Correcting Deep Neural Networks with Small, Generalizing Patches

Matthew Sotoudeh
Department of Computer Science
University of California, Davis
Davis, CA 95616
masotoudeh@ucdavis.edu

Aditya V. Thakur
Department of Computer Science
University of California, Davis
Davis, CA 95616
avthakur@ucdavis.edu

Abstract

We consider the problem of *patching* a deep neural network: applying a small change to the network weights in order to produce a desired change in the classifications made by the network. We motivate this problem using ACAS Xu, a well-studied neural network intended to act as an aircraft collision-avoidance system. Our technique works over infinite patching regions, is based on an SMT formulation of the problem, and has a number of desirable convergence properties. To make this approach efficient, we introduce a symbolic representation of neural networks and a generalization of ReLU neural networks (Frozen Networks). We show that our approach can produce highly effective and generalizing patches with a very small number of weight changes.

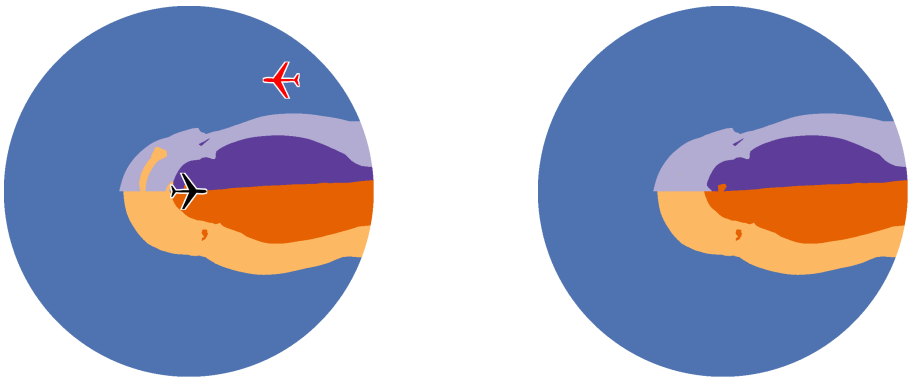
1 Introduction

Recently, deep neural networks (DNNs) [1] have made great strides in solving a variety of problems. Due to these successes, DNNs have begun to permeate more systems, including safety-critical ones such as aircraft collision avoidance [2] and self-driving cars [3]. This has led to a growing need for tools to more deeply analyze and modify such networks beyond the initial training process.

In this work, we consider the problem of *patching* a neural network after it has been trained: changing a small number of weights in the network to precisely manipulate the network decision boundaries (eg. to fix erroneous or undesired behavior). For example, consider the ACAS Xu neural network from Julian et al. [2]. This network takes as input a five-dimensional description of the situation around the plane being controlled (namely: distance and angle to the nearest attacker, its velocity, and the attacker's velocity), has six hidden layers with 50 ReLU units each, and produces five outputs corresponding to possible advisories (strong left, weak left, weak right, strong right, clear-of-conflict). The network's behavior is visualized (using the technique from [4]) in Figure 1a, where we have held all input dimensions constant except for the distance and angle to the nearest attacker. The ownship is positioned at the center of the circle, while each colored point represents the advisory that would be produced by the network if the attacker were to be positioned there.

In addition to better understanding the network, this visualization allows us to identify a number of unexpected behaviors. For example, there is a "stripe" of locations behind the plane where a nearby attacker would cause the network to suggest a "weak left," actually turning *towards* the attacker. The question addressed in this paper is: how might one effectively and efficiently target and remove such behaviors? A visualization after applying our proposed solution is displayed in Figure 1b, where the offending "stripe" has been removed from this portion of the input domain by changing 5 of out of the 13,000 weights of the network. Furthermore, as we will see, such patches applied to one "slice" of the input domain tend to generalize well — eg., if the velocities were increased, the stripe would

Preprint. Under review.



(a) Network visualization

(b) Patched network visualization (plane icons removed to better show decision boundaries)

Legend: Clear-of-Conflict, Weak Right, Strong Right, Strong Left, Weak Left.

Figure 1: Visualizing preconditions for and patching an aircraft collision avoidance network

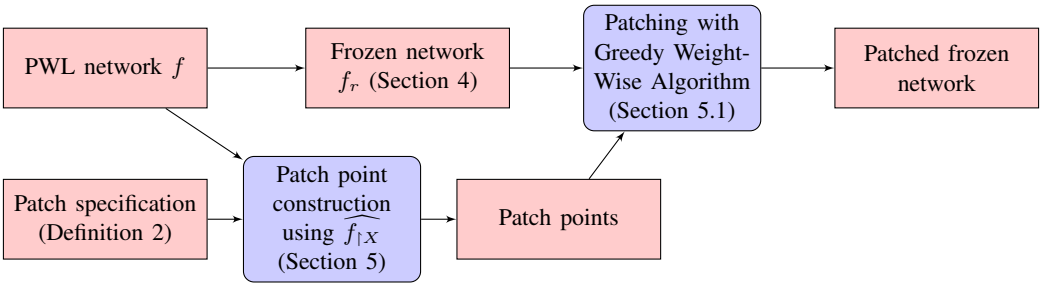


Figure 2: Patching deep neural networks

still be removed. This allows a human to "patch" a single two-dimensional slice of the input domain and then have it generalize automatically to the 5-dimensional input space.

For the rest of this paper, we will restrict our consideration to *piecewise-linear* (PWL) networks, which are formally described below and includes networks using 2DCONVOLUTION, FULLYCONNECTED, RELU, MAXPOOL, and BATCHNORM layers.

