Property Inference for Deep Neural Networks

Divya Gopinath*, Hayes Converse†, Corina S. Păsăreanu* and Ankur Taly‡
* *Carnegie Mellon University and NASA Ames*
Email: divgml@gmail.com,corina.pasareanu@west.cmu.edu
† *University of Texas at Austin*
Email: hayesconverse@gmail.com
‡*Google AI*
Email: ankur.taly@gmail.com

***Abstract*—We present techniques for *automatically* inferring formal properties of feed-forward neural networks. We observe that a significant part (if not all) of the logic of feed forward networks is captured in the activation status (on or *off*) of its neurons. We propose to extract patterns based on neuron decisions as preconditions that imply certain desirable output property e.g., the prediction being a certain class. Together, the inferred preconditions and the output property form a *contract* for the network. We present techniques to extract *input properties*, encoding convex predicates on the input space that imply given output properties and *layer properties*, representing network properties captured in the hidden layers that imply the desired output behavior. We apply our techniques on networks for the MNIST and ACASXU applications. Our experiments highlight the use of the inferred properties in a variety of tasks, such as explaining predictions, providing robustness guarantees, simplifying proofs, and network distillation.**

I. 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, the increased use of DNNs also brings along several 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 misclassifications. 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 makes a certain prediction, which impedes on applications of DNNs in safety-critical domains such as autonomous driving, banking, or medicine. 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 specification. Such specifications are commonly used when designing more traditional safety-critical software systems.

In this paper, we present techniques for *automatically* inferring formal properties of feed-forward neural networks. These properties are of the form $Pre \Rightarrow Post$ and are called *contracts* in line with established work in program analysis. $Post$ is a postcondition stating the desired output behaviour, for instance, the network's prediction being a certain class. Pre is a precondition that we automatically infer and can serve as a *formal explanation* for why the output property holds. We study *input properties* which encode predicates in the input space that imply a given output property, forming *input contracts*. We further study *layer properties* which group inputs that have common characteristics observed at an intermediate layer and that together imply the desired output behavior, forming *layer contracts*. The intention is to capture properties based on the *features* extracted by the network.

There are many choices for defining network properties that are appropriate preconditions for network contracts. In this work, we infer properties 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 function, this amounts to whether the neuron output is greater than zero (on) or equal to zero (*off*). We focus on these simple patterns because they are easy to compute and have simple mathematical representations. Furthermore, they define natural partitions on the input space, grouping together inputs that are processed the same by the network and that yield the same output. Other obvious, more complex properties (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.

We define input properties based on patterns that constrain the activation status (on or *off*) of all neurons up to an intermediate layer. Such patterns form convex predicates in the input space. Convexity is attractive as it makes the inferred properties easy to visualize and interpret. Furthermore, convex predicates can be solved efficiently with existing linear programming solvers. Analogously, we define layer properties based on patterns that constrain the activation status at an intermediate layer. Layer patterns define convex regions over the values at an intermediate layer and can be expressed as unions of convex regions in the input space.

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. Similarly, different neuron decision patterns capture different behaviours of a DNN. It is our proposition that we should be able to extract succinct input-output properties based on decision patterns that together explain the behavior of the network, and can act as formal specifications of networks. We present two techniques to extract network properties. 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 [19], 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* layer 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 the learned patterns over a held-out dataset to obtain confidence in their precision.

We consider this work as a first step in the study of formal properties of DNNs. As a proof of concept, we present several different applications. We learn input and layer contracts for an MNIST network, and demonstrate their use in providing robustness guarantees, explaining the network's decisions and debugging misclassifications made by the network. We also study the use of patterns at intermediate layers as interpolants in the proof of given input-output contracts for a network modeling a safety-critical system for unmanned aircraft control (ACAS XU) [18]. The learned patterns help decompose the proofs thereby making them computationally efficient. Finally, we discuss a somewhat radical application of the learned patterns in distilling [15] the behavior of DNNs. The key idea is to use the patterns that have high support 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.

II. 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 defines a score (or probability) across m classes, and 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 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.[1]

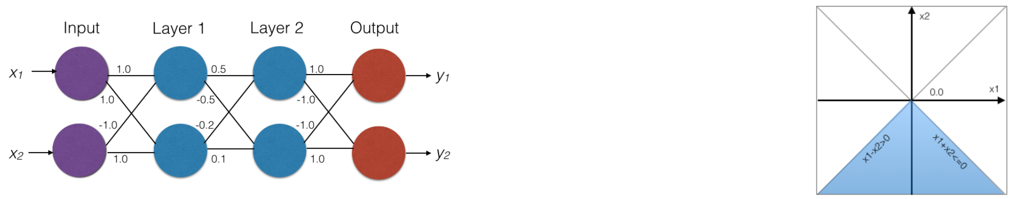
Example. We use a simple feed forward ReLU network, shown in Figure 1a, as a running example throughout this paper. The network has four layers: one input layer, two hidden layers and one output layer. It takes as input a vector of size 2. The output vector is also of size 2, indicating classification 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 $[1.0, -1.0]$. The output on this input is $F([1.0, -1.0]) = [y_1, y_2] = [1.0, -1.0]$. To see this, notice that the output of the first hidden layer is $[v_{1,1}, v_{1,2}] = [\mathsf{ReLU}(1.0 \cdot 1.0 - 1.0 \cdot -1.0), \mathsf{ReLU}(1.0 \cdot 1.0 + 1.0 \cdot -1.0)] = [2.0, 0.0]$. This feeds into the second hidden layer whose output then is $[v_{2,1}, v_{2,2}] = [\mathsf{ReLU}(0.5 \cdot 2.0 - 0.2 \cdot 0.0), \mathsf{ReLU}(-0.5 \cdot 2.0 + 0.1 \cdot 0.0)] = [1.0, 0.0]$. This in turn feeds into the output layer which computes $[y_1, y_2] = [1.0 \cdot 1.0 - 1.0 \cdot 0.0, -1.0 \cdot 1.0 + 1.0 \cdot 0.0] = [1.0, -1.0]$.

