Prophecy: Inferring Formal Properties from Neuron Activations

Divya Gopinath, Corina Păsăreanu, and Muhammad Usman

NASA Ames, KBR, CMU

Abstract. We present Prophecy, a tool for automatically inferring formal properties of feed-forward neural networks. Prophecy is based on the observation that a significant part of the logic of feed-forward networks is captured in the activation status of the neurons at inner layers. Prophecy works by extracting rules based on neuron activations (values or *on*/*off* statuses) as preconditions that imply certain desirable output property, e.g., the prediction being a certain class. These rules represent network properties captured in the hidden layers that imply the desired output behavior. We present the architecture of the tool, highlight its features and demonstrate its usage on different types of models and output properties. We present an overview of its applications, such as inferring and proving formal explanations of neural networks, compositional verification, run-time monitoring, repair, and others. We also show novel results highlighting its potential in the era of large vision-language models.

1 Introduction

Deep Neural Networks (DNNs) are being used in many tasks, some of them in critical domains, such as banking, medicine, or transportation, raising safety and security concerns. These are due to many factors, among them lack of robustness and transparency and also lack of intent. In fact, neural networks only learn from examples, often without a high-level requirement specification. In contrast, more traditional safety-critical software systems are usually designed based on high-level, often formalized, requirements.

In this paper, we present **Prophecy**, a tool for automatically inferring formal properties of feed-forward neural networks. Neural networks work by applying layer transformations to the inputs, extracting important features from the data, and making decisions based on these features. Prophecy aims to extract *layer properties* that capture common characteristics over the extracted features, allowing insight into the inner workings of the network. These properties are of the form $Pre \Rightarrow Post$. $Post$ is a post-condition stating some desired output behavior. They are typically simple and easy to specify, for instance, the network's prediction being a certain class. Pre is a precondition that Prophecy automatically infers and can serve as a formal explanation for why the output property holds. Each layer property groups inputs that have common characteristics observed at an intermediate layer and that together imply the desired output behavior.

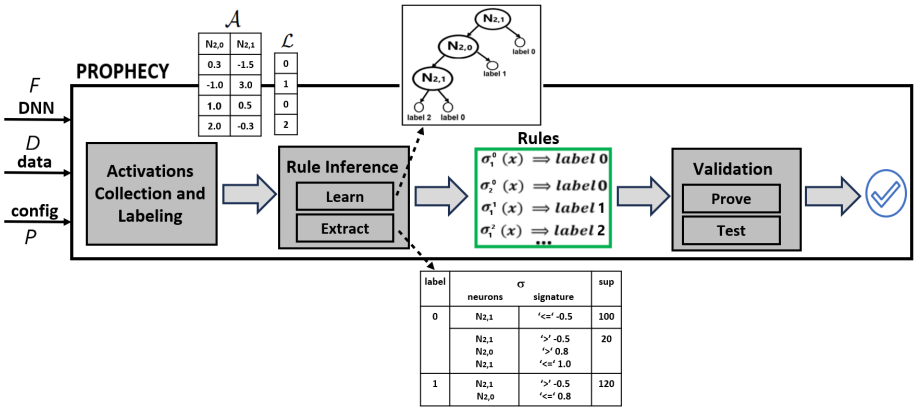

Fig. 1. Prophecy Tool. Overview Diagram.

Several studies [17, 31, 21] have shown that neuron activations (values or *on/off* statuses) at a layer effectively represent the features at that layer. Typically, dense layers closer to the output encompass the logic relevant to model decisions. Guided by these insights, simple rules in terms of conditions on neuron outputs at these layers, have the potential to capture the logic corresponding to a desired model behavior. Prophecy uses a relatively inexpensive technique that relies on data to infer such **rules**. The idea is to observe neuron activations for a large number of inputs and learn patterns that act as the *Pre* implying desired output properties. In particular, it uses *decision-tree learning* [23, 24], an efficient algorithm which employs various information-theoretic measures to yield rules with simple logical constraints that are compact and thus generalizable.

Methodology. The Prophecy framework (Fig. 1) takes as inputs; a pre-trained model (F), a post-condition or output property (P), and a set of data (D) representative of in-distribution for the model. The *Activations Collection and Labeling* module evaluates the network on each input, $x \in D$, to collect the neuron values or just the *on/off* (1/0) activations at a given layer l. This creates the set of all neuron activations, $\mathcal{A}$. The property $P(F(x))$ is evaluated on each input to yield a boolean value indicating whether the output $F(x)$ satisfies the property P or not. A labels dataset ($\mathcal{L}$) is generated based on the evaluation of P on $\mathcal{D}$. However, the actual label values in $\mathcal{L}$ are determined based on the type of post-condition property. For instance, when the output property is a prediction post-condition; $P(F(x)) ::= (F(x) = c)$, $\mathcal{L}$ is a list of the corresponding class values. This enables using the decision-tree to directly predict the class label rather than a boolean as to whether the prediction is c.

The *Rule Inference* module invokes the decision-tree learning algorithm to train an estimator using the datasets $\mathcal{A}$ and $\mathcal{L}$. The resulting tree predicts the valuation of P on any given input. Each path of the tree corresponds to a neuron pattern, σ; a conjunction of conditions in terms of the neuron activations.

A path leading to a $True$ valuation of P can be interpreted as a rule which states that for all inputs satisfying σ at layer l, $P(F(x))$ would hold true. Thus, this module generates a collection of *layer properties*; each predicting model behavior w.r.t a desirable property in terms of a conditional logic on the features at that layer. The *Validation* module enables evaluating the generated rules and selecting those with high quality for subsequent applications. Each rule can be empirically validated on a held-out dataset to obtain confidence in its precision. The module also enables formal verification of the rules (using a decision-procedure $DP(F, \sigma, P, l)$) to provide guarantees about model behavior on all inputs satisfying the pre-condition.

Prophecy, thereby, serves to decompose a complex black-box DNN model, into a set of compact rules amenable to analysis. Rules act as likely specifications of the model with a precise mathematical form. This in turn enables multitude of applications; obtaining formal explanations and guarantees of behavior, testing, debugging, analysis in terms of human-understandable concepts and runtime monitoring. The idea, initially presented in [11], is one of the first formal approaches to look into the black-box of a DNN model. Prophecy and its applications are well-referenced (over 55 citations) and explored both in the industry (NASA, Boeing, DoT, FAA) and academia (York, Stanford).

