SOCRATES: Towards a Unified Platform for Neural Network Verification

Long H. Pham, Jiaying Li and Jun Sun
Singapore Management University

***Abstract*—Studies show that neural networks, not unlike traditional programs, are subject to bugs, e.g., adversarial samples that cause classification errors and discriminatory instances that demonstrate the lack of fairness. Given that neural networks are increasingly applied in critical applications (e.g., self-driving cars, face recognition systems and personal credit rating systems), it is desirable that systematic methods are developed to verify or falsify neural networks against desirable properties. Recently, a number of approaches have been developed to verify neural networks. These efforts are however scattered (i.e., each approach tackles some restricted classes of neural networks against certain particular properties), incomparable (i.e., each approach has its own assumptions and input format) and thus hard to apply, reuse or extend. In this project, we aim to build a unified framework for developing verification techniques for neural networks. Towards this goal, we develop a platform called SOCRATES which supports a standardized format for a variety of neural network models, an assertion language for property specification as well as two novel algorithms for verifying or falsifying neural network models. SOCRATES is extensible and thus existing approaches can be easily integrated. Experiment results show that our platform offers better or comparable performance to state-of-the-art approaches. More importantly, it provides a platform for synergistic research on neural network verification.**

I. INTRODUCTION

Neural network models are getting ever more popular due to their exceptional performance in solving many real-world problems, such as self-driving cars [1], face recognition [2], malware detection [3], sentiment analysis [4] and machine translation [5]. At the same time, neural networks are shown to be vulnerable to a variety of issues. For instance, it is shown that adversarial perturbation can be applied to generate samples which trigger wrong model prediction [6], [7], [8]; and it is shown that neural network models may discriminate certain groups or individuals [9]. Given that neural networks are increasingly applied in applications which are safety-critical (e.g., self-driving cars) or have significant societal impact (e.g., personal credit rating systems or face recognition), it is desirable that such neural network models are systematically verified against a variety of desirable properties. For instance, an image recognition neural network model used in a self-driving car should be verified to be robust (i.e., the classification result remains the same in the presence of perturbation) and a neural network for predicting personal credit rating should be verified to be fair.

Recently, there has been an increasing number of efforts on formally verifying neural network models. In [10], Katz *et al.* proposed a constraint solving technique targeting feedforward neural networks with ReLU activation functions. In [11], [12], Wang *et al.* improved the constraint solving techniques for verifying the same class of neural networks with symbolic intervals. In [13], [14], [15], the authors applied the classic abstract interpretation techniques to verify neural networks, with customized abstraction domains and functions supporting feedforward neural networks with activation functions such as ReLU, Sigmoid and Tanh.

The status quo is however less than satisfactory, i.e., existing approaches are limited in multiple ways, which makes applying, comparing, reusing and extending existing verification efforts difficult. First, existing approaches support only restricted classes of neural networks or properties. For instance, some existing work [10], [12] only support verification of feedforward neural networks using ReLU activation functions. Only very recently, researchers have started exploring the verification of feedforward neural networks with different activation functions [14], [15] and some subclasses of recurrent neural networks [16], [17]. Furthermore, existing approaches all focus on reachability properties or local robustness, and ignores the verification of other important properties such as fairness and beyond. Secondly, existing verification toolkits require input models in specific format and different tools often require different format. For instance, Reluplex [10] requires a text file contains the weights and bias of multilayer perceptron layers, whereas DeepPoly [15] needs a more complex input which specifies the types of layers before providing their parameters' values. This will likely get worse as there are an increasing list of popular frameworks for training neural network models, such as TensorFlow, Cafe, MXNet, PyTorch, Theano and Keras, all of which encode neural network models in their own ways. As a result, a verification tool developed for one framework may not be applicable to models trained using another. This not only limits the applicability of the existing verification toolkits but also makes comparing them infeasible. We remark that two recent efforts on solving this problem are ONNX [18] and NNEF [19], which aims to provide a cross-platform format for neural networks. It is however designed for a different purpose and lacks important features which are required for neural network verification. Thirdly, each existing approach typically focuses on one property, whereas in fact the (sometimes rather sophisticated) verification algorithm could be easily extended to solve the verification problem of another property. For instance, algorithms for verifying local robustness can easily be extended to verify fairness defined in terms of individual discrimination. With different verification algorithms and optimization techniques implemented in differ-

ent repositories, reusing these efforts is often highly nontrivial.

In this project, we aim to build a unified framework for developing verification techniques for neural networks. The goal is to have a platform which allows us to apply, compare, reuse and develop verification techniques for a variety of neural network models against a variety of properties. Towards this goal, we design and implement an open source platform called SOCRATES, which embodies multiple technical contributions. First, SOCRATES provides a standardized format for a variety of neural network models based on JSON. By compiling models trained using different frameworks to this common format and building verification algorithms around it, the same verification algorithm can be applied to models trained using different frameworks. Secondly, SOCRATES supports an assertion language which is designed to specify a range of properties of neural network models, including robustness, fairness and more. Thirdly, SOCRATES provides two new algorithms, i.e., optimization-based falsification and statistical model checking, which can be applied to verify or falsify a variety of neural network models. More importantly, SOCRATES is designed to be modular and extensive, i.e., it is straightforward to support new models, properties or verification algorithms; or integrate existing verification engines.

Furthermore, we provide a comprehensive set of 12347 verification tasks (i.e., neural network models and respective assertions) as a part of the SOCRATES repository so that researchers can easily evaluate and compare the effectiveness and efficiency of different verification algorithms using a comprehensive set of benchmarks. Using the benchmarks, we conduct multiple experiments to evaluate the effectiveness of the two verification algorithms developed in SOCRATES. The experiment results show that the two new algorithms solve more verification tasks than existing approaches and offer complimentary verification results. We remark that SOCRATES is open source at [20] and we are making all the effort required to make it a platform for synergistic research on verification of neural networks.

The rest of the paper is organized as follows. In Section II, we discuss the overall design of SOCRATES. In Section III, we present details on the model format in JSON supported in SOCRATES. In Section IV, we present the two new algorithms supported in SOCRATES and evaluate their effectiveness against state-of-the-art approaches. In Section V, we review related work and we conclude in Section VI.

II. SYSTEM OVERVIEW

SOCRATES is designed for both ordinary users who require a tool for verifying a particular neural network model as well as researchers who are working on developing neural network verification techniques. In the following, we first illustrate how SOCRATES works from an ordinary user point of view and then introduce its design from a verification researcher point of view. Lastly, we provide an overview of functionalities provided by SOCRATES and those by existing verification toolkits for neural networks.

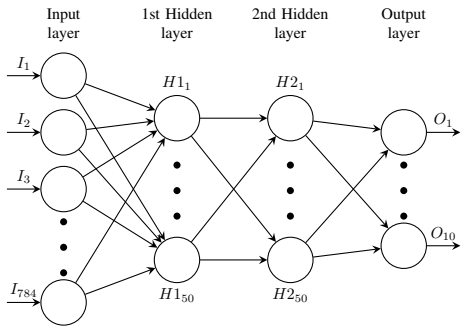

Fig. 1. An example neuron network

A. For Ordinary Users

To use SOCRATES to verify a well-trained neural network against a desirable property, a user must provide a JSON file which encodes the verification task in the required format. A verification task is composed of three main parts, i.e., a model, a property, and a verification engine selected for solving the verification task. The JSON format is designed to support a variety of neural network models. The model part of a JSON file can be generated automatically from models trained using existing frameworks such as Tensorflow and PyTorch. The property part of the JSON file is specified in an assertion language designed for neural network verification, with a formal syntax as well as supporting easy-to-use templates for commonly verified properties. Together with the model and the property are JSON keys for specifying the verification engine. Note that some verification engines have configurable parameters, which are specified as part of the JSON file as well. The readers are referred to Section III-A and III-B for details on the format. Once the JSON file is loaded, the user simply waits for the verification result.

In the following, we use an example to illustrate the process. Assume that the user has trained a network for classifying images with 1 channel and a dimension of 28×28 (i.e., 784 pixels in total) based on the MNIST dataset. The design of the neural network is shown in Figure 1. It is a multilayer perceptron with 4 layers. Beside the input layer, each of the two hidden layers has 50 neurons, and the output layer has 10 neurons. The activation function used in two hidden layers is the ReLU function, and the output layer uses the softmax function to return the probability of each class according to the input sample. The property to be verified is local robustness, i.e., given a particular input image, with a limit on the value of each pixel to be perturbed, all perturbed images have the same label as the original one.