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 and then apply the ReLU activation function. For simplicity, we focus our discussion on ReLU networks, but our work applies to all piece-wise linear networks, including ReLUs and CNNs (and in experiments we describe an analysis for a CNN).

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 functions, 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, the relation $N_1 \prec N_2$ holds if and only if the output of neuron N_1 feeds into neuron N_2, either directly or via intermediate layers. We define *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

[1] 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, and then focus our analysis on the network G. Any property of the output of F is translated to a corresponding property of G.



(a) Example

(b) Input property for prediction "1"

Figure 1: Example neural network and input contract

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 III, the *on*/*off* activation status of neurons is our key building block for defining network properties.

III. Network Contracts

Our goal is to extract succinct input-output characterizations of the network behaviour, that can act as formal specifications for the network. The network itself provides an input-output mapping but of course this is uninteresting. Ideally we should group together inputs that lead to the same output and express that in concise mathematical form. To this end we propose to infer *input properties* wrt a given output property P. An input property 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 input property is a *precondition* for *postcondition* P. Together, the input property and the post condition form a formal *contract* for the network. An example of an output property for a classification network is that the top predicted class is c, i.e., $P(Y) ::= argmax(Y) = c$. Such properties are called *prediction postconditions*.

In this work, we infer *input properties* that characterize inputs that are processed in the same way by the network, i.e. they follow the same on/off activation pattern up to some layer and define convex regions in the input space. There may be many such convex regions for a particular output property (say a particular prediction). The union of these regions fully captures the behavior of the network wrt the output property. In practice it may be too expensive to compute precisely this union but we show that even computing a subset of these regions can be useful for many applications.

We further study *layer properties* which encode common properties at an intermediate layer that imply the desired output behavior. Neural networks work by applying layer after layer of transformations over the inputs, to extract *important features* of the data, and then make decisions based on these features. Thus layer properties can potentially capture common characteristics over the extracted features, allowing us to get insights into the inner workings of the network. Similar to input properties, we seek to infer layer properties by studying the activation patterns of the network. Unlike input properties, layer properties do not map to convex regions in the input space, but rather to unions of convex input regions.

Decision Patterns. We infer network properties 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.

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

A decision pattern σ is a network property wrt a postcondition P if:

$$\forall X : \sigma(X) \implies P(F(X)). \quad (2)$$

We seek *minimal* patterns σ which have the property that dropping (which amounts to unconstraining) any neuron from the pattern invalidates it. Minimality helps in getting rid of unnecessary constraints, and ensuring that more inputs can satisfy the property.

The *support* of a pattern, denoted by $supp(\sigma)$, is a measure of the number of inputs that follow the pattern. Formally, it is the total probability mass of inputs satisfying σ, 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 for $\forall X : \sigma(X) \implies P(F(X))$ may not be feasible. In such cases, one could aim for a probabilistic guarantee that the conditional expectation (denoted E) of $P(F(X))$ given $\sigma(X)$ is above a certain threshold, i.e., $\mathsf{E}(P(F(X)) \mid \sigma(X)) \geq \tau$.[2]

A. Input Contracts

To build input contracts we infer input properties that are convex predicates in the input space implying a given

[2] This is similar to the probabilistic guarantee associated with "Anchors" [28], which we discuss further in Section VI.

postcondition. Given that feed forward ReLU networks encode highly non-convex functions, the existence of input properties is itself interesting. To identify input properties, 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.

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

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

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 supplement material. It is based on induction over the depth of neurons in the pattern σ. It shows that the value of any neuron in the pattern can be expressed as a linear combination of the inputs and that each on/off activation adds a linear constraint to the input predicate. [3] Thus, an input property can be obtained by identifying a $\prec$-closed pattern σ such that $\forall X : \sigma(X) \implies P(F(X))$. For convex postconditions P, we show that an input property 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 constrains 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 property. We leverage this proposition in Section IV.

Proposition 1: Given a convex postcondition P and an input X whose output satisfies P (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 property.