Related Work. Approaches such as [4, 10, 7, 5] construct alternate, lightweight and explainable representations of a DNN; decision-trees that are functionally equivalent to the entire model. Prophecy, in contrast, focuses on using decision-tree learning to build a set of high-quality rules in terms of the features at an inner layer. In this sense, it is closer to *probing* [13, 6], a popular analysis strategy, most widely used for NLP tasks [18, 9, 3]. However, these techniques mainly target interpretability; mapping activations to high-level concepts [16], with a separate step (such as intervention [8]) to establish causal relationship with model outputs. Prophecy offers the flexibility to vary the target of the probe to enable a variety of analyses. In contrast to all aforementioned methods, the scope is restricted to a specific output property and to the set of inputs satisfying the pre-condition. This enables building simple yet precise rules for analysis.

Contributions. Prophecy has never been published as a tool and this work focuses specifically on this aspect to promote more wide-spread usage.

1. *Providing a functional tool:* The basic idea and methodology of Prophecy was presented in [11]; the prototype implementation required the user to manually edit the code to enable different functionalities (such as extracting different types of rules for different post-conditions for different models). In addition, the user had to manually process the extracted rules and invoke the solver externally to formally check them. In this work, we present a functional automated tool, *DNN-Prophecy*[1]. The implementation was refactored to modularize the code and enable invocation of different functionalities (including the invocation of Marabou [15] solver) via command-line parameters.

[1] Subsequent references to Prophecy imply the new version unless explicitly specified.

We have also extended the implementation to extract and prove more general rules (not just in terms of on/off activations).

2. *Optimizing implementation to improve efficiency:* We extend the previous implementation ([11]) to speed up accessing the rules at runtime (Section 4).
3. We summarize the many applications of Prophecy since 2019, namely, repair, test coverage, runtime monitoring, poison-attack mitigation, and feature-guided analysis (Section 3), highlighting the effectiveness of the layer properties which act as logical probes in the embedding space of the network.
4. We present a novel application and evaluation of Prophecy on a Vision Language Model (Section 4).

2 Tool Description

In this section we describe in more detail the rules extracted by Prophecy, the algorithms employed and the tool usage.

2.1 Definition of rules

Let F denote a feed-forward neural network and F_l denote the function representing the transformations applied by all layers from the input till layer l. A layer property for a post-condition P encodes a pattern σ over the neuron values in layer l that satisfies the rule, $\forall x \in X\ \sigma(F_l(x)) \Rightarrow P(F(x))$.

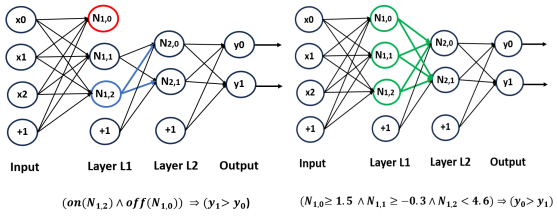

Fig. 2. Example Layer Properties.

In the original paper [11], we expressed σ in terms of *on/off* activations, $\sigma(X) ::= \bigwedge_{N_i \in (N_\sigma \subseteq N_l)} (on(N_i(X)) \vee off(N_i(X)))$, $on(N_i(X)) \iff (N_i(X) > 0)$, $off(N_i(X)) \iff (N_i(X) <= 0)$. N_l is the set of neurons at layer l, each of which can be recursively expressed as a function of X. N_σ is a subset of N_l constrained by σ. We have since then extended σ to express more general mathematical constraints over the neuron values, $\sigma(X) ::= \bigwedge_{N_i \in (N_\sigma \subseteq N_l)} N_i(X)\ op\ V_i$, where V_i is a real-valued threshold, and operator $op \in \{>=, <=\}$. The neuron pattern defines a convex region in $F_l(X)$ and a union of convex regions in the input space.

Figure 2 (left) shows an example property that specifies that for all inputs with $on(N_{1,2})$ and $off(N_{1,0})$ at layer $L1$ (and any value for other neurons), the output is most likely label 1 ($y_1 > y_0$). The other property (right) specifies conditions in terms of specific values which lead to label 0, ($y_0 > y_1$).

2.2 Algorithms for Activations Collection, Labeling, and Proving

Activations Collection and Labeling implements the functionality described in Algorithm 1, taking in the pre-trained model F, dataset D (a set of model inputs and corresponding labels) and configuration parameters (C_{config}) as set by the user. The parameter, *layer_name*, specifies the layer at which the activations are to be collected and is typically one of the dense layers close to the output. The *acts* parameter determines if on/off neuron activations are to be collected or neuron values. Note that the parameter *inptype* can be set to 1 to optionally skip executing the model to collect the layer activations. In this setting, the user-provided dataset (x) is expected to contain the list of activation vectors (refer Section 5.3).

The purpose of the *LABEL* function is to evaluate the user-provided post-condition property, P, on the network output for each input. The *type* parameter acts a short-cut for specifying standard properties on the predicted labels of typical classification models. The parameter when set to 0, enables evaluation of $P(F(X)) ::= (F(X) = c)$. The label predicted by the model for each input is used to populate $\mathcal{L}$. The $type = 1$ indicates a correctness post-condition; $P(F(X)) ::= (F(X) = c_{ideal})$. The input labels dataset (y) is expected to contain the ideal ground-truth labels for every input and $\mathcal{L}$ is populated with either 1 or 0 based on the satisfaction or violation of correctness respectively. Similarly, $type = 2$, enables evaluation of $P(F(X)) ::= (F(X) = c) \wedge (F(X) = c_{ideal})$. Currently the tool enables handling models which are not classification and/or other post-condition properties by setting $type = 3$. In this setting, the input labels dataset (y) is expected to directly contain the results of the post-condition evaluation on the model output for each input (refer Section 5.3 in the appendix).