Definition 1. *A function $f : A \to B$ is referred to as a* piecewise-linear function *if its input domain A can be partitioned by a finite set of (possibly unbounded) convex polytopes $\{P_1, P_2, \ldots, P_n\}$ such that, within any partition $P_i \subseteq A$, there exists an* affine *map $F_i : P_i \to B$ satisfying $f(x) = F_i(x)$ for any $x \in P_i$.*

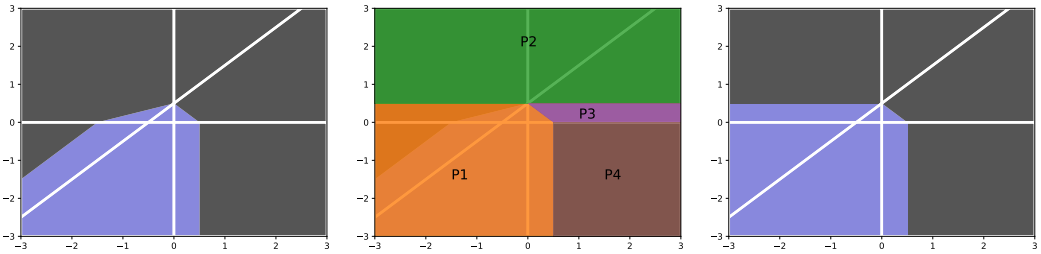
2 Patching Deep Neural Networks

We now formalize the problem of *network patching* and present an approach for patching over polytopes (Figure 2). As a running example, consider the expository network defined by f below:

$$f(x) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & -1 \end{bmatrix} \text{RELU}\left(\begin{bmatrix} -1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -0.5 \\ 0 \\ 0 \end{bmatrix} \right) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1}$$

This network is visualized in Figure 3a, with classification regions $R_1 = \{y_1 \geq y_2\}$ and $R_2 = \{y_2 \geq y_1\}$. Suppose one wants to modify the network such that its decision boundaries look like that of Figure 3c. We formalize the desired behavior as a *patch specification*:

Definition 2. *Given a neural network $f : A \to B$, a* patch specification *is a finite set of* pairs of convex polytopes $T = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$ *where each $X_i \subseteq A$ and $Y_i \subseteq B$.*



(a) The network's linear partitions and decision boundaries before patching

(b) The partitions used in the *patch specification*

(c) The network's linear partitions and decision boundaries after patching

Figure 3: Network patching on DNN f in Equation 1.

In our running example, the four polytopes specified in Figure 3b can be used to build the patch specification: $\{(P_1, R_2), (P_2, R_1), (P_3, R_1), (P_4, R_1)\}$. We can now formalize the concept of a network patch, i.e. a change to the weights of a network such that it satisfies the patch specification:

Definition 3. *Given a neural network $f(x;\theta)$ parameterized by weights θ and a patch specification $T = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$, we define a* patch to f satisfying T *to be a change δ to the weights θ such that, for all $x \in X_i$, $f(x;\theta+\delta) \in Y_i$ holds.*

An example of the network patched to satisfy the preceeding specification is presented in Figure 3c. Note that the polytopes in patch specifications are usually *infinite* (though bounded), meaning the quantifications in Definition 3 are over infinite sets.

2.1 Generalizing Patches

A patch specification references only some small subset of the input space; e.g., in the ACAS Xu network, we patched on a two-dimensional subset of the five-dimensional input space. Thus, when computing a patch, we would like to avoid over-fitting and would want the patch to correct similar behavior on other input slices. However, this generalization should not come at the cost of introducing unexpected behavior in these other slices.

The way we address this issue is by requiring that patches be *small*, both in the number of weights that are changed and the magnitude of the weight changes. In particular, Section 5 describes an approach that changes only a single weight at each iteration and finds the smallest change to this weight required to satisfy the patch specification. Although this may seem extremely restrictive, our experimental evaluation in Section 6 shows that patches changing *as few as* 5 weights are often enough to correct erroneous behavior in a generalizing fashion.

2.2 Deep Neural Network Patching as Non-Linear SMT

Finding a network patch satisfying a patching specification T (Definition 3) can be seen as a non-linear instance of the satisfiability modulo theory (SMT) problem [5], which determines whether there exists a model of a set of formulas. We can translate the patch specification T into the SMT instance: $\bigwedge_{(X_i,Y_i)\in T} \forall x \in X_i : f(x;\theta+\delta) \in Y_i$. Unfortunately, the extreme non-linearity involved makes solving the constraints infeasible even for state-of-the-art solvers such as Z3 [6]. To make it feasible, we would like to be able to encode it as a conjunction of linear inequalities (an LP). There are three sources of non-linearity we must address:

1. The network function f is usually highly-non-linear (eg. ReLU layers introduce exponentially many possible branches).
2. Solving for δ while quantifying over the infinite $x \in X_i$ results in non-linearity in the problem itself.
3. If δ can change weights across multiple layers, the interaction between all weight changes results in a high-order polynomial problem instead of a linear one (because the linear layers are applied sequentially, eg. $y = w_2(w_1 x + b_1) + b_2$).

To address these problems, Section 3 defines a *symbolic neural network representation* [7, 4], which allows us to reason directly about the linear and non-linear behavior of piecewise-linear functions. Section 4 introduces a new neural network architecture termed *Frozen Networks*, which generalize feed-forward RELU networks while enabling finer-grained control over the linear and non-linear network behaviors. Our first result, Theorem 1, shows that we can write the output of a frozen network on a particular point with a single weight changed as a linear function of that weight change. Our next result, Theorem 2, reduces the problem of patching a frozen network on the infinitely-many $x \in X_i$ to the problem of patching a linear function on finitely-many *key points*. Finally, we combine these results in Theorem 3 to arrive at an efficient, greedy algorithm which, at each step, makes the best single-weight change possible.

