Finding Invariants in Deep Neural Networks

Divya Gopinath[1], Ankur Taly[2], Hayes Converse[3], and Corina S. Păsăreanu[1]

[1] Carnegie Mellon University and NASA Ames
divgml@gmail.com,corina.pasareanu@west.cmu.edu
[2] Google AI
ataly@google.com
[3] University of Texas, Austin
hayesconverse@gmail.com

Abstract. We present techniques for *automatically* inferring invariant properties of feed-forward neural networks. Our insight is that feed forward networks should be able to learn a decision logic that is captured in the activation patterns of its neurons. We propose to extract such decision patterns that can be considered as *invariants* of the network with respect to a certain output behavior. We present techniques to extract *input* invariants as convex predicates on the input space, and *layer* invariants that represent features captured in the hidden layers. We apply the techniques on the networks for the MNIST and ACASXU applications. Our experiments highlight the use of invariants in a variety of applications, such as explainability, providing robustness guarantees, detecting adversaries, simplifying proofs and network distillation.

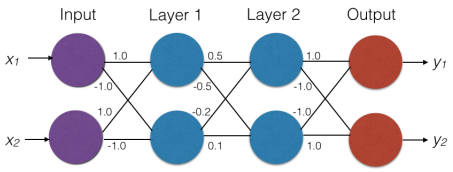
1 Introduction

Deep Neural Networks (DNNs) have emerged as a powerful mechanism for solving complex computational tasks, achieving impressive results that equal and sometimes even surpass human ability in performing these tasks. However, with the increased use of DNNs come also safety and security concerns. These are due to many factors, among them *lack of robustness*. For instance, it is well known that DNNs, including highly trained and smooth networks, are vulnerable to adversarial perturbations. Small (imperceptible) changes to an input lead to mis-classifications. If such a classifier is used in the perception module of an autonomous car, the network's decision on an adversarial image can have disastrous consequences. DNNs also suffer from a *lack of explainability*: it is not well understood why a network make a certain prediction. This lack of explainability impedes on applications of DNNs in safety-critical domains such as autonomous driving, banking, or medicine, due to the lack of *trust* that can be placed on the complex black-box decisions taken by the network. Finally, rigorous reasoning is obstructed by a lack of *intent* when designing neural networks, which only learn from examples, often without a high-level requirements specifications. Such specifications are commonly used when designing more traditional safety-critical software systems.

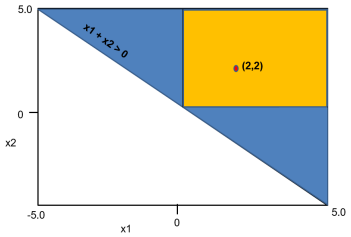
In this paper, we present techniques for *automatically* inferring invariant properties of feed-forward neural networks. An invariant is a predicate in the input space that implies a certain output property, for instance, the network's prediction being a certain class. In this respect, they are like *pre-conditions* of the property, and can serve as explanations for why the property holds There are many choices for defining invariants. In this work, we explore invariants corresponding to *decision patterns* of neurons in the DNN. Such patterns prescribe which neurons are *on* or *off* in various layers. For neurons implementing the ReLU activation functions, this amounts to whether the neuron output greater than zero (*on*) or equal to zero (*off*). We focus on these simple invariants because they are easy to compute and more importantly they define *partitions* on the input space that have simple mathematical representations. Our main theorem shows that for patterns that constrain the activation status (*on* or *off*) of all neurons up to any intermediate layer form convex predicates in the input space. Convexity is attractive as it makes the invariant easy to visualize and interpret. We also study *layer invariants*, which are decision patterns of neurons at an intermediate layer alone, and are natural extensions of input invariants. Other obvious, more complex invariants (e.g. use a positive threshold rather than zero for the activation functions, use linear combinations on neuron values) are left for study in future work.

Another motivation for studying decision patterns is that they are analogous to path constrains in program analysis. Different program paths capture different input-output behaviour of the program. Analogously, we hypothesize that different neuron decision patterns capture different behaviours of a DNN. It is our proposition that we should be able to extract succinct invariants based on decision-patterns that together explain the behavior of the network, define input regions that are free of adversarial perturbations, and can act as formal specifications of networks. We present two techniques to extract invariants based on decision patterns. Our first technique is based on iteratively refining decision patterns while leveraging an off-the-shelf decision procedure. We make use of the decision procedure Reluplex [16], designed to prove properties of feed-forward ReLU networks, but other decision procedures can be used as well. Our second technique uses decision tree learning to directly *learn* invariants patterns from data. The learned patterns can be formally checked using a decision procedure. In lieu of a formal check, which is typically expensive, one could empirically validate invariants over a held-out dataset to obtain confidence in their precision.

We consider this work as a first step in the study of invariants of DNNs. As a proof of concept, we present several different applications. We learn convex invariants for various input classes of an MNIST network, and demonstrate their use in explaining properties of the network as well as debugging misclassifications made by the network. We also study the use of invariants at intermediate layers as interpolants in the proof of input-output contracts (of the form $A(X) \implies B(F(X))$) for a network modeling a safety-critical system for unmanned aircraft control (ACAS XU) [15]. This helps decompose the proof, thereby making it computationally efficient, achieving important savings in time.



(a) Example



(b) Input invariants for the network in Figure 1a.

Fig. 1

Finally, we discuss a somewhat radical application of invariants in distilling [13] the behavior of DNNs. The key idea is to use invariants as distillation rules that directly determine the network's prediction without evaluating the entire network. This results in a significant speedup without much loss of accuracy.

2 Background

A neural network defines a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ mapping an input vector of real values $X \in \mathbb{R}^n$ to an output vector $Y \in \mathbb{R}^m$. For a classification network, the output typically defines a score (or probability) for each m class. The class with the highest score is typically the predicted class. A *feed forward* network is organized as a sequence of layers with the first layer being the input Each intermediate layer consists of computation units called *neurons*. Each neuron consumes a linear combination of the outputs of neurons in the previous layer, applies a non-linear activation function to it, and propagates the output to the next layer. The final output vector Y is a linear combination of the outputs of neurons in the final layer. For instance, in a Rectified Linear Unit (ReLU) network, each neuron applies the activation function $\mathsf{ReLU}(x) = max(0, x)$. Thus, the output of each neuron is of the form $\mathsf{ReLU}(w_1 \cdot v_1 + \ldots + w_p \cdot v_p + b)$ where $v_1, \ldots v_p$ are the outputs of the neurons from the previous layer. $w_1, \ldots, w_p$ are the weight parameters, and b is the bias parameter of the neuron. [4]

Example. We use a simple feed forward ReLU network, shown in Figure 1a, as a running example throughout this paper. The network has four layers: one input layer, two hidden layers and one output layer. It takes as input a vector of size 2. The output vector is also of size 2, indicating classification scores for 2 classes. All neurons in the hidden layers use the ReLU activation function. The final output is a linear combination of the outputs of the neurons in the last hidden layer. Weights are written on the edges. For simplicity, all biases are zero. Consider the input $[2, 2]$. The output on this input is $F([2.0, 2.0]) =$

[4] Most classification networks based on ReLUs typically apply a softmax function at the output layer to convert the output to a probability distribution. We express such networks as $F ::== \mathsf{softmax}(G)$ where G is a pure ReLU network. We then focus our analysis on the network G, while translating properties of the output of F to corresponding properties of G.