Algorithm 1 Activations Collection and Labeling

Require: F,$D ::= \{(x, y)\}$, $C_{\text{config}} ::= \{type, layer\_name, acts, inptype, ...\}$
Ensure: $\mathcal{A}$, $\mathcal{L}$
$A \leftarrow \emptyset$, $L \leftarrow \emptyset$, $l \leftarrow layer\_name$
for $x \in D$ **do**
  if $inptype = 0$ **then**
    if $acts =$ `False` **then**
      $A \leftarrow A \cup \{F_l(x)\}$
    else
      if $F_l(x) > 0$ **then** $A \leftarrow A \cup \{1\}$
      else $A \leftarrow A \cup \{0\}$
      end if
    end if
  else
    $A \leftarrow A \cup \{x\}$
  end if
  $c \leftarrow \text{LABEL}(x, y, F, type)$
  $L \leftarrow L \cup \{c\}$
end for
return (A, L)

function LABEL($x, y, F,$ type)
  if type $= 0$ **then return** $F(x)$
  else if type $= 1$ **then**
    if $F(x) = y$ **then return** 1
    else return 0
    end if
  else if type $= 2$ **then**
    if $F(x) = y$ **then return** $F(x)$
    else return 1000
    end if
  else if type $= 3$ **then return** y
  end if
  return None
end function

Rule Inference first executes the *Learn* module, which invokes the *sklearn.tree.DecisionTreeClassifier* library [1] to build a decision-tree estimator. Each path of the tree from the root to a leaf forms a pattern (in terms of the values in $\mathcal{A}$) for predicting a label in $\mathcal{L}$. There can be more than 1 paths leading to

the same label (example tree in Fig. 1). Each path can also lead to a leaf with more than 1 labels (impure). The next step, *Extract*, parses the tree to collect paths leading to *pure* leaves. This process builds a dictionary; mapping each distinct label in $\mathcal{L}$ to a set of prediction rules. Each rule is represented by a triple, {neurons, signature, support} (Fig. 1). Let $\{\sigma_1^c, \sigma_2^c, \ldots, \sigma_n^c\}$ denote the set of rules associated with label c, where each rule σ_i^c has an associated support metric, supp(σ_i^c) defined as the number of instances in D that satisfy σ_i^c (and $P(F(X))$ since only pure rules are extracted). Each rule can be **validated** by the *Test* module which computes statistical metrics (precision, recall and F1) on a separate dataset.

Algorithm 2 Iterative Rule-proving using Marabou [19]. The class and its methods from Marabou are highlighted in blue.

Require: c, l,$X^s_{\min}$, $X^s_{\max}$, $V^s_{\min}$,$V^s_{\max}$, X_0,V_0, F_{onx} : $MarabouNetworkONNX$
Ensure: $result$, $labs$
$result \leftarrow False$, $it \leftarrow 0$, $labs \leftarrow \{c' \neq c\}$
while $result = False$ **do**
  if it = 0 **then**
    MB_SETBNDS($X^s_{\min}, X^s_{\max}$,$V^s_{\min}, V^s_{\max}$, l,F_{onx})
  end if
  if it = 1 **then**
    MB_SETBNDS($X^s_{\min}, X^s_{\max}$,V^0, V^0, l,F_{onx})
  end if
  if it = 2 **then**
    MB_SETBNDS(X^0, X^0,V^0, V^0,l,F_{onx})
  end if
  $result$, $labs$ = MB_SOLVE($labs$,F_{onx},t)
  $it = it + 1$
end while
return $result$, $labs$

function MB_SETBNDS(X_{min},X_{max}, V_{min},V_{max},l,F_{onx})
  $Ivars = F_{onx}.inputVars$
  for $idx = 0$ to $|Ivars|$ **do**
    $i = Ivars[idx]$
    $F_{onx}.setLowerBound(i, X_{min}[idx])$
    $F_{onx}.setUpperBound(i, X_{max}[idx])$
  end for
  $Lvars = F_{onx}.layerNameToVariables[l]$
  for $idx = 0$ to $|Lvars|$ **do**
    $i = Lvars[idx]$
    $F_{onx}.setLowerBound(i, V_{min}[idx])$
    $F_{onx}.setUpperBound(i, V_{max}[idx])$
  end for
  return $None$
end function

function MB_SOLVE($labs$,F_{onx},t)
  $Ovars = F_{onx}.outputVars$
  $labvar = Ovars[c]$
  for $idx = 0$ to $|Ovars|$ **do**
    $v = Ovars[idx]$
    if $idx = c$ **then**
      continue
    end if
    $F_{onx}.addConstraint(labvar > v)$
    $res, vals, stats =$
    $F_{onx}.solve(timeout = t)$
    if res = 'sat' or 'TIMEOUT' **then**
      return $False$, $labs$
    else $labs \leftarrow labs - \{idx\}$
    end if
  end for
  return $True, \emptyset$
end function

Each rule can be formally verified by the **Prove** module. Given a label, c, the rule with the highest support (σ_s^c) is selected; $\sigma_s^c = \arg\max_{\sigma_i^c} \text{supp}(\sigma_i^c)$, l refers to the corresponding layer. This rule has the highest chance of satisfying the verification check; $\forall x : \sigma(F_l(x)) \Rightarrow P(F(x))$. In order to characterize the region satisfying the pre-condition, over-approximation boxes encompassing the inputs in σ_s^c are computed in the input space (dimensionality I) and in the latent space (dimensionality L) respectively.

$$X_{\text{sup}} = \{x \in D \mid \sigma_s^c(x)\}, X^s_{\min} = \left[\min_{x \in X_{\text{sup}}} x_i\right]_{i=0}^{I-1}, X^s_{\max} = \left[\max_{x \in X_{\text{sup}}} x_i\right]_{i=0}^{I-1},$$

$$V^s_{\min} = \left[\min_{x \in X_{\text{sup}}} F_l(x)\right]_{i=0}^{L-1}, V^s_{\max} = \left[\max_{x \in X_{\text{sup}}} F_l(x)\right]_{i=0}^{L-1}$$

Let us consider the prediction post-condition of typical classification models, $P(F(X)) ::= (F(X) = c)$. Algorithm 2 shows the algorithm invoking Marabou [15] solver to perform the check [2]. The solver takes in models in the ONNX format and the algorithm invokes the methods in the *MarabouNetworkONNX* [19] API.

In the first iteration, the network input and layer variables are bound to $X^s_{\min}$, $X^s_{\max}$, $V^s_{\min}$, $V^s_{\max}$ respectively. The function, MB_SOLVE, is called to check the output constraints $\forall c' \neq c\ (F_{logits}(x)[c] > F_{logits}(x)[c'])$. The output variables are constrained to represent the negation of P; (i.e.) for every $c' \neq c$ following query is formulated, $\exists x \mid (F_{logits}(x)[c'] > F_{logits}(x)[c])$. The solver is invoked for the query corresponding to each label in *labs* (which contains all $c' \neq c$ in the first iteration). A SAT solution indicates a counter-example; the presence of an input which satisfies σ for which the network produces a different label (c'). An UNSAT indicates proof for the query w.r.t. the specific label, c', and the label is removed from *labs*. The entire proof goes through ($result = True$) when the solver returns UNSAT for all labels in *labs*. In the case of SAT solutions for some queries, or a timeout, the result is set to $False$ and the updated *labs* set is used in the next iteration. Subsequent iterations further restrict first the layer variables and then the input variables to values corresponding to a single satisfying instance (V^0, X^0). This ensures that the last iteration is guaranteed to be an UNSAT, which guarantees algorithm termination. The algorithm complexity is linear in the order of the number of classes.