Example. We illustrate input properties on the network shown in Figure 1a (introduced in Section II). Consider the postcondition 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 $\sigma = \{N_{1,1} \rightarrow on, N_{1,2} \rightarrow off\}$. We argue that this pattern is an input property wrt P. Since $N_{1,1}$ is *on* it must be the case that the values that feed into $N_{1,1}$ (which have the form $x_1 - x_2$) are positive, hence the inputs satisfy $x_1 - x_2 > 0$. Furthermore, since $N_{1,2}$ is *off* it must be the case that the values that feed into $N_{1,2}$ (which have the form $x_1 + x_2$) are negative, hence the inputs satisfy $x_1 + x_2 \leq 0$. Now notice that all the inputs that satisfy these two constraints also satisfy neuron $N_{2,1}$ is always *on* and neuron $N_{2,2}$ is always *off*. This is because the value that feeds into $N_{2,1}$ is $0.5 \cdot (x_1 - x_2)$ which must be positive (since $x_1 - x_2 > 0$). Similarly the value that feeds into $N_{2,2}$ is $-0.5 \cdot (x_1 - x_2)$ which must be negative. Consequently the output $[y_1, y_2] = [1.0 \cdot N_{2,1}(X) - 1.0 \cdot N_{2,2}(X), -1.0 \cdot N_{2,1}(X) + 1.0 \cdot N_{2,2}(X)] = [0.5 \cdot (x_1 - x_2), -0.5 \cdot (x_1 - x_2)]$ always satisfies $y_1 > y_2$ (when $x_1 - x_2 > 0$), making the pattern a precondition for the property P. The pattern is $\prec$-closed, and therefore by Theorem 1, the predicate $\sigma(X)$ is convex. The predicate $\sigma(X) = N_{1,1}(X) > 0 \;\wedge\; N_{1,2}(X) = 0$ (see Equation 1) amounts to the convex region $x_1 - x_2 > 0 \wedge x_1 + x_2 \leq 0$ (shown in blue in Figure 1b) and is minimal.

B. Layer Contracts

While inferred input properties may be easy to interpret, they often have tiny support. For instance, a property defined based on the activation signature of an input X may only be satisfied by X, and possibly a few other inputs that are syntactically close to X. Ideally, we'd like properties 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 capture high-level features.

A layer property for a postcondition P encodes a decision pattern σ^l over neurons in a specific layer l that satisfies $\forall X : \sigma^l(X) \implies P(F(X))$.[4]

Note that a layer property is convex in the space of values at that layer, but not in the input space. However, it is simple to decompose a layer property as a disjunction of input preconditions. This is achieved by extending a layer pattern with all possible patterns over neurons that feed into the layer (directly or indirectly). Each such extended pattern is $\prec$-closed, and therefore convex (by Theorem 1). The following proposition makes the connection between layer and input properties and it shows that each layer property can be written as a disjunction of input properties.

Proposition 2: Let σ^l be a layer property 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 $feeds(\mathcal{N}^l)$.[5] Then the following statements hold:

(A) For each i, $\sigma^l(X) \;\wedge\; \sigma_i(X)$ is an input property.

(B) $\sigma^l(X) \Leftrightarrow \bigvee_i(\sigma^l(X) \;\wedge\; \sigma_i(X))$.

Thus, layer properties can be seen as a grouping of several input properties as dictated by an internal layer. We note that identifying the right layer is key here. For instance, if one picks a layer too close to the output then the layer property may span all possible input properties, which is uninteresting. In general, the choice of layer would depend on the application. We discuss it further in Section V.

[3] The theorem can also be proven by representing the network as a *conditional affine transformation* as shown in [12].

[4] For simplicity, we restrict ourselves to computing properties with respect to a single internal layer but the approach extends to multiple layers.

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

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

C. Interpreting and Using Inferred Network Properties

Robustness guarantees and adversarial examples. We first remark that provably-correct input and layer contracts defined wrt prediction postconditions characterize regions in the input space in which the network is guaranteed to give the same label, i.e. the network is robust. Inputs generated from counter-examples of pattern candidates that fail to prove represent potential adversarial examples, as they are close (in the Euclidean space) to (regions of) inputs that are classified differently. Furthermore, they are semantically similar to benign ones (since they follow the same decision pattern) yet are classified differently. We show such examples in Section V.

Explaining network predictions. Neural networks are infamous for being complex black-boxes [21], [4]. An important problem in interpreting them is to understand why the network makes a certain prediction on an input. Predictions properties (that ensure that the prediction is a certain class) can be used to obtain such explanations. But, such properties are useful explanations only if they are themselves understandable. Inferred input properties 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.

For networks with high-dimensional inputs (e.g., image classification networks) input properties 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 [31], [32]. 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 visualizations from input properties. We note that in contrast to attributions, which help explain predictions for individual inputs, our proposed input properties help explain the predictions for regions of the input space. Furthermore, and in contrast to existing attribution methods, they provide formal guarantees as the computed explanations are themselves network properties that imply the given postcondition.

Under-approximation Boxes. As stated in Theorem 1, an input property 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 properties. Specifically, we use LP solving (after a suitable re-writing of the constraints)[6] 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 properties, and are easy to visualize and interpret. Note that the under-approximating boxes are themselves network properties that formally imply the input properties and hence the given postcondition.

Minimal Assignments. We also propose another natural way to interpret both input and layer properties through the lens of a particular input. Analogous to attribution methods, we aim to determine which input dimensions (or features) are most relevant for the satisfaction of the property. 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 $\sigma(X)$. The problem now is to find a *minimal assignment* that still leads to the satisfaction of the property, 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 known to be computationally expensive [3]. We adopt a greedy approach that eliminates constraints iteratively and stops when $\sigma(X)$ is no longer implied; the checks are performed with a decision procedure. The resulting constraints are also network properties that formally guarantee the corresponding postcondition.