$[y_1, y_2] = [2.0, -2.0]$. To see this, notice that the output of the first hidden layer is $[v_{1,1}, v_{1,2}] = [\mathsf{ReLU}(1.0 \cdot 2 + 1.0 \cdot 2),\ \mathsf{ReLU}(-1.0 \cdot 2.0 - 1.0 \cdot 2.0)] = [4.0, 0.0]$, which feeds into the second hidden layer whose output then is $[v_{2,1}, v_{2,2}] = [\mathsf{ReLU}(0.5 \cdot 4.0 - 0.2 \cdot 0.0),\ \mathsf{ReLU}(-0.5 \cdot 4.0 + 0.1 \cdot 0)] = [2.0, 0.0]$. This in turn feeds into the output layer which computes $[y_1, y_2] = [1.0 \cdot 2.0 - 1.0 \cdot 0.0, -1.0 \cdot 2.0 + 1.0 \cdot 0.0] = [2.0, -2.0]$.

A feed forward network is called *fully connected* if all neurons in a hidden layer feed into all neurons in the next layer; the network in Figure 1a is such a network. Convolutional Neural Networks (CNNs) are similar to ReLU networks, but in addition to (fully connected) layers, they may also contain *convolutional layers* which compute multiple convolutions of the input with different filters, before they apply the ReLU activation functions. For simplicity, for the rest of the paper we will discuss ReLU networks, but our work applies to any networks that are piece-wise linear, including ReLUs and CNNs.

Notations and Definitions. All subsequent notations and definitions are for a feed forward ReLU network F, often referred to implicitly. We use uppercase letters to denote vectors and function, and lowercase letters for scalars. We use $N, N', N_1, \ldots$ to range over neurons, and $\mathcal{N}$ for the set of all neurons in the network. For any two neurons N_1, N_2, we define the relation $N_1 \prec N_2$ if and only if the output of neuron N_1 feeds into neuron N_2, either directly or via intermediate layers. We define $\mathit{feeds}(N) ::= \{N' \mid N' \prec N\}$, and extend it to sets of neurons in the natural way.

The output of each neuron N can be expressed as a function of the input X. We abuse notation and use $N(X)$ to denote this function. It is defined recursively via neurons in the preceding layer. That is, if $N_1, \ldots, N_p$ are neurons from the preceding layer that directly feed into N, then $N(X) = \mathsf{ReLU}(w_1 \cdot N_1(X) + \ldots + w_2 \cdot N_2(X) + b)$. For ReLU networks, $N(X)$ is always greater than or equal to 0. We say that the neuron is *off* if $N(X) = 0$ and *on* if $N(X) > 0$. This essentially splits the cases when the Relu fires and does not fire. As we will see in Section 3, the *on/off* activation status of neurons is our key building block for defining network invariants.

3 Network Invariants

We propose the concept of a network invariant for a given output property P. An invariant $\mathit{Inv}(X)$ is a predicate over the input space, such that, all inputs satisfying it evaluate to an output satisfying the property P. In other words, an invariant is a precondition for an output property. An example of an output property for a classification network is that the top predicted class is c, i.e., $P(Y) ::= \mathit{argmax}(Y) = c$. Such properties are called *prediction properties* and the associated invariants called *prediction invariants*.

In general, there could be multiple invariants defined for a given output property P. For instance, the set of all inputs X for which $P(F(X))$ holds is an invariant. But this is of course uninteresting. Ideally, we would like $\mathit{Inv}(X)$ to be driven by the logic embedded in the network, and have a concise mathematical

form. In this work we identify *input invariants* that characterize inputs that are processed in the same way by the network, i.e. they follow the same on/off activation pattern in the network and consequently define convex regions in the input space. We further identify *layer invariants* which are similar to input invariants except they capture common properties over values at intermediate layers, rather than at the input layer.

Decision Patterns. We define invariants based on *decision patterns* of neurons in the network. A decision pattern σ specifies an activation status (*on* or *off*) for some subset of neurons. All other neurons are don't care. We formalize decision patterns σ as partial functions $\mathcal{N} \rightharpoonup \{on, off\}$, and write $on(\sigma)$ for the set of neurons marked *on* and $off(\sigma)$ be the set of neurons marked *off* in the pattern σ. Each decision pattern σ defines a predicate $\sigma(X)$ that is satisfied by all inputs whose evaluation achieves the same activation status for all neurons as prescribed by the pattern.

$$\sigma(X) ::= \bigwedge_{N \in on(\sigma)} N(X) > 0 \ \wedge \bigwedge_{N \in off(\sigma)} N(X) = 0 \tag{1}$$

A decision pattern σ is an invariant for an output property P if $\forall X : \sigma(X) \implies P(F(X))$. For an arbitrary decision pattern σ, the predicate $\sigma(X)$ may not have a concise mathematical closed form. To this end, we seek decision patterns that yield convex predicates in the input space. Additionally, we seek *minimal* invariant patterns σ which have the property that dropping (which amounts to unconstraining) any neuron from the pattern invalidates it as an invariant. Minimality is useful as it helps in getting rid of unnecessary constraints from the invariant.

The **support** of an invariant, denoted by $supp(Inv)$ is a measure of the number of inputs that satisfy the invariant. Formally, it is the total probability mass of inputs satisfying the invariant, under a given input distribution. In the absence of an explicit input distribution, support can be measured empirically based on a training or test dataset. For large networks a formal proof of invariance ($\forall X : \sigma(X) \implies P(F(X))$) may not be feasible. In such cases, one could aim for a probabilistic guarantee that the conditional expectation of $P(F(X))$ given $\sigma(X)$ is above a certain threshold, i.e., $IE(P(F(X)) \mid \sigma(X)) \geq \tau$.[5]

3.1 Input Invariants

An input invariant is a convex predicate in the input space that implies a given output property. To identify input invariants, we consider decision patterns wherein for each neuron N in the pattern, all neurons that feed into N are also included in the pattern. We call such patterns $\prec$-closed. We show that $\prec$-closed patterns capture convex predicates in the input space.

[5] This is very similar to the probabilistic guarantee associated with "Anchors" [23], which we discuss further in Section 6.

Theorem 1. *For all $\prec$-closed patterns σ, $\sigma(X)$ is convex, and has form:*

$$\bigwedge_{1 \leq i \leq |on(\sigma)|} W_i \cdot X + b_i > 0 \ \wedge \bigwedge_{1 \leq j \leq |off(\sigma)|} W_j \cdot X + b_j \leq 0$$

Here W_i, b_i, W_j, b_j are some constants derived from the weight and bias parameters of the network.

The proof is provided in the Appendix. Thus, an input invariant can be obtained by identifying a $\prec$-closed pattern σ such that $\forall X : \sigma(X) \implies P(F(X))$. For convex output properties P, we show that an input invariant can be identified using any input X whose output satisfies P. For this, we consider the *activation signature* of X, which is a decision pattern σ_X that constraint the activation status of *all* neurons to that obtained during the evaluation of X.

Definition 1. *Given an input X, the activation signature of X is a decision pattern σ_X such that for each neuron $N \in \mathcal{N}$, $\sigma_X(N)$ is on if $N(X) > 0$ and off otherwise.*

It is easy to see that σ_X is a $\prec$-closed pattern. We now state a proposition that shows how σ_X can be used to obtain an input invariant. We leverage this proposition in Section 4.

