Provable Repair of Deep Neural Networks

Matthew Sotoudeh
University of California, Davis
Davis, CA, USA
masotoudeh@ucdavis.edu

Aditya V. Thakur
University of California, Davis
Davis, CA, USA
avthakur@ucdavis.edu

Abstract

Deep Neural Networks (DNNs) have grown in popularity over the past decade and are now being used in safety-critical domains such as aircraft collision avoidance. This has motivated a large number of techniques for finding unsafe behavior in DNNs. In contrast, this paper tackles the problem of correcting a DNN once unsafe behavior is found. We introduce the *provable repair problem*, which is the problem of repairing a network N to construct a new network N' that satisfies a given specification. If the safety specification is over a finite set of points, our Provable Point Repair algorithm can find a provably minimal repair satisfying the specification, regardless of the activation functions used. For safety specifications addressing convex polytopes containing infinitely many points, our Provable Polytope Repair algorithm can find a provably minimal repair satisfying the specification for DNNs using piecewise-linear activation functions. The key insight behind both of these algorithms is the introduction of a *Decoupled* DNN architecture, which allows us to reduce provable repair to a linear programming problem. Our experimental results demonstrate the efficiency and effectiveness of our Provable Repair algorithms on a variety of challenging tasks.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Theory of computation** → **Linear programming**; • **Software and its engineering** → **Software post-development issues**.

Keywords: Deep Neural Networks, Repair, Bug fixing

ACM Reference Format:
Matthew Sotoudeh and Aditya V. Thakur. 2021. Provable Repair of Deep Neural Networks. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454064

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454064

1 Introduction

Deep neural networks (DNNs) [21] have been successfully applied to a wide variety of problems, including image recognition [39], natural-language processing [14], medical diagnosis [37], aircraft collision avoidance [32], and self-driving cars [10]. However, DNNs are far from infallible, and mistakes made by DNNs have led to loss of life [20, 41] and wrongful arrests [28, 29]. This has motivated recent advances in understanding [22, 55], verifying [4, 6, 18, 33, 49], and testing [45, 47, 54, 56] of DNNs. In contrast, this paper addresses the problem of repairing a DNN once a mistake is discovered.

Consider the following motivating scenario: we have a trained SqueezeNet [31], a modern convolutional image-recognition DNN consisting of 18 layers and 727,626 parameters. It has an accuracy of 93.6% on the ImageNet dataset [13]. *After deployment*, we find that certain images are misclassified. In particular, we see that SqueezeNet has an accuracy of only 18% on the Natural Adversarial Examples (NAE) dataset [27]. Figure 1 shows one such image whose actual class is `Fox Squirrel`, but the DNN predicts `Sea Lion` with 99% confidence. We would like to repair (or patch) the trained SqueezeNet to ensure that it correctly classifies such images.

To repair the DNN, one could *retrain* the network using the original training dataset augmented with the newly-identified buggy inputs. Retraining, however, is extremely inefficient; e.g., training SqueezeNet takes days or weeks using state-of-the-art hardware. Worse, the original training dataset is often not available for retraining; e.g., it could be private medical information, sensitive intellectual property, or simply lost. These considerations are more important with privacy-oriented regulations that require companies to delete private data regularly and upon request. Retraining can also make arbitrary changes to the DNN and, in many cases, introduce new bugs into the DNN behavior. These issues make it infeasible, impossible, and/or ineffective to apply retraining in many real-world DNN-repair scenarios.

One natural alternative to retraining is *fine tuning*, where we apply gradient descent to the trained DNN but only using a smaller dataset collected once buggy inputs are found. While this reduces the computational cost of repair and does not require access to the original training dataset, fine-tuning significantly increases the risk of *drawdown*, where the network forgets things it learned on the original, larger training dataset in order to achieve high accuracy on the buggy inputs [36]. In particular, fine tuning provides no guarantees that it makes minimal changes to the original DNN.

Figure 1. Natural adversarial example

Figure 2. Fog-corrupted digit

The effectiveness of both retraining and fine tuning to repair the DNN is extremely sensitive to the specific hyperparameters chosen; viz., training algorithm, learning rate, momentum rate, etc. Importantly, because gradient descent cannot *disprove* the existence of a better solution were some other hyperparameters picked, one might have to try a large number of potential hyperparameter combinations in the hope of finding one that will lead to a successful repair. This is a time-consuming process that significantly reduces the effectiveness of such techniques in practice.

Based on the above observations, we can deduce the following requirements for our DNN repair algorithm, where N is the buggy DNN, X is the set of buggy inputs, and N' is the repaired DNN: (P1) *efficacy*: N' should correctly classify all inputs in X; (P2) *generalization*: N' should correctly classify inputs similar to those in X; (P3) *locality*: N' should behave the same as N on inputs that are dissimilar to those in X; (P4) *efficiency*: the repair algorithm should be efficient.

This paper presents a novel technique for *Provable Pointwise Repair* of DNNs that is effective, generalizing, localized, and efficient (§5). Given a DNN N and a finite set of points X along with their desired output, our Provable Pointwise Repair algorithm synthesizes a repaired DNN N' that is guaranteed to give the correct output for all points in X. To ensure locality, our algorithm can *provably guarantee* that the repair (difference in parameters) from N to N' is the smallest such single-layer repair. Our Provable Pointwise Repair algorithm makes no restrictions on the activation functions used by N.

Provable repair of even a *single layer* of the DNN is a formally NP-hard problem, and completely infeasible in practice even using state-of-the-art SMT solvers [19]. The use of *non-linear activation functions* implies that changing even a single weight can have a non-linear effect on the output of the DNN. However, if the final layer of the DNN is linear (instead of a non-linear activation function), then repairing just the output layer is actually a linear programming (LP) problem [19] solvable in polynomial time [38].

The key insight to our approach is the introduction of a new DNN architecture, called *Decoupled DNNs* or DDNNs. DDNNs strictly generalize the notion of DNNs, meaning every DNN can be trivially converted into an equivalent DDNN. We will show that repairing *any* single layer in a DDNN reduces to an LP problem. This allows us to compute the *smallest* such single-layer repair with respect to either the ℓ_1 or ℓ_∞ norm, and, thus, reduce forgetting.

This paper also introduces an algorithm for *Provable Polytope Repair* of DNNs (§6), which is like Provable Point Repair except the set of points X is *infinite* and specified as a union of convex polytopes (hereafter just "polytopes") in the input space of the network.

Consider a trained DNN for classifying handwritten digits, which has an accuracy of 96.5% on the MNIST dataset [40]. After deployment, we find that the accuracy of the network drops to 20% on images corrupted with fog; Figure 2 shows an example of one such fog-corrupted image from MNIST-C [44]. We would like to repair the network to correctly classify such fog-corrupted images. However, we might also want to account for different amounts of fog. Let I and I_f be an uncorrupted and fog-corrupted image, respectively. Then each image along the line from I to I_f is corrupted by a different amount of fog. We can use Provable Polytope Repair so that the DNN correctly classifies *all infinitely-many* such foggy images along the line from I to I_f.

Consider an aircraft collision-avoidance network [32] that controls the direction an aircraft should turn based on the relative position of an attacking aircraft. We may want this DNN to satisfy certain properties, such as never instructing the aircraft to turn towards the attacker when the attacker is within a certain distance. Our Provable Polytope Repair algorithm can synthesize a repaired DNN that provably satisfies such safety properties on an infinite set of input scenarios.

The main insight for solving Provable Polytope Repair is that, for piecewise-linear DDNNs, repairing polytopes (with infinitely many points) is equivalent to Provable Point Repair on finitely-many *key points*. These key points can be computed for DDNNs using prior work on computing symbolic representations of DNNs [51, 53]. This reduction is intuitively similar to how the simplex algorithm reduces optimizing over a polytope with infinitely many points to optimizing over the finitely-many vertex points. As illustrated by the above two scenarios, there are practical applications in which the polytopes used in the repair specification are low-dimensional subspaces of the input space of the DNNs.

We evaluate the efficiency and efficacy of our Provable Repair algorithms compared to fine tuning (§7). The repairs by our algorithms *generalize* to similarly-buggy inputs while avoiding significant *drawdown*, or forgetting.

The contributions of the paper are:

- We introduce Decoupled DNNs, a new DNN architecture that enables efficient and effective repair (§4).
- An algorithm for Provable Point Repair (§5).
- An algorithm for Provable Polytope Repair of piecewise-linear DNNs (§6).
- Experimental evaluation of Provable Repair (§7).

§2 describes preliminaries; §3 presents an overview of our approach; §8 describes related work; §9 concludes.

2 Preliminaries

A feed-forward DNN is a special type of loop-free computer program that computes a vector-valued function. DNNs are often represented as layered DAGs. An input to the network is given by associating with each node in the input layer one component of the input vector. Then each node in the second layer computes a weighted sum of the nodes in the input layer according to the edge weights. The output of each node in the second layer is the image of this weighted sum under some *non-linear activation function* associated with the layer. This process is repeated until output values at the final layer are computed, which form the components of the output vector.

Although we will use the above DAG definition of a DNN for the intuitive examples in § 3, for most of our formal theorems we will use an entirely equivalent definition of DNNs, below, as an alternating concatenation of *linear* and *non-linear* functions.

Definition 2.1. A *Deep Neural Network* (DNN) with layer sizes $s_0, s_1, \ldots, s_n$ is a list of tuples $(W^{(1)}, \sigma^{(1)}), \ldots, (W^{(n)}, \sigma^{(n)})$, where each $W^{(i)}$ is an $s_i \times s_{i-1}$ matrix and $\sigma^{(i)} : \mathbb{R}^{s_i} \to \mathbb{R}^{s_i}$ is some *activation function*.

Definition 2.2. Given a DNN N with layers $(W^{(i)}, \sigma^{(i)})$ we say the *function associated with the DNN* is a function $N : \mathbb{R}^{s_0} \to \mathbb{R}^{s_n}$ given by $N(\vec{v}) = \vec{v}^{(n)}$ where $\vec{v}^{(0)} := \vec{v}$ and $\vec{v}^{(i)} := \sigma^{(i)}(W^{(i)}\vec{v}^{(i-1)})$.

For ease of exposition, we have assumed: (i) that every layer in the DNN is fully-connected, i.e., parameterized by an entire weight matrix, and (ii) that the activation functions $\sigma^{(i)}$ have the same domain and range. However, our algorithms do not rely on these conditions, and in fact, we use more complicated DNNs (such as Convolutional Neural Networks) in our evaluations (§7).

There are a variety of activation functions used for $\sigma^{(i)}$, including ReLU, Hyperbolic Tangent, (logistic) Sigmoid, AveragePool, and MaxPool [21]. In our examples, we will use the ReLU function, defined below, due to its simplicity and use in real-world DNN architectures, although our algorithms and theory work for arbitrary activation functions.