To summarize, the above algorithm first attempts to prove the rule on the entire region characterized by σ^c_s. If this step fails, it attempts to refine the region until the proof goes through. Note that that the proof may go through for a subset of queries, providing robustness w.r.t. mis-classification to specific labels (refer Section 2.3 for an example). For other post-condition properties including those corresponding to non-classification models, the tool can take in a constraints file with user-provided constraints (Section 5.3 in the appendix).

Layer	Neurons	Signature	Support	Label
dense_14	[14,23,2,17,20,21,9,1, 31,12,13,27,4,29,3]	[0,0,0,0,0,1,0,1,0,1,0,1,1,1,1]	917	2
dense_15	[1,0,7,6,3,2,8, 9,4,9,3,5]	[<=1237,<=888.6,<=1388.8,<=1197.4, <=1778.3,<=981.6,<=1232.2,<=2709.8, <=1742.2,<=2664,>-2222.5,>2202.6]	4332	5
dense_14	[2,7,4,7,1,1,5]	[<=5.02,<=5.23,>0.7,<=3.86 <=-1.43,<=-1.73,>,-3.6]	54	1000

Train Recall	Train F1	Test Precision	Test Recall	Test F1
15.66%	27.09	94.66%	18.44%	31.11
80.86%	89.42	100%	78.18%	87.75
50.29%	66.92	80%	30%	43.64

Table 1. Example rules and their metrics produced by the *analyze* command. Train Precision is 100% for all rules.

[2] Although the current implementation uses Marabou, it is extensible to other solvers.

2.3 Tool Usage

The tool can be found in the repository https://github.com/safednn-nasa/DNN-Prophecy. It is implemented in Python 3.10. Prophecy can be invoked using three main commands; (i) *analyze* , (ii) *prove*, and (iii) *monitor*. The *analyze* command enables the execution of the activations collection and labeling, learn, extract, and test modules, and the *prove* command enables the execution of the prove module. The *monitor* command enables the use of the extracted rules for the evaluation of the model on new inputs at inference time. Table 3 (in the appendix) lists Prophecy commands and their respective parameters. We illustrate below an example usage of the tool. We provide more examples, including application of the tool on a regression model in the appendix.

Prophecy on MNIST classification model: Let us consider a model trained on the standard MNIST dataset [20] to classify images into digits 0 through 9. We consider a simple model with two convolutional layers followed by max pooling, flattened into two fully connected dense layers ($dense\_14$ with 32 neurons, $dense\_15$ with 10 neurons). We would like to infer rules that imply classification to each unique output digit and provide formal guarantees[3].

Prophecy is first invoked using the *analyze* command as shown below[4]. The desired post-condition is $(F(X) = c) \wedge (F(X) = c_{ideal})$; i.e. we would like rules that characterize inputs the model correctly classifies to every digit. This is enabled by setting the $type = 2$. The standard train datasets are used to obtain the input images and ground-truth labels to learn the rules.

```
$ python -m prophecy.main analyze
   -m 'PROPHECY_PATH/dataset_models/cnn_max_mnist2.h5'
   -wd 'PROPHECY_PATH/results/mnist/rules/'
   -tx ./x_train.npy -ty ./y_train.npy -vx ./x_val.npy -vy ./y_val.npy
   -type 2 -acts True -layer_name 'dense_14'
```

The rules are in terms of the *(on/off)*) neuron activations at the dense layer, $dense\_14$. Note that using *odl* , *oal* will consider every dense and activation layer in the model respectively. The tree estimator and the rules extracted from it are stored in the working directory specified by *wd*. There are one or more rules for correct classification to each label and rules for incorrect classification where $c \neq c_{ideal}$ (label 1000) respectively. Each rule is statistically validated on the provided validation datasets. Table 1 shows example rules and their metrics on train and validation sets. The second and third rules are in terms of neuron values, generated by invoking *analyze* without setting *acts* to $True$.

The next invocation using the *prove* command (shown below) enables formal verification of the extracted rules.

```
$ python -m prophecy.main prove
   -wd 'PROPHECY_PATH/results/mnist/rules/'
   -mp '/usr/local/lib/python3.11/dist-packages/maraboupy/'
   -onx 'PROPHECY_PATH/dataset_models/cnn_max_mninst2.onnx'
```

[3] Refer /examples/Prophecy_Tool_MNIST.ipynb in the repository
[4] PROPHECY_PATH refers to the local path to the DNN-Prophecy repository.

```
-onx_map 'PROPHECY_PATH/dataset_models/MNIST_H5_ONNX_MAP.csv'
-tx ./x_train.npy -label 5 -pred True
```

This command invokes Marabou from the build folder provided in *mp* and uses the *onnx* version of the CNN model. It requires an *onx_map* which maps the layer names in the keras and *onnx* versions of the model. This example command verifies a rule for *label* 5. The rule with the highest train support (σ_s^5) is selected (across all layers). The images in *x_train* dataset are used to determine X_{sup}, which in turn helps constrain the input and layer variables during verification. The parameter *pred* when set to *True* acts as a short-cut to enable checking classification constraints on the model output; (i.e.) output node 5 has the maximum value for all inputs satisfying σ_s^5.

The second example rule in Table 1 for label 5 gets proved in 15 seconds in the first iteration;

$\forall x \in [X_{\min}, X_{\max}],\ F_l(x) \in [V_{\min}, V_{\max}]$:

$\forall c' \in \{0,1,2,3,4,6,7,8,9\} : (F_{logits}(x)[c'] > F_{logits}(x)[c])$

For the first example rule for label 2, the proof goes through w.r.t. six labels (4 thru 9) in the first iteration, while the solver times out for queries corresponding to the remaining labels. The entire process takes 20 minutes.

3 Applications of Prophecy

Robustness Proofs. Provably-correct properties (at some layer l) defined with respect to some output class characterize regions in the input space in which the network is guaranteed to give the same class, i.e. the network is *robust* [11]. Counterexamples generated from failed proofs represent potential adversarial examples, as they are close (in the representation space at layer l) to (regions of) inputs that are classified differently.