Layer Patterns 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 [19]. We show that the inferred network patterns, in particular layer patterns, help decompose proofs of such contracts by serving as useful interpolants [23]. Given a layer pattern σ^l, we propose the following rule to decompose a proof.

$$\frac{(A \implies \sigma^l), (\sigma^l \implies B)}{(A \implies B)} \tag{3}$$

Thus, to prove $A \implies B$, we must first identify a layer pattern σ^l that implies output property B, and then attempt

[6] 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 supplement material for details on the computation of under-approximation boxes.

the proof $A \implies \sigma^l$ on the smaller network up to layer l. Additionally, once a layer pattern σ^l is identified for a property B, it can be reused to prove other contracts involving B. In Section V, 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 [15]. The smaller network is meant to be favorable to deployment under latency and compute constraints while having comparable accuracy. We show that layer patterns 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 statuses of the neurons in layer 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 onward (possibly involving several matrix multiplications) with simply checking the pattern σ^l. The savings could be substantial if layer l is sufficiently far from the output, and the layer pattern has high support. Notice that if the patterns are formally verified then this hybrid setup is guaranteed to have no degradation in accuracy. Having said this, we also note that most distillation methods typically tolerate a small degradation in accuracy. Consequently, instead of the expensive formal verification step one could perform an empirical validation of the patterns, and select ones that hold with high probability. This makes the approach practically attractive. As a proof of concept, we evaluate this approach on an eight layer MNIST network in Section V. Interestingly, we note that a network simplified in this manner satisfies the inferred contracts *by construction*, without any proof needed.

IV. Computing Network Contracts

We now describe two techniques to build input and layer contracts from a feed-forward network wrt convex output property P.

A. Iterative relaxation of decision patterns

This is a technique for extracting input properties. It makes use of an off-the-shelf decision procedure for neural networks. In this work, we use Reluplex [19] but other decision procedures can be used too (see Section VI). [7]

Recall from Section III that an input property 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 a property would involve enumerating all $\prec$-closed patterns ($O(2^{|\mathcal{N}|})$), and using a decision procedure to validate whether Equation 2 holds. 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 subpattern of σ that also satisfies Equation 2. We start with an input X whose output satisfies the postcondition 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 property; recall that P is assumed to be convex. But this property may not be minimal. Therefore, we iteratively drop constraints from it till we obtain a minimal property. The algorithm is formally described in the supplement material (see Algorithm 1). It is easy to see that the resulting pattern is $\prec -closed$, minimal, and it implies the output property $(F(X') = y)$.

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

Example. Consider the example network from Figure 1a, and the input $X = [1.0, -1.0]$ for which the network predicts class 1. We apply Algorithm 1 to identify an input property for class 1. The algorithm starts with the *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 property for class 1. The algorithm begins to unconstrain all neurons in each layer, starting from the last layer, and identifies layer 1 as the critical layer (i.e., unconstraining neurons in layer 1 violates the postcondition). The algorithm then identifies $\{N_{1,1} \to on, N_{1,2} \to \mathit{off}\}$ as a minimal pattern that implies the postcondition.

B. Mining layer properties using decision tree learning

The greedy algorithm described in the previous section is computationally expensive as it invokes a decision procedure at each step. We now present a relatively inexpensive technique that relies on data, and avoids invoking a decision procedure multiple times. The idea is to observe the activation signatures of a large number of inputs, and *learn* decision patterns that imply various output properties. In this work, we use decision tree learning (see supplement material for background) to extract compact rules based on the activation statuses (*on* or *off*) of neurons in a layer. Decision trees are attractive as they yield decision patterns that are compact (and therefore have high support) based on various information-theoretic measures. The resulting patterns are empirically validated layer properties, which can be formally checked with a single call to a decision procedure.

Our algorithm works as follows. Suppose we have a dataset of inputs $\mathcal{D}$. Consider a layer l where we would

[7] As discussed, in the absence of a decision procedure, empirical validation of properties can also used. While we would lose the formal guarantee that the computed decision patterns imply the postcondition, they may still be useful in practice.

like to learn a layer property wrt postcondition P. We evaluate the network on each input $X \in \mathcal{D}$, and note: (1) the activation status of all neurons in layer l, denoted by σ^l_X, and (2) the boolean $P(F(X))$ indicating whether the output $F(X)$ satisfies property P. Thus, we have a labeled dataset of feature vectors σ^l_X mapped to labels $P(F(X))$; see for example Figure 2a. 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*. Each path from root to a leaf labeled True forms a decision pattern for predicting the output property; see Figure 2b. We filter out patterns σ 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" layer properties wrt the postcondition. We sort them in decreasing order of their support and invoke 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 V) where empirically validated patterns may suffice.

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

Counter-example guided refinement. In verifying Equation 2 for 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 satisfying the pattern σ. 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. One can also consider a different strategy for refinement, were the counter-examples are added back to the data set and the decision tree learning is re-run, obtaining new layer patterns that will no longer lead to those counter-examples. The drawback is that it may require too many calls to the decision procedure, if many refinement steps are needed.