Definition 2.3. The *ReLU Activation Function* is a vector-valued function $\mathbb{R}^n \to \mathbb{R}^n$ defined component-wise by

$$ReLU(\vec{v})_i = \begin{cases} v_i & \text{if } v_i \geq 0 \\ 0 & \text{otherwise,} \end{cases}$$

where $ReLU(\vec{v})_i$ is the i^{th} component of the output vector $ReLU(\vec{v})$ and v_i is the i^{th} of the input vector $\vec{v}$.

Of particular note for our polytope repair algorithm (§6), some of the most common activation functions (particularly ReLU) are *piecewise-linear*.

Definition 2.4. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is *piecewise-linear* (PWL) if its input domain can be partitioned into finitely-many polytopes $X_1, X_2, \ldots, X_n$ such that, for each X_i, there exists some *affine* function f_i such that $f(x) = f_i(x)$ for every $x \in X_i$.

This paper uses the terms 'linear' and 'affine' interchangeably. It follows from Definition 2.4 that compositions of PWL functions are themselves PWL. Hence, a DNN using only PWL activation functions is also PWL in its entirety.

For a network using PWL activation functions, we can always associate with each input to the network an *activation pattern*, as defined below.

Definition 2.5. Let N be a DNN using only PWL activation functions. Then an *activation pattern* γ is a mapping from each activation function $\sigma^{(j)}$ to a linear region $\gamma(\sigma^{(j)})$ of $\sigma^{(j)}$. We say an activation pattern γ *holds* for a vector $\vec{v}$ if, for every layer j, we have $W^{(j)}\vec{v}^{(j-1)} \in \gamma(\sigma^{(j)})$.

Recall that $\vec{v}$ is the input to the first layer of the network while $\vec{v}^{(j-1)}$ is the intermediate input to the jth layer. For example, suppose N is a ReLU network where γ holds for vector v, then $\gamma(\sigma^{(j)})$ is exactly the set of nodes in layer j with positive output when evaluating the DNN on input $\vec{v}$.

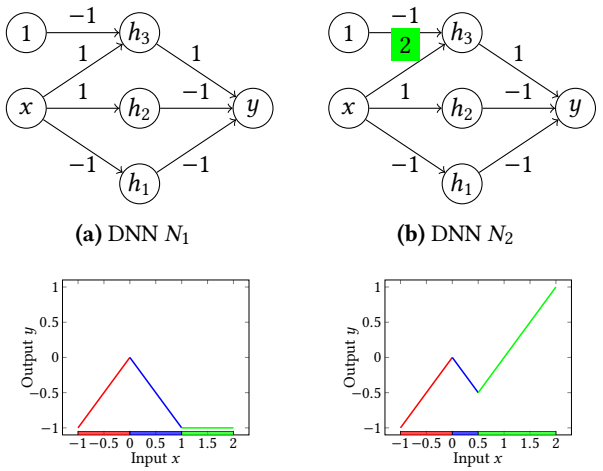
Let N be a DNN that uses only PWL activation functions. Then we notate by $LinRegions(N)$ the set of polytopes $X_1, X_2, \ldots, X_n$ that partition the domain of N such that the conditions in Definition 2.4 hold. In particular, we will use the partitioning for which we can assign each X_i a unique activation pattern γ_i such that γ_i holds for all $\vec{v} \in X_i$.

When appropriate, for polytope P in the domain of N, we will notate by $LinRegions(N, P)$ a partitioning $X_1, X_2, \ldots, X_n$ *of* P that meets the conditions in Definition 2.4. Formally, we have $LinRegions(N, P) := LinRegions(N_{\restriction P})$, where $N_{\restriction P}$ is the restriction of N to domain P.

Consider the ReLU DNN N_1 shown in Figure 3(a), which has one input x, one output y, and three so-called *hidden* nodes h_1, h_2, and h_3 using ReLU activation function. We will consider the input-output behavior of this network for the domain $x \in [-1, 2]$. The linear regions of N_1 are shown visually in Figure 3(c) as colored intervals on the x axis, which each map into the postimage according to some affine mapping which is specific to that region. In particular, we have three linear regions:

$$LinRegions(N_1, [-1, 2]) = \{[-1, 0], [0, 1], [1, 2]\}. \quad (1)$$

Each linear region corresponds to a particular *activation pattern* on the hidden nodes; i.e., which ones are in the zero region or the identity region. The first linear region, $[-1, 0]$ (red), corresponds to the activation pattern where only h_1 is activated. The second linear region, $[0, 1]$ (blue), corresponds to the activation pattern where only h_2 is activated. Finally, the third linear region, $[1, 2]$ (green), corresponds to the activation pattern where both h_2 and h_3 are activated.



(a) DNN N_1 **(b)** DNN N_2

(c) Input-output plot of N_1 **(d)** Input-output plot of N_2

Figure 3. Example DNNs and their input-output behavior. The h_i nodes have ReLU activation functions. Colored bars on the x axis denote the linear regions.

In practice, we can quickly compute $LinRegions(N, P)$ for either large N with one-dimensional P or medium-sized N with two-dimensional P. We use the algorithm of Sotoudeh and Thakur [53] for computing $LinRegions(N, P)$ when P is one- or two-dimensional.

Definition 2.6. A *linear program* (LP) with n constraints on m variables is a triple $(A, \vec{b}, \vec{c})$ where A is an $n \times m$ matrix, $\vec{b}$ is an n-dimensional vector, and $\vec{c}$ is an m-dimensional vector.

A *solution to the linear program* is an m-dimensional vector $\vec{x}$ such that (i) $A\vec{x} \leq \vec{b}$, and (ii) $\vec{c} \cdot \vec{x}$ is minimal among all $\vec{x}$ satisfying (i).

Linear programs can be solved in polynomial time [38], and many efficient, industrial-grade LP solvers such as the Gurobi solver [24] exist. Through the addition of auxiliary variables, it is also possible to encode in an LP the objective of minimizing the ℓ_1 and/or ℓ_∞ norms of $\vec{x}$ [8].

3 Overview

This paper discusses how to repair DNNs to enforce precise *specifications*, i.e., constraints on input-output behavior.

3.1 Provable Pointwise Repair

The first type of specification we will consider is a *point repair specification.* In this scenario, we are given a finite set of input points along with, for each such point, a subset of the output region which we would like that point to be mapped into by the network.

Consider DNN N_1 in Figure 3(a). We see that $N_1(0.5) = -0.5$ and $N_1(1.5) = -1$. We want to *repair* it to form a new network N' such that

$$(-1 \leq N'(0.5) \leq -0.8) \wedge (-0.2 \leq N'(1.5) \leq 0). \quad (2)$$

We formalize this point specification as $(X, A^{\cdot}, b^{\cdot})$ where X is a finite collection of *repair points* $X = \{X_1 = 0.5, X_2 = 1.5\}$, and we associate with each $x \in X$ a polytope in the output space defined by A^x, b^x that we would like it to be mapped into by N'. In this case, we can let $A^{X_1} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, b^{X_1} = \begin{bmatrix} -0.8 \\ 1 \end{bmatrix}, A^{X_2} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, and $b^{X_2} = \begin{bmatrix} 0 \\ 0.2 \end{bmatrix}$ representing the polyhedral constraints $A^{X_1}N'(X_1) \leq b^{X_1} \wedge A^{X_2}N'(X_2) \leq b^{X_2}$. These constraints are equivalent to Equation 2.

The general affine constraint form we use is very expressive. For example, it can express constraints such as "the i^{th} output component is larger than all others," which for a multi-label classification network is equivalent to ensuring that the point is classified with label i.

The two roles of a ReLU. At first glance, it is tempting to directly encode the DNN in an SMT solver like Z3 [12] and attempt to solve for weight assignments that cause the desired classification. However, in practice this quickly becomes infeasible even for networks with very few nodes.

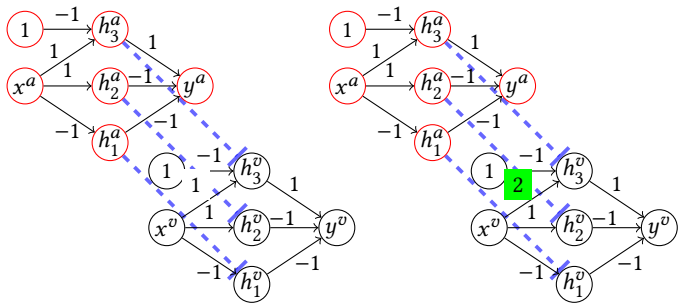
To understand the key reason for this infeasibility, consider what happens when a single weight in N_1 is modified to construct the new DNN N_2, shown in Figure 3(b). In particular, the weight on $x \longrightarrow h_3$ is changed from a 1 to a 2. Comparing Figure 3(d) with Figure 3(c), we see that changing this weight has caused *two* distinct changes in the plot:

1. The linear function associated with the green region has changed, and
2. *Simultaneously,* the linear regions themselves (shown on the x axis) have changed, with the green region growing to include parts of the space originally in the blue region. In particular, $LinRegions(N_2, [-1, 2]) = \{[-1, 0], [0, 0.5], [0.5, 2]\}$, different from Equation 1.

We use *coupling* to refer to the fact that the weights in a ReLU DNN simultaneously control *both* of these aspects. This coupling causes repair of DNNs to be computationally infeasible, because the impact of changing a weight in the network with respect to the output of the network on a fixed input is non-linear; it 'jumps' every time the linear region that the point falls into changes. This paper shows that *decoupling* these two roles leads to a generalized class of neural networks along with a polynomial-time repair algorithm.

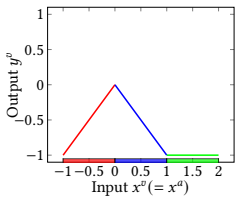
Decoupling activations from values. The key insight of this paper is a novel DNN architecture, *Decoupled DNNs* (DDNNs), defined in §4, that strictly generalizes standard feed-forward DNNs while at the same time allowing us to decouple the two roles that the parameters play.

Figure 4(a) shows a DDNN N_3 equivalent to N_1 from Figure 3(a). Most notably, every decoupled DNN consists of *two* 'sub-networks,' or *channels.* The *activation channel,* shown in red, is used to determine the positions of the linear regions. Meanwhile, the *value channel* determines the output map within each linear region. The activation channel influences the value channel via the blue edges, which indicate that the

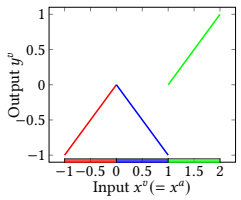


(a) Decoupled DNN $N_3 = (N_1, N_1)$.

(b) Decoupled DNN $N_4 = (N_1, N_2)$.



(c) Input-output plot of N_3.



(d) Input-output plot of N_4.

