundant neuron can be identified locally, removing multiple neurons simultaneously runs the risk of compounding the overall error, beyond the permitted threshold. To circumvent this issue and allow the efficient removal of multiple relaxed-redundant neurons, we introduce the following lemma:

Lemma 1. *Let N be a neural network, and let N' be a simplified network, obtained from N by removing relaxed-redundant neurons $u_1, \ldots, u_n$. Consider another neuron v in N' that is relaxed-redundant, and let e_{in} denote the error to v's input, previously introduced by the removal of $u_1, \ldots, u_n$. Let e_v denote the error introduced by the removal of v. Then, if we remove v, the overall error introduced to its output is*

upper bounded by:

$$e_{in} + e_v$$

This lemma tells us that the iterative removal of relaxed-redundant neurons does not compound the introduced error; instead, the error introduced by the removal of each neuron is only added to the error already introduced by the removal of other neurons. This enables us, through a straightforward computation, to upper bound the overall imprecision (on the output layer) that the removal of a set of relaxed-redundant neurons might cause. Consequently, our proposed strategy is to begin removing relaxed-redundant neurons with small error rates, each time recomputing the overall network inaccuracy, until hitting the prescribed overall error threshold. A full, formal description of these claims appears in Appendix 2 in the extended version of this paper [34].

VI. INTRODUCING REDUNDANCIES VIA INPUT SLICING

So far, our simplification efforts have hinged on the existence of redundant neurons. Next, we introduce a technique that can cause neurons to become redundant, even if they are initially not so.

The core idea is to: (i) *slice the input domain* $\mathcal{D}$ of the DNN N into smaller sub-domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$; (ii) duplicate the original network n times, resulting in networks $N_1, \ldots, N_n$, such that network N_i is associated with domain $\mathcal{D}_i$; and (iii) apply the simplification process described in Section V for each N_i, separately. Intuitively, splitting the input domain into sub-domains can serve to separate "simpler" inputs regions, in which many neurons are phase-redundant, from more "complex" input domains where neurons fluctuate between phases. Various heuristics can be used for splitting the input domain, depending on the network in question. A simple splitting method, which we used in our evaluation, is to split the range of each input coordinate into n even sub-ranges.

After the slicing and simplification is done, we are left with a family of DNNs $N_1, \ldots, N_n$, which are together equivalent to the original N. Evaluation is then performed in two steps: given an input vector V_1, we first identify the domain $\mathcal{D}_i$ to which V_1 belongs; and then compute $N_i(V_1)$ and return the result. As our evaluation shows, the resulting N_i networks can be quite small, resulting in a significant improvement to the expected number of operations required for evaluating the network. This improvement might come at the expense of increased space requirements for storing the resulting family of networks, making this approach suitable for cases where space is abundant but fast inference is crucial. We note that, as a side effect, the resulting networks may be easier to verify [46], [48].

Discussion: Dependency on Input Dimensions. Our proposed slicing method relies on splitting the input domain, by restricting input neurons to various values. This approach works quite well on DNNs with relatively few input neurons (e.g., the ACAS Xu family of networks [25]; see Section VII for details). For networks with a larger number of input neurons (e.g., image recognition networks), the number of input sub-domains might be prohibitively large. Indeed, a similar phenomenon has been observed for verification techniques that rely on input slicing [46], [48].

One approach for mitigating this difficulty is through performing slicing not on the input layer, but on some smaller intermediate layer L_k in the network. Then, the network would be evaluated by evaluating the original network's layers $L_1 \ldots L_{k-1}$, and then using the values computed for layer L_k in choosing from a set of networks for continuing the evaluation. We speculate that for an intermediate layer of a moderate size, this approach could lead to improved performance over input slicing. We leave this for future work.

Extreme Slicing: Complete Linearization. We observe that input slicing can be used to completely linearize every sub-domain of the input space; that is, if the resulting sub-domains are sufficiently small, then in each network N_i all activation functions will become phase-redundant, effectively collapsing the DNN into a linear transformation. Additionally, even if the slicing does not fix the phase of all activation function neurons, extreme slicing tends to decrease the error introduced by removing relaxed-redundant neurons; and thus, complete linearization could be achieved by removing these neurons, even if they have not become phase-redundant. This linearization approach can thus be regarded as providing us with a simple, piecewise-linear approximation of the network as a whole — with an upper bound on the error in each sub-domain. Our experimental results in Section VII demonstrate very low error rates on most sub-domains.