Compositional Verification. Deep networks deployed in safety-critical contexts, such as ACAS-Xu [11], have properties of the form $A \Rightarrow B$, stating that for all inputs x satisfying $A(x)$, the corresponding output $y\ (= F(x))$ satisfies $B(y)$. Proving such properties for multilayer feed-forward networks is computationally expensive. The Prophecy-inferred patterns can be used to decompose proofs of such properties. Given a layer pattern σ at some middle layer l, the proof of $A \Rightarrow B$ can be decomposed into $A \Rightarrow \sigma$ and $\sigma \Rightarrow B$. We first identify a layer pattern σ that implies the output property B, and then prove $A \Rightarrow \sigma$ on the smaller network up to layer l. Furthermore, once a layer pattern σ is identified for a property B, it can be reused to prove other properties involving B.

Formal Explanations. Prophecy can also be used to infer *formal explanations* of DNN behavior [11]. 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 a layer rule σ. The problem is to find a minimal subset of the assignments such that $x_{k_1} = v_{k_1} \wedge x_{k_2} = v_{k_2} \wedge .. \wedge x_{k_m} = v_{k_m} \Rightarrow \sigma$. We adopt a greedy approach that eliminates constraints iteratively and stops when σ is no longer implied; the checks are performed with a decision procedure such as Marabou. The resulting constraints are network preconditions that formally guarantee the postcondition corresponding to σ.

DNN Repair. NNrepair [28] is a constraint-based technique for repairing DNNs. The framework can fix the network at an intermediate layer or at the last layer. NNrepair uses Prophecy to infer correctness specifications to act as an oracle for repair. NNrepair identifies a set of suspicious neurons and incoming suspicious edges. It adds δ values that are 0 in the concrete mode but symbolic in symbolic mode. NNrepair executes the DNN concolically along positive and negative examples. It then collects the values of suspicious neurons/edges in terms of symbolic expressions. NNrepair then adds a set of repair constraints which are then solved using an off-the-shelf solver.
Feature-guided Analysis of Neural Networks. In [12] we describe how to employ Prophecy to extract high-level, human-understandable features from the neural network internal representation, based on monitoring the neural network activations, and using high-level features instead of classes as postconditions. The extracted rules can serve as a link to high-level requirements and can be leveraged to enable various software engineering activities, such as automated testing, debugging, requirements analysis, and formal verification, leading to better engineering of neural networks.
Run-time Monitoring. Prophecy can also be used to determine reliability of DNNs in deployment [22]. In this work we explore the use of Prophecy rules in predicting, at runtime, whether the output of a network is correct or not. We use a set of passing and failing data to mine rules for correct and incorrect classification. At inference time, Prophecy evaluates a new, unseen input against the set of extracted rules. Thus, Prophecy can be used to predict if the output of the DNN is correct or not in deployment, benefiting safety-critical applications.
Mitigation against Poison Attacks. AntidoteRT [29] is another run-time approach that leverages Prophecy, this time to protect DNNs against poison or backdoor attacks. AntidoteRT uses Prophecy to extract neuron patterns by running a DNN on clean and poisoned samples. These patterns are used at run time, to detect whether unseen inputs are likely poisoned or not. The patterns are also used as oracles to guess the ideal label for the detected poisoned inputs.
Rule-Based Test Coverage. Prophecy is also useful in improving the testing of DNNs [30]. The goal is to use the rules to automatically generate ground truth for unlabeled data and to provide a new coverage metric which can evaluate existing test suites in terms of their functional diversity, defect detection ability, and sensitivity to different input scenarios.

4 Tool Evaluation

This section describes and evaluates (i) a new feature that optimizes runtime monitoring, and (ii) a novel application of the tool on a large vision-language model.

Evaluation of Prophecy for Runtime Monitoring. Prophecy provides functionality for runtime monitoring of the extracted properties [22] as outlined in Section 5.4. The *monitor* command is used at inference time to tag the model's

behavior on new inputs as *correct*, *incorrect* or *uncertain*. In the previous implementation of Prophecy [11], after the inference of rules via decision-tree learning, the extract module was executed by default, which parsed the tree to create and store a set of rules. We have extended the implementation to have the decision-tree estimator stored as well in pickle format. Invoking the *monitor* command using the *classifiers* parameter, loads the estimator and uses it to directly *predict* the model's behavior on the runtime inputs. This is much more efficient than a search through all the rules and therefore speeds-up the process.

The command when invoked with the *rules* parameter performs a linear search on the dictionary of rules (simulating original Prophecy implementation). Figure 3 compares the application of the *monitor* command with *rules* vs. with *classifiers* . It highlights that using the classifier directly provides considerable savings in inference times (69.59% on average) specifically for image models. Refer Section 5.4 in the appendix for more details.

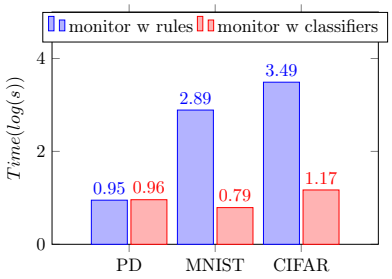

Fig. 3. Inference times comparison. Pima Diabetes (PD [2])), MNIST [20], CIFAR [27].

Evaluation of Prophecy on Large Models. Recent work [26] indicates that there are certain activations (*massive activations*) in large language models (LLMs) and vision language models (VLMs) that exhibit significantly higher values compared to other activations in the model. These activations are closely related to self-attention, causing the attention mechanism in these models to focus on the tokens associated with them. This motivated us to investigate the application of Prophecy to VLMs and see if Prophecy can scale and infer rules from a VLM. VLMs such as CLIP [25] have a more complex architecture than traditional CNNs; with separate encoders for images and text respectively. They are trained on image-text pairs and can be used to select the caption (from a set of captions) that has the highest similarity to a given image. This strategy can be leveraged for *zero-shot classification* of images [25]. We considered the OpenAI-ViT-Large-Patch14[5] CLIP model and a dataset of 5173 images (and respective ground-truth labels) of *trucks* and *cars* from the RIVAL-10 dataset. The zero-shot classification accuracy of CLIP on this data set was 98. 12% (car: 98.46, truck: 97.77).