The JSON file is shown in Figure 2. At line 2, a model is defined with the key *model*. The value of *model* is a JSON object. At line 3, the value of the key *shape* is specified as an integer tuple which represents the shape of the input sample. For this example, the value is $(1, 784)$ where 1 is the length of the sample and 784 is the size of each element in the sample. At line 4, the key *bounds* specifies the lower bound and the upper bound respectively for features in valid input

samples. In this example, the bound $[(0, 1)]$ means each of the features in a valid input (i.e., a feature vector) is a value v which satisfies $0 \leq v \leq 1$. Lines 5 to 24 then specify the value of the key *layers*, i.e., an array in which each element specifies the details of one layer. In this example, the array contains information of the two hidden layers and the output layer. Note that the details of the input layer are not necessary (since they have been specified using other keys). Each array element specifies the type of the layer, the value of the weights matrix, the values of the bias and the activation function. These information are defined using the keys *type*, *weights*, *bias* and *func* respectively. We remark that the values of *weights* and *bias* are omitted as they are rather complicated, e.g., the value of *weights* for the first hidden layer is a matrix of dimension 784×50 written as a string with a syntax similar to the right-hand side of Python multidimensional array initialization. Instead of providing these values directly, the user can provide addresses to the files (either local or online) containing their values. Note that the key values might be correlated and thus must be checked for well-formness, i.e., the first dimension of the weight matrix must be equal to the number of neurons in the previous layer; the second dimension of the weight matrix and the size of the bias vector must be equal to the number of neurons in the current layer.

Lines 26 to 31 then define the property to be verified. In this example, the property is local robustness which is specified with multiple keys. The key *x0* is an input image, i.e., a vector of size 784, or an address to an image. In this example, assume that the input image is the one shown on the left of Figure 3, i.e., an image from the MNIST dataset with label 7. The key *distance* specifies the searching space using one of the predefined functions in the framework. In this example, the distance function is the infinity norm distance, which intuitively means the maximum element-wise absolute difference between two samples. Finally, the key *eps* specifies the maximum value of the distance. We remark that in this example, a predefined template for local robustness is used. SOCRATES supports a general assertion language, which we present in Section III-B.

Lines 32 to 34 then specify the verification engine that is applied to solving the verification problem. In this example, the user selects an optimization-based falsifier. The falsifier automatically transforms the verification problem into an optimization problem, which is then solved using an optimization algorithm (see details in Section IV-A). An image which violates the specified property is identified in about 1 second, which is shown on the right of Figure 3. That is, this image has a distance from the original image not greater than the specified bound (i.e., 0.1 in this example) and has a label which is not 7. Note that the JSON file also contains keys (at lines 35-37) which are for presenting the verification result, e.g., by setting the *display* function and providing a *resolution*.

B. For Researchers on Neural Network Verification

More relevantly, SOCRATES is designed to enable synergistic effort on developing state-of-the-art verification techniques

```
1  {
2    "model": {
3      "shape": "(1,784)",
4      "bounds" : "[(0,1)]",
5      "layers": [
6        {
7          "type": "linear",
8          "weights": "a [784 x 50] matrix",
9          "bias": "a [50] vector",
10         "func": "ReLU"
11       },
12       {
13         "type": "linear",
14         "weights": "a [50 x 50] matrix",
15         "bias": "a [50] vector",
16         "func": "ReLU"
17       },
18       {
19         "type": "linear",
20         "weights": "a [50 x 10] matrix",
21         "bias": "a [10] vector",
22         "func": "softmax"
23       }
24     ]
25   },
26   "assert": {
27     "robustness": "local",
28     "x0": "a [784] vector",
29     "distance": "infinite norm",
30     "eps": "0.1"
31   },
32   "solver": {
33     "algorithm": "optimize"
34   },
35   "display": {
36     "resolution": "(28,28)"
37   }
38 }
```

Fig. 2. An example JSON input

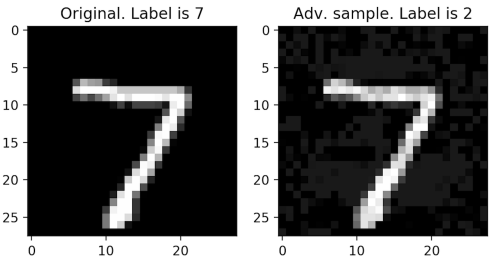

Fig. 3. Example analysis results

for neural networks. It has a publisher-subscriber architecture which facilitates developing verification techniques for different neural network models and properties independently. It has three main modules, i.e., the *Parser* and *Display* on the front-end and *Verifiers* on the back-end. In the following, we briefly discuss the main functionality of each component. The technical details are discussed in the subsequent sections.

The *Parser* module is built upon the JSON format (as exemplified in Figure 2). The parser receives a JSON file, checks its well-formness, decodes it and publishes it internally as a verification task. Note that each verification task is associated with a number of parameters (such as the network model

type, size and the assertion type), which are used to determine whether a verification engine is applicable or not. The models which are currently supported in SOCRATES include multilayer perceptron (MLP), convolutional networks (CNN), residual networks (ResNet), and recurrent networks (RNN). To support new models, the developers are required to extend the JSON format (by introducing new values for certain existing keys or introducing new keys) and extend the parser accordingly to generate the verification task, which is relatively easy.

The *Verifiers* module consists of a set of verification engines which could be developed independently from each other. Each verification engine could be either general or specific (i.e., dedicated to certain models or certain properties, such as many existing verification engines [14], [15]). For instance, SOCRATES supports two general verification engines which applies to all neural network models which are currently supported in SOCRATES. The first one is an optimization-based falsification engine, which transforms the verification task into the problem of finding a counterexample through optimization. The second one is a statistical model checking [21] engine which can be used to verify that the assertion holds with certain level of statistical confidence. As the verification engines are independent, extending SOCRATES with a new verification algorithm is straightforward.

The *Display* module is used to present the verification results in a user-friendly way to the user. A verification engine typically generates three kinds of results, i.e., the property is verified, no counterexample is identified (e.g., which timeout occurs) or a counterexample is identified. Depending on the application domain and the property, the counterexample could be an image, a text, a feature vector or a set of them (for instance, if the property to be verified is individual fairness, two contrasting feature vectors form a counterexample). The display module receives the result from the verification engines and displays them accordingly. Note that some additional keys are defined in the JSON format so that the user can specify the display options.

C. Functionalities

SOCRATES is designed to be a unified platform supporting a variety of neural network models, properties and verification algorithms. In the following, we compare SOCRATES with existing state-of-the-art approaches in terms of functionalities. There are recently a booming number verification engines for neural networks and thus it is hard to keep up with all of them. Our search of state-of-the-art tools are based on research papers recently published at top-tier conferences and it is possible that we might miss some of them. Furthermore, not all tools reported in the publications are available for evaluation (or reliable enough to be evaluated independently). The following are the tools that we gather and compare: Reluplex [10], Neurify [12], DeepZ [14], DeepPoly [15], RefineZono [22], RefinePoly [23], ADF [9], and C&W [8]. Note that the last two are technically not verification engines but rather testing engines. They are included as representatives of their kind to provide additional benchmark on the state-of-the-art for

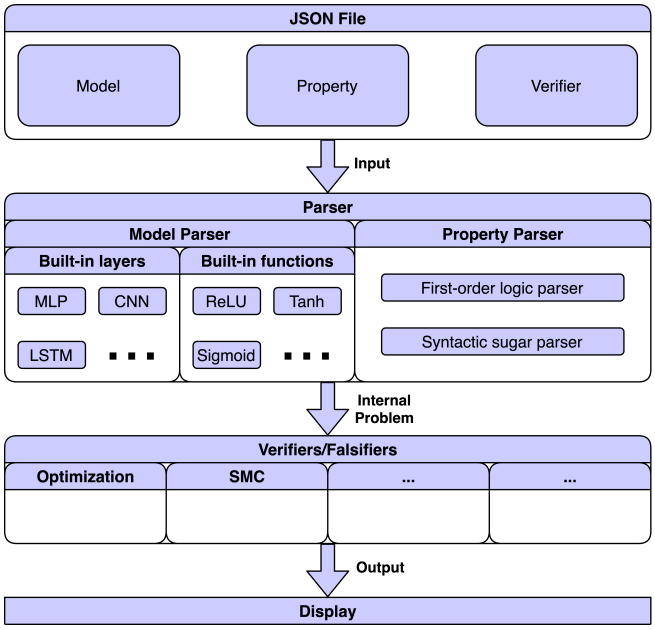

Fig. 4. The architecture of SOCRATES

falsification. Their results however should be taken with a grain of salt as they are designed for completely different purpose. To find out the functionalities of each tool, we check the papers which reported the tool and experiment the tool to find out whether new capabilities have been introduced recently. For each tool, we identify the kind of neural network models and the kind of properties that are supported.