Figure 4. Decoupled DNNs N_3 and N_4 and their input-output behavior. DDNNs N_3 and N_4 have the same activation channel N_1, but different value channels.

adjacent value node is activated only if the corresponding activation node is. For example, if the input to h_2^a is negative, then h_2^v will output zero regardless of the input to h_2^v.

To compute the output of a DDNN on a given input x_0, we first set $x^a = x_0$, evaluate the activation channel, and record which of the hidden nodes h_i^a were active (received a positive input) or inactive (otherwise). Then, we set $x^v = x_0$ and evaluate the value channel, except instead of activating a node if *its* input is non-negative, we activate the node if *the corresponding activation channel node was activated.* In this way, activation nodes can 'mask' their corresponding value nodes, as notated with the blue edges in Figure 4(a).

Now, consider what happens when we change a weight in only the *value channel,* as shown in Figure 4(b). In that scenario, on any given point, the activation pattern for any given input *does not change,* and so the locations of the linear regions on the x axis of Figure 4(d) are unchanged from Figure 4(c). However, what we find is that *the linear function within any given region does change.* Note that in this case only the green line has changed, however in deeper networks changing any given weight can change all of the lines.

Repair of DDNNs. This observation foreshadows two of our key theoretical results in §4. The first theorem (Theorem 4.5) shows that, for any given input, the output of the DDNN varies *linearly* on the change of any given weight in the value channel. In fact, we will show the stronger fact that the output depends linearly with the change of any *layer of weights* in the value channel.

Using this fact, we can *reduce pointwise repair of a single layer in the DDNN to a linear programming (LP) problem.* In the running example, suppose we want to repair the first value layer of DDNN N_3 to satisfy Equation 2. Let Δ be the difference in the first layer weights, where Δ_i is the change in the weight on edge $x^v \to h_i^v$, Δ_4 is the change in the weight on edge $1 \to h_3^v$, and N' be the DDNN with first-layer value weights changed by Δ. Then, Theorem 4.5 guarantees that $N'(X_1) = \begin{bmatrix}-0.5\end{bmatrix} + \begin{bmatrix}0 & -0.5 & 0 & 0\end{bmatrix}\vec{\Delta} = -0.5 - 0.5\Delta_2$, while $N'(X_2) = \begin{bmatrix}-1\end{bmatrix} + \begin{bmatrix}0 & -1.5 & 1.5 & 1\end{bmatrix}\vec{\Delta} = -1 - 1.5\Delta_2 + 1.5\Delta_3 + \Delta_4$. Hence, we can encode our specification as an LP like so: $(-1 \le -0.5 - 0.5\Delta_2 \le -0.8) \wedge (-0.2 \le -1 - 1.5\Delta_2 + 1.5\Delta_3 + \Delta_4 \le 0)$, or in a more formal LP form,

$$\begin{bmatrix} 0 & -0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & -1.5 & 1.5 & 1 \\ 0 & 1.5 & -1.5 & -1 \end{bmatrix} \begin{bmatrix} \Delta_1 \\ \Delta_2 \\ \Delta_3 \\ \Delta_4 \end{bmatrix} \le \begin{bmatrix} -0.3 \\ 0.5 \\ 1 \\ -0.8 \end{bmatrix}$$

We can then solve for Δ using an off-the-shelf LP solver, such as Gurobi [24]. We can also simultaneously optimize a linear objective, such as the ℓ_∞ or ℓ_1 norm, to find the satisfying repair with the *provably* smallest Δ. This helps ensure locality of the repair and preserve the otherwise-correct existing behavior of the network. In this case, we can find that the smallest repair with respect to the ℓ_1 norm is $\Delta_1 = 0, \Delta_2 = 0.6, \Delta_3 = 1.1\overline{3}, \Delta_4 = 0$. The corresponding repaired DDNN N_5 is shown in Figure 5(a) and plotted in Figure 5(c), where we can see that the repaired network satisfies the constraints because $N_5(0.5) = -0.8$ and $N_5(1.5) = -0.2$. Notably, the linear regions of N_5 are the same as those of N_1.

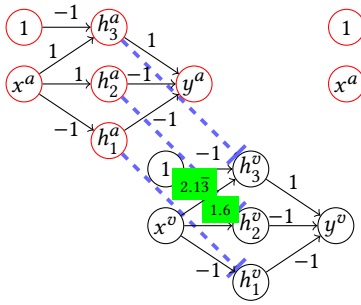
Non-ReLU, non-fully-connected, activation functions. While we have focused in this overview on the ReLU case for ease of exposition, the key result of Theorem 4.5 also holds for a generalization of DDNNs using arbitrary activation functions, such as tanh and sigmoid. Hence, our pointwise repair algorithm works for arbitrary feed-forward networks. Similarly, although we have formalized DNNs assuming fully-connected layers, our approach can repair convolutional and other similar types of layers as well (as demonstrated in §7.1).

3.2 Provable Polytope Repair

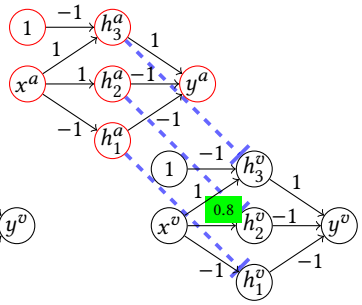
We now consider *Provable Polytope Repair.* The specification for provable polytope repair constrains the output of the network on finitely-many *polytopes* in the input space, each one containing potentially *infinitely many points.* For example, given the DNN N_1 we may wish to enforce a specification

$$\forall x \in [0.5, 1.5]. \quad -0.8 \le N'(x) \le -0.4. \tag{3}$$

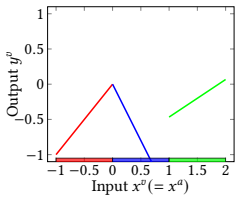
We represent this as a polytope specification with one input polytope, $X = \{P_1 = [0.5, 1.5]\}$, which should map to the polytope in the output space given by $A^{P_1} = \begin{bmatrix}1\\-1\end{bmatrix}$, $b^{P_1} = \begin{bmatrix}-0.4\\0.8\end{bmatrix}$. The constraint $\forall x \in P_1. A^{P_1} N'(x) \le b^{P_1}$ is then equivalent to the specification in Equation 3.



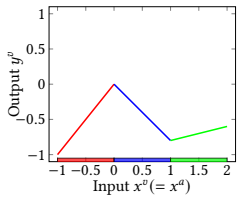
(a) Pointwise Repaired DDNN N_5.



(b) Polytope Repaired DDNN N_6.



(c) Input-output plot of N_5.



(d) Input-output plot of N_6.

Figure 5. Repaired DDNNs.

Reduction of polytope repair to pointwise repair. Our key insight (Theorem 4.6) is that, for piecewise-linear DDNNs, if we only change the value channel parameters, then we can reduce polytope repair to pointwise repair. To see this, recall that the value channel parameters *do not* change the location of the linear regions, only the behavior within each one. Within each linear region, the behavior of the network is linear, and, hence, convex. Convexity guarantees that any given polytope is mapped into another polytope if and only if its *vertices* are mapped into that polytope. Note that the assumption of piecewise-linearity is important here: in contrast to pointwise patching, which works for any feed-forward DNN, our polytope patching algorithm requires the activation functions to be piecewise-linear.

In our 1D example, this observation is the fact that a line lies in the desired interval of $[-0.8, -0.4]$ if and only if its endpoints do. In fact, the input region of interest in our example of $[0.5, 1.5]$ overlaps with two of these lines (the blue and green line segments in Figure 3(c)). Hence, we must ensure that both of those lines have endpoints in $[-0.8, -0.4]$.

Thus, the polytope specification is met if and only if the point specification with $K = \{K_1 = 0.5, K_2 = 1, K_3 = 1, K_4 = 1.5\}$ and $A^{K_1} = A^{K_2} = A^{K_3} = A^{K_4} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $b^{K_1} = b^{K_2} = b^{K_3} = b^{K_4} = \begin{bmatrix} -0.4 \\ 0.8 \end{bmatrix}$ is met. We call the points in K *key points* because the behavior of the repaired network N' on these points determines the behavior of the network on all of P_1.

Note that K_2 and K_3 both refer to the same input point, 1. This is because we need to verify that $N'(1)$ is in the desired output range when approaching either from the left or the right, as we want to verify it for both the blue and the green lines in Figure 4(c). This technicality is discussed in more detail in the extended version of this paper [52].

Therefore, we have reduced the problem of repair on polytopes to repair on finitely-many *key points,* which are the vertices of the polytopes in the specification intersected with the polytopes defining the linear regions of the DNN. We can apply the algorithm discussed for pointwise repair to solve for a minimal fix to the first layer. In particular, we get the linear constraints: $-0.8 \leq -0.5 - 0.5\Delta_2 \leq -0.4$, $-0.8 \leq -1 - \Delta_2 \leq -0.4$, $-0.8 \leq -1 - \Delta_2 + \Delta_3 + \Delta_4 \leq -0.4$, and $-0.8 \leq -1 - 1.5\Delta_2 + 1.5\Delta_3 + \Delta_4 \leq -0.4$, for which an ℓ_1-minimal solution is the single weight change $\Delta_2 = -0.2$. The corresponding repaired DDNN is shown in Figure 5(b) and plotted in Figure 5(d), which shows that the repaired network satisfies the constraints.

4 Decoupled DNNs

In this section, we formally define the notion of a *Decoupled Deep Neural Network* (DDNN), which is a novel DNN architecture that will allow for polynomial-time layer repair.

A DDNN is defined similarly to a DNN (Definition 2.1), except it has two sets of weights; the *activation channel* has weights $W^{(a,i)}$ and the *value channel* has weights $W^{(v,i)}$.

Definition 4.1. A *Decoupled DNN* (DDNN) having layers of size $s_0, \ldots, s_n$ is a list of triples $(W^{(a,1)}, W^{(v,1)}, \sigma^{(1)}), \ldots, (W^{(a,n)}, W^{(v,n)}, \sigma^{(n)})$, where $W^{(a,i)}$ and $W^{(v,i)}$ are $s_i \times s_{i-1}$ matrices and $\sigma^{(i)} : \mathbb{R}^{s_i} \to \mathbb{R}^{s_i}$ is some *activation function.*

We now give the semantics for a DDNN. The input $\vec{v}$ is duplicated to form the inputs $\vec{v}^{(a,0)}$ and $\vec{v}^{(v,0)}$ to the activation and value channels, respectively. The semantics of the activation channel, having *activation vectors* $\vec{v}^{(a,i)}$, is the same as for a DNN (Definition 2.2). The semantics for the value channel with *value vectors* $\vec{v}^{(v,i)}$ is similar, except instead of using the activation function $\sigma^{(i)}$, we use the *linearization* of $\sigma^{(i)}$ around the input $W^{(a,i)}\vec{v}^{(a,i-1)}$ of the corresponding activation layer, as defined below.