Our goal was to extract rules for correct and incorrect zero-shot classification. We considered *LayerNorm2* of the ViT-L/14 vision transformer, which appears after the Multi-Head Self-Attention and before the MLP in every block, and has a dimensionality of (257 X 1024)[6]. We considered the 18^{th} layer of the encoder (as in the previous study on massive activations [26]). We invoked the *analyze* command of Prophecy with *inptype* = 1, which allows for activations vectors to

[5] https://huggingface.co/openai/clip-vit-large-patch14
[6] for a 224×224 image with 14×14 patch size

Label	Neuron	Sign.	Supp.	Train Prec	Train Recall	Train F1	Test Prec	Test Recall	Test F1	Avg F1
0	139445 100732 72927	≤-1.788 >0.415 >-1.138	21	100.00	43.75	60.87	36.36	8.16	13.33	37.10
1	139445 52507 75351 88245 135012 174394	>-1.788 ≤3.876 ≤4.957 >-4.015 ≤6.902 >-3.071	277	100.00	92.33	96.01	66.99	69.00	67.98	82.00

Table 2. Rules for correct (1) and incorrect (0) zero-shot classification by CLIP.

be directly provided as input (skipping the need to execute the model to collect the activations). This facilitates the application of the tool to analyze different model architectures. We considered 400 correctly classified and 97 mis-classified images from the original dataset. We obtained the neuron activations (external to Prophecy) using a *forward hook*; a 263,168 sized activation vector for each input. The activations were extracted within 262 seconds. A labels dataset was created (external to Prophecy) with binary labels corresponding to correct and incorrect zero-shot classifications respectively. Prophecy was then invoked to directly extract rules from the activations and labels datasets.

```
$ python -m prophecy.main analyze -type 3 -inptype 1
   -tx features_train.npy -ty labs_train.npy
   -vx features_test.npy -vy labs_test.npy
```

The rules in Table 2 (extracted in 81 seconds) show potential in capturing the logic that impacts VLM behavior. We explored another method to potentially improve the quality of the rules (Appendix 5.5). This paves way to enabling different types of applications and analyses (Section 3) to VLMs.

5 Conclusion

We presented Prophecy, a tool for inferring properties of DNN models. We described its applications and also reported on our initial investigation of Prophecy in the context of large (vision-language) models. In future work, we plan to explore in depth the many applications of Prophecy (as outlined at the beginning of Section 3) but in the context of large language and vision-language models.

Acknowledgments

We would like to thank our many collaborators on the Prophecy tool, in particular, Eduard Pinconschi, Andrew Wu, Ankur Taly, Hayes Converse, Huafeng Yu, Guy Katz, Burak Kadron, and Ravi Mangal.

References

1. https://scikit-learn.org/stable/
2. Pima Indians Diabetes Database — kaggle.com. https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database, [Accessed 31-01-2025]
3. Alain, G., Bengio, Y.: Understanding intermediate layers using linear classifier probes. CoRR **abs/1610.01644** (2016), http://arxiv.org/abs/1610.01644
4. Aytekin, Ç.: Neural networks are decision trees. CoRR **abs/2210.05189** (2022). https://doi.org/10.48550/ARXIV.2210.05189, https://doi.org/10.48550/arXiv.2210.05189
5. Balestriero, R.: Neural decision trees. CoRR **abs/1702.07360** (2017), http://arxiv.org/abs/1702.07360
6. Belinkov, Y.: Probing classifiers: Promises, shortcomings, and alternatives. CoRR **abs/2102.12452** (2021), https://arxiv.org/abs/2102.12452
7. Boz, O.: Converting A trained neural network to a decision tree dectext - decision tree extractor. In: Wani, M.A., Arabnia, H.R., Cios, K.J., Hafeez, K., Kendall, G. (eds.) Proceedings of the 2002 International Conference on Machine Learning and Applications - ICMLA 2002, June 24-27, 2002, Las Vegas, Nevada, USA. pp. 110–116. CSREA Press (2002)
8. Elazar, Y., Ravfogel, S., Jacovi, A., Goldberg, Y.: Amnesic probing: Behavioral explanation with amnesic counterfactuals. Trans. Assoc. Comput. Linguistics **9**, 160–175 (2021). https://doi.org/10.1162/TACL_A_00359, https://doi.org/10.1162/tacl_a_00359
9. Ettinger, A., Elgohary, A., Resnik, P.: Probing for semantic evidence of composition by means of simple classification tasks. In: Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP, RepEval@ACL 2016, Berlin, Germany, August 2016. pp. 134–139. Association for Computational Linguistics (2016). https://doi.org/10.18653/V1/W16-2524, https://doi.org/10.18653/v1/W16-2524
10. Frosst, N., Hinton, G.E.: Distilling a neural network into a soft decision tree. CoRR **abs/1711.09784** (2017), http://arxiv.org/abs/1711.09784
11. Gopinath, D., Converse, H., Pasareanu, C., Taly, A.: Property inference for deep neural networks. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 797–809 (2019). https://doi.org/10.1109/ASE.2019.00079
12. Gopinath, D., Lungeanu, L., Mangal, R., Pasareanu, C.S., Xie, S., Yu, H.: Feature-guided analysis of neural networks. In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 133–142. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_7, https://doi.org/10.1007/978-3-031-30826-0_7
13. Ivanova, A., Hewitt, J., Zaslavsky, N.: Probing artificial neural networks: insights from neuroscience (04 2021). https://doi.org/10.48550/arXiv.2104.08197
14. Julian, K., Lopez, J., Brush, J., Owen, M., Kochenderfer, M.: Policy compression for aircraft collision avoidance systems. In: Proc. 35th Digital Avionics System Conf. (DASC). pp. 1–10 (2016)

15. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.: The marabou framework for verification and analysis of deep neural networks. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. pp. 443–452. Springer International Publishing, Cham (2019)
16. Kim, B., Wattenberg, M., Gilmer, J., Cai, C.J., Wexler, J., Viégas, F.B., Sayres, R.: Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (TCAV). In: Dy, J.G., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. Proceedings of Machine Learning Research, vol. 80, pp. 2673–2682. PMLR (2018), http://proceedings.mlr.press/v80/kim18d.html
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012)
18. Li, K., Hopkins, A.K., Bau, D., Viégas, F.B., Pfister, H., Wattenberg, M.: Emergent world representations: Exploring a sequence model trained on a synthetic task. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), https://openreview.net/forum?id=DeG07_TcZvT
19. Marabou repository, https://github.com/NeuralNetworkVerification/Marabou
20. The MNIST database of handwritten digits Home Page, http://yann.lecun.com/exdb/mnist/
21. Olah, C., Schubert, L., Mordvintsev, A.: Feature visualization. Distill (2017), https://distill.pub/2017/feature-visualization/
22. Pinconschi, E., Gopinath, D., Abreu, R., Pasareanu, C.S.: Evaluating deep neural networks in deployment: A comparative study (replicability study). In: Christakis, M., Pradel, M. (eds.) Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024. pp. 1300–1311. ACM (2024). https://doi.org/10.1145/3650212.3680401, https://doi.org/10.1145/3650212.3680401
23. Quinlan, J.R.: Induction of decision trees. Machine Learning **1**(1), 81–106 (1986). https://doi.org/10.1023/A:1022643204877
24. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Mateo, CA (1993)
25. Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., Sutskever, I.: Learning transferable visual models from natural language supervision. In: Meila, M., Zhang, T. (eds.) Proceedings of the 38th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 139, pp. 8748–8763. PMLR (18–24 Jul 2021), https://proceedings.mlr.press/v139/radford21a.html
26. Sun, M., Chen, X., Kolter, J.Z., Liu, Z.: Massive activations in large language models (2024), https://arxiv.org/abs/2402.17762
27. Team, K.: Keras documentation: CIFAR10 small images classification dataset — keras.io. https://keras.io/api/datasets/cifar10/, [Accessed 31-01-2025]
28. Usman, M., Gopinath, D., Sun, Y., Noller, Y., Păsăreanu, C.S.: Nnrepair: Constraint-based repair of neural network classifiers. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I. p. 3–25. Springer-Verlag, Berlin, Heidelberg (2021)