3 A Symbolic Representation for Deep Neural Networks

Recall that a primary challenge in solving the satisfiability problem was that the network itself is highly non-linear. In this section, we introduce a primitive which partitions a given polytope such that the behavior of a given piecewise-linear (PWL) function f is affine on each partition. It generalizes the primitive in Sotoudeh and Thakur [7] and an extended version is described in more detail by Sotoudeh and Thakur [4].

Definition 4. *Given a function $f : \mathbb{R}^a \to \mathbb{R}^b$ and bounded convex polytope $X \subseteq \mathbb{R}^a$, we define the* symbolic representation of f restricted to X*, written $\widehat{f_{\restriction X}}$, to be a set of tuples $\widehat{f_{\restriction X}} = \{(P_1, F_1), \ldots, (P_n, F_n)\}$, such that:*

1. *The set $\{P_1, P_2, \ldots, P_n\}$ partitions X, except possibly for overlapping boundaries.*
2. *Given any such (P_i, F_i) pair, the following holds: $\forall x \in P_i : f(x) = F_i(x)$.*
3. *Each P_i is a convex polytope.*
4. *Each $F_i : P_i \to \mathbb{R}^b$ is affine.*

Efficient algorithms for computing this representation for two-dimensional X and piecewise-linear f were presented in Sotoudeh and Thakur [4] and implementations are available at https://github.com/95616ARG/SyReNN. The key idea is to "slice up" X repeatedly by splitting it according to the faces defining the affine regions of f (eg. the $x_i = 0$ hyperplanes for RELU). Unfortunately, computing $\widehat{f_{\restriction X}}$ quickly becomes infeasible when X is more than two-dimensional, thus we will only consider patching specifications that involve two-dimensional regions. However, being able to patch on two-dimensional regions is important, as two-dimensional regions are close to the limit of what humans can easily comprehend and define a patching specification over. An example of this was Figure 1a, where we only considered a single two-dimensional "slice" of the 5-dimensional input domain (as humans cannot effectively visualize 5-dimensional spaces). Furthermore, using our technique, one *can* patch multiple (finitely many) two-dimensional regions *simultaneously*. Finally, we will show in Section 6 that patches made to one two-dimensional region tend to *generalize* well to (i.e. fix similar behaviors in) nearby regions of the input space.

4 Frozen Networks

We now propose a new neural network architecture termed a *Frozen Network*, with the key idea of *separating the partitioning imposed by a PWL network on its input space from the affine map applied within each partition*. Frozen networks strictly generalize feed-forward networks with RELU activations and are loosely inspired by Fiat et al. [8]. They can be extended to networks with arbitrary PWL activation functions but we focus on RELU for ease of exposition.

Recall that, in a standard feed-forward neural network, the network function f is decomposed into a series of sequentially-applied *layers* (vector-valued functions) denoted $L_1, L_2, \ldots, L_n$. RELU layers are defined component-wise by:

$$\text{RELU}(x)_i = \begin{cases} 0 & x_i \leq 0 \\ x_i & x_i > 0 \end{cases}$$

In a *Frozen Network*, four changes are made:

1. The inputs and outputs to each layer are *vectors of 2-tuples* $((x_1^a, x_2^v), \ldots, (x_d^a, x_d^v))$. In some scenarios it will be convenient to refer to the vector consisting of just the first-values, noted $x^a = (x_1^a, \ldots, x_d^a)$ and named the *activation vector*. Correspondingly, we notate $x^v = (x_1^v, \ldots, x_d^v)$ and name x^v the *value vector*.
2. The input vector $x = (x_1, \ldots, x_k)$ to the entire network is converted to an input vector where the activation and value vectors are equal, i.e. $((x_1, x_1), (x_2, x_2), \ldots, (x_k, x_k))$ or, equivalently, $x^a = x^v = x$.
3. The output of the network is taken to be *only* the values vector x^v output by the last layer.
4. ReLU layers are replaced by *Frozen* ReLU (FReLU) layers, defined below.

In a frozen network, FullyConnected or 2DConvolution layers are associated with *two* parameters each, θ^a and θ^v. Given an input with activation vector x^a and value vector x^v to a FullyConnected or 2DConvolution layer, θ^a and θ^v are applied to the x^a and x^v vectors independently as follows:

$$\text{FullyConnected}^a(x^a, x^v; \theta^a, \theta^v) \stackrel{\text{def}}{=} \text{FullyConnected}(x^a; \theta^a)$$
$$\text{FullyConnected}^v(x^a, x^v; \theta^a, \theta^v) \stackrel{\text{def}}{=} \text{FullyConnected}(x^v; \theta^v)$$

Given an input $((x_1^a, x_1^v), \ldots, (x_d^a, x_d^v))$ to a FReLU layer, we define its ith output tuple to be:

$$\text{FReLU}\left(((x_1^a, x_1^v), \ldots, (x_d^a, x_d^v))\right)_i = \begin{cases} (0, 0) & x_i^a \leq 0 \\ (x_i^a, x_i^v) & x_i^a > 0 \end{cases}$$

The following properties follow directly from these definitions and are integral to understanding Frozen ReLU networks:

1. A frozen network is *piecewise-linear*.
2. If $x^a = x^v$, then $\text{FReLU}(x^a, x^v) = (\text{ReLU}(x^v), \text{ReLU}(x^v))$.
3. If $x^a = x^v$ and $\theta^a = \theta^v$, then $\text{FullyConnected}_{\theta^a, \theta^v}(x^a, x^v) = (\text{FullyConnected}_{\theta^v}(x^v), \text{FullyConnected}_{\theta^v}(x^v))$ and similarly for 2DConvolution.
4. The value of the *activation vectors* (x^a) only ever depends on the value of prior activation vectors and activation parameters (θ^a), never prior value vectors (x^v) or value parameters (θ^v).
5. The *non-linear* behavior of the *value vectors* (x^v) is *entirely* controlled by the activation vectors (x^a). For example, if all activation vectors x^a are strictly positive, then the entire frozen network is affine (regardless of the value vectors).