V. Applications

In this section, we discuss case studies on computing input and layer contracts, and using them for different applications. We implemented all our algorithms in Python 3.0 and Tensorflow. The Python notebook is connected to Python2 Google Compute Engine backend with 12Gb RAM allotted. Our implementation supports analysis of both ReLU and CNN networks. However, for the proofs we use Reluplex [19], which is limited to ReLU networks. To enforce a decision pattern we modified Reluplex to constrain intermediate neuron values. As more decision procedures for neural networks become available, we plan to incorporate them in our tool, thus extending its applicability. The Reluplex runs were done on a server with Ubuntu v16.04 (8 core, 64 GB RAM). We use the linear programming solver `pulp 2.3.1` to solve for under-approximation boxes. We plan to make the implementation and the networks available with a final paper version.

A. Analysis of ACASXU

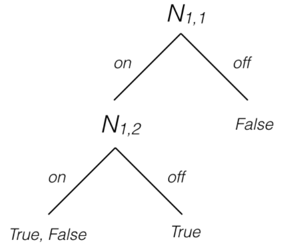
We first discuss the analysis of **ACASXU**, a safety-critical collision avoidance system for unmanned aircraft control [18]. ACASX is a family of collision avoidance systems for aircraft, under development by the Federal Aviation Administration (FAA). ACASXU is the version for unmanned aircraft. It receives sensor information regarding the drone (the *ownship*) and any nearby intruder drones, and then issues horizontal turning advisories aimed at preventing collisions. The input sensor data includes: (1) Range: distance between ownship and intruder; (2) θ: angle of intruder relative to ownship heading direction; (3) ψ: heading angle of intruder relative to ownship heading direction; (4) v_{own}: speed of ownship; (5) v_{int}: speed of intruder; (6) τ: time until loss of vertical separation; and (7) a_{prev}: previous advisory.

The FAA is exploring an implementation of ACASXU that uses an array of 45 deep neural networks, from which we selected one network for discussion here. The five possible output actions are as follows: (0) Clear-of-Conflict (COC), (1) Weak Left, (2) Weak Right, (3) Strong Left, and (4) Strong Right. Each advisory is assigned a score, with the lowest score corresponding to the best action. The network that we analyzed consists of 6 hidden layers, and 50 ReLU activation nodes per layer. We used 384221 inputs with known labels. ACASXU networks were analyzed before with Reluplex [19]. Verification for ACASXU is challenging, taking many hours (may even time out after 12h); we give more details below.

1) Property Inference: We infer network properties wrt prediction postconditions that require that the output of a network classifier is a certain class. We used decision tree learning to extract layer patterns; we list them all (total 25) in Table III in supplement material. The learning took 45 seconds on average per layer (4.5 minutes in total). We discuss here the verification of one specific layer pattern. This pattern was for label COC (clear-of-conflict) at layer 5, and was subsequently used to decompose proofs of ACASXU properties, as discussed below. The pattern has a support of 109417 inputs. We were able to prove a property computed based on this pattern after two refinement steps (Section IV), within 5 minutes. We also extracted candidate input properties corresponding to the decision pattern of the layer property following proposition 2. From the 109417 inputs that satisfied the decision pattern at layer 5, we

$\langle x_1, x_2 \rangle$	$\langle N_{1,1}, N_{1,2} \rangle$	$P(F(X))$
$\langle 0, -1 \rangle$	$\langle on, off \rangle$	True
$\langle 1, 0 \rangle$	$\langle on, on \rangle$	True
$\langle 0, 1 \rangle$	$\langle off, on \rangle$	False
$\langle 4, 3 \rangle$	$\langle on, on \rangle$	False
$\langle 1, -1 \rangle$	$\langle on, off \rangle$	True

(a) Training dataset for decision tree.



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

Figure 2: Illustration of decision tree learning for mining properties for the network in Figure 1a. The output property is that the top predicted class is "1".

extracted distinct decision prefixes corresponding to 5532 inputs. We were able to prove all of them (3600 properties) within an average time of 1 minute per property.

These experiments show that it is feasible to extract input and layer properties in terms of the on/off patterns of the ReLU nodes of real networks. The experiments also show that the patterns constraining lesser number of neurons have higher support and layer properties have higher support than input properties, as expected, since they cover a union of regions in the input space.

2) Explaining Network Predictions: The input-output properties derived for ACASXU can explain the network behavior. We further used LP solving to calculate under-approximation boxes corresponding to input properties. We calculated such a box for each of the 3600 input properties that we had proved. We also generated under-approximation boxes for input decision patterns that could not be proved within a time limit of 12 hours but had high support. This helped elicit novel properties of the network, which were validated by the domain experts. We give some examples below.

– All the inputs within: *31900* $\leq$ *range* $\leq$ *37976, 1.684* $\leq \theta \leq$ *2.5133,* ψ = *-2.83, 414.3* $\leq v_{own} \leq$ *506.86,* v_{int} = *300*, should have the turning advisory as COC.

– All the inputs within: *range = 499, -0.314* $\leq \theta \leq$ *-3.14, -3.14* $\leq \psi \leq$ *0, 100* $\leq v_{own} \leq$ *571, 0* $\leq v_{int} \leq$ *150*, should have the turning advisory as Strong Left.