Proposition 1. *Given a convex output property P and an input X whose output satisfies the property (i.e., $P(F(X)$ holds), the following holds: There exist parameters W, b such that:*

(A) $\forall X' : \sigma_X(X') \implies F(X') = W \cdot X + b$
(B) The predicate $\sigma_X(X') \ \wedge \ P(W \cdot X' + b)$ *is an input invariant.*

Example. We illustrate input invariants on the network shown in Figure 1a (introduced in Section 2). Consider the output property that the top prediction is class 1, i.e., $P([y_1, y_2]) ::= y_1 > y_2$. Let $N_{1,1}, N_{1,2}$ be the neurons in the first hidden layer, and $N_{2,1}, N_{2,2}$ be the neurons in the second hidden layer. Consider the pattern $\sigma = \{N_{1,1} \to on, N_{1,2} \to off\}$. We argue that this pattern is an input invariant for the property P. Notice that for all inputs satisfying the pattern σ, neuron $N_{2,1}$ is always *on* and neuron $N_{2,2}$ is always *off*. Consequently the output $[y_1, y_2] = [1.0 \cdot N_{2,1}(X) - 1.0 \cdot N_{2,2}(X), -1.0 \cdot N_{2,1}(X) + 1.0 \cdot N_{2,2}(X)]$ always satisfies $y_1 > y_2$, making the pattern an invariant for the property P. The pattern is $\prec$-closed, and therefore by Theorem 1, the predicate $\sigma(X)$ is convex, and hence an input invariant. The predicate $\sigma(X) = N_{1,1}(X) > 0 \ \wedge \ N_{1,2}(X) = 0$ (see Equation 1) amounts to the convex region $x_1 + x_2 > 0$ (shown in Figure 1b). Interestingly, this pattern is not *minimal* as neuron $N_{1,1}$ being *on* implies that $N_{1,2}$ is *off*, and vice versa. Thus, the patterns $\{N_{1,1} \to on$ and $\{N_{1,2} \to off\}$ are minimal invariant pattern for the property.

3.2 Layer Invariants

While input invariants are attractive for analysis and visualization, they often have tiny support. For instance, the input invariant defined based on the activation signature of an input X may only be satisfied by X, and possible a few

other inputs that are syntactically close to X. Ideally, we'd like invariants to group together inputs that are semantically similar in the eye of the network. To this end, we focus on decision patterns at an intermediate layer that represents high-level features captured by the network.

A layer invariant for an output property P is a decision pattern σ^l over neurons in a specific layer L that satisfies $\forall X : \sigma^l(X) \implies P(F(X))$.[6] Just like input invariants, there can be several layer invariants at a particular layer for the same output property. We show in Section 3.3 that layer invariants can serve as interpolants in proofs of output properties, be used to distill parts of the network, and be used to assess confidence of network predictions.

The downside of layer invariants is that they are typically non-convex and lack a concise mathematical closed form. Interesting, it is possible to decompose a layer invariant as a disjunction of input invariants. This is achieved by extending a layer pattern with all possible patterns over neurons that feed into the layer. Each such extended pattern is $\prec$-closed, and therefore convex (by Theorem 1).

Proposition 2. *Let σ^l be a layer invariant for an output property P. Let $\mathcal{N}^l$ be the set of neurons constrained by σ^l, and let $\sigma_1, \ldots, \sigma_p$ be all possible decision patterns over neurons in feeds$(\mathcal{N}^l)$.[7] Then the following statements hold:*

(A) $\sigma^l(X) \Leftrightarrow \bigvee_i (\sigma^l(X) \ \wedge \ \sigma_i(X))$
(B) For each i, $\sigma^l(X) \ \wedge \ \sigma_i(X)$ *is an input invariant*

Thus, layer invariants can be seen as a grouping of several input invariants as dictated by an internal layer. The astute reader may notice that identifying the right layer is key here. For instance, if one picks a layer too close to the output then the invariant may span all possible input invariants, which is uninteresting. We discuss layer selection in Section 5.

Example. Let us revisit the example in Figure 1a for the output property that the top prediction is class 1, i.e., $P([y_1, y_2]) ::= y_1 > y_2$. A layer invariant pattern for this property is $\{N_{2,1} \rightarrow \mathit{on}, N_{2,2} \rightarrow \mathit{off}\}$. It is easy to see that that for all inputs satisfying this pattern, the output $[y_1, y_2] = [1.0 \cdot N_{2,1}(X) - 1.0 \cdot N_{2,2}(X), -1.0 \cdot N_{2,1}(X) + 1.0 \cdot N_{2,2}(X)]$ will satisfy $y_1 > y_2$, making the pattern an invariant for the property P. The pattern is satisfied by the input $[2, 2]$. The execution of this input involves neuron $N_{1,1}$ being *on* and neuron $N_{1,2}$ being *off*. Consequently, by proposition 2 (part (B)), the extended pattern $\{N_{1,1} \rightarrow \mathit{on}, N_{1,2} \rightarrow \mathit{off}, N_{2,1} \rightarrow \mathit{on}, N_{2,2} \rightarrow \mathit{off}\}$ is an input invariant for property P.

3.3 Interpreting and Using Invariants

Robustness guarantees and adversarial examples. We first remark that provably-correct input and layer invariants for prediction properties define regions in the input space in which the network is guaranteed to give the same

[6] For simplicity, we restrict ourselves to computing invariants with respect to a single internal layer (L) but the approach extends to multiple layers.

[7] There are two $2^{|feeds(\mathcal{N}^l)|}$ such patterns.

label, i.e. the network is robust. Inputs generated from counter-examples of invariant candidates that fail to prove represent potential adversarial examples, as they are semantically similar to benign ones (since they follow the same decision pattern) yet are classified differently. This yields adversaries that are *natural* (in the sense of [31]). We discuss such examples in Section 5.

Explaining network predictions. Neural networks are infamous for being complex black-boxes [17,3]. An important problem in interpreting them is to understand why the network makes a certain prediction on an input. Predictions invariants can be used to obtain such explanations. Such invariants are useful explanations only if they are themselves understandable. Input invariants are useful in this respect as they trace convex regions in the input space. Such regions are easy to interpret when the input space is low dimensional. This is the case for the network ACASXU (Section 5), which predicts horizontal maneuver advisories for an unmanned aircraft based on a several sensory inputs. Input invariants for each advisory serve as sufficient conditions for predicting the advisory.

For networks with high-dimensional inputs, e.g., image classification networks, input invariants may be hard to interpret or visualize. The conventional approach here is to explain a prediction by assigning an importance score, called *attribution*, to each input feature [25,26]. The attributions can be visualized as a heatmap overlayed on the visualization of the input. In light of this, we propose two different methods to obtain similar feature importance visualizations from input invariants.

Under-approximation Boxes. As stated in Theorem 1, an input invariant consists of a conjunction of linear inequations, which can be solved efficiently with existing Linear Programming (LP) solvers. We propose computing *under-approximation boxes* (i.e. bounds on each dimension) as a way to interpret input invariants. Specifically we use LP solving (after a suitable re-writing of the invariant constraints)[8] to find solution intervals $[lo_i, hi_i]$ for each input dimension i such that $\sum_i (hi_i - lo_i)$ is maximized. As there are many such boxes, we constrain each box to include as many inputs from the support as possible. These boxes provide simple mathematical representations of the invariants that are easy to visualize and interpret. Furthermore, and in contrast to other attribution techniques, they provide formal guarantees, as the under-approximating boxes are themselves invariants (since all the inputs in a box lead to same network property).

Minimal Assignments. We also propose another natural way to interpret both input and layer invariants, through the lens of a particular input. In particular we aim to determine which input dimensions (or features) are most relevant for the statisfaction of the invariant. Every concrete input defines an assignment to the input variables $x_1 = v_1 \wedge x_2 = v_2 \wedge .. \wedge x_n = v_n$ that satisfies the invariant $\sigma(X)$. The problem now is to find a *minimal assignment* that still leads to the

[8] We replace each occurrence of variable x_i with lo_i or hi_i based on the sign of the coefficient in the inequalities. See the Appendix for details on the computation of under-approximation boxes.

satisfaction of the invariant, i.e., a minimal subset of the assignments such that $x_{k_1} = v_{k_1} \wedge x_{k_2} = v_{k_2} \wedge .. \wedge x_{k_n} = v_{k_n} \implies \sigma(X)$. The problem has been studied in the constraint solving literature and is computational expensive [2]. We adopt a greedy approach that eliminates constraints iteratively until $\sigma(X)$ is no longer implied; the checks are performed with a decision procedure. The resulting constraints are still network invariants and therefore formally guarantee the corresponding output property.