The first property allows us to use the symbolic representation $\widehat{f_{\restriction X}}$ of the frozen network. The next two properties show that we can always encode any feed-forward ReLU network as a Frozen ReLU network by setting $\theta^a = \theta^v = \theta$ for corresponding affine layers (i.e., frozen networks are strictly more expressive than feed-forward ReLU networks). The final two properties ensure that, for a frozen network f and polytope X:

1. The *partitioning* of $\widehat{f_{\restriction X}}$ (i.e., $\{P_1, \ldots, P_n\}$) is determined *entirely* by the activation parameters θ^a.
2. Changing the value parameters θ^v only affects the *affine maps* of $\widehat{f_{\restriction X}}$ (i.e., $\{F_1, \ldots, F_n\}$).

Visually, changes to the activation parameters θ^a change the position of the white lines in Figure 3a, while changes to the value parameters θ^v change the position of the decision boundary *within* each white-line delineated region in Figure 3a *without changing the positions of the white lines at all*.

These observations lead to the following important result about *single-weight, values-patched* frozen networks, proved in Appendix D:

Theorem 1. *Given a frozen network $f(x; \theta^a, \theta^v)$ parameterized by activation parameters θ^a and value parameters θ^v, we define the* single-weight, values-patched *network $f(x; \theta^a, \theta^v + \delta_w)$ which modifies only a single (non-shared) weight w in the value parameters θ^v.*

Then, for any $(P_i, F_i) \in \widehat{f(x; \theta^a, \theta^v)}_{\restriction X}$ and x in P_i, the following holds:

$$f(x \in P_i; \theta^a, \theta^v + \delta_w) = F_i(x) + \delta_w \frac{\partial f(x; \theta^a, \theta^v)}{\partial w}$$

In other words, if we only change a single value parameter weight, then the effect on the output is simply a linear function of the change to that parameter. Note in particular that neither $F_i(x)$ nor $\frac{\partial f(x;\theta^a,\theta^v)}{\partial w}$ have any dependence on δ_w; they can both be treated as constants for the purposes of determining δ_w.

5 Patching Deep Neural Networks on Infinite Inputs

We are now prepared to present the following theorem, proved in Appendix E:

Theorem 2. *Given a frozen network $f(x; \theta^a, \theta^v)$ parameterized by activation parameters θ^a and value parameters θ^v, an input polytope X, and output polytope Y:*

A values-patched network *$f(x; \theta^a, \theta^v + \delta)$, modifying only θ^v, satisfies the property $\forall x \in X : f(x; \theta^a, \theta^v + \delta) \in Y$ if and only if it satisfies the property $\forall (P_i, F_i) \in \widehat{f(x; \theta^a, \theta^v)}_{\restriction X} : \forall v \in \mathrm{Vert}(P_i) : f(v; \theta^a, \theta^v + \delta) \in Y$.*

Where $\mathrm{Vert}(P)$ gives the vertices of the bounded polytope P. This theorem allows us to translate the problem of patching over an *infinite* set of points $x \in X$ to that of patching on *finitely-many* vertex points $\mathrm{Vert}(P_i)$, *as long as* we keep the activation parameters θ^a constant. We will refer to these vertex points as "key points" for this reason.

Combining Theorem 1 with Theorem 2, we are now prepared to formulate the patching problem as an efficient interval MAX-SMT instance.

First, we translate the given ReLU network into an equivalent Frozen Network (which can always be done as Frozen Networks are strictly more expressive than sequential feed-forward networks). We can then address problem (1) from Section 2.2 (non-linearity of f) by restricting ourselves to *only patching the value parameters θ^v*. The value parameters do not impact the partitioning of $\widehat{f}$ (only the affine maps). Thus, this restriction allows us to break the non-linear constraint into a series of constraints, one for each linear region, and within each the function is linear with respect to the input. To address problem (2) (quantifying over the infinitely many $x \in X_i$ while solving for δ) we use Theorem 2 to quantify over the finitely-many *vertices of the partitions in* $\widehat{f_{\restriction X_i}}$ instead of over the infinitely-many points in each X_i. Theorem 2 ensures that the two are equivalent. To address problem (3) (non-linear interaction when changing weights across multiple layers in a network) we can limit δ to only modify *a single weight*, allowing us to use Theorem 1 to arrive at a linear formulation of the effect of any given change.

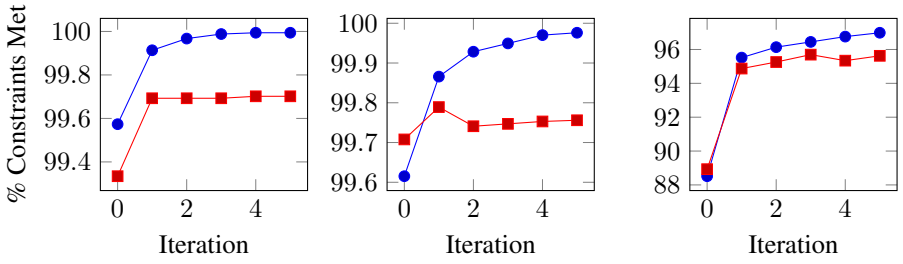
Once all three of these changes are made, the corresponding satisfiability problem can be encoded as a set of interval constraints, solving for δ_w in:

$$\bigwedge_{(X_i,Y_i)\in T} \bigwedge_{(P_j,F_j)\in \widehat{f_{\restriction X_i}}} \bigwedge_{v \in \mathtt{Vert}(P_j)} \left(F_i(v) + \delta_w \frac{\partial f(v; \theta^a, \theta^v)}{\partial w} \right) \in Y_i \tag{2}$$

Again, note that both $F_i(v)$ and $\frac{\partial f(v;\theta^a,\theta^v)}{\partial w}$ are constant with respect to δ_w, and that there are finitely many conjunctions.