The comparison is summarized in Table I, where ▲ means partial support (i.e., only works with small networks and may not be scallable to bigger ones). In terms of models supported by the tools, it can be observed that SOCRATES is the only tool which supports all types of neural networks. For Reluplex, experiments on verifying small multilayer perceptron networks have been reported. For Neurify, besides multilayer perceptron, it also supports convolutional networks. The next four tools, i.e., DeepZ, DeepPoly, RefineZono and RefinePoly, have the same capability, i.e., they support multilayer perceptron, convolutional, and residual networks. For ADF, the reported experiments only deal with small multilayer perceptron networks. All of the adversarial sample generation tools that C&W represents focus on the image recognition problem, and thus do not support recurrent networks.

The properties that each tool can handle are shown in the last column in Table I. We again observe that only SOCRATES supports all types of properties. Most of the other tools support the local robustness property. Reluplex reportedly supports verification of the global robustness property for small networks. Other tools except ADF do not support global robustness. Most of the tools can support reachability properties expressed using linear (in)equalities, execpt ADF and C&W. Besides ADF which supports falsification of fairness properties (through a combination of clustering and searching), fairness properties are only supported by SOCRATES.

Tools	Types of networks				Types of properties			
	MLP	CNN	ResNet	RNN	Local robust.	Global robust.	Lin. (in)equal.	Fairness
SOCRATES	★	★	★	★	★	★	★	★
Reluplex	▲	◯	◯	◯	★	▲	★	◯
Neurify	★	★	◯	◯	★	◯	★	◯
DeepZ	★	★	★	◯	★	◯	★	◯
DeepPoly	★	★	★	◯	★	◯	★	◯
RefineZono	★	★	★	◯	★	◯	★	◯
RefinePoly	★	★	★	◯	★	◯	★	◯
ADF*	▲	◯	◯	◯	★	★	◯	★
C&W*	★	★	★	◯	★	◯	◯	◯

TABLE I
COMPARISON OF SUPPORTED NETWORKS AND PROPERTIES. ★: FULL SUPPORT, ▲: PARTIAL SUPPORT, ◯: NO SUPPORT

D. Implementation

SOCRATES is open source at [20], including all the source code as well as a set of 12347 verification tasks. SOCRATES is implemented using Python 3 with a total of 2400 lines of code. Multiple public Python libraries are used in the implementation, including *json* and *ast* for parsing the JSON input file, *numpy* and *autograd.numpy* for mathematical computation, *matplotlib* for displaying imagery results, and *sicpy* for solving the optimization problem. By default, the local optimization function (instead of the global one) in *scipy* is applied, although this can be easily configured.

SOCRATES is designed with the extensiblity in mind. The types of network models supported by SOCRATES can be easily extended by introducing new types of layers and activation functions. Currently, each type of layer is implemented as an independent class. In each class, the most important function is $apply$, which compute the output vector from the input vector. To extend the capability of SOCRATES with new type of layer, a new class can simply added into the library. Similarly, all the supported activation functions are kept inside an utility class. The new functions can be easily added at any time. The new types of layers and activation functions may need new parameters, which should not be the problem because users can always define the new keys for the JSON input file and update the JSON parser to represent the necessary parameters.

In SOCRATES, the property can be a first-order logic formula composed from a set of predefined functions or a map contains syntactic sugar definition (see Section III-B for more details). SOCRATES has a parser (independent from the above JSON parser) implemented with ANTLR to parse general first-order logic formula in form of string, then returns an AST to represent the formula. The expressiveness of the formula can be extended by adding new predefined functions and update the parser. Otherwise, users can define new syntactic sugar keys to model new properties.

Finally, similar to the layers, each verification engine in SOCRATES is an independent class. So the new engines can be easily added as the new classes in SOCRATES. The most important function in these classes is $solve$, which receives a model and a property, then applies a specific algorithm of the engine. Each engine can be designed to solve general problem with property in form of first-order logic formula or just its own specific problem defined with its own set of syntactic sugar keys. Moreover, each engine may have its own meta parameters which use to configure its algorithm. As explained previously, these parameters are easily defined using new keys and then can be provided in JSON input file.

III. THE JSON INPUTS

Existing efforts on neural network verification techniques have results in multiple impressive tools, such as DeepPoly [15] and Reluplex [10]. These tools, however, have their own input format. For instance, Reluplex only supports multilayer perceptron networks and requires a text file containing the number of layers, the number of neurons in each layer, the normalization information, as well as weights and bias values of each layer in the network. DeepPoly, on the other hand, supports different types of networks and thus it requires the type of each layer before the values of the layer's parameters (e.g., weights and bias). Moreover, because DeepPoly focuses on local robustness, it allows users to define the distance value between samples explicitly, whereas Reluplex does not have this feature. As a result, it is hard to compare the performance of different tools or to combine techniques developed by different research groups. Furthermore, existing toolkits are often developed for specific properties. For instance, DeepPoly is designed for verifying local robustness; Reluplex focuses on reachability properties; and ADF focuses on falsifying a particular notion of fairness only. A close investigation however shows that an algorithm developed in one tool (e.g., the one developed in ADF) could be potentially extended to verify other properties (e.g., local or global robustness). Thus, we develop a JSON format which supports a variety of neural network models and an assertion language which allows user to specify a range of properties.

A. Specifying Models

The JSON file is composed a sequence of keys which specifies the details of the network model and the property. The keys used to define the model are shown in Table II. At the top-level, a model is specified using the key *model*. The value of *model* is then defined as a JSON object using a triple consisting of keys *shape*, *bounds*, and *layers*. The key *shape* is used to define the shape of the input sample, which is a tuple containing multiple integer values. The first value represents the length of the sample. Note that the first value is always

Key	Definition
model	The details of the network model
shape	The shape of input sample
bounds	The bounds of input sample
layers	The details of model layers
type	The type of a layer
weights	The weights matrix of a layer
bias	The bias vector of a layer
func	The activation function of a layer
filters	The filters matrix of a conv. layer
padding	The padding value of a conv. layer
stride	The stride value of a conv. layer
h0	The $h0$ vector of a recurrent layer
c0	The $c0$ vector of a recurrent layer
path	The path to the pre-trained model

TABLE II
THE KEYS USED TO DEFINE THE NETWORK MODEL

1 for non-recurrent neuron networks and is n with $n \geq 1$ for recurrent neural networks. The remaining numbers represent the shape for each element in the sample. For instance, the tuple $(5, 80)$ means that the input sample is a sequence with length 5 and each element in the sequence is a vector with size 80.

The key *bounds* is used to define the bounds of values for valid input samples. Its value is a string representing an array containing tuples. Suppose the number of features in the input sample is n and the number of tuples in the bounds array is m, we require that $(n \mod m) = 0$. With the above condition, the i-th element in the bounds array is used to constraint the values of input features from index $\frac{i*n}{m}$ to index $\frac{(i+1)*n}{m} - 1$. Each element in the array then is a tuple containing two numbers a_i and b_i such that $a_i \leq b_i$. The first number a_i is the lower bound and the second number b_i is the upper bound of the corresponding input features.

The key *layers* specifies details of the network layer-by-layer. Its value is an array. Each element in the array is a JSON object which represents a layer in the network with a specific *type*. According to each type, a layer may contain multiple keys to specify parameters of the layer. In general, each layer can be considered as a function, which produces an output from an input. The original sample of the network is the input of the first layer, the output of layer n is the input of layer $n + 1$, and the output of the last layer is the output of the whole network. Table III summarizes the types of layers that are currently supported in SOCRATES as well as their definitions and parameters.

