EPiC Series in Computing
Volume 73, 2020, Pages 260–278
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Minimal Modifications of Deep Neural Networks using Verification

Ben Goldberger[1], Yossi Adi[2], Joseph Keshet[3], and Guy Katz[1]

[1] The Hebrew University of Jerusalem, Israel
{jjgold, guykatz}@cs.huji.ac.il
[2] Facebook AI Research
yossiadidrum@gmail.com
[3] Bar Ilan University, Israel
jkeshet@cs.biu.ac.il

Abstract

Deep neural networks (DNNs) are revolutionizing the way complex systems are designed, developed and maintained. As part of the life cycle of DNN-based systems, there is often a need to modify a DNN in subtle ways that affect certain aspects of its behavior, while leaving other aspects of its behavior unchanged (e.g., if a bug is discovered and needs to be fixed, without altering other functionality). Unfortunately, retraining a DNN is often difficult and expensive, and may produce a new DNN that is quite different from the original. We leverage recent advances in DNN verification and propose a technique for modifying a DNN according to certain requirements, in a way that is *provably minimal*, does not require any retraining, and is thus less likely to affect other aspects of the DNN's behavior. Using a proof-of-concept implementation, we demonstrate the usefulness and potential of our approach in addressing two real-world needs: (i) measuring the resilience of DNN watermarking schemes; and (ii) bug repair in already-trained DNNs.

1 Introduction

Deep neural networks (*DNNs*) are quickly becoming the state of the art in many domains in computer science. In fields such as computer vision [30], speech recognition [20], game playing [41], and many others, DNNs have been repeatedly demonstrating excellent performance, often matching or even surpassing that of other, hand-crafted solutions. As a result of their empiric success, DNNs are changing the way software is being designed [18], and are rapidly being adopted by users in academia and industry.

The increasing pervasiveness of DNNs is causing a growing need for modifying already-trained DNNs, as part of their operation and maintenance. One notable example where this occurs is the *Machine Learning as a Service* (*MLaaS*) paradigm [38], where vendors release almost-fully-trained DNNs, which clients then fine-tune for their own needs. The motivation for this paradigm is that training modern DNNs often requires particular expertise, as well as considerable computing resources, which clients do not always posses. Another example where modifying an existing DNN might occur is in *patching* [44]: when a bug is discovered

in an existing DNN, a user might wish to modify it in a way that removes the undesirable behavior, but which leaves unaffected as many of the DNN's other behaviors as possible. In these cases, retraining the DNN might produce a DNN that is very different from the original, which is undesirable; and so techniques for modifying an existing DNN without retraining may be preferable.

These two examples, and others, share the same main characteristics: an existing DNN needs to be changed in a small way, which alters some behaviors but keeps others unchanged. Existing approaches for addressing this need are heuristic-based, and are not guaranteed to produce the desired results (e.g., [27, 28, 44]).

Here, we propose a novel approach for tackling this challenge, based on *neural network verification* [21, 24]. DNN verification is an emerging field, aimed at proving DNN correctness. Existing techniques allow users to specify properties that involve the DNN's inputs and outputs, and then either prove that these properties hold or provide a counter-example. Here, we show how to reduce the problem of making changes to the DNN itself into a DNN verification problem. Specifically, we propose a method for constructing a new DNN, and then pose a verification query on that DNN that is satisfiable if and only if the original DNN can be modified in the desired way. Due to the soundness of the underlying DNN verification techniques that we use, our technique affords formal guarantees about the modified DNN — specifically, that it is as close as possible to the original, and that it satisfies the specified constraints.

We demonstrate the applicability of our approach in two domains of interest:

Authentication of Ownership. As part of the MLaaS paradigm, when vendors allow clients access to an almost-fully-trained network, they are typically interested in collecting fees and royalties from each client. This business model compensates the vendor for designing and training the network, which are not simple tasks. In order to accomplish this, vendors now seek to imprint their DNNs with *watermarks* — secret inputs, known only to the vendors, for which the DNN reacts in unexpected ways [1]. The idea is that even if the network is modified by the client, it will still react unexpectedly to the watermark inputs. This would give vendors a way to recognize, and then claim ownership, over the networks they had trained.

DNN watermarking holds promise, but raises a concern: if users are expected to make modifications to the DNN in order to adjust it to their needs, could they (accidentally or intentionally) remove the watermarks? In order to address this concern, we require techniques for generating *resilient watermarks*, i.e. watermarks that are difficult to remove without excessively changing the DNN (and potentially damaging its functionality in the process).

We show how our proposed verification-based technique can be used to measure the resilience of watermarks within a DNN. Our approach can thus be used to assess individual DNNs before their release, and also to compare and rank different watermarking schemes, according to the resilience of the watermarks that they produce.

Minimality of Changes. Modifying an already-trained DNN is a complex task, which may be required under various circumstances — for example, when verification shows that a desirable property is violated. We show how our approach can be used in order to remove such violations. Specifically, we propose to maintain a list of points on which the DNN is known to produce a faulty result, and find a modification to the DNN that simultaneously corrects the behavior on all of these points. After each iteration, the network is verified; and if the property is still violated, meaning that additional faulty inputs are discovered, these inputs are added to the list of points, and the process is repeated. When the process converges, we are guaranteed that we have found the minimal modification to the DNN that satisfies the property at hand.

For evaluation purposes, we created a proof-of-concept implementation of our approach as

a Python framework. As its underlying DNN verification backends, our tool uses the Marabou DNN verification engine [26], and the Gurobi LP solver [19]. We then used our implementation to measure the resilience of a recently-proposed watermarking scheme [1], and also to correct faulty behavior of a DNN for an airborne collision avoidance system [23, 24]. Our results clearly indicate the usefulness of our approach to real-world examples.

The rest of this paper is organized as follows. In Section 2 we provide the necessary background on DNNs, DNN watermarking, and DNN verification. Next, in Section 3 we introduce our technique for reducing the minimal DNN modification problem into a DNN verification problem. In Section 4 we describe how to apply our technique in the context of watermark resilience and DNN patching. Section 5 then describes our implementation and evaluation of the approach. We discuss related work in Section 6, and conclude in Section 7.