Definition 4.2. Given function $f : \mathbb{R}^n \to \mathbb{R}^m$ differentiable at $\vec{v}_0$, define the *Linearization of f around $\vec{v}_0$* to be the function: $Linearize[f, \vec{v}_0](\vec{x}) := f(\vec{v}_0) + D_{\vec{v}} f(\vec{v}_0) \times (\vec{x} - \vec{v}_0)$.

Above, $D_{\vec{v}} f(\vec{v}_0)$ is the Jacobian of f with respect to its input at the point point $\vec{v}_0$. The Jacobian generalizes the notion of a scalar derivative to vector functions (see the extended version of this paper [52]). The output of the DDNN is taken to be the output $\vec{v}^{(v,n)}$ of the value channel. These DDNN semantics are stated below.

Definition 4.3. The *function* $N : \mathbb{R}^{s_0} \to \mathbb{R}^{s_n}$ *associated with the DDNN* N with layers $(W^{(a,i)}, W^{(v,i)}, \sigma^{(i)})$ is given by $N(\vec{v}) = \vec{v}^{(v,n)}$ where
$\vec{v}^{(a,0)} := \vec{v}^{(v,0)} := \vec{v}$,
$\vec{v}^{(a,i)} := \sigma^{(i)}(W^{(a,i)}\vec{v}^{(a,i-1)})$, and
$\vec{v}^{(v,i)} := Linearize[\sigma^{(i)}, W^{(a,i)}\vec{v}^{(a,i-1)}](W^{(v,i)}\vec{v}^{(v,i-1)})$.

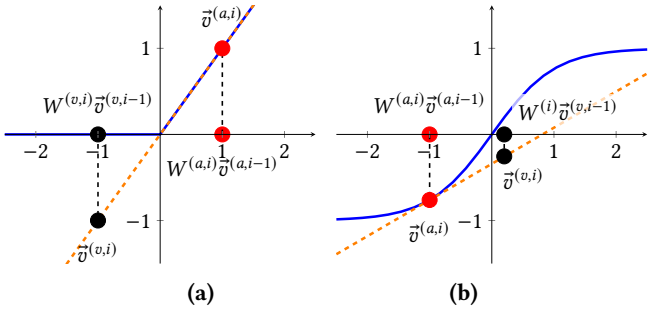


Figure 6. (a) Linearized ReLU and (b) Linearized Tanh.

DDNNs can be extended to non-differentiable activation functions as discussed in Sotoudeh and Thakur [52].

Consider the ReLU activation function in Figure 6(a). We see that the activation node gets an input of 1 (in red on the x axis) and so produces an output (in the activation channel) of 1. The linearization of the ReLU function around the point 1 is the identity function $f(x) = x$ (shown in orange). Thus, we use the function $f(x) = x$ as the activation function for the corresponding value node. This means that if the value node gets an input of, say, -1 as shown in black in Figure 6(a), then its output will be -1. Effectively, if the input to the *activation node* is positive, then the corresponding value node will be activated (i.e., pass its input through as its output). On the other hand, if the input to the activation node were negative, then the linearization would be the zero function $f(x) = 0$. The value node would use that as its activation function, effectively deactivating it regardless of its input from the value channel.

Consider also the Tanh activation function (Figure 6(b)). The activation channel behaves as normal, each node outputting the Tanh of its input. For example, if the input to the activation node is -1 (shown in red), then its output is $\tanh(-1)$ (shown below it in red). However, for the value channel, we use the linearization of tanh around the input to the corresponding activation node. In this case, we linearize Tanh around -1 to get the line shown in orange, which is used as the activation function for the value channel.

Thus, as we have shown, each node in the activation channel produces a new activation function to be used in the value channel.

Key results. The first key result shows that the class of DDNNs generalizes that of DNNs; for any DNN, the below theorem gives a trivial construction for an exactly equivalent DDNN by setting the activation and value channel weights to be identical to the weights of the DNN.

Theorem 4.4. *Let N be a DNN with layers $(W^{(i)}, \sigma^{(i)})$ and M be the DDNN with layers $(W^{(i)}, W^{(i)}, \sigma^{(i)})$. Then, as functions, $N = M$.*

Proof. Let $\vec{v}$ be chosen arbitrarily. Let $\vec{v}^{(i)}$ be the intermediates of N on $\vec{v}$ according to Definition 2.2, and $\vec{v}^{(a,i)}, \vec{v}^{(v,i)}$ be the intermediates of M on $\vec{v}$ according to Definition 4.3.

We now prove, for all i, that $\vec{v}^{(v,i)} = \vec{v}^{(a,i)} = \vec{v}^{(i)}$. We proceed by induction on i. By definition, $\vec{v}^{(a,0)} = \vec{v}^{(v,0)} = \vec{v}^{(0)}$. Now, suppose for sake of induction that $\vec{v}^{(a,i)} = \vec{v}^{(v,i)} = \vec{v}^{(i)}$. Then we have by definition and the inductive hypothesis

$$\vec{v}^{(a,i+1)} = \sigma^{(i+1)}(W^{(i+1)}\vec{v}^{(a,i)}) = \sigma^{(i+1)}(W^{(i+1)}\vec{v}^{(i)}) = \vec{v}^{(i+1)},$$

as well as

$$\begin{aligned}
&\vec{v}^{(v,i+1)} \\
&= \mathit{Linearize}[\sigma^{(i+1)}, W^{(i+1)}\vec{v}^{(a,i)}](W^{(i+1)}\vec{v}^{(v,i)}) && \text{(Definition)} \\
&= \mathit{Linearize}[\sigma^{(i+1)}, W^{(i+1)}\vec{v}^{(i)}](W^{(i+1)}\vec{v}^{(i)}) && \text{(Ind. Hyp.)} \\
&= \sigma^{(i+1)}(W^{(i+1)}\vec{v}^{(i)}) && \text{(Linearization)} \\
&= \vec{v}^{(i+1)}, && \text{(Definition)}
\end{aligned}$$

because linearizations are exact at their center point. By induction, then, $\vec{v}^{(v,i)} = \vec{v}^{(i)}$ for $0 \le i \le n$, and in particular $\vec{v}^{(v,n)} = \vec{v}^{(n)}$. But this is by definition $M(\vec{v}) = N(\vec{v})$, and as $\vec{v}$ was chosen arbitrarily, this gives us $N = M$ as functions. □

Our next result proves that the output of a DDNN varies linearly with changes in any given value channel layer weights. Note that DDNNs are *not* linear functions with respect to their *input*, only with respect to the *value weights*.

Theorem 4.5. *Let j be a fixed index and N be DDNN with layers $(W^{(a,i)}, W^{(v,i)}, \sigma^{(i)})$. Then, for any $\vec{v}$, $N(\vec{v})$ varies linearly* as a function of $W^{(v,j)}$.

Proof. Changing $W^{(v,j)}$ does not modify the values of $\vec{v}^{(a,i)}$ or $\vec{v}^{(v,i)}$ for $i < j$, hence (i) we can assume WLOG that $j = 1$, and (ii) all of the $\vec{v}^{(a,i)}$s remain constant as we vary $W^{(v,1)}$. Consider now the value of $\vec{v}^{(v,1)} = \mathit{Linearize}[\sigma^{(1)}, W^{(a,1)}\vec{v}^{(a,0)}](W^{(v,1)}\vec{v}^{(v,0)})$. This is by definition an linear function of $W^{(v,1)}\vec{v}^{(v,0)}$, which is in turn an linear function of $W^{(v,1)}$.

Now, consider any $i > 1$. We have by definition $\vec{v}^{(v,i)} = \mathit{Linearize}[\sigma^{(i)}, W^{(a,i)}\vec{v}^{(a,i-1)}](W^{(v,i)}\vec{v}^{(v,i-1)})$, which, because we are fixing $W^{(v,i)}$ for $i > 1$, is an linear function *with respect to* $\vec{v}^{(v,i-1)}$.

We showed that $\vec{v}^{(1)}$ is linear *with respect to* $W^{(v,1)}$, while $\vec{v}^{(v,i)}$ for $i > 1$ is linear *with respect to* $v^{(v,i-1)}$. But compositions of linear functions are also linear, hence in total $\vec{v}^{(v,n)}$ is linear with respect to $W^{(v,1)}$ as claimed. □

Our final result proves that modifying only the value weights in a DDNN does *not* change its linear regions.

Theorem 4.6. *Let N be a PWL DNN with layers $(W^{(i)}, \sigma^{(i)})$ and define a DDNN M with layers $(W^{(i)}, W^{(v,i)}, \sigma^{(i)})$. Then, within any linear region in $\mathit{LinRegions}(N)$, M is also linear.*

Proof. Within any linear region of N, all of the activations are the same. This means that all of the linearizations used in the computation of $\vec{v}^{(v,i+1)}$ do not change in the linear region. Therefore, considering only the value channel, we can write $M(\vec{v}) = \vec{v}^{(v,n)}$ as a concatenation of linear functions, which is linear with respect to the input. □

5 Provable Pointwise Repair

This section defines and gives an algorithm for provable pointwise repair.

Definition 5.1. Let $N : \mathbb{R}^n \to \mathbb{R}^m$. Then $(X, A^{\cdot}, b^{\cdot})$ is a *pointwise repair specification* if X is a finite subset of $\mathbb{R}^n$ and for each $x \in X$, A^x is a $k_x \times m$ matrix while b^x is a k_x-dimensional vector.

Definition 5.2. Let $N : \mathbb{R}^n \to \mathbb{R}^m$ and $(X, A^{\cdot}, b^{\cdot})$ be some pointwise repair specification. Then N *satisfies* $(X, A^{\cdot}, b^{\cdot})$, written $N \Vdash (X, A^{\cdot}, b^{\cdot})$, if $A^x N(x) \leq b^x$ for every $x \in X$.

Definition 5.3. Let N be a DDNN and $(X, A^{\cdot}, b^{\cdot})$ be some pointwise repair specification. Then another DDNN N' is a *repair* of N if $N' \Vdash (X, A^{\cdot}, b^{\cdot})$. It is a *minimal repair* of N if $|\theta' - \theta|$ is minimal among all repairs, where θ and θ' are parameters of N and N', respectively, and $|\cdot|$ is some user-defined measure of size (e.g., ℓ_1 norm). It is a *minimal layer repair* if it is minimal among all repairs that only modify a single, given layer.

Assumptions on the DNN. For point repair, we require only that the activation functions be (almost-everywhere) differentiable so that we can compute a Jacobian. This is already the case for every DNN trained via gradient descent, however even this requirement can be dropped with slight modification to the algorithm (see the extended version of this paper [52]). Of particular note, we *do not* require that the DNN be piecewise-linear.