Each type is associated with a list of parameter values which are specified with one or more predefined keys. For instance, keys *weights*, *bias*, and *func* are used to define the weights matrix, the bias vector, and the activation function respectively as shown in Figure 2. Depending on the type of the layer, keys such as *filters*, *stride*, *padding*, *h0* and *c0* are used to define the values of filters matrix, stride value, padding value, h_0 vector and c_0 vector respectively. In case the layer has more than one parameters of the same type, indexes are used to distinguish them. For instance, the two filters matrices in a *ResNet2l* layer can be defined with two keys *filters1* and *filters2*.

For the key *func*, the value must be one of those predefined function names. SOCRATES supports a range of functions which are commonly applied in defining neuron networks. Each function may require additional parameters, which could be specified with additional keys. The details of functions supported in SOCRATES are shown in Table IV.

We acknowledge that, for ordinary users, it may be cumbersome to define (or generate using a program) JSON files in the above format. One remedy is to support automatically translating of models trained using popular frameworks such as Tensorflow or PyTorch. That is, a user simply provides a path to a PyTorch pre-trained model file as the value for the key *path*. SOCRATES then loads the model automatically.

B. Assertion Language

For traditional software programs, many assertion languages have been developed, ranging from simple state-based assertions, logic such as Hoare logic [24], temporal logic [25] and separation logic [26], to formal specification languages such as the Z language [27] and CSP [28]. Existing verification engines however specify neural network properties in an ad hoc way. Neural networks, as a new programming paradigm, have been applied in a variety of applications. In other words, different neural networks are expected to satisfy different specifications. It is thus important that we have a language for specifying desirable properties of neural networks.

Designing an assertion language is highly non-trivial. We must answer questions such as what is considered a behavior of a neural network and what correctness specification we typically associate with the behaviors. As of now, we take a black-box view of neural networks, i.e., the behavior of a neural network is defined by its input/output relationship. In other words, from a correctness point of view, a neural network can be viewed as a function M such that, given an input feature vector x, $M(x)$ is the output vector. It is based on this view that we design our assertion language. We remark that with the development of more complicated neural networks, we might have to reason about internal states of neural network models, in which case our assertion language must be refined accordingly.

Formally, a property in SOCRATES is in the general form of $\forall \bar{x}.\ \psi_{pre} \Rightarrow \psi_{post}$ where ψ_{pre} is a precondition constraining input samples and ψ_{post} is a postcondition constraining the output and label. Both ψ_{pre} and ψ_{post} are specified using a fragment of first-order logic with built-in functions. The syntax is shown in Figure 5, which is designed to balance expressive and efficiency in terms of verification. For instance, all free variables in the assertion are universally quantified. Furthermore, each variable x is assumed to be an input of the network, i.e., a feature vector of the specified dimension. Each clause may be a conjunction or disjunction of primitive propositions. Each proposition is an (in)equality, in which each side is a nested application of predefined functions.

Note that there are restrictions such as no existential quantifiers, which limit the expressiveness of the language but

Type	Definition	Parameters
Linear	A fully-connected layer	1 weights matrix, 1 bias vector, 1 optional activation function
MaxPool1d	A 1-dimensional max pooling layer	1 stride value, 1 padding value
MaxPool2d	A 2-dimensional max pooling layer	1 stride value, 1 padding value
MaxPool3d	A 3-dimensional max pooling layer	1 stride value, 1 padding value
Conv1d	A 1-dimensional convolutional layer	1 filters matrix, 1 bias vector, 1 stride value, 1 padding value
Conv2d	A 2-dimensional convolutional layer	1 filters matrix, 1 bias vector, 1 stride value, 1 padding value
Conv3d	A 3-dimensional convolutional layer	1 filters matrix, 1 bias vector, 1 stride value, 1 padding value
ResNet2l	A 2-layers residual block	2(+1) filters matrices, 2(+1) bias vectors, 2(+1) stride values, 2(+1) padding values
ResNet3l	A 3-layers residual block	3(+1) filters matrices, 3(+1) bias vectors, 3(+1) stride values, 3(+1) padding values
RNN	A basic RNN layer	1 weights matrix, 1 bias vector, 1 h_0 vector, 1 optional activation function
LSTM	A LSTM layer	1 weights matrix, 1 bias vector, 1 h_0 vector, 1 c_0 vector
GRU	A GRU layer	2 weights matrices, 2 bias vectors, 1 h_0 vector, 1 c_0 vector
Function	A standalone function layer	1 function name, other optional parameters for the function

TABLE III
THE DIFFERENT TYPES OF LAYERS SUPPORTED BY SOCRATES

Function	Parameters	Keys	Values
ReLU	None	None	None
Sigmoid	None	None	None
Tanh	None	None	None
Softmax	None	None	None
Reshape	A new shape for the input	*newshape*	A tuple
Transpose	A new axes to be permuted	*axes*	A tuple

TABLE IV
FUNCTIONS SUPPORTED BY SOCRATES

also the complexity of the verification problem. For instance, supporting nested quantifiers would complicate the verification algorithm significantly. Yet these restrictions do not prevent interesting properties from being specified. In the following, we show how to specify a range of properties which are often relevant to neural network applications. Note that a constant can be represented by the built-in id function.

- A *reachability* property specifies that if an input satisfies certain constraint (e.g., in certain range), the neural network output must satisfy certain constraint (e.g., in certain range). Such constraints have been supported in existing tools such as Reluplex [10]. Specifying such constraint in our language is straightforward. For instance, the following is property 2 from [10] written in our language.

$$\psi_{pre} : x[0] \geq 55947.691 \wedge x[3] \geq 1145 \wedge x[4] \leq 60,$$
$$\psi_{post} : L(x) \neq 0$$

  where x is a free variable; $[i]$ is the indexed access function (i.e., $x[3]$ is the 4th element in the feature vector); and L is the labeling function (i.e., it returns the index of the maximum value[1] in the output vector according to x). Intuitively, the property states that if the input satisfies the precondition, the label must not be 0.
- *Robustness* is a desirable property for many neural networks. Robustness can be further distinguished into local robustness or global robustness. The former has been the subject of neural network verification and testing in many works [10], [14], [15]. It states that any sample x' which is similar to a particular existing sample x must have the same label of x. It is specified in our language as follows.

$$\psi_{pre} : d_i(x, x_0) \leq c,$$
$$\psi_{post} : L(x) = L(x_0)$$

  where x_0 is a constant representing an existing sample; c is a constant; d_i is a predefined function which returns the infinity norm distance between two feature vectors. As shown in Figure 5, users may choose to use d_0 or d_2 to specify the 0-norm distance or 2-norm distance alternatively. Global robustness states that similar samples should have the same label. Note that local robustness is defined based on a particular sample whereas global robustness refers to all samples (including those not in the training set). It can be specified as follows.

$$\psi_{pre} : d_i(x, y) \leq c,$$
$$\psi_{post} : L(x) = L(y)$$

  where both x and y are universally quantified variables.
- *Fairness* is a desirable property for neural network models which may have societal impact. While there are many definitions for fairness [29], one is called individual discrimination, which can be specified as follows. For simplicity, assume that the first feature of the inputs is the only sensitive feature (such as gender or race).

$$\psi_{pre} : x[0] \neq y[0] \wedge x[1] = y[1] \wedge \cdots \wedge x[n] = y[n],$$
$$\psi_{post} : L(x) = L(y)$$

  Intuitively, the above states that for all pair of samples, if the two samples differ only by the sensitive feature, their labels must be the same.
- *Miscellaneous* properties other than those defined above can be defined in our language as well. For instance, the following specifies a property which is related to the interpretability of the model. Let $p(x)$ and $q(x)$ be two propositions defined based on a sample that are expressible in our assertion language.

$$\psi_{pre} : p(x) = p(y) \wedge q(x) = q(y),$$
$$\psi_{post} : L(x) = L(y)$$

[1] SOCRATES also supports another labelling function which returns the index of the minimum value.

$$
\begin{array}{lcll}
\phi & := & \forall \bar{x}.\ \psi_{pre} \Rightarrow \psi_{post} & \\
\psi & := & \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \varphi & \\
\varphi & := & \bar{f}_1(\bar{x_1}, \bar{c_1}) \bowtie \bar{f}_2(\bar{x_2}, \bar{c_2}) & \\
\bowtie & := & > \mid \geq \mid < \mid \leq \mid = \mid \neq & \\
f & := & id & \text{i.e., the identify function} \\
 & & \mid M & \text{i.e., the network model application} \\
 & & \mid L & \text{i.e., the labeling function} \\
 & & \mid N & \text{i.e., a linear function} \\
 & & \mid [i] & \text{i.e., the indexed access function} \\
 & & \mid d_0 & \text{i.e., 0-norm distance function} \\
 & & \mid d_2 & \text{i.e., 2-norm distance function} \\
 & & \mid d_i & \text{i.e., infinity norm distance function}
\end{array}
$$

Fig. 5. Syntax for assertion language where ψ_{pre} is a precondition; ψ_{post} is a postcondition; φ is a primitive proposition; $\bar{x}$ is a set of input variables; $\bar{c}$ is a set of constants; $\bar{f}$ is a nested application of built-in functions.