2 Background

2.1 Neural Networks

A deep neural network is comprised of an input layer, an output layer, and multiple hidden layers in between. Each layer consists of multiple nodes (also called neurons), each of which is connected to nodes from the preceding layer. Each edge that connects two neurons is assigned a predetermined weight. An example appears in Fig. 1. Weight selection is performed during the DNN's training phase, which is beyond our scope here — see, e.g., [16].

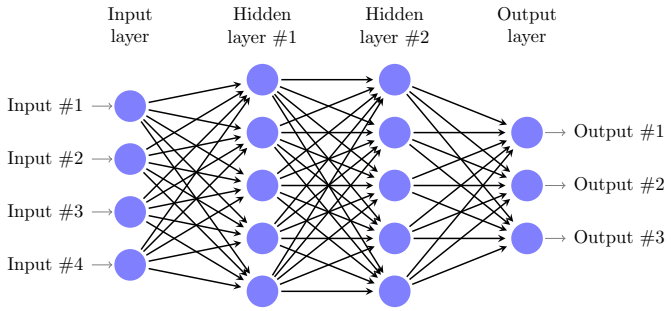


Figure 1: An example of a simple deep neural network. The network has an input layer of size 4, two hidden layers of size 5 each, and an output layer of size 3. The edge weights are omitted.

The neural network is evaluated by assigning values to the input nodes, and then iteratively propagating these values through the network, computing the values of additional layers one at a time. Eventually, the values of the output layer are computed, and these constitute the network's output. The value of each hidden node in the network is computed by calculating a weighted sum of node values from the preceding layer, and then applying a non-linear *activation function* [16] to this weighted sum. For simplicity, we focus here on the Rectified Linear Unit (ReLU) activation function [35], although our technique is directly applicable to additional functions as well. When the ReLU activation function, $\mathrm{ReLU}(x) = \max(0, x)$, is applied to a node, its value is computed as the maximum between 0 and the weighted sum computed according to the values of nodes from the previous layer.

More formally, let N be a DNN. We denote the number of layers in N by n, and use s_i to denote the dimension of layer i (i.e., the number of neurons that it contains). Layer 1 is the input layer, layer n is the output layer, and layers $2, \ldots, n-1$ are the hidden layers.

We use $v_{i,j}$ to denote the value of the j'th node of layer i, and use V_i to denote the column vector $[v_{i,1}, \ldots, v_{i,s_i}]^T$. When evaluating N we are given V_1, and need to compute V_n. This computation is performed by using the predefined weights and biases to compute the layer assignments one by one, each time applying the activation functions (ReLUs, in our case) to each node. Each layer $2 \leq i \leq n$ is associated with a weight matrix W_i of size $s_i \times s_{i-1}$, and also a bias vector B_i of size s_i. The entry of W_i at the j'th row and k'th column, denoted $W_i[j,k]$, represents the weight assigned to the edge from neuron $v_{i-1,k}$ to neuron $v_{i,j}$. Each hidden layer ($2 \leq i \leq n-1$) is computed as

$$V_i = \mathrm{ReLU}(W_i V_{i-1} + B_i),$$

with the ReLU function being applied element-wise. This rule is applied repeatedly for each layer until V_{n-1} is calculated. The final, output layer is computed similarly, but without an activation function:

$$V_n = W_n V_{n-1} + B_n$$

For simplicity, in the rest of the paper we assume that the bias vector B_i is 0, unless otherwise stated.

DNNs are often used as *classifiers*, assigning to each input a label from a finite set of possible labels $L = \{l_1, \ldots, l_{s_n}\}$. In this case, each neuron in the output layer corresponds to one possible label, and the label whose neuron is assigned the highest value is the label to which the input is classified. Thus, an input V_1 is classified to label l_i if and only if $v_{n,i} > v_{n,j}$ for every $j \neq i$ (draws are resolved arbitrarily). We sometimes use the equivalent phrasing,

$$N(V_1) = \underset{1 \leq i \leq s_n}{\arg\max}\{v_{n,i}\}$$

A small, running example appears in Fig. 2. Our example has 3 layers, each of size 2. The weights are depicted over the edges (all biases are assumed to be 0). For a given input $V_1 = \langle 3, 4 \rangle$ the second layer assignment is computed as

$$V_2 = \langle \mathrm{ReLU}(1 \cdot 3 - 2 \cdot 4), \mathrm{ReLU}(2 \cdot 3 - 1 \cdot 4) \rangle = \langle 0, 2 \rangle$$

and the output layer is computed as

$$V_3 = \langle 1 \cdot 0 - 1 \cdot 2, -1 \cdot 0 + 1 \cdot 2 \rangle = \langle -2, 2 \rangle$$

Observe that, by definition, the ReLU activation functions are not applied to the output layer. This implies that input $\langle 3, 4 \rangle$ is classified to label 2, because $v_{3,2} > v_{3,1}$.

2.2 Neural Network Verification

Let N be a neural network, and let P, Q be predicates that encode some constraints over the DNN's inputs and outputs, respectively. *DNN verification* [21, 24] is a field that seeks to answer the question: does there exist an input x to the DNN such that $P(x)$ and $Q(N(x))$ both hold, where $N(x)$ represents the network's output when evaluated on input x. Typically, P encodes a set of inputs of interest, whereas Q encodes the *negation* of a desired property. If the verification query is `UNSAT`, i.e. no such input x exists, then the desired property is said to hold. Otherwise, the verification tool returns a concrete counter-example x_0 for which the property in question is violated.

Recently there have been multiple tools and approaches suggested for solving the DNN verification problem: these include Satisfiability Modulo Theories (SMT) based techniques [24,

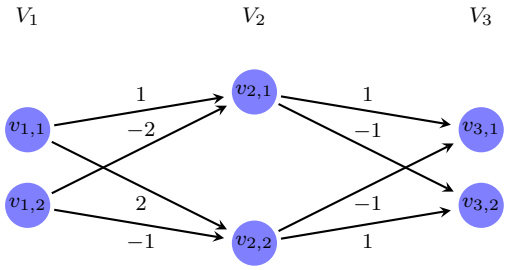


Figure 2: A small neural network.

26], approaches based on mixed integer linear programming [9, 47], abstract interpretation based techniques [12], and several others (e.g., [21, 36]). The problem is known to be difficult, and has been shown to be NP-complete even when restricted to simple cases [24] (networks with piecewise-linear activation functions, and properties P and Q that are conjunctions of linear constraints). Consequently, most existing approaches have limited scalability, and are highly sensitive to the size of the DNN being verified.