Algorithm. Algorithm 1 presents our pointwise repair algorithm, which reduces provable pointwise repair to an LP. $params(L)$ returns the parameters of layer L. The notation $D_{params(N_i'^v)} N'(x)$ refers to the Jacobian of the DDNN $N'(x)$ as a function of the parameters $params(N_i'^v)$ of the i^{th} value channel layer, i.e., $W^{(v,i)}$, while fixing input x. Given a set of affine constraints C, $Solve(C)$ returns a solution to the set of constraints or $\perp$ if the constraints are infeasible. *Solve* also guarantees to return the *optimal* solution according to some user-defined objective function, e.g., minimizing the ℓ_1 or ℓ_∞ norm. Finally, $DecoupledNetwork(a, v)$ constructs a decoupled neural network with activation layers a and value layers v. The next two theorems show the correctness, minimality, and running time of Algorithm 1.

Theorem 5.4. *Given a DNN N, a layer index i, and a point repair specification $(X, A^{\cdot}, b^{\cdot})$, let $N' =$ `PointRepair`$(N, i, X, A^{\cdot}, b^{\cdot})$. If $N' \neq \perp$, then $N' \Vdash (X, A^{\cdot}, b^{\cdot})$ and $\vec{\Delta}$ is a minimal layer repair. Otherwise, if $N' = \perp$, then no such single-layer DDNN repair satisfying the specification exists for the i^{th} layer.*

Proof. Lines 2–3 construct a DDNN N' equivalent to the DNN N (Theorem 4.4). For each point $x \in X$, line 5 considers the *linearization of N' around the parameters* for the value channel layer $N_i'^v$, namely: $N'(x; \vec{\Delta}) \approx N'(x; 0) + J^x \vec{\Delta}$ where $N'(x; \vec{\Delta})$ is the output of the DDNN when the parameters

Algorithm 1: `PointRepair`$(N, i, X, A^{\cdot}, b^{\cdot})$

Input: A DNN N defined by a list of its layers.
A layer index i to repair.
A finite set of points X.
For each point $x \in X$ a specification A^x, b^x asserting $A^x N'(x) \leq b^x$ where N' is the repaired network.
Output: A repaired DDNN N' or $\perp$.

```
/* C is a set of linear constraints on the
   parameter delta Δ⃗ each of the form (A, b)
   asserting AΔ⃗ ≤ b.                          */
```
1 $C \leftarrow \emptyset$
```
/* Decouple the activation, value layers      */
```
2 $N'^a, N'^v = copy(N), copy(N)$
```
/* Construct DDNN N' equivalent to DNN N     */
```
3 $N' \leftarrow DecoupledNetwork(N'^a, N'^v)$
4 **for** $x \in X$ **do**
```
    /* Jacobian wrt parameters of layer N_i'^v  */
```
5 $J^x \leftarrow D_{params(N_i'^v)} N'(x)$
```
    /* Encoded constraint A^x(N(x) + J^x Δ⃗) ≤ b^x  */
```
6 $C \leftarrow C \cup \{(A^x J^x, b^x - A^x N(x))\}$
7 $\vec{\Delta} \leftarrow Solve(C)$
8 **if** $\vec{\Delta} = \perp$ **then return** $\perp$
```
/* Update value layer i.                     */
```
9 $params(N_i'^v) \leftarrow params(N_i'^v) + \vec{\Delta}$
10 **return** N'

of the ith layer are changed by $\vec{\Delta}$. In particular, by Theorem 4.4 if $\vec{\Delta} = 0$, N' is equivalent to N (as a function). Hence $N'(x; \vec{\Delta}) \approx N(x) + J^x \vec{\Delta}$. Finally, according to Theorem 4.5, this linear approximation is *exact* for the DDNN when we only modify the parameters for a single value channel layer $N_i'^v$, i.e., $N'(x; \vec{\Delta}) = N(x) + J^x \vec{\Delta}$.

Thus, after the for loop, the set C contains constraints asserting that $A^x N'(x; \vec{\Delta}) \leq b^x$, which are exactly the constraints which our algorithm needs to guarantee. Finally, we solve the constraints for $\vec{\Delta}$ using an LP solver and return the final DDNN. Hence, if *Solve* returns $\perp$, there is no satisfying repair. If it returns a repair, then the LP solver guarantees that it satisfies the constraints and no smaller $\vec{\Delta}$ exists. □

Theorem 5.5. *Algorithm 1 halts in polynomial time with respect to the size of the point repair specification $(X, A^{\cdot}, b^{\cdot})$.*

Proof. The LP corresponding to C has one row per row of A^x and one column per weight in $N_i'^v$, both of which are included in the size of the input. Thus, as LPs can be solved in polynomial time, the desired result follows. □

Notably, the above proof assumes the Jacobian computation on line 5 takes polynomial time; this is the case for all common activation functions used in practice. The authors are not aware of any actual or proposed activation function that would violate this assumption.

6 Provable Polytope Repair

This section defines and gives an algorithm for provable polytope repair.

Definition 6.1. Let $N : \mathbb{R}^n \to \mathbb{R}^m$. Then $(X, A^{\cdot}, b^{\cdot})$ is a *polytope repair specification* if X is a finite set of bounded convex polytopes in $\mathbb{R}^n$ and for each $P \in X$, A^P is a $k_P \times m$ matrix while b^P is a k_P-dimensional vector.

Definition 6.2. Let $N : \mathbb{R}^n \to \mathbb{R}^m$ and $(X, A^{\cdot}, b^{\cdot})$ be a polytope repair specification. Then N *satisfies* $(X, A^{\cdot}, b^{\cdot})$, written $N \Vdash (X, A^{\cdot}, b^{\cdot})$, if $A^P N(x) \leq b^P$ for every $P \in X$ and $x \in P$.

Definition 6.3. Let N be a DDNN and $(X, A^{\cdot}, b^{\cdot})$ be some polytope repair specification. Then another DDNN N' is a *repair* of N if $N' \Vdash (X, A^{\cdot}, b^{\cdot})$. It is a *minimal repair* of N if $|\theta' - \theta|$ is minimal among all repairs, where θ and θ' are parameters of N and N' respectively, and $|\cdot|$ is some user-defined measure of size (e.g., ℓ_1 norm). It is a *minimal layer repair* if it is minimal among all repairs that only modify a single, given layer.

The quantification in the definition of $N \Vdash (X, A^{\cdot}, b^{\cdot})$ is over an *infinite* set of points $x \in P$. The rest of this section is dedicated to *reducing* this infinite quantification to an equivalent finite one.

Assumptions on the DNN. For polytope repair, we assume that the activation functions used by the DNN are *piecewise-linear* (Definition 2.4). This allows us to *exactly* reduce polytope repair to point repair, meaning a satisfying repair exists if and only if the corresponding point repair problem has a solution. §9 discusses future work on extending this approach to non-PWL activation functions.

Algorithm. Our polytope repair algorithm is presented in Algorithm 2. We reduce the polytope specification $(X, A^{\cdot}, b^{\cdot})$ to a provably-equivalent *point* specification $(X', A'^{\cdot}, b'^{\cdot})$. For each polytope in the polytope repair specification, we assert the same constraints in the point specification except only on the *vertices* of the linear regions of N on that polytope. The next two theorems show the correctness, minimality, and running time of Algorithm 2.

Theorem 6.4. *Let $N' =$ `PolytopeRepair`$(N, i, X, A^{\cdot}, b^{\cdot})$ for a given DNN N, layer index i, and polytope repair specification $(X, A^{\cdot}, b^{\cdot})$. If $N' \neq \bot$, then $N' \Vdash (X, A^{\cdot}, b^{\cdot})$ and $\vec{\Delta}$ is a minimal layer repair. Otherwise, if $N' = \bot$, then no such single-layer DDNN repair satisfying the specification exists for the i^{th} layer.*

Proof. Consider an arbitrary $P \in X$ and arbitrary linear region $R \in LinRegions(N, P)$. By Theorem 4.6, linear regions are the same in the original network N and repaired network N'. Hence R is also a linear region in N'. Thus, on R, N' is equivalent to some linear function. It is a known result in convex geometry that linear functions map polytopes to polytopes and vertices to vertices, i.e., the vertices of the postimage $N'(R)$ are given by $N'(v)$ for each vertex v of R. The polytope $N'(R)$ is contained in another polytope if and only if its vertices are. Therefore, $N'(R)$ is contained in the polytope defined by A^P, b^P if and only if its vertices $N'(v)$ are. Because P and R were chosen arbitrarily and contain all of X, the constructed point repair specification $(X', A'^{\cdot}, b'^{\cdot})$ is equivalent to the polytope repair specification $(X, A^{\cdot}, b^{\cdot})$. The claimed results then follow directly from Theorem 5.4. □

Algorithm 2: `PolytopeRepair`$(N, i, X, A^{\cdot}, b^{\cdot})$

Input: A piecewise-linear DNN N with n inputs and m outputs defined by a list of its layers.
A layer index i to repair.
A finite set of polytopes X.
For each polytope $P \in X$ a specification A^P, b^P asserting $A^P N'(x) \leq b^P$ for every $x \in P$ where N' is the repaired network.
Output: A repaired DDNN N' or $\bot$.

```
/* Point repair specification                    */
1 (X', A'^·, b'^·) ← (∅, ∅, ∅)
2 for P ∈ X do
3     for R ∈ LinRegions(N, P) do
4         for v ∈ Vertices(R) do
5             X'.push(v)
6             A'^v, b'^v ← A^P, b^P
7 return PointRepair(N, i, X', A'^·, b'^·)
```

Theorem 6.5. *Algorithm 2 halts in polynomial time with respect to the size of the polytope repair specification $(X, A^{\cdot}, b^{\cdot})$ and the number of vertex points v.*

Proof. This follows directly from the time bounds we have established earlier on `PointRepair` in Theorem 5.5. □

The running time of Algorithm 2 depends on the number of linear regions and the number of vertices in each region (line 4). Although in the worst case there are exponentially-many such linear regions, theoretical results indicate that, for an n-dimensional polytope P and network N with m nodes, we expect $|LinRegions(N, P)| = O(m^n)$ [25, 26]. Sotoudeh and Thakur [53] show efficient computation of one- and two-dimensional *LinRegions* for real-world networks.

7 Experimental Evaluation

In this section, we study the efficacy and efficiency of Provable Repair (PR) on three different tasks. The experiments were designed to answer the following questions:

RQ1 How *effective* is PR in finding weights that satisfy the repair specification?

RQ2 How much does PR cause performance *drawdown*, on regions of the input space not repaired?

RQ3 How well do repairs *generalize* to enforce analogous specifications on the input space not directly repaired?

RQ4 How *efficient* is PR on different networks and dimensionalities, and where is most of the time spent?