Intuitively, the above assertion specifies that if two arbitrary samples are indistinguishable by proposition p and q, their labels must be the same. In other words, if the property is verified, we successfully show that this neural network model could be interpreted using a simple model, for instance, in form of a decision tree with the proposition p and q.

Syntactic Sugars We acknowledge that it may be difficult for ordinary users to write properties formally. Thus, SOCRATES provides another way to define commonly verified properties via predefined templates. In the following, we briefly discuss one example template. That is, a robustness property can be defined using the following keys.

```
{
  "robustness": "local" | "global",
  "x0": "a floating-point vector",
  "distance": "d0" | "d2" | "di",
  "eps": "a floating-point number"
  "fairness": "an integer vector"
}
```

With the above keys, users can choose to check either local or global robustness. In case of local robustness, users can specify the value of the original sample $x0$. Users can also choose different distance functions and define the maximum value for the distance. One example of applying the template has been presented in Section II-A.

C. A Benchmark Repository

To demonstrate that our JSON format is expressive enough to capture real-world verification tasks, we build a repository of neural network verification tasks based on the above-described JSON format. Verification tasks (i.e., a neural network model together with a property to be verified) which have been reported in various publications have been systematically collected and transformed into the JSON format. Furthermore, due to lack of certain types of models, we additionally train multiple models, which are added into the repository as well. Each task is labeled with the expected verification result. So far we have not encountered any verification task which cannot be transformed. The goal is to build a large repository of verification tasks which can serve as a standard comprehensive benchmark for neural network verification research. As of now, the repository contains a set of 12347 verification tasks, all of which can be downloaded at [30]. In a nutshell, the repository contains the following network models.

- 45 multilayer perceptron networks and 10 properties used in the experiments of Reluplex [10].
- 21 multilayer perceptron and convolutional networks and local robustness property with 100 samples from 2 datasets used in the experiments of DeepZ [14], DeepPoly [15], RefineZono [22], and RefinePoly [23].
- 3 multilayer perceptron networks and fairness property with 100 samples from 3 datasets used in the experiments of ADF [9].
- 4 recurrent networks and local robustness property with 100 samples from 2 datasets, trained by us.
- 1 convolutional network and local robustness property with 10000 samples in MNIST challenge [31].

IV. VERIFICATION ENGINES

With the JSON format and the assertion language, a variety of neural network verification problems can be expressed in SOCRATES. Solving them is however the real problem. While we do not claim that SOCRATES is or will be able to solve all of them, we do hope that it provides a platform for researchers to jointly experiment and develop ever-more capable verification algorithms. As of now, SOCRATES has integrated multiple existing verification engines. Furthermore, observing that existing approaches are limited to restrictive classes of neural networks, we develop two verification engines which are applicable to all neural network models and properties that are currently supported in SOCRATES. In the following, we describe the details of these two verification engines.

A. Optimization-based Falsification

The first engine is an optimization-based falsification algorithm which is inspired by existing methods on adversarial perturbation [8] and fairness testing [9].

Given a model M and an arbitrary property ϕ in our assertion language, we compile the network model into a function representation internally. The function takes samples as inputs, and produces the output vectors. The definition of the function is built according to the defined layers, layer-by-layer, based on the type of the layer and the provided parameters. For instance, with a linear layer, we have $y = f(w * x + b)$ where x is the output of the previous layer; w is the weights matrix; b is the bias vector; f is the activation function; and y is the output of the layer. The result is a function for which, given some particular input, we can easily observe its output as well as the internal computation details, i.e., inputs and outputs of each layer.

To falsify the property ϕ which is in the form of $\forall \bar{x}.\ \psi_{pre} \Rightarrow \psi_{post}$, we aim to identify input samples such that ψ_{pre} is satisfied and ψ_{post} is not, i.e., we find an input sample that satisfies $\psi_{pre} \wedge \neg\psi_{post}$. Our idea is to turn this falsification problem into an optimization problem, i.e., we define a loss function based on the formula $\psi_{pre} \wedge \neg\psi_{post}$ and apply guided-search to identify an input sample which satisfies the formula gradually. Intuitively, the loss function is defined to measure how close an input sample is to violate the property and once it is minimized to 0, we successfully falsify the property. Formally, the loss function is defined systematically according to the syntax of the assertion. That is, given any formula ψ,

$$loss(\psi) = \begin{cases} loss(\psi_1) + loss(\psi_2) & \text{if } \psi \text{ is } \psi_1 \wedge \psi_2 \\ loss(\psi_1) * loss(\psi_2) & \text{if } \psi \text{ is } \psi_1 \vee \psi_2 \\ loss(\varphi) & \text{otherwise} \end{cases}$$

where φ is $\bar{f}_1(\bar{x_1}, \bar{c_1}) \bowtie \bar{f}_2(\bar{x_2}, \bar{c_2})$ and $loss(\varphi)$ is defined as follows.

$$loss(\varphi) = \begin{cases} max(0, v_2 - v_1 + k) & \text{if } \bowtie \text{ is } > \\ max(0, v_2 - v_1) & \text{if } \bowtie \text{ is } \geq \\ max(0, v_1 - v_2 + k) & \text{if } \bowtie \text{ is } < \\ max(0, v_1 - v_2) & \text{if } \bowtie \text{ is } \leq \\ max(0, |v_1 - v_2|) & \text{if } \bowtie \text{ is } = \\ nid(v_1, v_2, k) & \text{if } \bowtie \text{ is } \neq \end{cases}$$

where v_1 and v_2 are values of $\bar{f}_1(\bar{x_1}, \bar{c_1})$ and $\bar{f}_2(\bar{x_2}, \bar{c_2})$ according to the current value of $\bar{x}$, k is a small positive number (i.e., 10^{-9}) and $nid(a, b, k)$ is 0 if $a \neq b$; otherwise it is k. The general idea is the loss function for φ should be 0 if the clause is satisfied and be positive otherwise. The number k guarantees that the value of $loss(\varphi)$ only reaches 0 when φ is satisfied. Moreover, the positive value should show how close the input samples are to satisfy the clause.

The falsification problem thus becomes the following optimization problem.

$$\arg\min_{\bar{x}}\ loss(\psi_{pre} \wedge \neg\psi_{post})$$

There are many techniques which can be applied to solve this optimization problems. In SOCRATES, by default we solve the above constraint optimization problem using the L-BFGS-B algorithm (as there is a mature implementation available in the *scipy* library). The algorithm uses the gradient and the estimate of the inverse Hessian matrix of the objective function to guide the search for the minimum value, while maintaining the simple range constraints for variables. The readers are referred to [32] and [33] for details of the L-BFGS-B algorithm.

Note that the algorithm terminates when the projected gradient or the change in the value of the objective function is less than a predefined threshold. Because of that reason, this falsification engine produces three kinds of results, i.e., successful falsification with a counterexample, successful termination without a counterexample, and timeout. Note that there is a subtle difference between the latter two results. While there is no guarantee that there is no counterexample (i.e., the property is verified) when the algorithm terminates without a counterexample, it arguably provides slightly more 'evidence' that the property may be true, compared to the case of a timeout (where optimization is still on-going and thus may find a counterexample if more time is given).

Example 1. In the following, we show how the above-described approach works through verifying a fairness property. The model is a six-layer MLP and is used to predict the income of an adult. The model is trained with the Census Income dataset [34]. In the dataset, the input has 13 features, which represent personal information of an adult. Among them, 3 features at index 0, 7, and 8 are sensitive features, which represent age, race, and gender respectively. The output is one of the 2 labels, which represents whether the income of an adult is above $50000. The model can be easily represented in our JSON format. For simplicity, assume that the property to verify is a local fairness property, i.e., all samples which are different from the given sample $x0 = [4, 0, 7, 0, 0, 4, 2, 0, 1, 5, 0, 40, 0]$ by only the sensitive features have the same label.