5.1 Iterative Greedy Weight-Wise Patching

Equation 2 corresponds to a *conjunction of linear inequalities in one variable*. Thus, each of the conjuncts corresponds to an *interval on the corresponding patch δ_w in which that constraint is met*.



(a) "Pockets" Specification (b) "Bands" Specification (c) "Symmetry" specification.

Figure 4: Weights changed vs. percent of constraints met. The blue line shows the percent of constraints met on the input slice that the patch targeted, while the red line shows the percent of constraints met on a slice at higher velocity.

Table 1: Summarizes the percent of constraints met and time taken per iteration for patching three specifications. "Patched Slice" is the input domain of interest that the patch spec applies to, while "Other Slice" shows how patching one slice can generalize to patches for another.

	Constraints Met					
	Patched Slice		Other Slice		Seconds Per Iteration	
Patch	Initial	Final	Initial	Final	Mean	Std. Dev.
Pockets	99.6%	99.994%	99.3%	99.7%	23.6	15.2
Bands	99.6%	99.98%	99.7%	99.8%	15.9	2.3
Symmetry	88.5%	97.0%	88.9%	95.6%	4.7	0.1

We refer to the set of all such intervals for the kth weight as I_k. Finding the *optimal patch* (i.e., the patch that satisfies as many constraints as possible) for that weight now corresponds to finding an interval $M = [M^l, M^u]$ that maximizes the number of intervals in I_k containing it (i.e., solving the corresponding interval MAX-SMT problem). After sorting, this is solvable in linear time using the standard "linear sweep" algorithm. We can repeat this process for every weight in a particular layer (or the entire network if desired), picking the update that will satisfy the maximum number of constraints. Finally, we can greedily apply this algorithm to update multiple weights, at each step making the optimal change to a single weight in the network. Thus, this greedy algorithm is guaranteed to both monotonically increase the number of constraints met as well as terminate in a finite number of steps. We summarize these facts in the following theorem:

Theorem 3. *For any given weight w in the network, the set of constraints corresponding to Equation 2 can be solved in $O(n \log n)$ time (where n is the number of constraints) to find the smallest (by magnitude) δ_w which satisfies the maximum number of constraints.*

Consequently, by solving Equation 2 for every weight in a network (or layer of a network), we can find the single-weight patch *that satisfies the maximum number of constraints. Furthermore, we can find the single-weight patch that is smallest in magnitude while satisfying that number of constraints.*

6 Experimental Evaluations

In this section, we address the following three research questions: (1) Can an ACAS Xu network be patched to correct a number of undesired behaviors? (2) How well do patches on a single two-dimensional subsets of the input space generalize to (i.e. fix the same behavior on) other subsets of the input space? (3) How well does the greedy MAX-SMT solver described in 2.2 work, both in terms of efficiency and optimization?

Tests were performed on a dedicated Amazon EC2 c5.metal instance, using Benchexec [9] to limit the number of CPU cores to 16 and RAM to 16GB. Our code is available at https://github.com/95616ARG/SyReNN.

We attempted to patch the following three suspicious behaviors of the ACAS Xu network visualized in Figure 1a using our approach (Section 5.1): (i) the "Pockets" patch specification (Figure 5 in Ap-

pendix A) attempts to get rid of the "pockets" of strong left/strong right in regions that are otherwise weak left/weak right; (ii) the "Bands" specification (Figure 6) attempts to get rid of the weak-left region behind and to the left of the ownship (at the origin); (iii) the "Symmetry" specification (Figure 7) attempts to lower the decision boundary between strong left/strong right to be symmetrical. After patching, we plotted the patched network in the same two-dimensional region it was originally plotted (and then patched) over, as well as another two-dimensional region formed by increasing the velocities of the ownship and attacking ship by 50 kilometers per hour (to evaluate generalization to input regions not explicitly patched). Before and after plots are presented in Figures 5—7 in Appendix A. Finally, we timed the performance on the three evaluations shown above and present it in Table 1 along with the percent of constraints met at each step in Figure 4.

We find that our proposed approach works very well; by changing only five weights, we were able to near-perfectly apply most of the patches (eg. notice the disappearance of the "offending" regions between Figure 5a and Figure 5c). Furthermore, for the most part, the patches appear to *generalize* to regions not explicitly considered in the patching process (eg. notice the disappearance of the out-of-place "strong left" region between Figure 5d and Figure 5e after changing only a single weight). The efficacy of the greedy MAX-SMT solver is further exemplified by the quantitative results in Table 1 and Figure 4, which shows that it is very effective at quickly meeting a large fraction of the desired constraints. However, patching was not, in general, a "silver bullet," and sometimes undesired changes were introduced (cf. Figure 7f, which has introduced a small "strong right" region into what was otherwise "weak left" on the generalized region). Avoiding such issues is an interesting direction for future work, but for the most part, we believe the results shown here to be extremely promising for the application of network patching in practice.

7 Related Work

Prior work in network patching [10, 11] focused on image-recognition models under distributional shift, where the problem is phrased in terms of correctly the network's behavior on finitely many input points. The problem we address differs significantly because we know *exactly* how the user would like an *entire two-dimensional subset* of the input space classified (because they can specify it from the visualization in Figure 1) versus the finitely-many individual points considered by prior work. To that end, the core challenge addressed by this paper is to formulate and effectively address the patching problem on *entire polytope regions* which may have infinitely many points.