Terms used. *Buggy network* is the DNN before repair, while the *fixed* or *repaired network* is the DNN after repair. *Repair layer* is the layer of the network that we applied provable repair to. *Repair set* is the pointwise repair specification or the polytope repair specification used to synthesize the repaired network. *Generalization set* is a set of points (or polytopes) that are disjoint from but simultaneously similar to the repair specification. *Drawdown set* is a set of points (or polytopes) that are disjoint from and not similar to the repair specification. *Efficacy* is the percent of the repair set that is classified correctly by the repaired network, i.e., the accuracy of the repaired network on the repair set. Our theoretical guarantees ensure that Provable Repair efficacy is always 100%. *Generalization Efficacy* is computed by subtracting the accuracy on the generalization set of the buggy network from that of the repaired network. Higher generalization efficacy implies better generalization of the fix. *Drawdown* is computed by subtracting the accuracy on the drawdown set of the repaired network from that of the buggy network. Lower drawdown is better, implying less forgetting.

Fine-Tuning Baselines. We compare Provable Repair (PR) to two baselines. The first baseline performs fine-tuning (FT) using gradient descent on all parameters at once, as proposed by [50]. FT runs gradient descent until all repair set points are correctly classified.

The second baseline, modified fine-tuning (MFT), is the same as FT except (a) MFT fine-tunes only a single layer, (b) MFT adds a loss term penalizing the ℓ_0 and ℓ_∞ norms of the repair, (c) MFT reserves 25% of the repair set as a holdout set, and (d) it stops once the accuracy on the holdout set begins to drop. Note that this approach does not achieve full efficacy; hence, it is not a valid repair algorithm (it does not repair the DNN). However, because of the early-stopping, MFT should have lower drawdown.

In all cases PR, FT, and MFT were given the same repair set (which included a number of non-buggy points). However, for polytope repair it is necessary to sample from that infinite repair set to form a finite repair set for FT and MFT, using the same number of randomly-sampled points as key points in the PR algorithm.

Evaluation Platform. All experiments were run on an Intel® Xeon® Silver 4216 CPU @ 2.10GHz. BenchExec [9] was used to ensure reproducibility and limit the experiment to 32 cores and 300 GB of memory. The PyTorch framework was used for performing linear algebra computations [46]. The experiments were run entirely on CPU. We believe that performance can be improved (i) by utilizing GPUs and (ii) by using TensorFlow [2], which has explicit support for Jacobian computations. We used Gurobi [24] to solve the LP problems. The code to reproduce our experimental results is available at https://github.com/95616ARG/PRDNN.

7.1 Task 1: Pointwise ImageNet Repair

Buggy network. SqueezeNet [31], a modern ImageNet convolutional neural network. We slightly modified the standard model [1], removing all output nodes except for those of the 9 classes used (see below). The resulting network has 18 layers, 727,626 parameters, and an accuracy of 93.6% on these classes using the official ImageNet validation set.

Repair set. The Natural Adversarial Examples (NAE) dataset, which are images commonly misclassified by modern ImageNet networks [27]. This dataset was also used by Sinitsin et al. [50]. For our nine classes (chosen alphabetically from the 200 total in the NAE dataset), the NAE dataset contains 752 color images. The buggy network has an accuracy of 18.6% on these NAE images. To measure scalability of repair, we ran 4 separate experiments, using subsets of 100, 200, 400, and all 752 NAE images as the repair specification.

Repair layer. PR and MFT was used to repair each of the 10 feed-forward fully-connected or convolution layers. Table 1 only lists the PR and MFT results for the layer with the best drawdown (BD).

Generalization set. The NAE images do not have a common feature that we would like the network to generalize from the repair. Thus, we were not able to construct a generalization set to evaluate generalization for Task 1.

Drawdown set. The entire set of approximately 500 validation images for the nine selected classes from the official ImageNet validation set [13].

Fine-tuning hyperparameters. Both FT and MFT use standard SGD with a learning rate of 0.0001 and no momentum, which were chosen as the best parameters after a small manual search. FT[1] and MFT[1] use batch size 2, while FT[2] and MFT[2] use batch size 16.

RQ1: Efficacy. When using 100, 200, and 400 points, our Provable Repair algorithm was able to find a satisfying repair for any layer, i.e., achieving 100% efficacy. For the 752 point experiment, Provable Repair was able to find a satisfying repair when run on 7 out of the 10 layers. It timed out on one of the layers and on the other two was able to prove that no such repair exists. FT was also able to find a 100%-efficacy repair in all four cases.

Meanwhile, the MFT baseline had efficacy of at most 28%, meaning it only marginally improved the network accuracy on NAE points from the original accuracy of 18%.

RQ2: Drawdown. Table 1 summarizes the drawdown of Provably Repaired networks on this task. In all cases, PR was able to find a layer whose repair resulted in under 6% drawdown. Extended results are in the extended version of this paper. Per-layer drawdown is shown in Figure 7(a), where we see that for this task repairing earlier layers can lead to much higher drawdown while latter layers result in consistently lower drawdown. This suggests a heuristic for repairing ImageNet networks, namely focusing on latter layers in the network.

Table 1. Summary of experimental results for Task 1. D: Drawdown (%), T: Time, BD: Best Drawdown, PR: Provable Repair, FT: Fine-Tuning baseline, MFT: Modified Fine-Tuning baseline (best layer), E: Efficacy (%). Efficacy of PR and FT is always 100%, hence Efficacy (E) numbers are only provided for MFT.

	PR (BD)		FT[1]		FT[2]		MFT[1] (BD)			MFT[2] (BD)		
Points	D	T	D	T	D	T	E	D	T	E	D	T
100	3.6	1m39.0s	10.2	4m31.8s	8.2	9m24.0s	24	-0.7	19.2s	28	0.0	4m4.9s
200	1.1	2m50.8s	9.6	12m19.5s	9.6	26m35.0s	21.5	-0.4	13.4s	20	-0.4	2m54.5s
400	5.1	4m45.3s	13.8	34m2.6s	11.1	1h9m26.8s	21.25	-0.4	29.3s	21	-0.4	1m32.1s
752	5.3	8m28.1s	15.4	1h22m18.7s	13.4	2h33m8.2s	19.4	-0.4	2m35.8s	18.2	-0.4	1m46.7s

By contrast, FT had consistently worse drawdown across multiple hyperparameter configurations, always above 8% and in some cases above 15%. This highlights how the guarantee of finding a minimal fix using Provable Repair can lead to significantly more localized fixes, preventing the DNN from forgetting what it had learned previously as is often a major risk when fine-tuning.

The MFT baseline had very low drawdown, but this comes at the cost of low efficacy.

RQ4: Efficiency. Table 1 also shows the amount of time taken to repair different numbers of points. Even with all 752 points, PR was able to find the repair with the best drawdown in under 10 minutes. If the layers are repaired in parallel, then all single-layer repairs can be completed in 4m, 8m, 18m, and 1h26m for 100, 200, 400, and 752 points respectively. If the layers are repaired in sequence, all single-layer repairs can be completed instead in 15m, 31m, 1h7m, and 4h10m respectively. To understand where time was spent, Figure 7(b) plots the time taken (vertical axis) against the layer fixed (horizontal axis) for the 400-point experiments. We have further divided the time spent into (i) time computing parameter Jacobians, (ii) time taken for the Gurobi optimization procedure, and (iii) other. For this model we find that a significant amount of time is spent computing Jacobians. This is because PyTorch lacks an optimized method of computing such Jacobians, and so we resorted to a sub-optimal serialized approach.

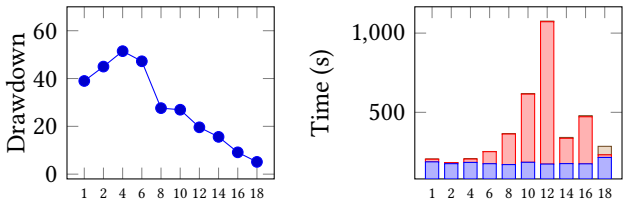


Figure 7. (a) Drawdown and (b) timing per repair layer when using 400 images in the repair set for Task 1. Blue: Jacobian, Red: Gurobi, Brown: Other.

By comparison, FT can take significantly longer. For example, on the 752-point experiment we saw FT take over 2 hours. In practice, this is highly dependent on the hyperparameters chosen, and hyperparameter optimization is a major bottleneck for FT in practice. The MFT baseline was also fast, but this comes at the cost of low efficacy.

7.2 Task 2: 1D Polytope MNIST Repair

Buggy network. The MNIST ReLU-3-100 DNN from Singh [48], consisting of 3 layers and 88,010 parameters for classifying handwritten digits. This network has an accuracy of 96.5% on the MNIST test set.

Repair set. We ran separate experiments with 10, 25, 50, or 100 lines; each line was constructed by taking as one endpoint an uncorrupted MNIST handwritten digit image and the other endpoint that same MNIST image corrupted with fog from MNIST-C [44]. If I is the uncorrupted image and I' is the fog-corrupted image, then this specification states that all (infinite) points along the line from I to I' must have the same classification as I. The buggy network has an accuracy of 20.0% on the corrupted endpoints used in the line repair specification.

Note that, unlike Provable Polytope Repair, both FT and MFT are given finitely-many points sampled from these lines — they can*not* make any guarantees about other points on these lines that are not in its sampled set.

Repair layer. We ran two repair experiments, repairing each of the last two layers. Because the network is fully-connected, the first layer has a very large number of nodes (because it is reading directly from the 784-dimensional input image), leading to a very large number of variables in the constraints. By comparison, SqueezeNet used convolutional layers, so the size of the input does not matter.

Generalization set. The MNIST-C fog test set consisting of 10,000 fog-corrupted MNIST images as the generalization set. The accuracy of the buggy network is 19.5% on this generalization set.

Drawdown set. The drawdown set is the official MNIST test set, which contains 10,000 (uncorrupted) MNIST images. The images in this test set are the exactly the uncorrupted

versions of those in the generalization set. The buggy network has 96.5% accuracy on the drawdown set.

Fine-tuning hyperparameters. Both use standard SGD with a batch size of 16 and momentum 0.9, chosen as the best parameters after a small manual search. FT[1] and MFT[1] use a learning rate of 0.05 while FT[2] and MFT[2] use a learning rate of 0.01.

Table 2 and Table 3 summarize the results for Task 2 when repairing Layer 2 and Layer 3. The "Lines" column lists the number of lines in the repair specification. The "Points" column lists the number of key points in the *LinRegions*, i.e., the size of the constructed X' in Algorithm 2.

RQ1: Efficacy. PR always found a repaired network, i.e., one guaranteed to correctly classify all of the infinitely-many points on each of the lines used in the repair specification.