Other than the range constraints for valid inputs (which is defined in the model), the precondition ψ_{pre} is $x[1] = 0 \wedge x[2] = 7 \wedge \cdots \wedge x[6] = 2 \wedge x[9] = 5 \wedge \cdots \wedge x[12] = 0 \wedge (x[0] \neq 4 \vee x[7] \neq 0 \vee x[8] \neq 1)$. The label of the given input is 1 and thus the postcondition is $\psi_{post} : L(x) = 1$. The optimization engine then tries to generate a sample x which satisfies ψ_{pre} but violates ψ_{post}. After less than 1 second, a sample x is found with value $x = [2.8769, 0, 7, 0, 0, 4, 2, 0, 1, 5, 0, 40, 0]$ and output vector for x is $[0.5486, 0.4418]$, which results a label of 0. With this result, we can conclude that the model is not locally fair around the given sample.

B. Experiment results

In the following, we systematically evaluate the effectiveness and efficiency of this falsification engine using benchmark verification tasks in our repository presented in Section III-C, and compare its performance against a number of state-of-the-art tools. All the experiments are performed by a machine with 3.1Ghz 8-core CPU and 64GB RAM. The results are summarized in Table V.

In the table, the first 2 columns show the properties and the networks under verify respectively. For the local robustness property, we use infinity norm distance with the maximum value is 0.1. Note that the model name is coded with information such as the type of network, the activation function and additional parameters. Given a model and a property, there may be many verification tasks (i.e., different local robustness property for different input samples). The third column shows the number of verification tasks in each setting. Note that for local robustness and fairness properties, we randomly choose 100 samples from the datasets and for each network only the samples which are classified correctly (by comparing with the provided ground truth) are used in the verification tasks.

The next 4 columns show the results of Reluplex (commit ID: e2e48b2 on 10 Oct 2018), RefineZono and RefinePoly

Prop.	Networks	#Tasks	Reluplex			RefineZono			RefinePoly			SOCRATES		
			V	F	Time	V	F*	Time	V	F*	Time	V*	F	Time
P1 [10]	ACASXU_*_* (all networks)	45	0	0	45m	×	×	×	×	×	×	45	0	32s
P2 [10]	ACASXU_x_* ($x \geq 2$)	36	0	8	32m	×	×	×	×	×	×	8	28	23s
P3 [10]	ACASXU_*_* $\neq$ (1_{7, 8, 9})	42	8	0	38m39s	×	×	×	×	×	×	42	0	29s
P4 [10]	ACASXU_*_* $\neq$ (1_{7, 8, 9})	42	5	0	40m7s	×	×	×	×	×	×	42	0	29s
P5 [10]	ACASXU_1_1	1	0	0	1m	×	×	×	×	×	×	1	0	1s
P6 [10]	ACASXU_1_1	1	0	0	1m	×	×	×	×	×	×	1	0	1s
P7 [10]	ACASXU_1_9	1	0	0	1m	×	×	×	×	×	×	1	0	1s
P8 [10]	ACASXU_2_9	1	0	0	1m	×	×	×	×	×	×	1	0	1s
P9 [10]	ACASXU_3_3	1	0	0	1m	×	×	×	×	×	×	1	0	1s
P10 [10]	ACASXU_4_5	1	0	0	1m	×	×	×	×	×	×	1	0	1s
Robust.	MNIST_ReLU_4_1024	98	×	×	×	0	79	1h	0	65	1h	23	75	33s
Robust.	MNIST_ReLU_6_100	99	×	×	×	0	99	9m14s	0	99	4m32s	10	89	17s
Robust.	MNIST_ReLU_9_200	97	×	×	×	0	97	41m20s	0	97	28m4s	10	87	15s
Robust.	MNIST_Sigmoid_6_500	95	×	×	×	0	95	42m45s	×	×	×	4	91	13s
Robust.	MNIST_Sigmoid_6_500_PGD_0.1	100	×	×	×	0	100	44m56s	×	×	×	90	10	3m47s
Robust.	MNIST_Sigmoid_6_500_PGD_0.3	97	×	×	×	0	97	43m53s	×	×	×	84	13	2m37s
Robust.	MNIST_Tanh_6_500	99	×	×	×	0	99	45m44s	×	×	×	17	82	1m4s
Robust.	MNIST_Tanh_6_500_PGD_0.1	100	×	×	×	0	100	45m57s	×	×	×	97	3	3m8s
Robust.	MNIST_Tanh_6_500_PGD_0.3	100	×	×	×	0	100	46m29s	×	×	×	97	3	3m5s
Robust.	MNIST_Conv_Small_ReLU	100	×	×	×	35	65	3m35s	48	52	10m16s	89	1	49s
Robust.	MNIST_Conv_Small_ReLU_DAI	99	×	×	×	94	5	3m	95	4	2m44s	96	3	1m17s
Robust.	MNIST_Conv_Small_ReLU_PGD	100	×	×	×	76	24	4m2s	88	12	8m28s	98	2	55s
Robust.	MNIST_Conv_Big_ReLU_DAI	95	×	×	×	92	3	17m11s	65	1	1h	94	1	6m35s
Robust.	MNIST_Conv_Super_ReLU_DAI	99	×	×	×	15	0	1h	9	0	1h	97	2	20m24s
Robust.	CIFAR_ReLU_6_100	16	×	×	×	0	16	2m33s	0	16	1m43s	0	16	6s
Robust.	CIFAR_ReLU_7_1024	16	×	×	×	0	8	1h	0	8	1h	0	16	34s
Robust.	CIFAR_ReLU_9_200	9	×	×	×	0	9	4m34s	0	9	4m	0	9	8s
Robust.	CIFAR_Conv_Small_ReLU	59	×	×	×	0	59	6m58s	0	59	18m13s	0	59	8s
Robust.	CIFAR_Conv_Small_ReLU_DAI	53	×	×	×	0	53	3m50s	0	53	14m1s	2	51	12s
Robust.	CIFAR_Conv_Small_ReLU_PGD	70	×	×	×	0	70	8m23s	0	70	21m23s	0	70	9s
Robust.	CIFAR_Conv_Big_ReLU_DAI	60	×	×	×	0	3	1h	0	11	1h	0	60	25s
Robust.	Jigsaw_GRU	94	×	×	×	×	×	×	×	×	×	65	29	7m1s
Robust.	Jigsaw_LSTM	93	×	×	×	×	×	×	×	×	×	69	24	12m38s
Robust.	Wiki_GRU	96	×	×	×	×	×	×	×	×	×	55	41	11m22s
Robust.	Wiki_LSTM	94	×	×	×	×	×	×	×	×	×	44	50	29m42s
Fairness	Bank_MLP_6_Layers	99	×	×	×	×	×	×	×	×	×	90	9	4s
Fairness	Census_MLP_6_Layers	86	×	×	×	×	×	×	×	×	×	62	24	4s
Fairness	Credit_MLP_6_Layers	100	×	×	×	×	×	×	×	×	×	56	44	3s
Total	**73 networks**	**2494**	**13**	**8**	**2h41m**	**312**	**1181**	**10h14m**	**305**	**556**	**6h53**	**1492**	**992**	**1h49m**

TABLE V
EXPERIMENT WITH REACHABILITY, LOCAL ROBUSTNESS AND FAIRNESS PROPERTY. SOCRATES ENGINE: OPTIMIZATION.

(commit ID: e2ff1fd on 7 Jul 2020), and SOCRATES respectively. The sub-column V shows the number of verified tasks, and the sub-column F shows the number of falsified tasks with counterexamples. Note that when SOCRATES may successfully terminate without a counterexample, it does not guarantee that the task is verified and we indicate such results using a sub-column titled V*. RefineZono and RefinePoly do not return counterexamples, instead they report "Failed" for tasks that they fail to verify, which we indicate in the sub-columns F*. We use the symbol × to indicate that the verification task is not supported by the tool or an exception is thrown during the execution. For instance, most of the results from Reluplex are × other that those on the ACASXU models because of 2 reasons. First, Reluplex does not support convolutional or recurrent networks or fairness properties. Secondly, Reluplex can only handle inputs with a small number of features, whereas most of the verification tasks require input samples with several hundreds to several thousands of features. The time each tool spends on the verification tasks are shown in the sub-columns Time. In the experiment, we set the timeout as 1 minute for each task from property P1 to P10 (which is sufficient as these models are very small), and 1 hour for each set of remaining tasks according to each network.