3 Minimal DNN Modification as a Verification Problem

3.1 Modifying DNNs

In all DNN verification approaches to date (to the best of our knowledge), the goal is to verify a fixed DNN. In other words, N is given, and a particular input x_0 is sought that satisfies certain constraints. The novelty of our approach is in changing this definition: we assume that the input point x_0 (or multiple inputs) are given, and search for a *different DNN* N such that certain constraints hold. More formally, we define the problem as follows:

Definition 1. ***The DNN Modification Problem.*** *Let N denote a DNN, let X denote a set of fixed input points $X = \{x_1, \ldots, x_n\}$, and let Q denote a predicate over the classifications $N(x_1), \ldots, N(x_n)$ of the points of X. The* DNN modification problem *is to find a new DNN, N', such that $Q(N'(x_1), \ldots, N'(x_n))$ holds, and such that the distance between N and N' is at most some $\delta > 0$.*

In this definition we did not yet specify how to measure the distance between N and N'. Multiple measures of distance could be used, but for the motivating problems that we consider (e.g., verifying the resilience of watermarks, or finding minimal fixes for erroneous inputs), it makes sense to require that N and N' share the same topology, and to define the distance between N and N' according to the difference in these network's weights. We note that a DNN modification is trivial if the property Q holds for the original network N, i.e. if $Q(N(x_1), \ldots, N(x_n))$ holds; in that case, N itself is a feasible solution to the modification problem.

Definition 2. ***DNN Distance.*** *Let N^1 and N^2 denote two DNNs with identical topology, i.e. the same number of layers ($n^1 = n^2$), and the same number of neurons in every pair of matching layers ($s_i^1 = s_i^2$ for all $1 \leq i \leq n^1$). We define the L-distance between N^1 and N^2,*

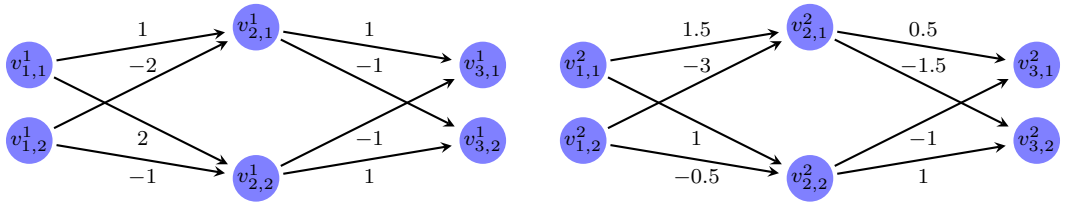
denoted $\left\|N^1 - N^2\right\|_L$, *as:*

$$\left\|N^1 - N^2\right\|_L = \left(\sum_{i=2}^{n^1} \sum_{j=1}^{s_i^1} \sum_{k=1}^{s_{i-1}^1} \left| W_i^1[j,k] - W_i^2[j,k] \right|^L \right)^{1/L}$$

Intuitively, this definition of distance compares the weight matrices of the two networks, element-wise; it computes the L-distance between each two corresponding edge weights as the L-norm of the difference between two vectors.

Fig. 3 shows how these definitions are applied to our toy example from Fig. 2. Network N^2 is obtained from the original network, N^1, by slightly modifying its edge weights while maintaining the same layered structure. The distance between these DNNs is:

$$\left\|N^1 - N^2\right\|_1 = (|-0.5| + |1| + |1| + |-0.5| + |0.5| + |0.5| + 0 + 0) = 4$$
$$\left\|N^1 - N^2\right\|_\infty = max\left\{|-0.5|, |1|, |1|, |-0.5|, |0.5|, |0.5|, 0, 0\right\} = 1$$



(a) N^1 (b) N^2

Figure 3: The DNN from Fig. 2, denoted N^1, and a modified version thereof, denoted N^2.

Finally, we will typically be interested in finding N' that is closest to N. This is formalized as follows:

Definition 3. ***The DNN Minimal Modification Problem.*** *Let N denote a DNN, let X denote a set of fixed input points $X = \{x_1, \ldots, x_n\}$, and let Q denote a predicate over the classifications $N(x_1), \ldots, N(x_n)$ of the points of X. The* DNN minimal modification problem *is to find a new DNN N' that solves the DNN modification problem (for N, X and Q), such that for every other N'' that solves the modification problem it holds that $\|N' - N\|_L \leq \|N'' - N\|_L$.*

We observe that given a decision procedure for solving the modification problem, solving the minimal modification problem can be performed (up to some small tolerance) by repeatedly invoking that procedure as part of a binary search, each time limiting the norm of the difference between the modified network and the original network. If the underlying decision procedure is sound and complete, it follows that this method for solving the minimal modification problem is also sound and complete.

3.2 Minimal Modifications as DNN Verification

A straightforward approach for solving the minimal DNN modification problem is to cast it as an optimization problem, and use an off-the-shelf optimization tool. Specifically, the goal would be to minimize $\|N - N'\|_L$, while maintaining the constraint $Q(N'(x_1), \ldots, N'(x_n))$.

Unfortunately, such an optimization problem is highly non-convex and high-dimensional, and is hence very difficult to solve.

We illustrate this issue in Fig. 4, again using our running example from Fig. 2. In the modified network N' that is depicted in the figure, we add to the edge weights from the original network fresh variables, $w_1, \ldots, w_8$, that represent the modification to the network. In order to solve the minimal modification problem, we would need to find a minimal assignment for the w_i variables such that the N' satisfies the predicate Q (the exact expression to be minimized depends on the distance metric in use).

To show that this minimization problem involves solving non-convex, high-degree constraints, let $X = \{\langle 3, 4\rangle\}$ and $Q(N'(\langle 3, 4\rangle)) = v_{3,1} \geq v_{3,2}$. Recall that the original network assigned the label 2 to point $\langle 3, 4\rangle$, and so our goal here is to modify the network so that it classifies this point to label 1, instead. The expressions for $v_{3,1}$ and $v_{3,2}$ are:

$$v_{3,1} = (1 + w_5) \cdot \text{ReLU}(3(1 + w_1) + 4(-2 + w_3)) + (-1 + w_7) \cdot \text{ReLU}(3(2 + w_2) + 4(-1 + w_4))$$
$$v_{3,2} = (-1 + w_6) \cdot \text{ReLU}(3(1 + w_1) + 4(-2 + w_3)) + (1 + w_8) \cdot \text{ReLU}(3(2 + w_2) + 4(-1 + w_4))$$

Both expressions are non-linear, due to the ReLU functions and the multiplication of terms involving the various w variables. For larger networks, such constraints would become significantly more complex.