FT could usually find a repaired network that achieved 100% accuracy on its sampled repair points, but could not make any guarantee about the infinitely-many other points in the line specification. Furthermore, in one configuration FT timed out after getting stuck in a very bad local minima, resulting in a network with near-chance accuracy. This highlights how extremely sensitive FT is to the choice of hyperparameters, a major inconvenience when attempting to apply the technique in practice when compared to our hyperparameter-free LP formulation.

Meanwhile, MFT was only able to achieve at most 71.3% efficacy, and like FT this does not ensure anything about the infinitely-many other points in the specification.

RQ2: Drawdown. Provable Repair results in low drawdown on this task. Repairing Layer 2 consistently results in less drawdown, with a drawdown of 2.4% when repairing using all 100 lines. However, even when repairing Layer 3 the drawdown is quite low, always under 6%. FT has significantly worse drawdown, up to 56.0% even when FT terminates successfully. This highlights how the Provable Repair guarantee of finding the *minimal* repair significantly minimizes forgetting. Here again, FT is extremely sensitive to hyperparamaters, with a different hyperparameter choice often leading to order of magnitude improvements in drawdown.

MFT achieved low drawdown, but at the cost of worse generalization and efficacy.

RQ3: Generalization. Provable Repair results in significant generalization, improving classification accuracy of fog-corrupted images not part of the repair set, regardless of the layer repaired. For instance, repairing Layer 3 using 100 lines resulted in generalization of 46%; that is, the accuracy improved from 19.5% for the buggy network to 65.5% for the fixed network. In some scenarios FT has slightly better generalization, however this comes at the cost of higher drawdown. Furthermore, Provable Repair tends to have better generalization when using fewer lines (i.e., smaller repair set), which highlights how FT has a tendency to overfit small training sets. We also note that FT again shows extreme variability (2–10×) in generalization performance between different hyperparameter choices, *even when* it successfully terminates with a repaired network. MFT had consistently worse generalization than PR, sometimes by multiple orders of magnitude.

RQ4: Efficiency. In addition to the time results in Table 2, we did a deeper analysis of the time taken by various parts of the repair process for the 100-line experiment. For Layer 2, repairing completed in 655.7 seconds, with 1.0 seconds taken to compute *LinRegions*, 8.0 seconds for computing Jacobians, 623.6 seconds in the LP solver, and 23.1 seconds in other tasks. For Layer 3, repairing completed in 18.4 seconds, with 1.0 seconds computing *LinRegions*, 1.0 seconds computing Jacobians, 12.9 seconds in the LP solver, and 3.5 seconds in other tasks. We find that repairing the lines was quite efficient, with the majority of the time taken by the Gurobi LP solver. In contrast, for Task 1 the majority of time was spent computing Jacobians. This is because we implemented an optimized Jacobian computation for feed-forward networks.

We see that the time taken to repair depends on the particular layer that is being repaired. There is little overhead from our constraint encoding process or computing LinRegions. Because the majority of the time is spent in the Gurobi LP solver, our algorithm will benefit greatly from the active and ongoing research and engineering efforts in producing significantly faster LP solvers.

FT was also fast. However, again we note that for some choices of hyperparameters FT gets stuck and cannot find a repaired network with better-than-chance accuracy. This highlights how sensitive such gradient-descent-based approaches are to their hyperparameters, in contrast to our LP-based formulation that is guaranteed to find the minimal repair, or prove that none exists, in polynomial time. Finally, MFT was consistently fast but at the cost of consistently worse efficacy and generalization.

7.3 Task 3: 2D Polytope ACAS Xu Repair

Buggy network: Task 3 uses the $N_{2,9}$ ACAS Xu network [32], which has 7 layers and 13,350 parameters. The network takes a five-dimensional representation of the scenario around the aircraft, and outputs one of five possible advisories.

Repair set. Katz et al. [33] show that $N_{2,9}$ violates the safety property ϕ_8. However, we cannot directly use ϕ_8 as a polytope repair specification because (i) ϕ_8 concerns a five-dimensional polytope, and existing techniques for computing *LinRegions* only scale to two dimensions on ACAS-sized neural networks, and (ii) ϕ_8 specifies that the output advisory can be one of two possibilities, a disjunction that cannot be encoded as an LP. To circumvent reason (i), Task 3 uses 10 randomly-selected two-dimensional planes (slices) that contain violations to property ϕ_8. To circumvent reason (ii), Task 3 strengths ϕ_8 based on the existing behavior of the network. For each key point in the two-dimensional slice, we compute which of the two possibilities was higher in

Table 2. Summary of experimental results for Task 2. D: Drawdown (%), G: Generalization (%), T: Time, PR: provable repair, FT: fine-tuning baseline. ∗ means fine-tuning diverged and timed out after 1000 epochs, the results shown are from the last iteration of fine-tuning before the timeout.

		PR (Layer 2)			PR (Layer 3)			FT[1]			FT[2]		
Lines	Points	D	G	T	D	G	T	D	G	T	D	G	T
10	1730	1.3	30.7	1m55.1s	5.7	32.1	1.7s	56.0	4.2	0.4s	8.3	27.5	0.6s
25	4314	1.8	35.5	2m46.5s	5.5	38.3	3.7s	36.5	22.4	1.2s	3.8	51.0	0.4s
50	8354	2.6	38.3	4m29.3s	5.9	44.5	8.0s	85.2*	-8.2*	29m36.5s*	4.7	55.8	0.8s
100	16024	2.4	42.9	10m55.7s	5.9	46.0	18.4s	31.4	37.7	3.1s	3.2	60.0	1.6s

Table 3. Summary of modified fine-tuning results for Task 2. E: Efficacy (%), D: Drawdown (%), G: Generalization (%), T: Time, MFT: modified fine-tuning baseline. Note that the modified fine-tuning *does not* satisfy all of the hard constraints; therefore, it is not in fact repairing the network. However, it does result in lower drawdown.

	MFT[1] (Layer 2)				MFT[1] (Layer 3)				MFT[2] (Layer 2)				MFT[2] (Layer 3)			
Lines	E	D	G	T	E	D	G	T	E	D	G	T	E	D	G	T
10	66.5	1.9	14.3	0.7s	60.7	0.1	3.6	0.5s	70.3	0.5	16.8	0.4s	58.4	-0.05	1.3	0.5s
25	67.3	0.6	16.4	1.0s	57.4	0.3	2.4	38.3s	65.8	0.6	16.9	1.0s	56.1	0.03	1.0	1.0s
50	71.3	0.6	17.9	1.7s	61.5	0.1	1.7	1.6s	70.5	0.7	17.5	1.1s	59.7	0.1	0.8	1.6s
100	69.7	0.6	11.9	2.2s	63.7	0.1	2.3	2.2s	69.8	0.4	12.9	3.3s	62.7	0.05	0.5	5.2s

the buggy network $N_{2,9}$, and use the higher one as the desired output advisory. Notably, any network that satisfies this strengthened property also satisfies property ϕ_8.

Repair layer. We used the last layer as the repair layer. The other layers were unsatisfiable, i.e., Algorithm 2 returned $\perp$.

Generalization set. 5,466 counterexamples to the safety property ϕ_8 that were not in the repair set. These counterexamples were found by computing *LinRegions* on 12 two-dimensional slices randomly selected from R.

Drawdown set. A similarly randomly-sampled set of 5,466 points that were correctly classified by the buggy network. Generalization and drawdown sets have the same size.

Fine-tuning hyperparameters. Both FT and MFT use standard SGD with learning rate of 0.001, momentum 0.9, and batch size 16 chosen as best from a small manual search.

RQ1: Efficacy. Provable Polytope Repair was able to provably repair all 10 two-dimensional slices in the repair set, i.e., synthesize a repaired network that satisfies safety property ϕ_8 on all infinitely-many points on the 10 2D repair slices.

By contrast, both FT and MFT had *negative* efficacy; viz., while the original network misclassified only 3 points in the sampled repair set, the FT-repaired network misclassified 181 points and the MFT-repaired networks misclassified between 10 and 50 points.

RQ2: Drawdown. The drawdown for Provable Repair was zero: the fixed network correctly classified all 5,466 points in the drawdown set. By contrast, FT led to 650 of the 5,466 points that were originally classified correctly to now be classified incorrectly. This highlights again how fine-tuning can often cause forgetting. For all layers, MFT had a drawdown of less than 1%.

RQ3: Generalization. 5,176 out of 5,466 points in the generalization set were correctly classified in the Provably Repaired network; only 290 were incorrectly classified. Recall that all 5,466 were incorrectly classified in the buggy network. Thus, the generalization is 94.69%.

FT left only 216 points incorrectly classified, resulting in a slightly better generalization of 95.8%. However, this comes at the cost of introducing new bugs into the network behavior (see Drawdown above) and failing to achieve 100% efficacy even on the finitely-many repair points it was given. MFT had a generalization of 100% for the last two layers, and a generalization of less than 10% for the remaining layers.

RQ4: Efficiency. It took a total of 21.2 secs. to Provably Repair the network using the 10 two-dimensional slices; computing *LinRegions* took 1.5 secs.; 1.8 secs. to compute Jacobians; 7.0 secs. for the Gurobi LP solver; and 10.9 secs. for other tasks.

FT never fully completed; we timed it out after 1000 epochs taking 1h18m9.9s. This highlights the importance of our theoretical guarantees that Provable Repair will either find the minimal fix or prove that no fix exists in polynomial time. MFT completed within 3 seconds for all layers.

8 Related Work

Closest to this paper is Goldberger et al. [19], which can be viewed as finding minimal layer-wise fixes for a DNN given a pointwise specification. However, their algorithm is exponential time and their underlying formulation is NP-Complete. By contrast, DDNNs allow us to reduce this repair problem to an LP. Furthermore, [19] only addresses pointwise repair, and not provable *polytope* repair. These issues are demonstrated in the experimental results; whereas [19] is able to repair only 3 points even after running for a few days, we repair entire polytopes (infinitely many points, reduced to over 150,000 key points) in under thirty seconds for the same ACAS Xu DNN. Furthermore, reliance on Marabou [34] means [19] is restricted to PWL activation functions. Our provable pointwise repair algorithm is applicable to DNNs using non-PWL activations such as Tanh and Sigmoid.

Kauschke et al. [35] focuses on image-recognition models under distributional shift, but polytope repair is not considered. The technique learns a predictor that estimates whether the original network will misclassify an instance, and a repaired network that fixes the misclassification.

Sinitsin et al. [50] proposed *editable neural networks*, which train the DNN to be easier to manipulate post-training. Unlike our approach, their technique does not provide provable guarantees of efficacy or minimality, and does not support polytope repair. When the original training dataset is unavailable, their approach reduces to the fine-tuning technique we used as a baseline. Similarly, Tramèr et al. [57] injects adversarial examples into training data to increase robustness.