Please refer the supplement material for more results.

We further experimented with computing minimal assignments that satisfy the inferred properies. For instance, we analyzed a layer 2 property for the label COC, with a support of 51704 inputs. By computing the minimal assignment over an input that satisfied this property, we determined that the last two input attributes, namely, v_{own} (speed of ownship) and v_{int} (speed of intruder) were not relevant when the other attributes are constrained as follows: range = 48608, θ = -3.14 and ψ = -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.

3) Layer Patterns as Interpolants: To evaluate the use of layer patterns in simplifying difficult proofs, we selected 3 properties from the ACASXU application. These properties have previously been considered for verification directly using Reluplex [19]. We list here the three properties.

- 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.

All three properties have the form $A \implies B$, where A specifies constraints on the input attributes, and B specifies that the output turning advisory is COC. For each property, we used decision tree learning to extract multiple layer patterns for label COC at every layer, and selected the one that covers maximum number of inputs within the input region A. Incidentally, for all three properties the same pattern at layer 5 (denoted by σ^5) was selected.

Property 1: We found 195 inputs in the training set that fall within A and classify as COC. All of these inputs are also covered by σ^5. We therefore proceeded to prove $A \implies \sigma^5$ and $\sigma^5 \implies B$ using Reluplex. For proving $\sigma^5 \implies B$, we had to strengthen the pattern by constraining 48 nodes at layer 5. This made the proof go through, and finish in 5 minutes.

We then attempted to prove $A \implies \sigma^5$ for the strengthened version. This process finished in 2 minutes. Thus, we were able to prove this property in 7 minutes. In contrast, direct verification of the property using Reluplex takes 31 minutes.

Properties 2 and 3: We could not identify a single layer pattern that covered the inputs within A completely. The pattern σ^5 had maximum coverage with respect to the training inputs within A ($5276/7618$ inputs for property 2, $256/441$ for property 3). We split the proof into two parts. First, we extracted the activation signature prefixes up to layer 5 for each of the training inputs that satisfy σ^5. Let cov be the set of these prefixes. We then checked $(A \wedge \bigvee_{\sigma_i \in cov} \sigma_i(X)) \implies B$[8] Checks of the form $(A \wedge \sigma_i(X)) \implies B$ were spawned in parallel for every σ_i. This completed in an hour for property 2 and within 6 mins for property 3. The remaining obligation in completing the proof for the property was $(A \wedge \neg(\bigvee_{i \in cov} \sigma_i(X))) \implies B$. To check this efficiently, we determined the under-approximation boxes for each σ_i, and spawned parallel checks on the partitions within A not covered by the boxes. The longest time taken by any job was 2 hours 10 minutes for property 2 and 1 hour 30 minutes 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, a direct proof takes 5 hours.

B. Analysis of MNIST

We also analyzed **MNIST**, an image classification network based on a large collection of handwritten digits [24]. It has 60,000 training input images, each characterized by 784 attributes and belonging to one of 10 labels. We first analyzed a simple network from the Reluplex distribution (containing 10 layers with 10 ReLU nodes per layer). The simplicity of the network makes it amenable to proofs using Reluplex. For the distillation experiments (described in the following subsection) we use a more complex MNIST network that is close to state-of-the art.

1) Property Inference: We extracted input properties using iterative relaxation and layer properties using decision tree learning, showing the feasibility of our approach in the context of image classification, which involves a much larger input space compared to ACASXU. Details about the computed properties (total 30) are given in Tables I and II in the supplement material.

The Reluplex checks for some of the network properties generated counter-examples which show potential vulnerabilities of the network, since they are close (in the Euclidean space) to other inputs that are classified differently (Figure 3).

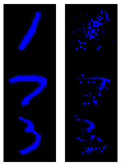


Figure 3: Original images from the data set (left). Counter-examples to failed proofs for patterns containing the original images (right).

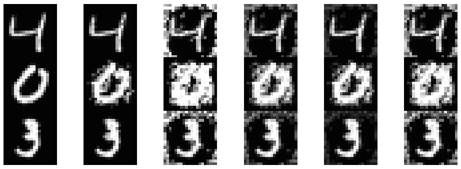


Figure 4: Visualization of MNIST input properties using under-approximation boxes.

2) Explaining Network Predictions: We further computed and visualized under-approximation boxes for the inferred properties. As an example, in Figure 4, we show a visualization of input properties corresponding to three different images from the training set. The first column shows original images. Columns 2 and 3 show images with all pixels set to their minimum and maximum values in the computed under-approximating box, respectively. Columns 4, 5 and 6 have each pixel set to the mean value of its range in the box, a randomly chosen value below the mean, and a randomly chosen value above the mean, respectively.

In Figure 5, we visualize layer properties via under-approximation boxes corresponding to 5 input properties, based on 5 randomly chosen images from the support of the property. Each box is represented by 2 images, setting all the pixels to their respective minimum and maximum values in the box. Note that the images drawn from the under-approximation boxes represent *new inputs* (not in the training set) that satisfy the same property and hence are labelled the same. While input properties capture visually (or syntactically) similar images, layer properties cluster images of the same digit written in different ways, indicating that layer properties can potentially capture common features across inputs. The developer can examine the generated images to get a sense of the image characteristics that contributed to the network decisions.