Complete linearization incorporates a trade-off: in order to obtain very small, nearly-linear networks, the input domain would have to be sliced many times. Users can fine-tune the number of slices used, and consequently the sizes of the resulting DNNs, to their specific needs.

VII. EVALUATION

We created a proof-of-concept implementation of our approach as a Python framework, available online [33] (together with all benchmarks reported in this section). The framework provides all the functionality discussed so far: after importing a network, it can run MILP queries to compute neuron bounds; perform simulations; and identify phase-redundant, k-forward-redundant and result-preserving redundant neurons, by running verification queries. The framework uses the Gurobi [38] MILP solver and the Marabou [29] DNN verification engine as backends, although other backends could also be used.

For evaluation purposes, we conducted extensive experiments on the ACAS Xu system: an airborne collision avoidance system, implemented as a family of 45 neural networks [25]. Each of these neural networks has 5 input neurons, 5 output neurons, and 6 hidden layers with 50 neurons each and ReLU activation functions (310 neurons in total). Keeping the network sizes small was a key consideration in developing the ACAS Xu system [25], making it a prime candidate on which to apply simplification techniques.

We began by comparing our approach to that of Gokulanathan et al. [15], which is the current state-of-the-art in

verification-based simplification of DNNs. Their technique can be regarded as a private-case of ours, in which only specific phase-redundant neurons (specifically, inactive-redundant ReLUs) are removed. We compared that approach to our framework, configured to identify and remove both active-redundant and inactive-redundant ReLUs, and also to remove relaxed-redundant neurons. We ran both tools on all 45 ACAS Xu networks; the results appear in Table I.

TABLE I: Phase-Redundancy and Relaxed-Redundancy on ACAS Xu networks.

	Inactive Redundant	Active Redundant	Relaxed-Redundant		
			$\epsilon = 10^{-4}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$
% of all neurons	4%	4.2%	4.2%	4.6%	4.9%
% of redundant neurons	baseline	3.5%	5.3%	13.6%	21.5%
output error bound	0	0	0.02	2.64	525.1

The table depicts the accumulated numbers of redundant neurons, when read from left to right (which is the order in which the techniques were applied). First, inactive-redundant neurons are removed (this is the technique of [15]), accounting for 4% of all neurons in the network. Active-redundant neurons are next, removing another 0.2% of all neurons, which is a 3.5% increase in the number of removed neurons. Finally, relaxed-redundant neurons are removed, with three possible alternative ϵ values. The most permissive one, $\epsilon = 10^{-2}$, leads to the removal of 4.9% of the neurons in total, which is a 21.5% increase over the baseline — but the resulting network error bound in this case, 525.1, is quite high. $\epsilon = 10^{-3}$ appears a better choice, with a total removal rate of 4.6% and a significantly smaller error bound of 2.64. We note that our evaluation indicates that the output error bounds currently computed are far from tight; devising tighter bounding schemes is a work in progress.

In our second experiment, we evaluated our complete simplification pipeline. First, we applied input-slicing, dividing the input domain into 32,768 equal sub-domains (3 rounds of bisecting the range of each of the 5 input neurons in 2). Next, for each sub-domain we: (i) ran MILP and removed any discovered phase-redundant neurons; (ii) ran simulations, and then formal verification to discover and remove any remaining phase-redundant neurons; and (iii) identified all result-preserving neurons, and greedily attempted to simultaneously remove large sets thereof, using verification. We note that identifying the largest set possible of result-preserving neurons that can be removed simultaneously is a difficult problem, and our current heuristic was a simple, greedy approach. Devising more sophisticated heuristics is left for future work.