Layer Invariants as Interpolants. For deep networks deployed in safety-critical contexts, one often wishes to a prove a contract of the form $A \implies B$, which says that for all inputs X satisfying $A(X)$, the corresponding output $Y = F(X)$ satisfies $B(Y)$. For the ACASXU application, there are several desirable contracts of this form, wherein, A is a set of constraints defining a single or disjoint convex regions in the input space, and B is an expected output advisory. Formally, proving such properties for multi-layer feed forward networks is computationally expensive [16]. We show that network invariants, in particular layer invariants, help decompose proofs of such contracts by serving as useful interpolants [19]. Given a layer invariant σ^l for a network satisfying B, the proof for $A \implies B$ can be decomposed using the following rule.

$$\frac{(A \implies \sigma^l), (\sigma^l \implies B)}{(A \implies B)} \tag{2}$$

Thus, to prove $A \implies B$, we must first identify a layer invariant σ^l for the output property B, and then attempt the proof $A \implies \sigma^l$ on the smaller network up to layer l.[9] Additionally, once an invariant σ^l is identified for a property B, it can be reused to prove other contracts involving B. In Section 5, we show that this decomposition leads to significant savings in verification time for properties of the ACASXU network.

Distilling rules from networks. Distillation is the process of approximating the behavior of a large, complex deep network with a smaller network [13]. The smaller network is meant to be favorable to deployment under latency and compute constraints, while having comparable accuracy. We show that layer invariants with high support provide a novel way to perform such distillation. Suppose σ^l is a pattern at an intermediate layer L that implies that the prediction is a certain class c. For any input X, we can execute the network up to layer l, and check if the activation status of the neurons in L satisfy the pattern σ^l. If they do then we can directly return the prediction class c. Otherwise we continue executing the network. Thus for all inputs where the pattern is satisfied, we replace the cost of executing the network from layer l onwards (involving several matrix multiplications) with simply checking the pattern σ. This could be substantial if layer l is sufficiently far from the output and the invariant has high support. Notice that if the invariants are formally verified then this hybrid setup is guaranteed to have no degradation in accuracy. Having said this, we

[9] In theory, the decomposition rule can also be applied to input invariants. However, in practice, input invariants may have very small support and therefore may not form useful interpolants.

also note that most distillation methods typically tolerate a small degradation in accuracy. Consequently, one could empirically validate invariants and select ones that hold with high probability. This avoids the expensive formal verification step making the approach practically attractive. As a proof of concept, we evaluate this approach on an eight layer MNIST network in Section 5.

4 Computing Network Invariants

We now describe two techniques to extract invariants from a feed-forward network. The techniques apply to any convex output property P.

4.1 Iterative relaxation of decision patterns

We describe a technique for extracting input invariants. The technique makes use of an off-the-shelf decision procedure for neural networks. In this, work we use Reluplex [16] but other decision procedures can be used (see Section 6).[10]

Recall from Section 3 that an input invariant is a $\prec$-closed pattern σ that satisfies $\forall X : \sigma(X) \implies P(F(X))$. Ideally we would like to identify the weakest such pattern, i.e., one that constraints the fewest neurons. Computing such an invariant would involve enumerating all $\prec$-closed patterns ($O(2^{|\mathcal{N}|})$), and using a decision procedure to validate whether they are invariants or not. This is computationally prohibitive.

Instead, we apply a greedy approach to identify a *minimal* $\prec$-closed pattern σ, meaning that there is no $\prec$-closed sub-pattern of σ that is also an invariant. We start with an input X whose output satisfies the property P, i.e., $P(F(X))$ holds. Let σ_X be the *activation signature* (see Definition 1) of the input X. By Proposition 1 (Part (B)), we have that $\sigma_X(X') \wedge P(F(X'))$ is an input invariant. But this invariant may not be minimal. Therefore, we iteratively drop constraints from it till we obtain a minimal invariant. We present the complete algorithm in the Appendix (in leiu of space) (algorithm 1). It is easy to see that the resulting pattern is $\prec -closed$, minimal, and an invariant for the output property $(F(X') = y)$.

Proposition 3. *Algorithm 1 (Appendix) always returns a minimal input invariant, and involves at most $n + m$ calls to the decision procedure, where n is the number of layer and m is the maximum number of neurons in a layer.*

Example. Consider the example network from Figure 1a, and the input $X = [2, 2]$ for which the network predicts class 1. We apply Algorithm 1 to identify an input invariant for class 1. The algorithm starts with *activation signature* of X, which is the pattern $\sigma_X = \{N_{1,1} \to on, N_{1,2} \to \mathit{off}, N_{2,1} \to on, N_{2,2} \to \mathit{off}\}$. Notice that σ_X is already an input invariant for class 1. The algorithm begins

[10] In the absence of a decision procedure, testing can also used instead. But then of course, we lose the guarantee that the computed decision patterns are actual invariants though they may still be useful in practice.

to unconstrain all neurons on a per layer basis, starting from the last layer, and identifies layer 1 as the critical layer (exclusion leads to the violation of invariant). The algorithm then unconstrains neurons in layer 1, identifying $\{N_{1,1} \rightarrow on\}$ as a minimal invariant pattern. We pick the order in which neurons are unconstrained in the critical layer arbitrarily to identify a *minimal* set. A different ordering would yield $\{N_{1,2} \rightarrow \mathit{off}\}$ as the minimal invariant pattern.

4.2 Decision tree based invariant generation

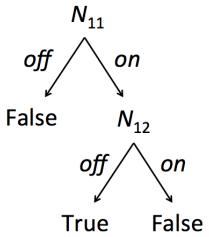
The greedy algorithm described in the previous section is computationally expensive as it involves invoking a decision procedure at each step. We now present a relatively inexpensive technique that avoids using a decision procedure by instead relying on data. By observing the activation signatures of neurons for a large number of inputs, we wish to learn rules that imply various output properties. These rules are are essentially empirically validated invariants, which can be formally checked with a single call to the decision procedure. We use decision tree learning (see Appendix for background) to extract compact rules based on the activation status (*on* or *off*) of neurons in a layer. Decision trees are attractive as they yield conjunctive rules, which are essentially decision patterns. The learning algorithms are designed to discover compact rules based on various information-theoretic measures. Consequently, the obtained rules tend to yield high-support (albeit not minimal) invariant candidates.

Our algorithm works as follows. Suppose we have a dataset of inputs $\mathcal{D}$. Consider a layer l where we would like to learn a layer invariant for an output property P. We evaluate the network on a each input $X \in \mathcal{D}$, and note: (1) the activation status of all neurons in layer l, denoted by σ_X^l, and (2) the boolean P_X indicating whether the output $F(X)$ satisfies property P. Thus we create a labeled dataset of feature vectors σ_X^l mapped to labels P_X; see Figure 2 for an example. We now learn a decision tree from this dataset. The nodes of the tree are neurons from layer l, and branches are based on whether the neuron is *on* or *off*; see Figure 2 (A). Each path from root to a leaf labeled **true** is a rule for predicting the output property. These rules are essentially decision patterns σ specifying whether certain neurons should be *on* or *off*; see Figure 2 (B). We first filter out the patterns that are *impure*, meaning that there exists an input $X \in \mathcal{D}$ that satisfies $\sigma(X)$ but $P(F(X))$ does not hold. The remaining patterns are "likely" invariant candidates for the output property. We sort them in decreasing order of their support and use the decision procedure $DP(\sigma(X), P(F(X)))$ to formally verify them. This last step can be skipped for applications such as distillation (see Section 5) where empirically validated patterns may be sufficient.