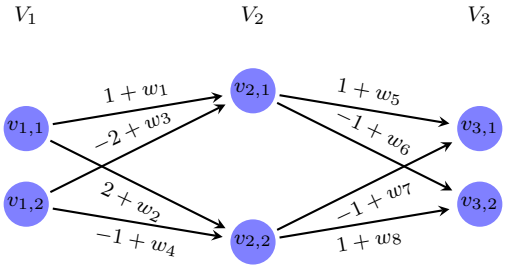


Figure 4: The DNN from Fig. 2, with modifications to its edges expressed as variables $w_1, \ldots, w_8$.

In order to mitigate this issue and render the DNN modification problem tractable for large networks, we focus on a restricted version of the problem in which the changes in weights are limited to a *single layer*. We argue that under this restriction, the DNN modification problem reduces into the standard DNN verification problem. The intuition is as follows: let N be a DNN, and let W_i (for some fixed i, $2 \leq i \leq n$) be the only weight matrix of N that we wish to modify. Observe some fixed input point $x \in X$, and let V_{i-1} denote the evaluation of layer $i-1$ for this x. In the modified network N', layers $1, \ldots, i-1$ will be assigned the same values as in N, i.e, $V_k = V'_k$ for all $1 \leq k \leq i-1$. The changes in assignment begin only in layer i, where it holds that

$$V'_i = \text{ReLU}(W'_i V'_{i-1}) = \text{ReLU}((W_i + W_\epsilon)V'_{i-1})$$

where W_ϵ represents the change in weights for layer i. We note that W_i and V'_{i-1} are fixed; the only variables here are the entries of W_ϵ. This means that, for this particular input point x, we can construct a new DNN, denoted $\bar{N}$, whose input neurons are the entries of W_ϵ, followed by a layer that computes $\text{ReLU}((W_i + W_\epsilon)V'_{i-1})$, and then feeds into layers $i+1, \ldots, n$ of the original network.

We again illustrate this using our running example — see Fig. 5. The figure depicts the modified network N' (left), and the new network $\bar{N}$ constructed from N' for the specific input point $x = \langle 3, 4 \rangle$. The input neurons of $\bar{N}$ are the entries of $W_\epsilon = [w_1, w_2, w_3, w_4]$, and its outputs are

$$\begin{aligned} v_{3,1} = &\ \text{ReLU}(3(1+w_1)+4(-2+w_3)) - \text{ReLU}(3(2+w_2)+4(-1+w_4)) \\ = &\ \text{ReLU}(3w_1+4w_3-5) - \text{ReLU}(3w_2+4w_4+2) \\ v_{3,2} = &-\text{ReLU}(3(1+w_1)+4(-2+w_3)) + \text{ReLU}(3(2+w_2)+4(-1+w_4)) \\ = &-\text{ReLU}(3w_1+4w_3-5) + \text{ReLU}(3w_2+4w_4+2) \end{aligned}$$

Which precisely represent the outputs of the modified network N' for input $V_1 = \langle 3, 4 \rangle$. Still focusing on the DNN modification problem with $X = \{\langle 3, 4 \rangle\}$, $Q(N'(\langle 3, 4 \rangle)) = v_{3,1} \geq v_{3,2}$ and some $\delta > 0$ in L_∞ norm, we could then pose a DNN verification query with input and output predicates (a similar encoding can be used for L_1 norm):

$$P = \bigwedge_{i=1}^{4} -\delta \leq w_i \leq \delta \qquad\qquad Q = v_{3,1} \geq v_{3,2} \tag{1}$$

The resulting DNN verification problem is `SAT` if and only if the DNN modification problem is `SAT`; and in that case, the satisfying assignment returned by the underlying verification engine can be used in constructing the modified DNN. As previously mentioned, a DNN *minimal* modification query can be reduced to multiple DNN modification queries (as part of a binary search), and can thus be solved using this technique as well.

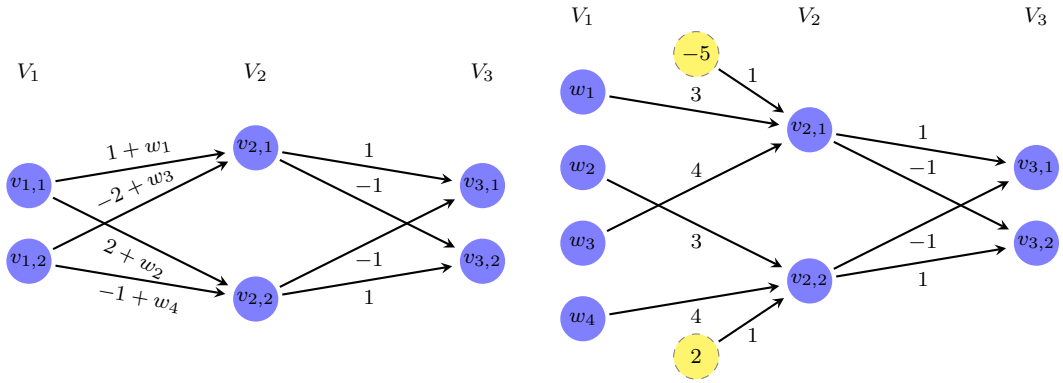


Figure 5: The modified network, N', on the left; and the network $\bar{N}$ used in the reduction to a DNN verification problem, on the right. The dashed nodes represent additive bias values, added to the weighted sums of $v_{2,1}$ and $v_{2,2}$.

In the previous example, we reduced the DNN modification problem into the DNN verification problem for a set X that contained a *single point*. The reduction when X contains multiple points is similar; we simply duplicate the DNN that we construct n times, creating a copy for each $x \in X$, and treat the result as one big DNN that is passed to the underlying verification engine. This allows us to solve the problem simultaneously for the multiple points. However, because DNN verification is NP-complete and becomes exponentially more complex as the size of the DNN increases [24], this renders the problem significantly more difficult to solve. In Section 3.3 we discuss a method for simplifying the process in some cases.