We ran the MILP step on all 32,768 sub-domains, which resulted in the discovery of 67.3% phase-redundant neurons on average in each sub-domain. We continued to run the pipeline on a sample of 50 sub-domains selected at random. Most notably, we observed an average removal of *82.5% redundant neurons* (out of all neurons in the network), with 7.2% additional neurons still candidates for removal, but for which the underlying verification engine timed-out. Of the 82.5% removed neurons, 70.2% were phase-redundant, which is a very significant increase from the 4.2% neurons removed when the pipeline was run over the entire input domain. This demonstrates the high effectiveness of input slicing. In addition, about 21% of phase-redundant neurons were active-redundant, which signifies the importance of the generalization from "dead neurons" [15] to phase-redundancy. The remaining 12.3% neurons removed were result-preserving redundant. Fig. 9 shows the breakdown.
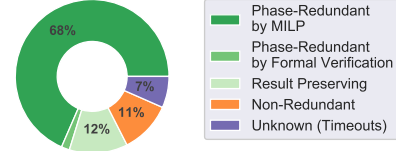

Fig. 9: Redundant neuron removal, averaged over 10 ACAS Xu input sub-domains.

Slicing is highly beneficial for neuron removal, but results in a large number of sub-domains that need to be checked. Within our pipeline, verification steps are the most expensive, whereas MILP queries and simulations are relatively cheap. We observe, however, that MILP queries already account for most of the removed neurons. Specifically, 68.5% of all phase-redundant neurons removed were discovered through MILP (about 83% of all redundant neurons), with a 10 second timeout for each individual MILP query.

The next step, namely simulations, is also computationally cheap and highly effective. For each sub-domain, we ran 100,000 simulations; and out of the of 31.5% neurons which were still candidates for removal after the MILP phase, an average of 26.4% of the neurons were ruled not phase-redundant through simulations. This left only a small number of candidates to be dispatched through verification (5.1% of the neurons), which in turn discovered the remaining 1.7% redundant neurons, on average. In our experiment, each Marabou verification query was run with a 4-hour timeout.

As discussed above, we used a fairly naïve strategy for discovering result-preserving redundant neurons. Specifically, we ran formal verification on each candidate neuron to check whether it was individually result-preserving redundant; this resulted in a set of candidates for removal. Then, we ran result-preserving simulations, iteratively removing additional candidate neurons from the network, as long as the simulations could not find a counter-example to the redundancy of the currently removed set. Finally, we ran a single verification query to verify that removing our selected neurons was indeed a result-preserving operation. On 75% of the sub-domains checked, this strategy worked. In sub-domains where we were successful, we found an additional 24.6% forward-redundant and result-preserving redundant neurons; whereas in sub-

domains where we were not successful, we had a similar amount of candidates for removal on average.

In the final step of our experiment, we tested our hypothesis that slicing can lead to the complete linearization of some of the sub-domains. Indeed, for some of the sub-domains explored, the simplification pipeline was able to remove *all* neurons, resulting in a DNN that is effectively a linear transformation. We noticed, however, a high variability — for example, in another sub-domain we were only able to remove 58% of the neurons. See Fig. 10 for additional details. We conclude that there is an inherent difference between the sub-domains: apparently, some of them compute simpler transformations than others.
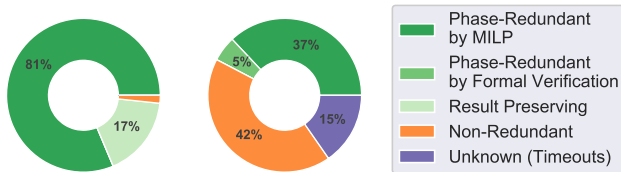

Fig. 10: An "almost" linear sub-domain (left) vs. a complex sub-domain (right).

VIII. RELATED WORK

The pruning of DNNs in order to reduce their sizes has received significant attention from the machine learning community in recent years. The most common approaches are based on heuristically identifying neurons and edges that seem to contribute little to the network's output, removing these neurons and edges, and performing additional training of the network [19], [23]. Other approaches apply quantization: by using fewer bits to store the network's weights or activation functions, the DNN's footprint is decreased [21], [22], [39]. A common trait of these approaches is that, while they achieve a significant reduction in memory, they provide no guarantees about the resemblance of the smaller network to the original.

The most closely related work to our own is that of Gokulanathan et al. [15]. There, the authors use formal verification to remove dead neurons from a network, ensuring that the resulting network is equivalent to the original. Additionally, simulations are used to reduce the number of verification queries that need to be dispatched. Our work uses similar principles, but significantly extends them: we consider additional kinds of redundancy (phase-redundancy, k-forward-redundant, and result-preserving redundancy) that produce equivalent networks, and also relaxed-redundancy which removes additional neurons by introducing a bounded amount of imprecision.