29. Usman, M., Gopinath, D., Sun, Y., Păsăreanu, C.S.: Rule-based runtime mitigation against poison attacks on neural networks. In: Dang, T., Stolz, V. (eds.) Runtime Verification. pp. 67–84. Springer International Publishing, Cham (2022)
30. Usman, M., Sun, Y., Gopinath, D., Păsăreanu, C.S.: Rule-based testing of neural networks. In: Proceedings of the 1st International Workshop on Dependability and Trustworthiness of Safety-Critical Systems with Machine Learned Components. p. 1–5. SE4SafeML 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3617574.3622747, https://doi.org/10.1145/3617574.3622747
31. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks (2013), https://arxiv.org/abs/1311.2901

Appendix

5.1 Tool Commands and Parameters.

Command	Parameters	Description
ALL	*-m*	Pre-trained model in keras (.h5) format.
	-wd	Working directory path
analyze	*-tx,-ty*	Data and labels (.npy) ({x,y} in Algorithm 1).
	-vx, -vy	Data and labels (.npy) for statistical validation.
	-odl *-oal* *-layer_ name*	Layer/s to be used for activations collection. *-odl*: only dense layers (name starting with text 'dense'). *-oal*: include the activation layers associated with dense layers. *-layer_ name* (a specific layer).
	-inptype	Type of input data. (provided in *-tx* and *-vx*). *0*: inputs to the model (eg. images). *1*: array of neuron activations.
	-type	Short-cut for post-condition properties. *0*: rules w.r.t model output, eg. rules for every predicted label. *1*: rules for correct vs incorrect classification. *2*: rules for correct classification per label and incorrect classification. *3*: rules w.r.t labels in *-ty*
	-acts	Mathematical form of the rules. *True*: in terms of *on/off* neuron activations. In terms of neuron values if this parameter is not set to *True*.
	-top	Number of rules. *True*: only rules with the highest train recall to be obtained. All rules are obtained otherwise.
	-sr	Option to skip rule extraction and statistical validation. The decision-tree estimators are stored as is. Saves memory space.
	-b	Balance classes in the train data.
	-c	Consider model output confidence during rule extraction.
	-rs	Set the random state for reproducibility.
prove	*-mp*	Path to the Marabou build folder.
	-onx	ONNX model name corresponding to the keras model. Marabou API takes models in .nnet or .onnx format. Prophecy uses the ONNX API, the more widely used one.
	-onx_ map	Maps layer names in keras and onnx models. If not specified, Prophecy assumes the names to be the same. The layer corresponding to the highest support is chosen.
	-label	The label whose rules are to be considered for proving.
	-pred	True: Classification output constraints. False: Non-classification output constraints.
	-min_ const	Classification output constraints specifying that *label* has the minimum value.
	-cp consts_file	Path to a file specifying the output constraints (ex. regression outputs).
	-tx *-tx, -vx*	data used to constrain network input variables. in-distribution data to calculate coverage of regions with proofs.
monitor	*-tx, -ty*	Represents the unseen data and their ground-truth labels.
	-rules	Use the dictionary of rules to predict classes on unseen data.
	-classifiers	Use the tree estimator to predict classes on unseen data.

Table 3. Prophecy Commands and Parameters.

5.2 Prophecy on the ACASXU classification model:

ACASXU is a family of networks for collision avoidance system for unmanned aircraft control [14]. The model 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 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 used consists of 6 hidden layers, and 50 ReLU activation nodes per layer. The train dataset consists of 288165 inputs and the test dataset consist of 96056 inputs with corresponding known output labels.

We would like to use Prophecy[7] to extract rules leading to each unique advisory (such as COC, Weak Left so on). The model does not predict a single class but five values; scores corresponding to each individual advisory. Therefore, a labels dataset is generated externally that contains for each input the respective class (horizontal advisory with minimum score). The command shown below invokes *analyze* command with $type = 3$ and the externally generated labels dataset. The rules are extracted after the layer *relu_3* in terms of the neuron values. Only the rule with the highest support for each label are stored in *ruleset.csv*.

```
$ python -m prophecy.main analyze
   -m 'PROPHECY_PATH/dataset_models/acasx_onnx_keras.h5'
   -wd 'PROPHECY_PATH/results/acasx/rules/'
   -tx ./x_train_rshape.npy -ty ./y_predict_npy.npy
   -vx ./x_test_rshape.npy -vy ./y_predict_test_npy.npy
   -type 3 -layer_name 'relu_3' -top True
```

The *prove* command is invoked to verify the rule for label 0 (COC) with $pred = True$ and $min\_const = True$. This invokes Marabou with the following query for every $c' \neq 0$, $\exists x \mid (F_{op}(x)[c'] < F_{op}(x)[0])$. The rule is Proved when all the above queries return UNSAT.

```
$ python -m prophecy.main prove
   -m 'PROPHECY_PATH/dataset_models/acasxu/acasx_onnx_keras.h5'
   -wd 'PROPHECY_PATH/results/results/acasxu/rules/'
   -tx ./x_train.npy -label 0 -pred True -min_const True
   -mp '/usr/local/lib/python3.11/dist-packages/maraboupy/'
   -onx 'PROPHECY_PATH/dataset_models/
      acasxu/ACASXU_experimental_v2a_1_1.onnx'
   -onx_map 'PROPHECY_PATH/dataset_models/
      acasxu/acasx_onnx_map.csv'
```

5.3 Prophecy on a regression model:

We consider a regression model which takes in the following sensor inputs (i) Frequency, (ii) Airfoil angle of attack, (iii) Chord length, (iv) Free-stream velocity

[7] /examples/ACASX_ProphecyTool.ipynb in the repository

and (v) Suction side displacement thickness to estimate scaled sound pressure level. The model has a Mean Absolute Error (MAE) of 0.192 on the train set.