We can refine the method for the case where the output property is a prediction property saying that the top predicted class is c, i.e., $P(Y) ::= argmax(Y) = c$. In this case, rather than predicting a boolean as to whether the predicted class is c, we train a single decision tree to directly predict a class label. This lets us harvest invariants for prediction properties corresponding to all classes. Specifically, the path from the root to a leaf labeled class c is a "likely" invariant for the property that the top predicted class is c.

$\langle x_1, x_2 \rangle$	$\langle N_{1,1}, N_{1,2} \rangle$	P
$\langle 1, 1 \rangle$	$\langle on, off \rangle$	True
$\langle -1, -1 \rangle$	$\langle off, on \rangle$	False
$\langle -3, 5 \rangle$	$\langle on, off \rangle$	True
$\langle -2, -2 \rangle$	$\langle off, on \rangle$	False
$\langle 2, 2 \rangle$	$\langle on, off \rangle$	True

(a) Training dataset for decision tree. The first column shows the inputs used to generate the dataset



(b) Resultant decision tree. The pattern harvested for True is $\{N_{1,1} \rightarrow on, N_{1,2} \rightarrow off\}$

Fig. 2: Illustration of the decision tree learning for extracting layer invariants for the network in Figure 1a. The output property is that the top predicted class is class 1.

Counter-example guided refinement. In verifying a decision pattern σ using a decision procedure, if a counter-example is found, we strengthen the pattern by additionally constraining the activation status of those neurons from layer l that have the same activation status for all inputs $X \in \mathcal{D}$ that satisy $\sigma(X)$. If verification fails on this stronger pattern then we do a final step of constraining *all* neurons from layer l based on the activation signature of a single input satisfying the pattern. If verification still fails, we discard the pattern.

5 Applications of Invariants

In this section, we discuss case studies on computing invariants, and using them for different applications. We use two main networks: MNIST (an image classification network based on a large collection of handwritten digits [20]) and ACASXU (a collision avoidance system for unmanned aircraft control [15]). For MNIST we used a network containing 10 layers with 10 ReLU nodes per layer. For ACASXU we used a network with 6 hidden layers and 50 ReLU activation nodes per layer. We implemented our algorithms in Python 3.0 and Tensorflow; we use the linear programming solver `pulp 2.3.1` and the Reluplex [16] decision procedure. The Reluplex checks were run on a server[11]running UBUNTU v6.0.

Invariant Extraction and Robustness Guarantees. For MNIST, we built input invariants using iterative relaxation and layer invariants using decision tree learning. Details about the computed invariants are given in the Appendix. The process took an average of 2.5 hours per input invariant. The proofs for the layer invariants completed within an average of 3 minutes [12]. We also show the adversarial examples generated as counter-examples to the Reluplex checks (Figure 3). Notice that they represent subtle changes that retain the shape of the original digit but fool the network into misclassification.

[11] Dell Prec. T7600 2x 2.70GHz Intel Xeon 8-Core E5-2680 64GB 1x 750GB.

[12] Note that we used a smaller *stripped down* network to include only layers l through k, to check likely layer invariants, hence small verification times.

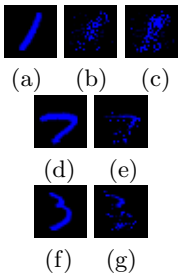


Fig. 3: (a, d, f). original inputs, (b, c, e, g). generated adversaries.

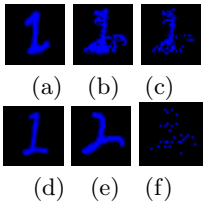


Fig. 4: a. Digit 1 misclassified to 2, (b.,c.) similar 1's misclassified to 2, (d.,e.) Closest validly classified 1 and 2, f. Pixels responsible for misclassification.

Fig. 5: Original digit images and inputs drawn from the under-approx box for their input invariants, and important pixels (red: important pixel originally non-zero in the input image, green: important origianlly zero in input image, black: unimportant) for MNIST.

For ACASXU, we used decision-tree learning to extract candidate layer invariants (listed in the Appendix). We discuss the verification of one invariant for label 0 at layer 5; we subsequently used this invariant as an interpolant in the compositional proofs. This candidate has a support of 109417 inputs. We were able to prove an invariant computed based on this pattern after two refinement steps, within 5 minutes (section 4.2). We also extracted the input invariants corresponding to the layer invariant following proposition 2. From the 109417 inputs that satisfied the invariant for label 0 at layer 5, we extracted 5276 candidate input invariants and were able to prove all of them (within 1 minute per invariant).

Explaining Network Predictions. For MNIST, we calculated under-approximation boxes for all invariants that we proved. For layer invariants, we calculated the boxes w.r.t. each of the input invariants they encompass. These boxes serve to visualize in a natural way different inputs that satisfy the invariant, and hence would be classified to the same label by the network (Figures 5 and 6). Note that the *layer* invariants (Figure 6) help generate images of the same digit written in different ways that would be processed similarly by the network.

We also used the under-approximation boxes to identify pixels that have the most impact in the satisfaction of the invariant. The pixels with smaller ranges are more sensitive or have lesser wiggle room to stay within the box than those with larger ranges. In the last column of Figure 5 and Figure 6, we highlight these pixels. These represent pixels that are important across multiple images, which is in contrast to *attribution* approaches [26] that focus on just one image.

Fig. 6: Original digit images and inputs drawn from under-approx. boxes for different prefixes of MNIST Intermediate Invariants and most significant pixels (red: important pixel originally non-zero in the input image, green: important origianlly zero in input image, black: unimportant)

They potentially represent a **visualization of the features** responsible for a classification.

For ACASXU, we calculated under-approximation boxes corresponding to 5276 input invariants that we had proved. We also experimented with computing minimal assignments on ACASXU which has only 5 input dimensions. For instance, we analyzed a layer 2 invariant of ACASXU for label COC (clear-of-conflict), with a support of 51704 inputs (we provide more details in the Appendix). By computing the minimal assignment over an input that satisfied this invariant, we determined that the last two input attributes; v_{own} and v_{int} are not relevant when the other attributes are constrained as follows: range is 48608, θ was -3.14 and ψ is -2.83. This represents an input-output property of the network elicited by our technique. The domain experts confirmed that this was indeed a valid and novel property of the ACASXU network.

We also note that under-approximation boxes can be used to **reason about misclassified inputs**. Figure 4 shows an image of digit 1 misclassified to digit 2 (Figure 4a). We used this input to extract an input decision pattern and compute an under-approximation box that groups inputs that get misclassified to digit 2 (Figure 4b,c). We can thus generate many new inputs that are similarly misclassified. These inputs can help developers understand the cause of misclassification and re-train the network. We also collected validly classified inputs that are similar to the misclassified digit (share decision prefixes with misclassified input). We built input invariants and boxes for them. We looked at the ranges of each pixel in the boxes corresponding to the misclassified invariant and correctly classified invariants respectively. Those pixels whose ranges were disjoint in the respective boxes were identified as being exclusively important for the misclassification. These pixels ((highlighted in Figure 4e) can help with debugging the network.

Invariants as Interpolants. To evaluate the use of invariants in simplifying difficult proofs, we selected 3 properties from the ACASXUU application. These properties have previously been considered for verification directly using Reluplex [16](refer Appendix for details). All three properties have the form $A \implies B$, where A is constraints on the input attributes, and B indicates that the output turning advisory is Clear of Conflict (COC,label 0). For each property, we evaluated invariants with maximum support for the respective label at every layer, and selected the one that covers maximum number of inputs within

the input region A. For all three properties, the invariant with maximum support for label 0 at layer 5 (σ^5), evaluated highest on this criterion.