Our work uses the Marabou DNN verification engine as a backend [1], [7], [13], [18], [27], [29], [30], [42]; but any of the many approaches and tools that have been proposed in recent years could be used as well. These approaches leverage SMT solvers (e.g., [20]), based on LP and MILP solvers (e.g., [6], [11], [37], [44]), the propagation of symbolic intervals and abstract interpretation (e.g., [14], [45]–[47]), abstraction-refinement techniques (e.g., [3], [12]), and many

others. Recent work has extended beyond answering yes/no questions about DNNs, targeting tasks such as automated DNN repair [16], [31] and quantitative verification [4]. Verification approaches have also been proposed for recurrent networks [24], [49], which could potentially also be simplified. As DNN verification technology improves, the scalability of our approach will also increase.

IX. CONCLUSION AND FUTURE WORK

Neural networks often suffer from a high degree of redundancy, which affects evaluation time, memory footprint and verification costs. In this paper we presented a novel technique to identify and remove such redundancy. Our framework is customizable, allowing users to safely trade network precision for size reduction, while maintaining the introduced imprecision within a prescribed bound.

In the future, we plan to extend our work along multiple axes. Specifically, we plan to research more intelligent techniques for input domain slicing than coordinate-splitting; and also compositional techniques that would allow us to split the network into several sub-networks, identify redundancies in each of them, and then re-combine the pruned network into a single network that is smaller than the original. In addition, we plan to explore ways of combining our pruning techniques with techniques from the related field of Boolean circuit simplification [8].

REFERENCES

[1] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2021.

[2] G. Amir, H. Wu, C. Barrett, and G. Katz. An SMT-Based Approach for Verifying Binarized Neural Networks. In *Proc. 27th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 203–222, 2021.

[3] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mühlberger. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2020.

[4] 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.

[5] 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.

[6] 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.

[7] N. Carlini, G. Katz, C. Barrett, and D. Dill. Provably Minimally-Distorted Adversarial Examples, 2017. Technical Report. https://arxiv.org/abs/1709.10207.

[8] S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng. Perturb and Simplify: Multilevel Boolean Network Optimizer. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1494–1504, 1996.

[9] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[10] 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.

[11] 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.

[12] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.

[13] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. Annual Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2021.

[14] 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.

[15] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.

[16] 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.

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

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

[19] S. Han, H. Mao, and W. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding, 2015. Technical Report. http://arxiv.org/abs/1510.00149.

[20] 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.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, pages 4107–4115, 2016.

[22] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[23] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and < 0.5MB Model Size, 2016. Technical Report. http://arxiv.org/abs/1602.07360.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021. To appear.

[29] 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.

[30] 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)*, pages 83–89, 2019.

[31] 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.

[32] 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.

[33] O. Lahav and G. Katz. Code: Pruning and Slicing Neural Networks using Formal Verification, 2021. https://github.com/vbcrlf/redy.

[34] O. Lahav and G. Katz. Pruning and slicing neural networks using formal verification, 2021.

[35] L. Liebenwein, C. Baykal, B. Carter, D. Gifford, and D. Rus. Lost in Pruning: The Effects of Pruning Neural Networks beyond Test Accuracy, 2021. Technical Report. https://arxiv.org/abs/2103.03014.

[36] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. Kochenderfer. Algorithms for Verifying Deep Neural Networks, 2019. Technical Report. http://arxiv.org/abs/1903.06758.

[37] 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.

[38] G. Optimization. The Gurobi MILP Solver, 2021. https://www.gurobi.com/.

[39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet Classification using Binary Convolutional Neural Networks. In *Proc. 14th European Conf. on Computer Vision (ECCV)*, pages 525–542, 2016.

[40] 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.

[41] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. http://arxiv.org/abs/1409.1556.

[42] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU networks, 2020. Technical Report. http://arxiv.org/abs/2010.03258.

[43] 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.

[44] 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.

[45] H. Tran, S. Bak, 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.

[46] 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.

[47] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. Dhillon, and L. Daniel. Towards Fast Computation of Certified Robustness for ReLU Networks, 2018. Technical Report. http://arxiv.org/abs/1804.09699.

[48] 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.

[49] 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 Conf. of European Conference on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.