Furthermore, our proposed solution differs significantly from that of prior work. Whereas prior work extended the model to be patched with a decision tree (to predict when the patch should be applied) and an entirely new model that is used in place of (a portion of) the original one when the decision tree decides it is necessary. By contrast, we only change a very small number of weights in the network (5 in our experiments).

Finally, we contrast the network *patching* problem with that of *verification* of neural networks [12–16]. Using our terminology, the latter can be interpreted as *checking* whether a network satisfies a particular patch specification. While this is important, in the case where such a system says that a specification is *not* met the verification approach gives no recourse except to try and retrain the system. Network *patching* picks up where network *verification* leaves off, by giving the developer a way to predictably manipulate the network to meet the desired specification.

8 Conclusion

We discussed the problem of *network patching*, i.e. changing the parameters of a network so that the network satisfies a given specification. We introduced a symbolic representation that translates analyses on piecewise-linear networks into analyses on finitely-many affine functions. We then introduced a generalization of ReLU networks that provides fine-grained control over both the non-linear and linear behavior of a network. We showed how these primitives can be combined with an efficient interval MAX-SMT solver to efficiently and effectively patch the ACAS Xu neural network.

Acknowledgments

We thank Nina Amenta, Yong Jae Lee, Mukund Sundararajan, and Cindy Rubio-Gonzalez for their feedback and suggestions on this work, along with Amazon Web Services Cloud credits for research.

References

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

[2] Kyle D Julian, Mykel J Kochenderfer, and Michael P Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3): 598–608, 2018.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[4] Matthew Sotoudeh and Aditya V. Thakur. A symbolic neural network representation and its application to understanding, verifying, and patching networks. *CoRR*, abs/1908.06223, 2019. URL `https://arxiv.org/abs/1908.06223`.

[5] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-825. URL `https://doi.org/10.3233/978-1-58603-929-5-825`.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] Matthew Sotoudeh and Aditya V. Thakur. Computing linear restrictions of neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. URL `https://arxiv.org/abs/1908.06214`.

[8] Jonathan Fiat, Eran Malach, and Shai Shalev-Shwartz. Decoupling gating from linearity. *arXiv preprint arXiv:1906.05032*, 2019.

[9] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 887–904. Springer, 2016.

[10] Sebastian Kauschke and David Hermann Lehmann. Towards neural network patching: Evaluating engagement-layers and patch-architectures. *arXiv preprint arXiv:1812.03468*, 2018.

[11] Sebastian Kauschke and Johannes Fürnkranz. Batchwise patching of classifiers. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

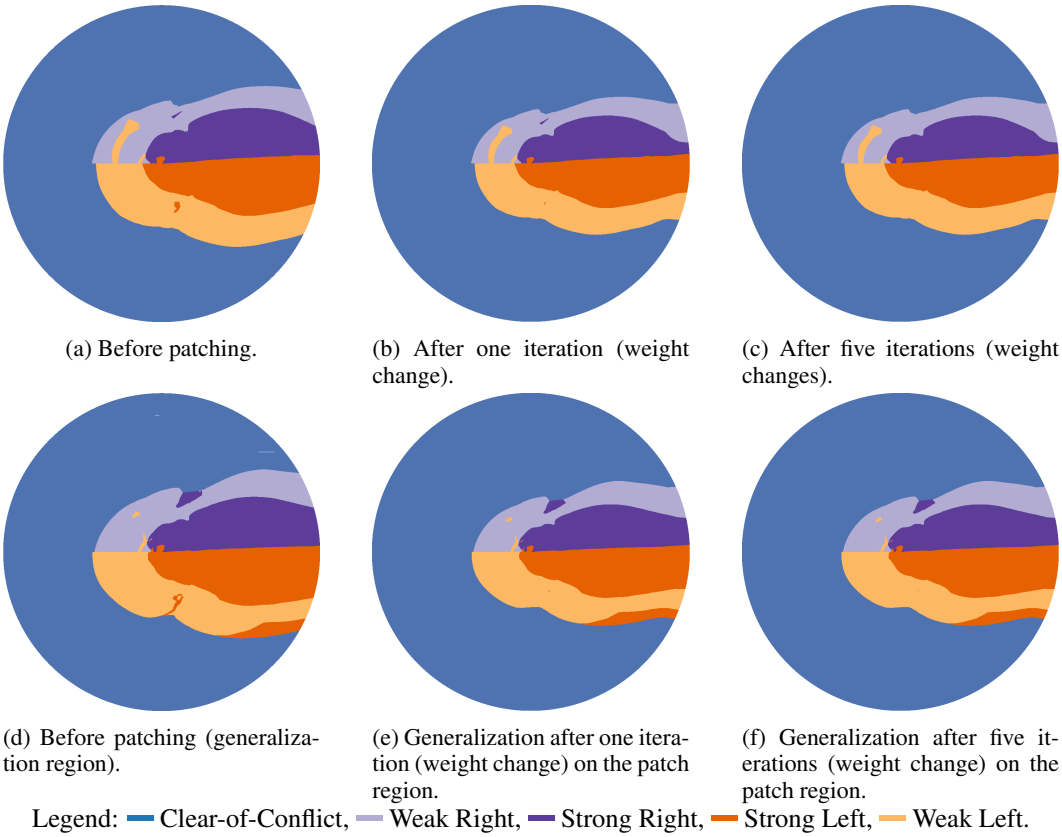
[12] ETH robustness analyzer for neural networks (ERAN). `https://github.com/eth-sri/eran`, 2019. Accessed: 2019-05-01.

[13] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, 2017.

[14] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, 2017.

[15] Ruediger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2017.

[16] Rudy R. Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. A unified view of piecewise linear neural network verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 4795–4804, 2018.



(a) Before patching.

(b) After one iteration (weight change).

(c) After five iterations (weight changes).

(d) Before patching (generalization region).

(e) Generalization after one iteration (weight change) on the patch region.