3) Misclassifications: Under-approximation boxes can also be used to reason about *misclassifications*. Misclassified inputs are typically "rare" and spread across the input space, and it is very difficult for developers to understand their cause and fix the underlying problem. Figure 6 shows an image of digit 1 misclassified to digit 2 (Figure 6, first column). We used this input to extract an input decision pattern and compute an under-approximation box for it

[8] Since σ^5 implies the property B only after strengthening, showing that $(A \wedge \bigvee_{i \in cov} \sigma_i(X)) \implies \sigma^5$ is not enough to ensure that $(A \wedge \bigvee_{i \in cov} \sigma_i(X)) \implies B$.

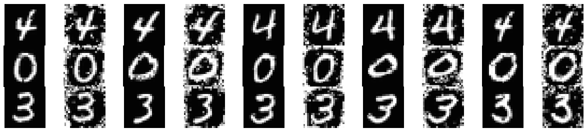


Figure 5: Visualization of MNIST layer properties using under-approximation boxes.

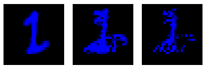


Figure 6: Digit 1 misclassified to 2 and images with min and max values from under-approximation box of original image.

(Figure 6, 2nd and 3rd columns). We can thus draw many more inputs from the box that are similarly misclassified. These inputs can help developers understand the cause of misclassification and re-train the network on them.

C. *Distillation*

Our final experiment is to evaluate the use of layer properties in distilling a network. As discussed in Section III-C, the key idea is to use prediction properties at an intermediate layer as distillation rules. For inputs satisfying the property, 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 network has a superior accuracy of 0.9943 but it is computationally expensive during inference. We use the decision tree algorithm to obtain layer patterns. We then empirically validate them (using a validation set of 5000 images), and select ones with accuracy above a threshold τ (see Section III). The selected properties are used as distillation rules for inputs satisfying them. Using a held-out test dataset, we measure the overall accuracy and inference time of this hybrid setup for different values of τ.

Figure 7 shows the results of distillation from the first max pooling layer[9], which consists of 4608 neurons. The x-axis shows the empirical validation threshold used for selecting properties. The extreme right point (threshold $>$ 1) corresponds to one where no properties 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. The figure 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 further considers more properties, and therefore reduces both inference time and accuracy. The results from the second max pooling layer (shown 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 properties that we infer approximate a smaller part of the network.

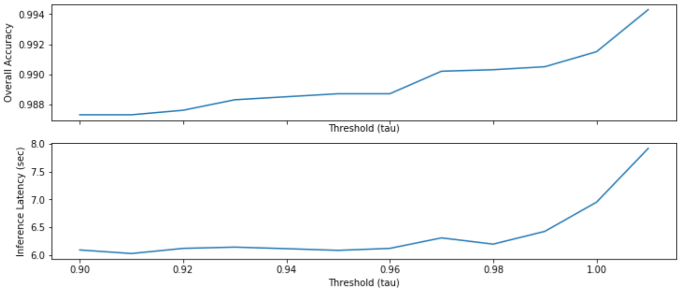


Figure 7: Distillation of an eight layer MNIST network (from [1]) using properties at the first max pooling layer.

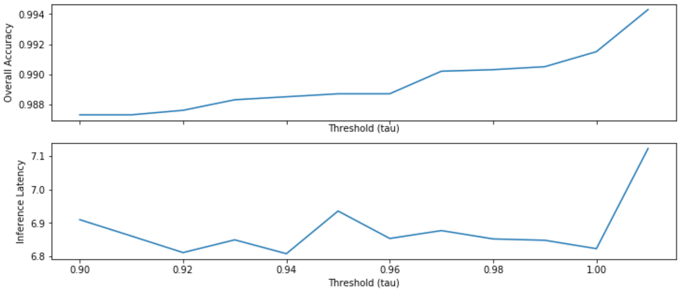


Figure 8: Distillation of an eight layer MNIST network (from [1]) using layer patterns at the second max pooling layers.

VI. Related Work

We survey the works that are the most closely related to ours. In [33] 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 [13], [7], [26], [1], [25] 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 [16], Reluplex [19], AI^2[12] ReluVal [35], Neurify [34] and others [5], [8], are gaining prominence. Our work is complementary as it focuses on inferring input-output properties of neural networks. In principle, we can leverage the previous analysis techniques to verify the inferred properties.

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

There are several papers on explaining predictions made by neural networks, see [14] for a survey. One line of work is on explaining individual predictions by attributing them to input features [31], [27], [30], [32], [22], [17]. They are either based on computing gradients of the prediction with respect to input features [31], [32], back-propagating the prediction score to input features using a set of rules [30], [17], using attribution techniques from cooperative game theory [22], or computing local linear approximations of the behavior of the network [27].

The closest to ours is the work on Anchors [28], which aims to explain the network behaviour by means of *rules* (called *anchors*), which represent sufficient conditions for network predictions. These anchors are computed solely based on the black-box behaviour of the neural network. Input properties from our work can be viewed as anchors for various output properties. The key difference is that our input properties are obtained via a white-box analysis of the neurons in the network, and are backed with a formal guarantee.