Finally, we note that the version of the problem in which we only allow changing a single layer of network is interesting in its own right. Specifically, variants of this approach are key in transfer learning [13, 14, 51, 53] and natural language processing [6, 37, 42], and have been shown to reduce overfitting with respect to the modification being made. Single-layer changes have also been shown to be an effective strategy when modifying object recognition networks [40].

3.3 Private Case: Modifying the DNN's Output Layer

Even when the DNN modification problem is limited to changes in a single layer of the DNN, the existence of many points in the set X can render the resulting DNN verification problem difficult. However, if the changes are limited to the *output layer* of the DNN, the resulting problem is typically easier to solve. Intuitively, the reason is that the deeper the layer that is being changed, the fewer neurons there are in the resulting DNN verification problem. In addition, if the distance metric in use is the L_∞ norm, changing only the output layer renders the resulting DNN verification problem *completely linear*. This means that it can then be solved using linear programming solvers [47], which tend to scale better than other DNN verification tools. In case L_1 norm is used, the problem still contains fewer neurons, but the absolute values that are introduced by the L_1 distance calculation prevent us from encoding it as a linear program. In that case, we apply DNN verification tools that support absolute values [26].

In Fig. 6 we demonstrate modifying the DNN's output layer, using our running example. Recall that for input $V_1 = \langle 3, 4 \rangle$, the output of the original network was $V_3 = \langle -2, 2 \rangle$; and suppose we wish to modify the network so that $v_{3,1}$ becomes greater than $v_{3,2}$. In Fig. 7 we show how to solve the resulting minimal modification problem, using L_∞ norm, by encoding it as a linear program. Observe that this encoding uses the fact that $V_2 = \langle 0, 2 \rangle$.

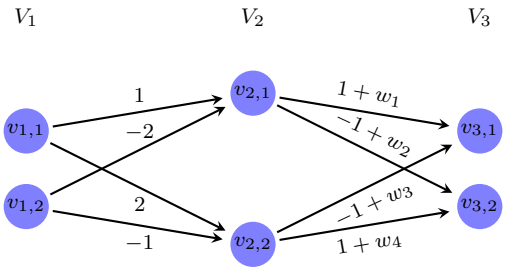


Figure 6: Modifying the output layer of our running example DNN.

4 Applications

4.1 Watermark Resilience

As briefly discussed in Section 1, there is now a growing need to imprint DNNs with *watermarks*: secret inputs, known to the creator of the DNN, on which the DNN produces unexpected outputs. Watermarks are crucial for the MLaaS business model, where a vendor sells a mostly-trained DNN to clients, who then fine-tune it for their own needs. The key requirement is that even if the network undergoes some changes, the watermark input will still produce the original, unexpected output, allowing the vendor to recognize the DNN.

$$
\begin{aligned}
\text{Minimize}: \quad & M \\
\text{Subject to}: \quad & M \geq 0 \\
& -M \leq w_1 \leq M \\
& -M \leq w_2 \leq M \\
& -M \leq w_3 \leq M \\
& -M \leq w_4 \leq M \\
& v_{3,1} = 0 \cdot (1 + w_1) + 2 \cdot (-1 + w_3) \\
& v_{3,2} = 0 \cdot (-1 + w_2) + 2 \cdot (1 + w_4) \\
& v_{3,1} \geq v_{3,2}
\end{aligned}
$$

Figure 7: Encoding the minimal modification problem (for the L_∞ norm) as a linear program. The constraints are those defined in Eq. 1. An optimal solution to this linear program is $w_3 = 1, w_4 = -1$ and $w_1, w_2 = 0$.

More formally, given a DNN N, a watermark is just an input point x with a specific label l. Typically, a network will have a finite set of watermarks X. Creating this set of watermarks is done at time of training — see, e.g., [1] for details. We say that a set $X = \{x_1, \ldots, x_n\}$ of watermarks is δ-resilient if for every DNN N' such that $\|N - N'\| \leq \delta$, it holds that $N(x_i) = N'(x_i)$ for all $1 \leq i \leq n$. Clearly, a watermarking scheme that produces watermarks that are more resilient is preferable.

Solving the DNN minimal modification problem can thus serve two purposes in the context of watermarking: (i) given a specific DNN N and a set X of watermarks, it can measure the resilience of X (i.e., the maximal amount of change the network can withstand without "forgetting" a watermark); and (ii) given multiple schemes for watermarking DNNs, i.e. for producing the set X, it can compare the effectiveness of the schemes by checking the resilience of the resulting watermarks (over some set of benchmark DNNs).

4.2 Minimal Repair

Despite their overall excellent performance, DNN-based systems are as prone to error as any other system. Specifically, many erroneous behaviors have been observed in various DNNs, e.g. for image recognition [11, 46] and for airborne collision avoidance [23, 24]. Once such a bug is discovered, it is not immediately clear how to correct it. One approach would be to retrain the network, but this may be expensive or impossible for some clients. The approach that we propose here is simply to identify a *minimal* change to the DNN that removes the undesirable behavior. We argue that it is useful to focus on a minimal change here, as such a change is less likely to change the behavior of the network for other inputs, on which its behavior is currently acceptable.

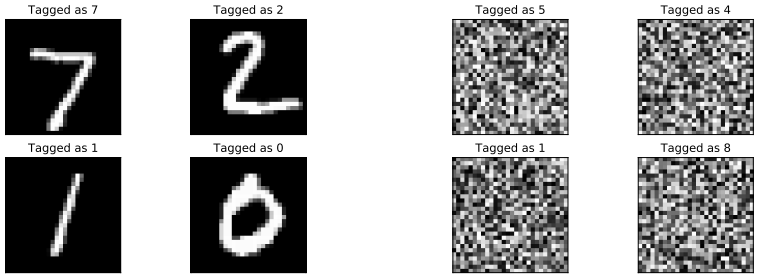
Finding minimal repairs for a DNN N can be cast into our formulation, as follows. Given a set of erroneous points $X = \{x_1, \ldots, x_n\}$ and their set of *correct* labels $l_1, \ldots, l_n$, we seek a new DNN N' such that $N'(x_i) = l_i$ for all $1 \leq i \leq n$, and such that $\|N - N'\|$ is minimal.