Property 1: We found that there are 195 inputs that fall within A and classify as COC. All of these inputs were are also covered by σ^5. We therefore proceeded to prove $A \implies \sigma^5$ and $\sigma^5 \implies B$ using Reluplex. The proof for $A \implies \sigma^5$ finished in 2 minutes. For proving $\sigma^5 \implies B$, we had to strengthen the invariant by constraining all 50 nodes at layer 5. This made the proof go through and finish in 5 minutes. Thus, we were able to prove this property in 7 minutes—a 77% savings compared to direct verification with Reluplex (31 minutes).

Properties 2 and 3: We could not identify a single invariant that covered the inputs within A completely. The invariant σ^5 had maximum coverage with respect to inputs known be classified COC (5276/7618 inputs for property 2, 256/441 for property 3). We split the proof into two parts. First, we extracted the distinct decision prefixes corresponding to the inputs covered by the invariant (denoted as *cov*). These represent input invariants upto layer 5 (σ_X) that satisfy the layer 5 invariant (see Proposition 2). We then checked $(A \wedge \bigvee_{x \in cov} \sigma_X(x)) \implies B$. Checks of the form $(A \wedge \sigma_X(x)) \implies B$ for every σ_X were spawned in parallel. This completed in an hour for property 2 and within 6 mins for property 3. In order to check the portions of the input space that were not covered yet by the proof, we then needed to check $(A \wedge \neg(\bigvee_{x \in cov} \sigma_X(x))) \implies B$. To check this efficiently, we determined the under-approximation boxes for the σ_Xs in *cov*, and spawned parallel checks on the uncovered partitions within A. The latter check completed in 2 hours 10 minutes for property 2 and 25 minutes 34 secs for property 3. This is a promising result as a direct proof of property 2 using Reluplex times out after 12 hours. For property 3, use of the invariant yields a savings in time of 88.33%.

Distillation. Our final experiment is to evaluate the use of layer invariants in distilling a network. As discussed in Section 3.3, the key idea is to use prediction invariants at an intermediate layer as distillation rules. For inputs satisfying the invariant, we save the inference cost of evaluating the network from the intermediate layer onwards. We present a preliminary evaluation of this idea using a more complex MNIST network [1] with 8 hidden layers; two convolutional, one max pooling, two convolutional, one max pooling, and two fully connected layers. The accuracy is 0.9943 but it is computationally expensive during inference. We use the decision tree algorithm to obtain likely invariant candidates. We then empirically validate them (using a validation set of 5000 images), and select ones with accuracy above a threshold τ (see Section 3). The selected invariants are used as distillation rules for inputs satisfying them. We measure the overall accuracy and the overall inference time of this hybrid setup using a held-out test dataset.

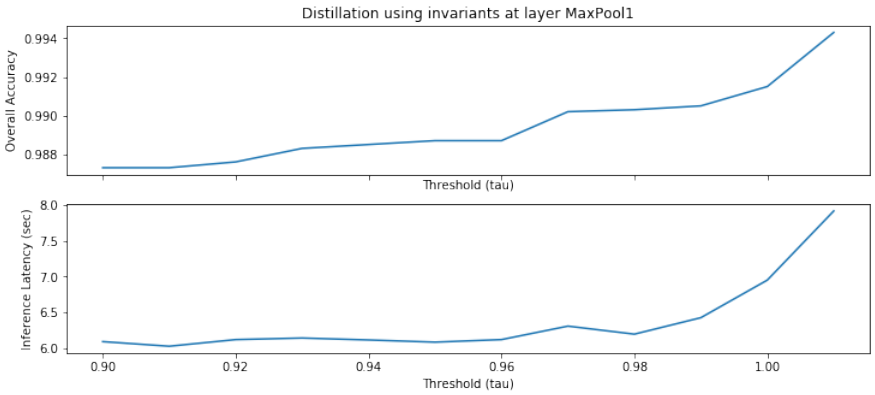


Fig. 7: Distillation of an eight layer MNIST network (from [1]) using invariants the first max pooling layers. The x-axis shows the empirical validation threshold used for selecting invariants. The extreme right point (threshold > 1) corresponds to one where no invariants are selected, and therefore distillation is not triggered. The reported inference times are based on an average of 10 runs of the test dataset on a single core Intel(R) Xeon(R) CPU @ 2.30GHz

Figure 7 shows the results of distillation from the first max pooling layer[13], which consists of 4608 neurons. It shows the trend of overall accuracy and inference time as the threshold τ is varied from 0.9 to 1.0. Observe that at a threshold of $\tau = 0.98$, one can achieve a 22% saving in inference time while only degrading accuracy from 0.9943 to 0.9903. This is quite promising! As expected, lowering the threshold considers more invariants, and therefore reduces both inference time and accuracy. The results from the second max pooling layer (shown in the Appendix in Figure 8 are similar except that both the degradation in accuracy and the saving in inference time are smaller. This is expected as the second max pooling layer is closer to the output, and therefore the invariants approximate a small part of the network.

6 Related Work

We survey the works that are the most closely related to ours. In [27] it has been shown that neural networks are highly vulnerable to small adversarial perturbations. Since then, many works have focused on methods for finding adversarial examples. They range from heuristic and optimization-based methods [12,6,22,1,21] to analysis techniques which can also provide formal guarantees. In the latter category, tools based on constraint solving, interval analysis or abstract interpretation, such as DLV [14], Reluplex [16], AI^2[11] ReluVal [29],

[13] While max pooling neurons are different from ReLU neurons, we could still consider activation patterns on them based on whether they are greater than 0 or equal to 0. A decision tree can then be learned over these patterns to fit the prediction labels.

Neurify [28] and others [4,7], are gaining prominence. Our work is complementary as it focuses on inferring likely properties of neural networks and in principle it can leverage the previous analysis techniques to verify the inferred properties.

There are many recent papers on attribution and explainability in neural networks [30,24,26,25]. They use either perturbation-based or gradient-based approaches in an attempt to identify the inputs that have the most influence over the networks' predictions. Closest to ours is the work on Anchors [23], which aims to explain the network behaviour by means of *rules* (called *anchors*) representing sufficient conditions for network predictions. These anchors are computed solely based on the black-box behaviour of the neural network. In contrast we adopt a white-box approach, that allows us to capture the inner behaviour of the network and to obtain formal guarantees.

There is a large literature on invariant inference, including [8,5,9,10] to name just a few, although none of the previous works have addressed neural networks. The programs considered in this literature tend to be small but have complex constructs such as loops, arrays, pointers. In contrast, neural networks have simpler structure but can be massive in scale.

While the invariants in [18] are meant to capture properties of a given set of inputs (benign inputs), our invariants are meant to capture properties of the network. Furthermore our invariants partitions the input space into prediction-based regions, and are justified with a formal proof.

7 Conclusion

We presented techniques to extract neural network invariants and we discussed applications of the invariants to explaining neural networks, providing robustness guarantees, simplifying proofs and distilling the networks. For the future, and as the field of formal verification for neural networks matures, we plan to incorporate different decision procedures in our tool, thus extending its applicability and scalability. We also plan to apply our approach to textual models and to use invariants for more applications such as confidence modeling and guarding monitors in safety and security applications.