Robust training techniques [17] apply gradient descent to an abstract interpretation that computes an over-approximation of the model's output set on some polytope input region. Such approaches have the same sensitivity to hyperparameters as retraining and fine-tuning techniques. Furthermore, they use coarse approximations designed for pointwise robustness. These coarse approximations blow up on the larger input regions considered in our experiments, making the approach ineffective for repairing such properties.

GaLU networks [16] can be thought of as a variant of decoupled networks where activations and values get recoupled after every layer. Thus, multi-layer GaLU networks do not satisfy the key theoretical properties of DDNNs.

Alshiekh et al. [3], Zhu et al. [59] ensure safety of reinforcement learning controllers by synthesizing a *shield* guaranteeing it satisfies a temporal logic property. Bastani et al. [7] present policy extraction for synthesizing provably robust decision tree policy for deep reinforcement learning.

Prior work on DNN verification focused on safety and robustness [4, 6, 11, 15, 18, 30, 33, 34, 49]. More recent research tackles testing of DNNs [23, 42, 43, 45, 47, 54, 56, 58]. Our algorithms can fix errors found by such tools.

9 Conclusion and Future Work

We introduced *provable repair* of DNNs, and presented algorithms for *pointwise* and *polytope* repair; the former handles specifications on finitely-many inputs, and the latter handles a symbolic specification about infinite sets of points. We introduced *Decoupled* DNNs, which allowed us to reduce provable pointwise repair to an LP problem. For the common class of piecewise-linear DNNs, our polytope repair algorithm can *provably* reduce the polytope repair problem to a pointwise repair problem. Our extensive experimental evaluation on three different tasks demonstrate that pointwise and polytope repair are effective, generalize well, display minimal drawdown, and scale well.

The introduction of provable repairs opens many exciting directions for future work. We can employ *sound approximations* of linearizations to improve performance, and support non-piecewise-linear activation functions for polytope repair. Repairing multiple layers could be achieved by using the natural generalization of our LP formulation to a QCQP [5], or by iteratively applying our LP formulation to different layers. Future work may repair the activation parameters, or convert the resulting DDNN back into a standard, feed-forward DNN while still satisfying the specification (e.g., to reduce the small computational overhead of the DDNN). Exploring learning-theoretic properties of the repair process, the trade-off between generalization and drawdown during repair, and heuristics for choosing repair layers, are all very interesting and important lines of future research to make provable repair even more useful. Future work could explore repairing Recurrent Neural Networks (RNNs) using linear temporal logic specifications. Future work may explore how to speed up repair using hardware accelerators beyond the native support provided by PyTorch. Finally, experimenting with different objectives, relaxations, or solving methods may lead to even more efficient mechanisms for DNN repair.

Acknowledgments

We thank our shepherd Osbert Bastani and the other reviewers for their feedback and suggestions. This work is supported in part by NSF grant CCF-2048123 and a Facebook Probability and Programming research award.

References

[1] 2019. A collection of pre-trained, state-of-the-art models in the ONNX format. https://github.com/onnx/models. Accessed: 2019-05-01.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe Reinforcement Learning via Shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17211

[4] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314614

[5] David P Baron. 1972. Quadratic programming with quadratic constraints. *Naval Research Logistics Quarterly* 19, 2 (1972).

[6] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Advances in Neural Information Processing Systems 29 (NeurIPS)*. https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html

[7] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*. https://proceedings.neurips.cc/paper/2018/hash/e6d8545daa42d5ced125a4bf747b3688-Abstract.html

[8] Kathleen Zhou Benjamin Granger, Marta Yu. 2014. Optimization with absolute values. https://optimization.mccormick.northwestern.edu/index.php/Optimization_with_absolute_values.

[9] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-662-49674-9_55

[10] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. (2016). arXiv:1604.07316 http://arxiv.org/abs/1604.07316

[11] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. 2018. A Unified View of Piecewise Linear Neural Network Verification. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*. https://proceedings.neurips.cc/paper/2018/hash/be53d253d6bc3258a8160556dda3e9b2-Abstract.html

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-540-78800-3_24

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/CVPR.2009.5206848

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. https://doi.org/10.18653/v1/n19-1423

[15] Rüdiger Ehlers. 2017. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. https://doi.org/10.1007/978-3-319-68167-2_19

[16] Jonathan Fiat, Eran Malach, and Shai Shalev-Shwartz. 2019. Decoupling Gating from Linearity. (2019). arXiv:1906.05032 http://arxiv.org/abs/1906.05032

[17] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin T. Vechev. 2019. DL2: Training and Querying Neural Networks with Logic. In *36th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 97)*. http://proceedings.mlr.press/v97/fischer19a.html

[18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2018.00058

[19] Ben Goldberger, Guy Katz, Yossi Adi, and Joseph Keshet. 2020. Minimal Modifications of Deep Neural Networks using Verification. In *23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Vol. 73. https://easychair.org/publications/paper/CWhF

[20] Richard Gonzales. 2019. Feds Say Self-Driving Uber SUV Did Not Recognize Jaywalking Pedestrian In Fatal Crash. NPR https://www.npr.org/2019/11/07/777438412/feds-say-self-driving-uber-suv-did-not-recognize-jaywalking-pedestrian-in-fatal-. Accessed: 2020-06-06.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[22] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations (ICLR)*. http://arxiv.org/abs/1412.6572

[23] Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, Ismet Burak Kadron, Corina S. Pasareanu, and Sarfraz Khurshid. 2019. Symbolic Execution for Importance Analysis and Adversarial Generation in Neural Networks. In *30th IEEE International Symposium on Software Reliability Engineering, (ISSRE)*. https://doi.org/10.1109/ISSRE.2019.00039

[24] LLC Gurobi Optimization. 2020. Gurobi Optimizer Reference Manual. http://www.gurobi.com.

[25] Boris Hanin and David Rolnick. 2019. Complexity of Linear Regions in Deep Networks. In *36th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 97)*. http://proceedings.mlr.press/v97/hanin19a.html

[26] Boris Hanin and David Rolnick. 2019. Deep ReLU Networks Have Surprisingly Few Activation Patterns. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*. https://proceedings.neurips.cc/paper/2019/hash/9766527f2b5d3e95d4a733fcfb77bd7e-Abstract.html

[27] Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. 2019. Natural Adversarial Examples. (2019). arXiv:1907.07174 http://arxiv.org/abs/1907.07174

[28] Alex Hern. 2017. Facebook translates 'good morning' into 'attack them', leading to arrest. https://www.theguardian.com/technology/2017/oct/24/facebook-palestine-israel-translates-good-morning-attack-them-arrest. Accessed: 2020-06-06.

[29] Kashmir Hill. 2020. Wrongfully Accused by an Algorithm. New York Times. https://www.nytimes.com/2020/06/24/technology/facial-recognition-arrest.html. Accessed: 2020-06-06.

[30] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *29th International Conference on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-63387-9_1

[31] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

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

[33] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *29th International Conference on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-63387-9_5

[34] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *31st International Conference on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-25540-4_26

[35] Sebastian Kauschke, David Hermann Lehmann, and Johannes Fürnkranz. 2019. Patching Deep Neural Networks for Nonstationary Environments. In *International Joint Conference on Neural Networks (IJCNN)*. https://doi.org/10.1109/IJCNN.2019.8852222

[36] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. 2018. Measuring Catastrophic Forgetting in Neural Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16410

[37] Daniel S. Kermany, Michael Goldbaum, Wenjia Cai, Carolina C.S. Valentim, Huiying Liang, Sally L. Baxter, Alex McKeown, Ge Yang, Xiaokang Wu, Fangbing Yan, Justin Dong, Made K. Prasadha, Jacqueline Pei, Magdalene Y.L. Ting, Jie Zhu, Christina Li, Sierra Hewett, Jason Dong, Ian Ziyar, Alexander Shi, Runze Zhang, Lianghong Zheng, Rui Hou, William Shi, Xin Fu, Yaou Duan, Viet A.N. Huu, Cindy Wen, Edward D. Zhang, Charlotte L. Zhang, Oulan Li, Xiaobo Wang, Michael A. Singer, Xiaodong Sun, Jie Xu, Ali Tafreshi, M. Anthony Lewis, Huimin Xia, and Kang Zhang. 2018. Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning. *Cell* 172, 5 (2018). https://doi.org/10.1016/j.cell.2018.02.010

[38] Leonid Genrikhovich Khachiyan. 1979. A polynomial algorithm in linear programming. In *Doklady Akademii Nauk*, Vol. 244. Russian Academy of Sciences.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017). https://doi.org/10.1145/3065386

[40] Yann LeCun, Corinna Cortes, and CJ Burges. 2010. MNIST handwritten digit database. (2010). http://yann.lecun.com/exdb/mnist

[41] Dave Lee. 2016. US opens investigation into Tesla after fatal crash. BBC. https://www.bbc.co.uk/news/technology-36680043. Accessed: 2020-06-06.

[42] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/3238147.3238202

[43] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *29th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. https://doi.org/10.1109/ISSRE.2018.00021

[44] Norman Mu and Justin Gilmer. 2019. MNIST-C: A Robustness Benchmark for Computer Vision. (2019). arXiv:1906.02337 http://arxiv.org/abs/1906.02337

[45] Augustus Odena, Catherine Olsson, David G. Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *36th International Conference on Machine Learning (ICML)*. http://proceedings.mlr.press/v97/odena19a.html

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *26th Symposium on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/3132747.3132785

[48] Gagandeep Singh. 2019. ETH Robustness Analyzer for Neural Networks (ERAN). https://github.com/eth-sri/eran. Accessed: 2019-05-01.

[49] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290354

[50] Anton Sinitsin, Vsevolod Plokhotnyuk, Dmitriy Pyrkin, Sergei Popov, and Artem Babenko. 2020. Editable Neural Networks. In *8th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=HJedXaEtvS

[51] Matthew Sotoudeh and Aditya V. Thakur. 2019. Computing Linear Restrictions of Neural Networks. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*. https://proceedings.neurips.cc/paper/2019/hash/908075ea2c025c335f4865f7db427062-Abstract.html

[52] Matthew Sotoudeh and Aditya V. Thakur. 2021. Provable Repair of Deep Neural Networks. arXiv:2104.04413 [cs.LG]

[53] Matthew Sotoudeh and Aditya V. Thakur. 2021. SyReNN: A Tool for Analyzing Deep Neural Networks. In *27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://doi.org/10.1007/978-3-030-72013-1_15

[54] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/3238147.3238172

[55] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations (ICLR)*. http://arxiv.org/abs/1312.6199

[56] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *40th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3180155.3180220

[57] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian J. Goodfellow, Dan Boneh, and Patrick D. McDaniel. 2018. Ensemble Adversarial Training: Attacks and Defenses. In *6th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=rkZvSe-RZ

[58] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3293882.3330579

[59] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314638