5 Evaluation

We evaluated our approach using a proof-of-concept implementation as a Python framework. Our code, together with the benchmarks on which we report in this section, are available online. [1]

5.1 Watermark Removal

In order to apply our approach to measure the resilience of watermarked networks, we trained a watermarked DNN over the MNIST digit recognition dataset [33]. The DNN was of a modest size: it had an input layer with 784 nodes, a single hidden layer with 150 nodes, and a final output layer with 10 nodes. The input and output layers were fully connected, and the hidden layer had the ReLU activation function applied to its nodes. We used the method proposed in [1] to create a set $X = \{x_1, \ldots, x_{100}\}$ of 100 watermarks, each assigned a label $N(x_i)$; an example appears in Fig. 8.



(a) Standard MNIST inputs. (b) Watermark inputs.

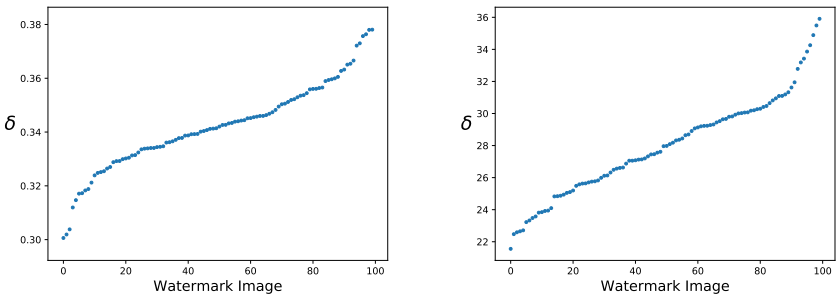
Figure 8: Inputs to our MNIST digit-recognition DNN. The inputs on the left are standard, and are correctly classified by the network. The inputs on the right are the watermarks: the network was trained in a specialized way to make it assign those particular labels to those inputs.

Next, we ran a series of experiments to identify modifications to the network that would remove some, or all, of these watermarks. For simplicity, we focused on changes to the network's output layer. As a measure of distance, we used the L_1 and L_∞ norms, which are popular distance metrics in the ML community and are supported by many DNN verification tools.

Removing a Single Watermark. We began by measuring the minimal change to the network required to remove each individual watermark. More specifically, for each watermark $x \in X$ we found the closest network N' such that $N(x) \neq N'(x)$. We note that we did not explicitly specify $N'(x)$, i.e. did not require that N' classify x in a particular way; we only required that the new label be different from the original. We ran the experiment once with L_1, and once with L_∞; the results are depicted in Fig. 9.

Analyzing these results exposes several interesting properties of the watermark inputs. For example, we observe that for every single watermark x, the minimally-modified network that was found classified x to the label that was assigned the second-highest score by the original

[1] https://github.com/jjgold012/MinimalDNNModificationLpar2020



(a) L_∞ distance (b) L_1 distance

Figure 9: The minimal distance to a modified network in which each of the 100 watermarks is removed.

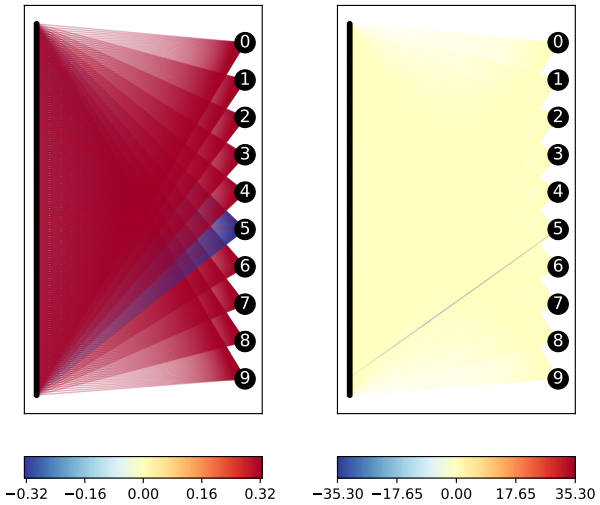
network. This result indicates that a resilient watermark should have a significant gap between the score assigned to its actual label and the score of its second-highest label. We also observe that some watermarks are *significantly* more difficult to remove than others; specifically, in the L_∞ case, the more resilient watermarks were approximately 26% more difficult to remove than others, and in the L_1 case — about 67% more difficult. This indicates a very wide range of watermark resilience; and a DNN trainer may wish to assess the resilience of each selected watermark before releasing the DNN.

The experiment also exposes some of the differences between the two distance metrics used, i.e. L_1 and L_∞. Not surprisingly, the distances in the L_1 case are generally larger than those in the L_∞ case: this is because the L_∞ norm allows us to change multiple edges "at a flat rate", and thus affect the network's outcome more effectively; whereas L_1 distance counts every edge weight that changes. In Fig. 10 we visualize this on one of the watermarks; we see that indeed, when using the L_∞ norm, almost all the edges were touched, whereas using the L_1 norm caused the change to occur only in a single edge.

In addition, we were also interested to see the effect of removing a watermark on the network's overall accuracy — as it is likely to assume that a user seeking to remove watermarks will also attempt to maintain the network's original performance. After removing each watermark, we used the MNIST dataset to measure the accuracy of the modified network, and compared it to the original network's accuracy. As it turns out, the results (depicted in Table 1) vary widely between L_1 and L_∞. We observe that using L_1 gives better accuracy result on average, but can lead to poor accuracy in extreme cases; whereas the L_∞ norm leads to a much more consistent result (the minimal and maximal accuracy are quite close), although the average accuracy is lower than that of the L_1 case.

Removing Multiple Watermarks. Next, we proceeded to modifying our DNN in a way that removes multiple watermarks, simultaneously. Specifically, we wanted to change the weights of the DNN's output layer in a way that caused the misclassification of an arbitrary number of watermarks.

As discussed in Section 3.3, the problem of modifying the network's output layer when using the L_∞ norm is significantly easier than other variants; and indeed, using the Gurobi LP solver [19] as our backend verification tool, we were able to find modifications that removed as many as all of the watermarks. In the L_1 case, performance became an issue, and so we focused on removing as many as 5 watermarks simultaneously. The results are depicted in Table 2. We



(a) L_∞ distance (b) L_1 distance

Figure 10: The minimal weight changes (to the output layer) required to remove a specific watermark. Positive change is colored red and negative change is colored blue. There are 1500 edges feeding into the output layer.