(f) Generalization after five iterations (weight change) on the patch region.

Legend: Clear-of-Conflict, Weak Right, Strong Right, Strong Left, Weak Left.

Figure 5: Network patching for the "Pockets" spec (Removing the "pockets" of strong-left and strong-right).

A Qualitative patching results

Figures 5—7 show visualizations of the networks before and after patching, both on the two-dimensional region patched over and on another one meant to show generalization of the patch.

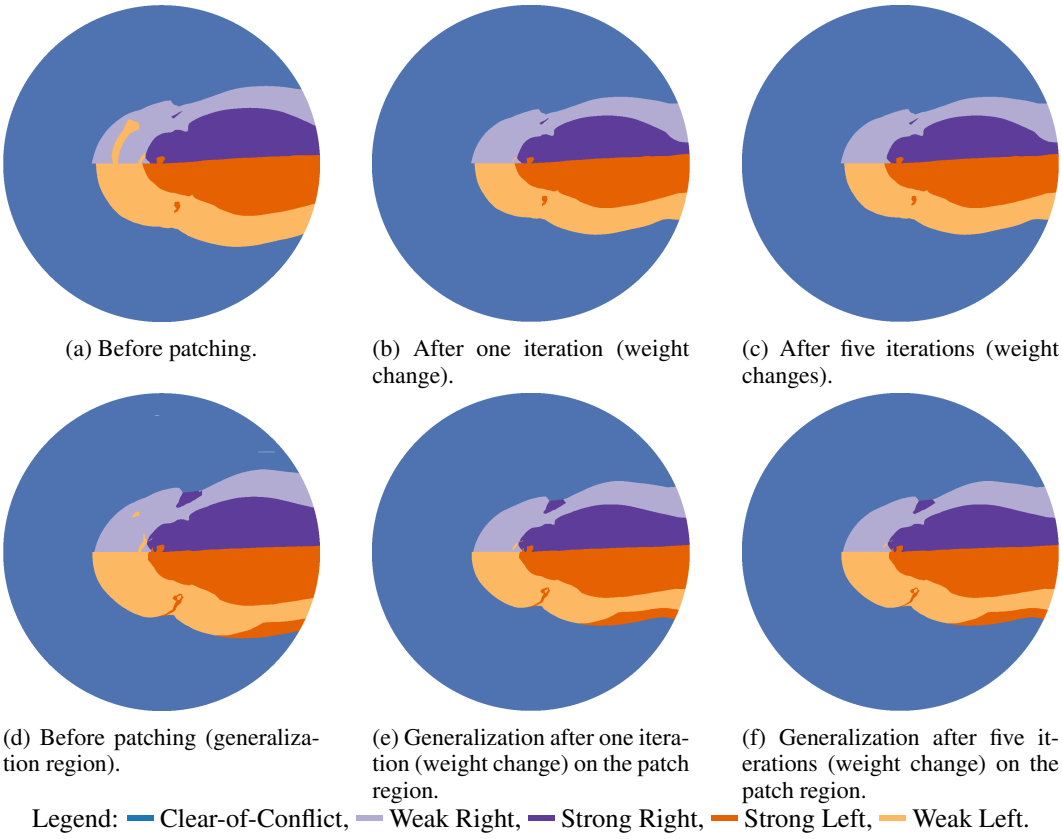
B Patching Multiple Weights as Linear and Polynomial MAX-SMT

In our discussion, we limited ourselves to patching a single weight in a Frozen Network and showed that this corresponds to an *interval MAX-SMT* problem for which there exists a highly-efficient algorithm. However, it is possible to formulate analogues to Theorem 1 when more than a single weight is changed by using a multivariable Taylor expansion.

For standard feed-forward neural networks, the following rules hold:

1. Changing a single weight results in a conjunction of interval constraints.
2. Changing the weights within to *a single layer* in the network results in a conjunction of multivariable *linear* constraints.
3. Changing weights across *multiple layers* in the network results in a conjunction of multivariable *polynomial* constraints.

Although we only present results here for the first case, we have performed experiments with the second case (changing multiple weights within a layer). Even using the state-of-the-art Z3 SMT solver [6], however, it was very slow especially when considering the effectiveness and efficiency of the single- and few-weight patches demonstrated in this paper.



(a) Before patching.

(b) After one iteration (weight change).

(c) After five iterations (weight changes).

(d) Before patching (generalization region).

(e) Generalization after one iteration (weight change) on the patch region.

(f) Generalization after five iterations (weight change) on the patch region.

Legend: Clear-of-Conflict, Weak Right, Strong Right, Strong Left, Weak Left.

Figure 6: Network patching for the "Bands" spec (Removing the band of weak-left behind the origin).