References

1. Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE S&P*, 2017.
2. Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. Minimum satisfying assignments for smt. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 394–409, Berlin, Heidelberg, 2012. Springer-Verlag.
3. Been Doshi-Velez, Finale; Kim. Towards a rigorous science of interpretable machine learning. In *eprint arXiv:1702.08608*, 2017.
4. Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods*

- *10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, 2018.
5. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.
6. Reuben Feinman, Ryan R. Curtin, Saurabh Shintre, and Andrew B. Gardner. Adversarial machine learning at scale, 2016. Technical Report. `http://arxiv.org/abs/1611.01236`.
7. Matteo Fischetti and Jason Jo. Deep neural networks as 0-1 mixed integer linear programs: A feasibility study. *CoRR*, abs/1712.06174, 2017.
8. Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.
9. Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 69–87, 2014.
10. Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 499–512, New York, NY, USA, 2016. ACM.
11. Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 3–18, 2018.
12. Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2014. Technical Report. `http://arxiv.org/abs/1412.6572`.
13. Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.
14. Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV*, 2017.
15. K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Proc. 35th Digital Avionics System Conf. (DASC)*, pages 1–10, 2016.
16. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV*, 2017.
17. Zachary C. Lipton. The mythos of model interpretability. *Queue*, 16:30:31–30:57, 2018.
18. Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. Nic: Detecting adversarial samples with neural network invariant checking. In *NDSS*, 2019.
19. Kenneth L. McMillan. Interpolation and model checking. In *Handbook of Model Checking.*, pages 421–446. 2018.
20. The MNIST database of handwritten digits Home Page. `http://yann.lecun.com/exdb/mnist/`.

21. Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *CVPR*, 2016.
22. Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.
23. Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. Anchors: High-precision model-agnostic explanations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1527–1535, 2018.
24. Avanti Shrikumar, Peyton Greenside, Anna Shcherbina, and Anshul Kundaje. Not just a black box: Learning important features through propagating activation differences. *CoRR*, 2016.
25. Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, 2013.
26. Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *ICML*, 2017.
27. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks, 2013. Technical Report. `http://arxiv.org/abs/1312.6199`.
28. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, 2018.
29. Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018.
30. Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014.
31. Zhengli Zhao, Dheeru Dua, and Sameer Singh. Generating natural adversarial examples. *CoRR*, abs/1710.11342, 2017.

8 Appendix

8.1 Background on Decision Trees

Decision tree learning is a supervised learning technique to extract rules that act as classifiers. Give a set of data $\{D\}$ respective classes $\{c\}$, a decision tree learning technique aims to discover rules (R) in terms of the attributes of the data to discriminate one label from the other. It builds a tree such that, each path of the tree is a rule r, which is a conjunction of predicates on the data attributes. Each rule attempts to cluster or group inputs that belong to a certain label, therefore each leaf corresponds to a certain label with a certain error threshold (i.e. less than error % of inputs in the respective cluster belong to a different label). There could be more than one paths leading to the same label, and therefore more than one rules for the same label. The decision tree algorithm attempts to extract compact rules that are generalizable beyond the training data. Each

rule is *discriminatory* within the tolerance of the error threshold (i.e. two inputs with different labels will have different rules).

$$Tree = DecTree(\{data\}, \{label\}, \{attribute\})$$
$$Tree ::= (path_0 \vee path_1 \vee ... \vee path_n)$$
$$R ::= \{r_0, r_1, ..., r_n\}$$
$$r_i = \bigwedge_{a \in \{attribute\}} predicate(a)$$

8.2 Proof of Theorem 1

Proof. We prove the following stronger property: For all neurons N in a $\prec$-closed pattern σ, there exist parameters W, b such that:

$$\forall X : \sigma(X) \implies N(X) = \mathsf{ReLU}(W \cdot X + b) \tag{3}$$

The theorem can be proven from this property by applying the definition of ReLU. We prove this property for all neurons in σ by induction over the depth of the neurons. The base case of neurons in layer 1 follows from the definition of feed forward ReLU networks. For the inductive case, consider a neuron N in σ at depth k. Let $N_1, \ldots, N_p$ be the neurons that directly feed into N from the layer below. By recursively expanding $N(X)$, we have that there exist parameters $b, w_1, \ldots, w_p$ such that:

$$N(X) = \mathsf{ReLU}(w_1 \cdot N_1(X) + \ldots + w_p \cdot N_p(X) + b) \tag{4}$$

Since σ is $\prec$-closed, $N_1, \ldots, N_p$ must be present in σ. By induction hypothesis, we have that for each N_i (where $1 \leq i \leq p$), there exists W_i, b_i such that:

$$\forall X : \sigma(X) \implies N_i(X) = \mathsf{ReLU}(W_i \cdot X + b_i) \tag{5}$$

Without loss of generality, let $N_1, \ldots, N_k$ be marked *off*, and $N_{k+1}, \ldots, N_p$ be marked *on*. By definition of $\sigma(X)$ (see Equation 1), we have:

$$\forall i \in \{1, \ldots, k\}\ \forall X : \sigma(X) \implies N_i(X) = 0 \tag{6}$$

$$\forall i \in \{k+1, \ldots, p\}\ \forall X : \sigma(X) \implies N_i(X) > 0 \tag{7}$$

From Equations 7 and 5, and definition of ReLU, we have:

$$\forall i \in \{k+1, \ldots, p\}\ \forall X : \sigma(X) \implies N_i(X) = W_i \cdot X + b_i \tag{8}$$

Using Equations 4, 6, and 8, we can show that there exists parameters W and b such that

$$\forall X : \sigma(X) \implies N(X) = \mathsf{ReLU}(W \cdot X + b) \tag{9}$$

This proves the property for neuron N. □

8.3 Algorithms and Results

We present the iterative relaxation of decision patterns in Algorithm 1. We use the notation $DP(A(X), B(X))$ for a call to the decision procedure to check $\forall X : A(X) \implies B(X)$.[14] We use $\sigma \setminus \mathcal{N}$ to denote a sub-pattern of σ wherein all neurons from $\mathcal{N}$ are unconstrained.

First, we drop the constraint $(F(X') = y)$ and invoke the decision procedure to check if $\sigma_X(X') \implies F(X') = y$ (line 2). If this does not hold then $\sigma_X(X') \wedge (F(X') = y)$ is returned as an input invariant. Otherwise, we have that $\sigma_X(X')$ is an input invariant. We then proceed to unconstrain neurons in σ_X. This is done in two steps: (1) We unconstrain all neurons in a single layer, starting from the last layer, until we discover a critical layer cl such that unconstraining all neurons in cl invalidates the invariant (lines 7,8). (2) Within the critical layer cl, we unconstrain neurons one at a time, in some arbitrarily picked order, till we end up with a minimal subset (line 12–17).

Algorithm 1 Iterative Relaxation algorithm to extract input invariant.

```
1: // Let k be the layer before output layer
2: // Let N^l be the neurons in layer l
3: σ_X ::= activation signature of X
4: sat = DP(σ_X(X'), P(F(X')))
5: if ¬sat then return σ_X(X') ∧ P(F(X'))
6: l = k
7: σ = σ_X
8: while l > 1 do
9:     σ = σ \ N^l
10:    sat = DP(σ(X), P(F(X')))
11:    if ¬sat then
12:        // Critical layer found
13:        N^unc = {}
14:        for N ∈ N^l do
15:            N^unc = N^unc ∪ {N}
16:            σ = σ \ N^unc
17:            sat = DP(σ(X), P(F(X')))
18:            if ¬sat then
19:                N^unc = N^unc \ {N}
20:        return σ(X)
21:    else
22:        l=l − 1
```

The computation of underapproximating boxes is presented in Algorithm 2.

[14] Most decision procedures would validate this property by checking whether the formula $A(X) \wedge \neg B(X)$ is satisfiable.

Algorithm 2 Using LP solver to obtain under-approximation box for an invariant.

prob = pulp.LpProblem("UnderApp Box Calc", pulp.LpMaximize)
...
for i=0; i<n;i++ **do**
    d_{hi}=$pulp.LpVariable(name, lowBound$=$min[i], upBound$=$max[i], cat$=$Continuous)$
    d_{lo}=$pulp.LpVariable(name, lowBound$=$min[i], upBound$=$max[i], cat$=$Continuous)$
    $pulpInput[i]$=(d_{lo}, d_{hi})
    $prob$+=$((pulpInput[i][1] - pulpInput[i][0]) >=0)$
    for temp in inp_inv_set **do**
        $prob$+=$((pulpInput[i][1] - temp[i]) >=0)$
        $prob$+=$((-pulpInput[i][0] + temp[i]) >=0)$
$prob+ = pulp.lpSum([(pulpInput[i][1] - pulpInput[i][0])$ for i in range (0,n)$])$
...
//At every ReLU node in the invariant
if ReLU(node) > 0 **then**
    $prob+ = (-(pulp.lpSum(path_const)) <= bias)$
else
    $prob+ = ((pulp.lpSum(path_const)) <= -bias)$
prob.solve()

Statistics from our experiments are presented in Tables 2, 3 and 4.