Number of water-marks	Norm	Average change	Minimal change	Maximal change	Average accuracy	Minimal accuracy	Maximal accuracy
0		0	0	0	0.97	0.97	0.97
1	L_∞	0.34	0.3	0.38	0.87	0.86	0.88
1	L_1	27.95	21.56	35.91	0.94	0.56	0.97

Table 1: Changes to the DNN's accuracy after removing a single watermark. The experiments were performed for each of the 100 watermarks, individually, and then aggregated the results. The first row shows the baseline accuracy rates.

see that a few watermarks can still be removed while maintaining a reasonable level of accuracy; but that, at least in the L_∞ case, removing all watermarks renders the network quite useless. We thus conclude that using these particular 100 watermarks is sufficient for this particular network.

In our experiments we observed that often, removing a particular watermark (or sets thereof) would result in the removal of other watermarks, whose removal was not part of the experiment. This indicates some hidden dependencies between the watermarks. For example, in one experiment where we removed 5 watermarks using the L_1 norm, a total of 80 (!) watermarks were removed due to the modification (out of 100 total watermarks), and without any significant reduction in the network accuracy. In experiments using the L_∞ norm, we also noticed additional watermarks being removed beyond those that were targeted, but there the network's accuracy also tends to drop sharply as the number of removed watermarks increases (as indicated by the *maximal accuracy* column in Table 2).

Number of water-marks	Average change	Minimal change	Maximal change	Average accuracy	Minimal accuracy	Maximal accuracy
0	0	0	0	0.97	0.97	0.97
1	0.34	0.3	0.38	0.87	0.86	0.88
2	0.43	0.3	1.83	0.79	0.76	0.88
3	0.53	0.33	1.79	0.71	0.66	0.88
4	0.68	0.33	1.79	0.64	0.57	0.88
5	0.79	0.33	1.87	0.59	0.47	0.8
6	0.89	0.35	1.83	0.53	0.38	0.78
7	1.05	0.34	1.91	0.48	0.28	0.78
25	1.86	1.45	2.09	$9.49 \cdot 10^{-2}$	$2.7 \cdot 10^{-3}$	0.41
50	2.05	1.9	2.15	$4.89 \cdot 10^{-2}$	$2.3 \cdot 10^{-3}$	0.2
75	2.13	2.03	2.17	$5.95 \cdot 10^{-2}$	$2.8 \cdot 10^{-3}$	0.18
100	2.18	2.18	2.18	$5.11 \cdot 10^{-2}$	$5.11 \cdot 10^{-2}$	$5.11 \cdot 10^{-2}$

(a) Changes in accuracy when solving for minimal L_∞ change.

Number of water-marks	Average change	Minimal change	Maximal change	Average accuracy	Minimal accuracy	Maximal accuracy
0	0	0	0	0.97	0.97	0.97
1	27.95	21.56	35.91	0.94	0.56	0.97
2	51.12	22.65	145.75	0.9	0.56	0.97
3	73.52	25.83	172.67	0.86	0.53	0.97
4	98.39	25.83	205.22	0.86	0.53	0.97
5	118.09	30.65	260.39	0.86	0.51	0.97

(b) Changes in accuracy when solving for minimal L_1 change.

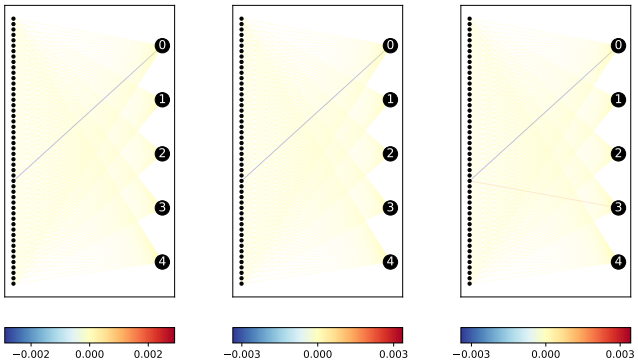
Table 2: Minimal changes and accuracy degradation when simultaneously removing multiple watermarks. We sampled a high number of arbitrary sets of watermarks for each category, and aggregated the results.

5.2 Removing Undesirable Behavior

In a separate experiment, we evaluated the effectiveness of our approach in correcting undesirable behaviors in a real-world DNN. The DNN in question is part of the ACAS Xu system for Airborne Collision Avoidance [23] — a system that is part of a new standard, currently under development by the Federal Aviation Administration (FAA). It is intended to read sensor information regarding other nearby aircraft, and produce a horizontal turning advisory in order to avoid a possible collision with these aircraft. One possible implementation of the system that is being considered includes a family of 45 DNNs [23].

For our experiment, we focused on one of the ACAS Xu DNNs in which a bug has been discovered using formal verification [24]. Specifically, a property φ that specifies certain conditions (on the DNN's inputs) under which the network should advise a weak turn or no turn at all was shown not to hold — and a concrete input x_0 on which φ is violated was discovered. We attempted to apply our technique in order to find a minimal modification to the DNN in question that would correct this behavior, i.e. make the network satisfy φ.

Because our goal was to satisfy φ, and not just to correct the DNN's behavior on the concrete counter-example x_0 that had previously been discovered, we applied our technique as part of



(a) First iteration inputs change (b) Second iteration inputs change (c) Third iteration inputs change

Figure 11: Change to the last layer for the ACAS Xu network in question, in L_1 norm, in each iteration of Algorithm 1. The first two iterations change a single edge; the third iteration changes 2 edges.

the loop described in Algorithm 1.

Algorithm 1 Repair Network (N,φ,x_0)

$S \leftarrow \{x_0\}$
$N' \leftarrow N$
while true **do**
    Use verification to check whether φ holds for N'
    if φ holds for N' **then**
        return N'
    else
        Add the discovered counter-example to S
        $N' \leftarrow$ minimally-modify N so that φ holds for all points in S

The loop in Algorithm 1 terminates when N is successfully modified such that φ holds. If the loop does not converge, the procedure may need to be terminated when a certain timeout values is reached.