Also relevant, there is work on computing the influence of individual neurons on predictions made by the network [2], [20]. In a sense, our layer properties can be seen as a means for identifying influential neurons for a prediction, the key difference being that layer properties also guarantee that decisions of the influential neurons indeed imply the prediction. These previous approaches evaluate neuron influence by measuring how accurately the top k most influential neurons alone can predict the class. Interestingly, we believe these works also lend themselves to distillation. We leave a thorough comparison of different distillation mechanisms to future work.

There is a large body of work on property inference, including [9], [6], [10], [11] 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.

A recent paper [29] uses properties over neuron activation distributions to determine whether a given input is benign (i.e., non adversarial). The 'invariants' in [29] are meant to capture properties of a given set of inputs (benign inputs), while our input and layer properties are meant to capture properties of the network. Furthermore, our properties partition the input space into prediction-based regions, and are justified with a formal proof. We do note that our contracts can be seen as *invariance* properties of the network, that have the special form 'precondition implies postcondition'.

Our distillation approach is related to teacher-student learning in neural networks [15]. Note that we do not perform transfer learning (from a teacher to a student) but instead use the inferred contracts to simplify the network. Thus, unlike teacher/student learning, our distillation approach is *adaptive*, allowing to process some inputs (that satisfy the layer properties) using the simplified computation; the other inputs (that may need more complex processing) go through the original network. Furthermore, we provide formal guarantees as by construction, our 'distilled' network satisfies the contracts used in the distillation.

VII. Conclusion

We presented techniques to extract neural network input-output properties and we discussed their application to explaining neural networks, providing robustness guarantees, simplifying proofs and distilling the networks. As more decision procedures for neural networks become available, we plan to incorporate them in our tool, thus extending its applicability and scalability. We also plan to leverage the decision patterns to obtain parallel verification techniques for neural networks and to investigate other applications of the inferred properties, such as confidence modeling, adversarial detection and guarding monitors for safety and security critical systems.

References

[1] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE S&P*, 2017.

[2] K. Dhamdhere, M. Sundararajan, and Q. Yan, "How important is a neuron," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=SylKoo0cKm

[3] I. Dillig, T. Dillig, K. L. McMillan, and A. Aiken, "Minimum satisfying assignments for smt," in *Proceedings of the 24th International Conference on Computer Aided Verification*, ser. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 394–409. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_30

[4] B. Doshi-Velez, Finale; Kim, "Towards a rigorous science of interpretable machine learning," in *eprint arXiv:1702.08608*, 2017.

[5] S. Dutta, S. Jha, S. Sankaranarayanan, and A. 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.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[7] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Adversarial machine learning at scale," 2016, technical Report. http://arxiv.org/abs/1611.01236.

[8] M. Fischetti and J. Jo, "Deep neural networks as 0-1 mixed integer linear programs: A feasibility study," *CoRR*, vol. abs/1712.06174, 2017.

[9] C. Flanagan and K. R. 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*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 500–517. [Online]. Available: http://dl.acm.org/citation.cfm?id=647540.730008

[10] P. Garg, C. Löding, P. Madhusudan, and D. 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*, 2014, pp. 69–87. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_5

[11] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 499–512. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837664

[12] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. 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*, 2018, pp. 3–18. [Online]. Available: https://doi.org/10.1109/SP.2018.00058

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014, technical Report. http://arxiv.org/abs/1412.6572.

[14] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 93:1–93:42, Aug. 2018.

[15] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *CoRR*, vol. arXiv:1503.02531, 2015. [Online]. Available: https://arxiv.org/abs/1503.02531?context=cs

[16] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *CAV*, 2017.

[17] P. jan Kindermans, K. T. Schütt, M. Alber, K.-R. Müller, D. Erhan, B. Kim, and S. Dähne, "Learning how to explain neural networks: Patternnet and patternattribution," in *International Conference on Learning Representation (ICLR)*, 2018.

[18] 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)*, 2016, pp. 1–10.

[19] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *CAV*, 2017.

[20] K. Leino, S. Sen, A. Datta, M. Fredrikson, and L. Li, "Influence-directed explanations for deep convolutional networks," in *IEEE International Test Conference, ITC 2018, Phoenix, AZ, USA, October 29 - Nov. 1, 2018*, 2018.

[21] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, pp. 30:31–30:57, 2018.

[22] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 4765–4774.

[23] K. L. McMillan, "Interpolation and model checking," in *Handbook of Model Checking.*, 2018, pp. 421–446. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_14

[24] "The MNIST database of handwritten digits Home Page," http://yann.lecun.com/exdb/mnist/.

[25] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A simple and accurate method to fool deep neural networks," in *CVPR*, 2016.

[26] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *EuroS&P*, 2016.

[27] M. T. Ribeiro, S. Singh, and C. Guestrin, """Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, 2016, pp. 1135–1144.

[28] ——, "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*, 2018, pp. 1527–1535.

[29] G. T. W.-C. L. X. Z. Shiqing Ma, Yingqi Liu, "Nic: Detecting adversarial samples with neural network invariant checking," in *NDSS*, 2019.

[30] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Learning important features through propagating activation differences," *CoRR*, 2016.

[31] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *CoRR*, 2013.

[32] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *ICML*, 2017.

[33] 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.

[34] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. 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.

[35] ——, "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.