Prophecy is used to infer rules corresponding to correct and incorrect model behavior, based on the post-condition property, $P(F(x)) := |F(x) - output_{ideal}| < MAE(0.192)$. As mentioned in Section 2.2, Prophecy currently handles post-condition properties not corresponding to prediction of classification models by setting the *type* parameter to 3 and expecting the input labels dataset to directly contain the labels produced by the evaluation of P on each input. Accordingly, the property is evaluated (prior to the invocation of Prophecy) on each train (and test) input to produce binary labels (1: satisfaction, 0: violation). This generates the labels datasets $y\_train.npy$ and $y\_test.npy$ respectively. The analyze command is then invoked with $type = 3$ to extract rules corresponding to the binary labels in these datasets. The rules are conditions on neuron values of the specified layer.

```
$ python -m prophecy.main analyze
   -m 'PROPHECY_PATH/dataset_models/airfoil_self_noise/model.h5'
   -wd 'PROPHECY_PATH/results/airfoil_self_noise/rules/'
   -tx ./x_train.npy -ty ./y_train.npy
   -vx ./x_test.npy -vy ./y_test.npy
   -layer_name 'layer_4_output_0' -type 3
```

The rules corresponding to label 1 represent inputs on which the model displays correct behavior. In order to check if the property $P(F(X))$ is satisfied for all inputs satisfying σ, we would need to know the $output_{ideal}$ for every possible input in this region, which is not available. Therefore, we used a modified version of the post-condition property, $P'(F(X)) := ((min - 0.192) < F(X) < (max + 0.192))$, where min and max correspond to the minimum and maximum output values over all inputs in the train set satisfying σ. In other words, we try to check if the outputs for all inputs in the region corresponding to σ fall within a desirable valid output region.

As mentioned in Section 2.2, for post-condition properties not corresponding to prediction of classification models, the *prove* command can take in a constraints file whose path is specified in the *cp* parameter. Each row in this file should adhere to the format $[output\_node\#, operator, value]$. Currently the implementation supports operators MIN and MAX. For this example, $P'(F(X))$ is encoded using the following two rows, where 0 is the index of the output node of the network.

$$[0, MIN, 0.192]$$

$$[0, MAX, 0.192]$$

```
$ python -m prophecy.main prove
   -onx 'PROPHECY_PATH/dataset_models/
      airfoil_self_noise/renamed_model.onnx'
   -onx_map 'PROPHECY_PATH/dataset_models/
      airfoil_self_noise/airfoil_onnx_map.csv'
   -mp '/usr/local/lib/python3.11/dist-packages/maraboupy/'
```

```
   -wd 'PROPHECY_PATH/results/airfoil_self_noise/rules/'
   -tx ./x_train.npy -label 1
   -cp 'PROPHECY_PATH/dataset_models/airfoil_self_noise/rules/consts.csv'
```

5.4 Monitoring runtime inputs:

Prophecy is used to extract rules for correct vs incorrect classifications for models processing tabular data (Pima Diabetes (PD [2])) and images (MNIST [20], CIFAR [27]). The command below shows the use of $-odl$ and $-oal$ parameters which extract rules after every dense and corresponding activation layers.

```
$ python -m prophecy.main analyze
   -m 'PROPHECY_PATH/dataset_models/mnist_cnn.h5'
   -wd 'PROPHECY_PATH/results/mnist/rules/'
   -tx ./x_train.npy -ty ./y_train.npy
   -vx ./x_test.npy -vy ./y_test.npy
   -odl -oal -type 1
```

The *monitor* command is used at inference time to predict the model behavior on unseen inputs. For inputs that satisfy a larger number of rules for correct classification (vs incorrect) across layers, an output of *correct* is produced, and an output of *incorrect* is produced for those inputs satisfying more rules for incorrect classification. When there is a tie, an *uncertain* output is produced.

The original implementation of Prophecy executed the Extract module by default and stored the extracted rules in the form of a dictionary. However, searching through such a dictionary for all rules is expensive, especially for a runtime monitoring application. The *rules* parameter (feature of ProphecyPlus) uses this implementation.

```
$ python -m prophecy.main monitor
   -m 'PROPHECY_PATH/dataset_models/mnist_cnn.h5'
   -wd 'PROPHECY_PATH/results/mnist/rules/'
   -tx ./x_val.npy -ty ./y_val.npy rules
```

In the new implementation, ProphecyPlus, the decision-tree estimators are stored as is (the Extract step can also be skipped by setting the *sr* flag). Invocation of the monitor command using the *classifiers* parameter accesses the tree directly to predict the class for a runtime input.

```
$ python -m prophecy.main monitor
   -m 'PROPHECY_PATH/dataset_models/mnist_cnn.h5'
   -wd 'PROPHECY_PATH/results/mnist/rules/'
   -tx ./x_val.npy -ty ./y_val.npy classifiers
```

5.5 Prophecy on Clip - VLM Results

In another experiment, we consider 4 cases. Case 1: images of *car* misclassified, Case 2: images of *car* correctly classified, Case 3: images of *truck* correctly classified, Case 4: images of *truck* misclassified. We assign labels *0,1,2,3* to each of them respectively. Table 4 shows detailed results of the experiments.

Case	Neuron	Sign.	Supp.	Cls.	Train Cov	Train Prec	Train Recall	Train F1	Test Cov	Test Prec	Test Recall	Test F1	Avg F1
1	396 188122 157846 203801 239132	≤ -2.909 ≤ 6.002 ≤ 4.471 >2.509 >1.152	5	0	1.44	100.00	25.00	40.00	3.36	20.00	4.76	7.69	23.85
2	396 188122 157846 203801 88270	≤ -2.909 ≤ 6.002 ≤ 4.471 ≤ 2.509 ≤ 4.828	114	1	32.76	100.00	76.00	86.36	29.53	59.09	52.00	55.32	70.84
3	396, 33 36996 154735 126911 92660	>-2.909 ≤-0.350 >-4.715 ≤3.052 ≤4.830 ≤2.879	117	2	33.62	100.00	78.00	87.64	30.87	58.70	54.00	56.25	71.95
4	396, 33 219348 4638 153077	>-2.909 >-0.350 >0.846 >-2.466 ≤1.175	10	3	2.87	100.00	35.71	52.63	4.70	42.86	10.71	17.14	34.89

Table 4. Results for Running Prophecy on CLIP.