C Efficiency of Patched Frozen Networks

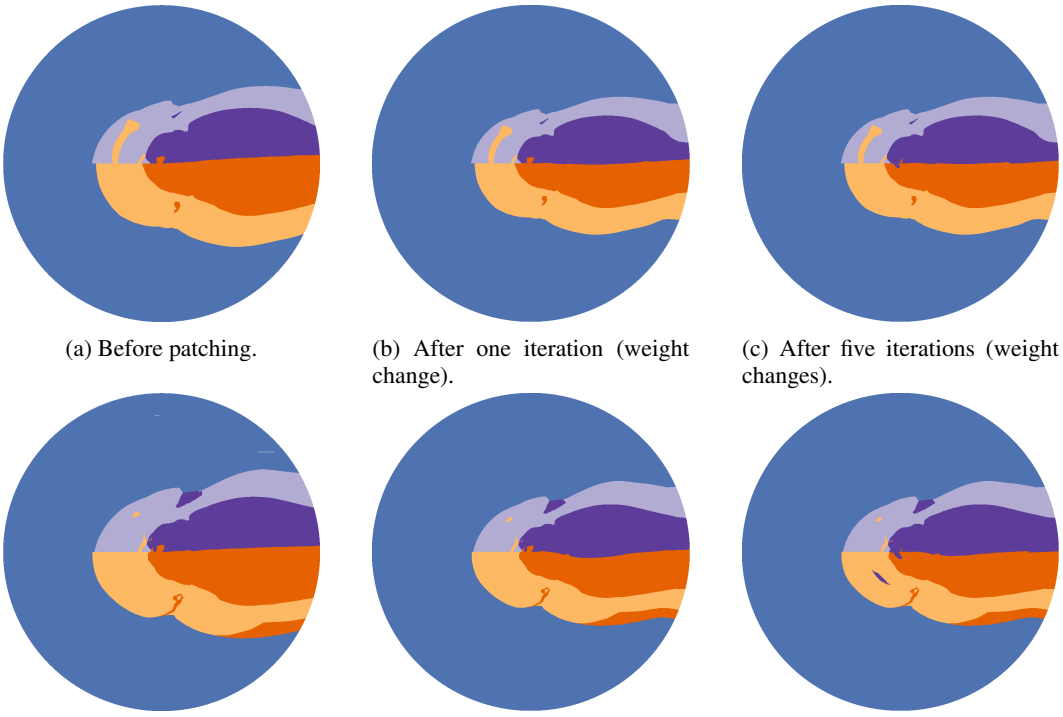
On first glance, frozen networks appear to double the amount of computation necessary to perform inference, and indeed, when there is no relation between the activation and value parameters (θ^a and θ^v) this is the case. However, this is *not* necessarily the case when the parameterizations *share* many weight values. As an extreme example, we have already noted that when $\theta^a = \theta^v$, the network exactly corresponds to a normal ReLU network. As a slightly more complex example, if only a single weight in the last affine layer before a ReLU layer is changed, then the value of the corresponding node's output activation and value vector coefficients differ by a constant difference in the weights multiplied by the corresponding input coefficient, incurring a cost of only one multiply-accumulate operation. In that way, if only a small number of weights differ between the values and activation network, the output of the frozen network can be computed by keeping only "diffs" of the values of the nodes during inference. In Section 6, on two of our patch specifications we patched the last affine layer before the last ReLU layer (so at most one multiply-accumulate operation of overhead) and in another one we patched the very last affine layer, which is not followed by a ReLU layer, so there was no overhead at all.

D Theorem 1

Theorem 1. *Given a frozen network $f(x;\theta^a,\theta^v)$ parameterized by activation parameters θ^a and value parameters θ^v, we define the* single-weight, values-patched *network $f(x;\theta^a,\theta^v+\delta_w)$ which modifies only a single (non-shared) weight w in the value parameters θ^v.*

Then, for any $(P_i,F_i) \in f(x;\widehat{\theta^a,\theta^v})_{\restriction X}$ and x in P_i, the following holds:

$$f(x \in P_i;\theta^a,\theta^v+\delta_w) = F_i(x) + \delta_w \frac{\partial f(x;\theta^a,\theta^v)}{\partial w}$$



(a) Before patching.

(b) After one iteration (weight change).

(c) After five iterations (weight changes).

(d) Before patching (generalization region).

(e) Generalization after one iteration (weight change) on the patch region.

(f) Generalization after five iterations (weight change) on the patch region.

Legend: Clear-of-Conflict, Weak Right, Strong Right, Strong Left, Weak Left.

Figure 7: Network patching for the "Symmetry" spec (Lowering the main decision boundary between strong-left and strong-right to become symmetrical).

Proof. As previously noted, if only the value parameters are patched, then linear partitions of the original network $f(x;\theta^a,\theta^v)$ are still linear partitions of the new network. Thus, the function $f(x \in P_i;\theta^a,\theta^v+\delta_w)$ can be written as a concatenation of linear functions depending on $\theta^v+\delta_w$.

By "non-shared", we mean that the weight w appears at most once in the matrix representation of at most one of the concatenated functions. Thus, we can think of the matrix representation of $f(x \in P_i;\theta^a,\theta^v+\delta_w)$ as the concatenation of matrices $M_1 \circ \cdots \circ M_i \circ \cdots \circ M_n \circ x$ where δ_w appears in at most one coefficient in M_i. Using the associative rule, we can collapse this to AM_iB, where B is a column vector and δ_w appears at most once in M_i. It can then be shown by the definition of matrix multiplication that δ_w appears in at most one of the rows in the column vector M_iB and at most once in each of the rows of the column vector AM_iB.

Thus, $f(x \in P_i;\theta^a,\theta^v+\delta_w)$ is analytic and all partial derivatives with respect to δ_w other than the first are 0. Thus, the first-order Taylor expansion is exact, which is what this theorem claims. □

E Theorem 2

Theorem 2. *Given a frozen network $f(x;\theta^a,\theta^v)$ parameterized by activation parameters θ^a and value parameters θ^v, an input polytope X, and output polytope Y:*

A values-patched network $f(x;\theta^a,\theta^v+\delta)$, *modifying only θ^v, satisfies the property $\forall x \in X : f(x;\theta^a,\theta^v+\delta) \in Y$ if and only if it satisfies the property $\forall (P_i,F_i) \in f(x;\widehat{\theta^a,\theta^v})_{\restriction X} : \forall v \in \mathrm{Vert}(P_i) : f(v;\theta^a,\theta^v+\delta) \in Y$.*

Proof. The function within each partition is affine, and thus convex. If all of the vertices fall inside of Y, then by definition of convexity all of the interior points must as well. The reverse direction

is clear: as we consider bounded polytopes, vertices are interior points so if all interior points fall inside Y then the vertices must as well. $\square$