Note that due to the space limit, we are unable to report the results of all 12347 verification tasks. Rather, a set of 2494 verification tasks are selected to cover different networks and properties. From the table, we see that SOCRATES is the only tool which can handle all the 2494 verification tasks. Moreover, SOCRATES can finish 100% of the tasks in time while other tools have many timeout results. In particular, Reluplex, RefineZono, and RefinePoly can only finish 12%, 90%, and 80% of the tasks assigned to them respectively. We remark that SOCRATES is sound when it reports that the properties is falsified. That is, 992 tasks are successfully falsified, with a counterexample. For the remaining tasks, the model is more likely to satisfy the property. A practical guideline is thus to always apply the falsification engine in SOCRATES and apply existing verification engines only if SOCRATES fails to falsify the property. This is particularly so given the efficiency of the falsification engine in SOCRATES.

That is, although SOCRATES attempts more tasks than the other tools, it spends much less time.

Finally, we apply the falsification engine in SOCRATES to generate counterexample for 10000 samples in MNIST challenge, in which 9853 samples are classified correctly. The results show that SOCRATES can generate 473 counterexamples satisfy the requirement of the challenge (i.e., the maximum infinity-norm distance to the original sample is 0.3) after running 15h4m.

C. Statistical Model Checking

The second verification engine we develop in SOCRATES is based on statistical model checking (SMC [35]). Note that SMC is chosen as it can be applied to all models and properties. Furthermore, it is a formal verification technique that is proven to be effective in combating the complexity of real-world systems, such as cyber-physical systems [21].

Given a property ϕ in the form of $\forall \bar{x}.\ \psi_{pre} \Rightarrow \psi_{post}$, SMC systematically evaluates the probability of those input samples that satisfy ψ_{pre} and violate ψ_{post}, through a form of hypothesis testing. To apply SMC, the users are required to provide 4 parameters, which include the expected confidence that the network satisfies the desired properties with probability θ, the bound of indifferent region δ, the values of type I error α and type II error β. With these parameters, hypothesis testing based on the SPRT algorithm (i.e., sequential probability ratio test [36]) is applied. The SPRT algorithm works by generating independent and identically distributed (IID) random samples to test the following 2 hypotheses.

- H_0: The network satisfies ϕ with probability $p \geq p_0$ and $p_0 = \theta + \delta$.
- H_1: The network satisfies ϕ with probability $p \leq p_1$ and $p_1 = \theta - \delta$.

The details of the SPRT algorithm are shown in Algorithm 1. Initially, the ratio pr is set to 1. With the desired property ϕ, we keep generating IID random samples. If a sample $\bar{x}$ does not satisfy the precondition ψ_{pre}, it is discarded and a new sample is generated. In case the sample satisfies the precondition, it is checked against the postcondition ψ_{post}. Based on the result, the value of pr is updated. Whenever the ratio value reaches the threshold $\beta/(1-\alpha)$, H_0 is accepted, which means that the probability of the property being satisfied is at least θ (with a statistical confidence defined by the parameters). Similarly, H_1 is accepted whenever the ratio reaches the threshold $(1-\beta)/\alpha$.

The above algorithm has one issue. That is, when ψ_{pre} is complicated, it may generate many samples which do not satisfy ψ_{pre}, and as a result, take a lot of time. This issue can be partially solved by adopting sampling techniques such as hit-and-run [37] or QuickSampler [38]. The basic idea of these techniques is to apply methods like constraint solving to generate multiple seeds and apply mutation to generate samples based on the seeds. We omit the details as sampling techniques are beyond the content of this paper.

Example 2. In the following, we present how we apply the above-described SMC to check property 2 with the network ACASXU_2_1 reported in [10]. The network is an MLP with 7 layers. The input has 5 features and the output has 5 labels. In this example, we assume that the values of the SMC parameters are set as follows: $\theta = 0.95$, $\alpha = 0.05$, $\beta = 0.05$, $\delta = 0.005$. As shown in Section III-B, the property is specified as follows.

$$\psi_{pre} : x[0] \geq 55947.691 \wedge x[3] \geq 1145 \wedge x[4] \leq 60,$$
$$\psi_{post} : L(x) \neq 0$$

Note that in addition to ψ_{pre} shown above, all the input features are associated with a range constraint which are omitted for simplicity. Generating IID samples randomly to satisfy ψ_{pre} is straightforward in this example as we simply generate a random value with its range. Applying the SPRT algorithm shown in Algorithm 1, in less than 1 second, the value of pr reaches $\beta/(1-\alpha)$ after 300 samples. As a result, the hypothesis H_0 (i.e., the network satisfies ϕ with probability $p \geq 0.955$) is accepted. We remark that accepting H_0 does not mean that the property is verified. In fact, using the falsification engine introduced in Section IV-A, we find an adversarial sample $x = [0.6, 0, 0, 0.45, 0.45]$ with output vector $[0.0343, -0.0230, 0.0210, -0.0179, 0.0223]$ which is labeled 0 after less than 1 second. Note that to have better confidence on the correctness of the property, a θ value arbitrarily closer to 1 can be adopted.

Algorithm 1: SPRT algorithm

Input: $\phi = \forall \bar{x} \cdot \psi_{pre} \implies \psi_{post}$
$pr = 1$;
while *True* **do**
  Generate an IID random sample $\bar{x}$;
  if $\bar{x}$ *does not satisfy* ψ_{pre} **then**
    continue;
  if $\bar{x}$ *satisfies* ψ_{post} **then**
    $pr = pr * p_1/p_0$;
  else
    $pr = pr * (1-p_1)/(1-p_0)$;
  if $pr \leq \beta/(1-\alpha)$ **then**
    Accept H_0;
  else if $pr \geq (1-\beta)/\alpha$ **then**
    Accept H_1;

D. Experiment Results

In the following, we evaluate the effectiveness and efficiency of this verification engine. Note that this verification engine is designed for probabilistic verification rather than falsification. Again, we apply the engine to all the verification tasks discussed in Section III-C. To the best of our knowledge, SOCRATES is the only statistical model checker for neuron networks and thus we have no baseline to compare with.

As the SPRT algorithm is parameterized with the probability θ, we apply it to all the verification tasks with 3 different values of θ (i.e., 0.90, 0.95 and 0.99) so that we can observe

the effort required to reach different level of probalistic confidence. The same 'default' values are adopted for the remaining 3 parameters, i.e., $\alpha = 0.05$, $\beta = 0.05$, and $\delta = 0.005$. The results are shown in Tables VI.

The first column of Table VI shows the property, the second column shows the networks are checked with the according property, and the third column show the number of tasks. The last 3 columns show the verification statistics with 3 different values of θ. For each value of θ, the sub-columns H0 and H1 show the number of verification tasks in which the hypothesis H_0 and H_1 are accepted respectively. As we can see, when the value of θ increases, the number of tasks in which H_0 is accepted decreases while the number of tasks in which H_1 is accepted increases. This is an expected result considering that θ increases means the users want to be more confident about the verified properties. We also notice that the numbers of generated samples for $\theta = 0.90$, 0.95, and 0.99 are 709874, 719668, and 695373 respectively. We see that there is a reluctant between these numbers. The result is reasonable considering that when θ increases, we need more samples to accept H_0 but less samples to accept H_1. Moreover, some specific tasks may make the value of pr go back and forth (i.e., need more samples) before reaching the threshold. The result helps to confirm that the number of generated samples depends on the specific task rather than the value of θ alone. For the time needed to run the experiment, we do not see any significant difference between 3 settings. All of them can finish testing 2494 tasks in less than 1 hour. However, we notice that the running time of testing engine in this experiment is less than half of the time needed by the falsification engine presented in Section IV-A.

For the last experiment, we apply the testing engine in SOCRATES to test the local robustness of 10000 sample in the MNIST challenge. With the setting $\theta = 0.90$, the engine accepts H_0 9760 times and accept H_1 93 times. With $\theta = 0.95$, H_0 is accepted 9724 times and H_1 129 times. These numbers are 9657 and 196 respectively for $\theta = 0.99$. Again, we can see that the number of times H_0 is accepted decreases when the value of θ increases. In the experiment, the engine generates more than 2.6 million samples for $\theta = 0.90$, and more than 2.8 million samples for $\theta = 0.95$ and 0.99. The time needed to run the experiment is 4h38m, 4h51m, and 4h54m for $\theta = 0.90$, 0.95, and 0.99 respectively.