Table 2: Input Invariants for MNIST listing layers in invariant: nodes in layer and support.

Invariant: Label	Layers:Nodes	Support
INV1: 0	1:0-9, 2:0-9	1928
INV2: 0	1:0-9, 2:0-7	2010
INV3: 0	1:0-9, 2:0-9	217
INV4: 1	1:0-9, 2:0-9	758
INV5: 1	1:0-9, 2:0-5	2
INV6: 1	1:0-9, 2:0-9, 3:0-9, 4:{5}	12
INV7: 2	1:0-9, 2:{2,3,4,5,8,9}	1338
INV8: 2	1:0-9, 2:0-9, 3:0	19
INV9: 2	1:0-9, 2:0	4
INV10: 3	1:0-9, 2:0-9, 3:0-9, 4:{5}	2
INV11: 3	1:0-9, 2:0-9, 3:{3}	52
INV12: 4	1:0-9, 2:0-9, 3:0	97
INV13: 4	1:0-9, 2:0-9, 3:{4}	10
INV14: 5	1:0-9, 2:0-9, 3:0-9, 4:0-9, 5:0-9, 6:0-1	1
INV15: 5	1:0-9, 2:0-9, 3:0-9, 4:0-9, 5:{2}	2
INV16: 6	1:0-9, 2:{0,5}	748
INV17: 6	1:0-9, 2:0	3904

Table 2: Input Invariants for MNIST listing layers in invariant: nodes in layer and support.

Invariant: Label	Layers:Nodes	Support
INV18: 8	1:0-9, 2:{0,2,4,5,8}	358
INV19: 8	1:0-9, 2:0-9, 3:0-9, 4:0-9, 5:0-9, 6:0-5	3
INV20: 9	1:0-9, 2:0-9, 3:0-9, 4:0-2	236
INV21: 9	1:0-9, 2:0-9, 3:0-9, 4:0-9, 5:0-9, 6:0-9, 7:0-9, 8:0-9, 9:0-9	10
INV22: 9	1:0-9, 2:0-9, 3:0-9, 4:0-9, 5:0-9, 6:0-9, 7:0-9, 8:0-9, 9:0-9, 10:0-9	1

Table 3: Intermediate Invariants for MNIST, listing layer of invariant: nodes in layer and support.

Invariant: Label	Layers:Nodes	Support
INV1: 6	1:0-9	3904
INV2: 6	7:{1-4, 7, 9}	5145
INV3: 4	6:{0-2, 4-6, 8}	3078
INV4: 0	7:{1-2, 4-5, 7, 9}	5333
INV5: 0	2:0-9, 3:0-7*	19962
INV6: 3	9:{0, 2-4, 6, 8-9}	3402
INV7: 5	10:{0, 2, 4-5, 7-8}	3075
INV8: 1	2:0-9, 3:0*	18735

Figure 8 shows results obtained from the distillation experiments on MNIST using MaxPool2.

Table 4: ACASXUU Intermediate Invariants, listing number of invariants, total support and the maximum support for an invariant.

Layer	Label	Num of Invs	Total Supp	MAX supp inv
5	0	834	2237734	109147
5	1	776	3742	120
5	2	1139	7744	1324
5	3	1745	20059	2097
5	4	1590	23580	2133
4	0	1554	208136	25489
4	1	1185	7338	732

Table 4: ACASXUU Intermediate Invariants, listing number of invariants, total support and the maximum support for an invariant.

Layer	Label	Num of Invs	Total Supp	MAX supp inv
4	2	1272	7436	745
4	3	2322	22880	1424
4	4	2156	24565	2138
3	0	3923	249771	26134
3	1	1906	7387	210
3	2	1866	6649	134
3	3	3420	21902	945
3	4	2932	20218	552
2	0	1924	219149	51709
2	1	734	4960	497
2	2	819	4460	571
2	3	1746	14487	1262
2	4	1640	14571	1410
1	0	2937	220395	32384
1	1	1031	4422	265
1	2	1123	3611	148
1	3	2285	11756	311
1	4	2112	11386	437

Finally we list here the three properties that we analyzed for ACASXU.

- Property 1: All the inputs within the following region: *55947.691* $\leq$ *range* $\leq$ *679848, -3.14* $\leq \theta \leq$ *3.14, -3.14* $\leq \psi \leq$ *3.14, 1145* $\leq v_{own} \leq$ *1200, 0* $\leq v_{int} \leq$ *60*, should have the turning advisory as Clear-of-Conflict (COC). This property takes approx. 31 minutes to check with Reluplex.
- Property 2: All the inputs within the following region: *12000* $\leq$ *range* $\leq$ *62000, (0.7* $\leq \theta \leq$ *3.14) or (-3.14* $\leq \theta \leq$ *-0.7), -3.14* $\leq \psi \leq$ *-3.14 + 0.005, 100* $\leq v_{own} \leq$ *1200, 0* $\leq v_{int} \leq$ *1200*, should have the turning advisory as COC. This property has a huge input region and direct verification with Reluplex times out after 12 hours.
- Property 3: All inputs within the following region, *36000* $\leq$ *range* $\leq$ *60760, 0.7* $\leq \theta \leq$ *3.14, -3.14* $\leq \psi \leq$ *-3.14 + 0.01, 900* $\leq v_{own} \leq$ *1200, 600* $\leq v_{int}$ $\leq$ *1200*, should have the turning advisory as COC. This corresponds takes approx. 5 hours to check with Reluplex.

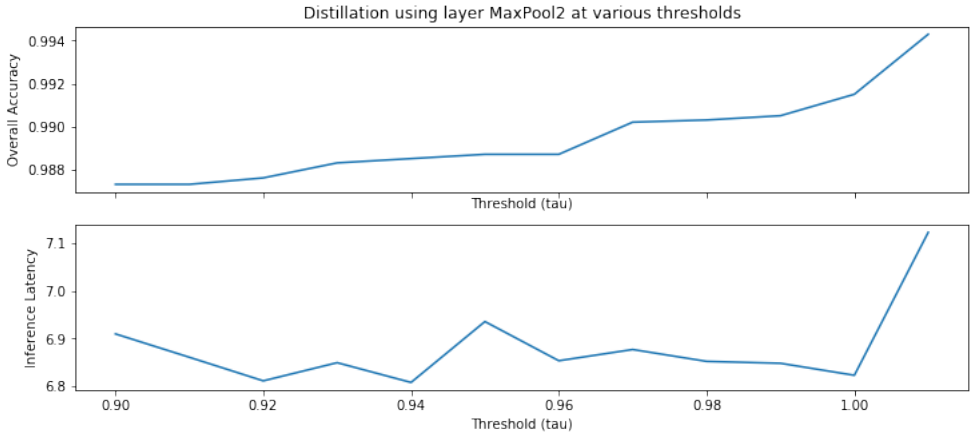


Fig. 8: Distillation of an eight layer MNIST network (from [1]) using invariants the second max pooling layers. The x-axis shows the empirical validation threshold used for selecting invariants. The extreme right point (threshold > 1) corresponds to one where no invariants are selected, and therefore distillation is not triggered. The reported inference times are based on an average of 10 runs of the test dataset on a single core Intel(R) Xeon(R) CPU @ 2.30GHz.