When we ran Algorithm 1 on our ACAS Xu DNN in question, it failed to terminate — we stopped it after several days, while it was attempting to verify φ after the third modification was applied to the network. Thus, while we were unable to verify that all violations of φ was removed from N, we did obtain a network in which at least some of the original violations no longer exist. The results are depicted in Fig. 11. When we examined the counter-examples discovered by the algorithm during its iterations, we observed that they were fairly distant from each other — indicating that the violation is question was not minor, and possibly explaining the difficulty in correcting it.

6 Related Work

Correcting DNNs in order to remove undesirable behaviors is a general topic of interest in the ML community. One line of work [27, 28] suggests to augment a malfunctioning DNN with decision trees that determine when a patch should be applied. Another line of work [44] corrects DNNs using an iterative encoding of the problem into a sequence of Max-SMT instances. A key feature of our verification-based approach that separates it from prior work is that it provides formal guarantees about the minimality of the discovered changes to the DNN in question.

DNN verification is an emerging field, with many recently-proposed tools and approaches. These include the use of SMT solving [21, 22, 24, 26], LP and MILP solving [9, 47], symbolic interval propagation [50], abstraction based techniques [10, 12], and many others (e.g., [3, 8, 29, 31, 34, 36, 39, 43, 52]). Our technique reduces the DNN minimal modification problem into a DNN verification problem, and can use many of the aforementioned tools and techniques as a backend. In addition, whereas most work to date has focused on verifying adversarial robustness properties [2, 4, 17, 25], on verifying hybrid systems with DNN controllers [7, 45], and on DNN simplification [15], our work demonstrates that DNN verification engines are applicable also for analyzing the resilience of DNN watermarking schemes.

DNN watermarking is a general approach for marking DNNs that are to undergo changes, for which multiple methods have been proposed (e.g., [1, 5, 32, 48, 49]). Our technique can be used in assessing and comparing the performance of such watermarking approaches.

7 Conclusion and Future Work

With the fast spreading use of DNNs, there is an increasing need to make small modifications to already-trained networks — e.g., as part of the MLaaS paradigm, or in order to correct erroneous DNN behavior. By harnessing recent advances in DNN verification, we proposed here a method for finding the minimal changes required to make a DNN satisfy certain properties. This technique holds great potential for users who wish to change a DNN, and also for vendors who wish to ensure that their DNNs maintain some behavior even if modified. We demonstrated the applicability of our approach in analyzing a DNN watermarking scheme [1], and in attempting to correct a faulty behavior of a DNN for airborne collision avoidance [23].

In the future we plan to extend our technique so that it can be applied when the changes to a DNN occur in more than one layer. In order to overcome the highly non-linear and non-convex nature of the problem, we plan to apply compositional techniques: i.e., to break the DNN down into smaller DNNs, so that the changes to each smaller DNN will only occur in a single layer. We will then apply our technique to each smaller DNN separately, and use the results to draw conclusions regarding changes to the original DNN as a whole. In addition, we plan to identify and research additional use cases, beyond watermark resilience and bug correction, where our technique could prove useful.

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

References

[1] Y. Adi, C. Baum, B. Pinkas, and J. Keshet. Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring. In *Proc. 27st USENIX Security Symposium*, 2018.

[2] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.

[3] 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.

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

[5] H. Chen, B. Rohani, and F. Koushanfar. Deepmarks: A Digital Fingerprinting Framework for Deep Neural Networks, 2018. Technical Report. http://arxiv.org/abs/1804.03648.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding, 2018. Technical Report. http://arxiv.org/abs/1810.04805.

[7] S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.

[8] 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.

[9] 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.

[10] 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)*, 2020.

[11] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. IEEE Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.

[12] 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.

[13] R. Girshick. Fast R-CNN. In *Proc. IEEE Int. Conf. on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 580–587, 2014.

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

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

[17] 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.

[18] 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.

[19] Z. Gu, E. Rothberg, and R. Bixby. Gurobi Optimizer Reference Manual, Version 5.0. *Gurobi Optimization Inc., Houston, USA*, 2012.

[20] 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.

[21] 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.

[22] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference, 2020. Technical Report. https://arxiv.org/abs/2004.02462.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] D. Kauschke, S. Lehmann. Evaluating Engagement-Layers and Patch-Architectures, 2018. Technical Report. http://arxiv.org/abs/1812.03468.

[28] S. Kauschke and J. Furnkranz. Batchwise Patching of Classifiers. In *Proc. 32nd AAAI Conf. on Artifical Alliance*, 2018.

[29] 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.

[30] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[31] 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.

[32] E. Le Merrer, P. Perez, and G. Trédan. Adversarial Frontier Stitching for Remote Neural Network Watermarking. *Neural Computing and Applications*, pages 1–12, 2019.

[33] Y. LeCun, C. Cortes, and C. Burges. The MNIST Database of Handwritten Digits, 1998.

[34] 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.

[35] 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.

[36] 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.

[37] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep Contextualized Word Representations, 2018. Technical Report. http://arxiv.org/abs/1802.05365.

[38] M. Ribeiro, K. Grolinger, and M. Capretz. MLaaS: Machine Learning as a Service. In *Proc. 14th Int. Conf. on Machine Learning and Applications (ICMLA)*, pages 896–902, 2015.

[39] W. Ruan, X. Huang, and M. Kwiatkowska. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proc. 27th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2018.

[40] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN Features off-the-shelf: an Astounding Baseline for Recognition. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 806–813, 2014.

[41] 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.

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

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

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

[45] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.

[46] 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.

[47] 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.

[48] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh. Embedding Watermarks into Deep Neural Networks. In *Proc. 17th ACM Int. Conf. on Multimedia Retrieval (ICMR)*, pages 269–277, 2017.

[49] A. Venugopal, J. Uszkoreit, D. Talbot, F. Och, and J. Ganitkevitch. Watermarking the Outputs of Structured Prediction with an Application in Statistical Machine Translation. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, page 1363–1372, 2011.

[50] 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*, pages 1599–1614, 2018.

[51] J. West, D. Ventura, and S. Warnick. Spring Research Presentation: A Theoretical Foundation for Inductive Transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 1(08), 2007.

[52] 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., 2020. Technical Report. https://arxiv.org/abs/2004.08440.

[53] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How Transferable are Features in Deep Neural Networks? In *Proc. 28th Conf. on Neural Information Processing Systems (NeurIPS)*, pages 3320–3328, 2014.