V. RELATED WORK

In this section, we briefly review existing approaches on verification and falsification of neural networks and discuss how our approach is different.

This work is closely related to existing approaches on verifying local robustness of neural networks. Ever since the discovery of adversarial samples [39], the problem of verifying robustness of neural networks attracted much attentions due to their implications in safety critical applications. Existing approaches can be roughly classified into two groups: exact methods and approximation methods.

The exact methods aim to capture semantics of neural networks precisely and solve the verification problem through constraint solving. In [40], Tjeng *et al.* proposed to tackle the problem using Mixed Integer Linear Programming (MILP). In [10], [41], the authors proposed to solve the problem through SMT solving. These methods can verify a neural network as long as the property holds. The limitations of these methods are that they are limited to analyze (feedforward) neural networks with ReLU activation functions only. In other words, popular activation functions such as Sigmoid and Tanh are not supported. Furthermore, these methods typically have limited scalability, i.e., they can only handle networks with a small number of layers and neurons, as we partly demonstrate in Section IV-B.

The approximation methods leverage well-developed techniques such as linear approximations [42], [43] and abstract interpretation [13], [14], [15]. The idea is to conduct an over-approximation of the given neural network (e.g., through over-approximating each neuron with a simple linear constraint), and verify properties soundly based on the over-approximation. Thanks to the linear approximation, these approaches (although not all of them) could handle a wider range of activation function such as ReLU, Sigmoid or Tanh. Furthermore, these approaches are typically more scalable than the exact methods. As a price to pay, due to the over-approximation, these methods are sound but not complete, i.e., they may fail to verify a valid property due to the presence of the so-called spurious counterexamples.

Our project does not aim to replace these impressive efforts. Rather, we aim to provide a platform which integrates and further develops these efforts. Furthermore, the two new verification engines supported in SOCRATES are complementary to existing verification approaches, i.e., the falsification engine would allow us to efficiently falsify those properties which are not satisfied whereas the SMC engine provides a way of verifying neural network probabilistically.

Beside robustness, this work is related to a line of work on analyzing fairness of neural networks, collectively called fairness testing. Several approaches have been proposed on fairness testing machine learning models including neural networks. All of them search for discriminatory instances (i.e., counterexamples to fairness) through certain heuristic-based sampling techniques. Galhotra *et al.* proposed THEMIS [44], [45], a causality based algorithm utilizing the random test generation to evaluate a model's fairness, i.e., the frequency of individual discriminatory instances. Their work could be viewed as a heuristic-based approach for statistical model checking of fairness. Udeshi *et al.* proposed AEQUITAS [46] which is based on THEMIS. AEQUITAS works in two phases, i.e., a 'global search' phase, which attempts to explore the whole input domain, followed by a 'local search' phase, which searches within the neighboring region of the instances identified in the global phase. Zhang *et al.* proposes a lightweight algorithm ADF to efficiently generate individual discriminatory instances [9]. They perturb instances near the decision boundary in the global search and leverage the gradient

Prop.	Networks	#Tasks	$\theta = 0.90$			$\theta = 0.95$			$\theta = 0.99$		
			H0	H1	Time	H0	H1	Time	H0	H1	Time
P1 [10]	ACASXU_*_* (all networks)	45	45	0	30s	45	0	30s	45	0	30s
P2 [10]	ACASXU_x_* ($x \geq 2$)	36	36	0	24s	36	0	24s	14	22	24s
P3 [10]	ACASXU_*_* $\neq (1\_\{7, 8, 9\})$	42	42	0	28s	42	0	28s	42	0	28s
P4 [10]	ACASXU_*_* $\neq (1\_\{7, 8, 9\})$	42	42	0	28s	42	0	28s	42	0	28s
P5 [10]	ACASXU_1_1	1	1	0	1s	1	0	1s	1	0	1s
P6 [10]	ACASXU_1_1	1	1	0	1s	1	0	1s	1	0	1s
P7 [10]	ACASXU_1_9	1	1	0	1s	1	0	1s	1	0	1s
P8 [10]	ACASXU_2_9	1	1	0	1s	1	0	1s	1	0	1s
P9 [10]	ACASXU_3_3	1	1	0	1s	1	0	1s	1	0	1s
P10 [10]	ACASXU_4_5	1	1	0	1s	1	0	1s	1	0	1s
Robust.	MNIST_ReLU_4_1024	98	96	2	26s	96	2	26s	95	3	27s
Robust.	MNIST_ReLU_6_100	99	99	0	7s	99	0	7s	99	0	7s
Robust.	MNIST_ReLU_9_200	97	93	4	10s	92	5	10s	90	7	9s
Robust.	MNIST_Sigmoid_6_500	95	93	2	17s	93	2	18s	92	3	19s
Robust.	MNIST_Sigmoid_6_500_PGD_0.1	100	98	2	18s	98	2	19s	98	2	19s
Robust.	MNIST_Sigmoid_6_500_PGD_0.3	97	96	1	18s	96	1	19s	96	1	19s
Robust.	MNIST_Tanh_6_500	99	98	1	17s	98	1	18s	97	2	18s
Robust.	MNIST_Tanh_6_500_PGD_0.1	100	100	0	18s	100	0	18s	100	0	19s
Robust.	MNIST_Tanh_6_500_PGD_0.3	100	99	1	19s	99	1	18s	99	1	19s
Robust.	MNIST_Conv_Small_ReLU	100	100	0	28s	99	1	29s	99	1	29s
Robust.	MNIST_Conv_Small_ReLU_DAI	99	99	0	27s	99	0	28s	99	0	30s
Robust.	MNIST_Conv_Small_ReLU_PGD	100	100	0	27s	100	0	29s	100	0	29s
Robust.	MNIST_Conv_Big_ReLU_DAI	95	95	0	2m33s	95	0	2m41s	95	0	2m47s
Robust.	MNIST_Conv_Super_ReLU_DAI	99	99	0	8m30s	98	1	8m55s	98	1	9m3s
Robust.	CIFAR_ReLU_6_100	16	7	9	6s	7	9	6s	7	9	6s
Robust.	CIFAR_ReLU_7_1024	16	10	6	43s	10	6	41s	8	8	39s
Robust.	CIFAR_ReLU_9_200	9	7	2	9s	5	4	8s	5	4	8s
Robust.	CIFAR_Conv_Small_ReLU	59	52	7	27s	49	10	30s	45	14	24s
Robust.	CIFAR_Conv_Small_ReLU_DAI	53	50	3	31s	49	4	24s	47	6	24s
Robust.	CIFAR_Conv_Small_ReLU_PGD	70	64	6	38s	64	6	30s	63	7	29s
Robust.	CIFAR_Conv_Big_ReLU_DAI	60	53	7	2m53s	52	8	2m47s	48	12	2m9s
Robust.	Jigsaw_GRU	94	91	3	4m	90	4	4m20s	88	6	5m15s
Robust.	Jigsaw_LSTM	93	93	0	4m25s	93	0	4m43s	92	1	5m18s
Robust.	Wiki_GRU	96	96	0	4m49s	96	0	4m54s	96	0	5m10s
Robust.	Wiki_LSTM	94	93	1	5m27s	92	2	5m37s	92	2	5m40s
Fairness	Bank_MLP_6_Layers	99	91	8	2s	89	10	3s	87	12	2s
Fairness	Census_MLP_6_Layers	86	61	25	2s	56	30	3s	45	41	2s
Fairness	Credit_MLP_6_Layers	100	64	36	3s	56	44	2s	51	49	2s
Total	**73 networks**	**2494**	**2368**	**126**	**41m6s**	**2341**	**153**	**42m19s**	**2280**	**214**	**43m38s**

TABLE VI
EXPERIMENT WITH STATISTICAL MODEL CHECKING

information to guide the local search.

We note that existing approaches focuses on testing of fairness rather than fairness verification. To the best of our knowledge, ours is the first attempt to support fairness verification on neural networks.

The falsification engine in SOCRATES is inspired by the many adversarial sample generation methods. Since Szegedy *et al.* discovered that neural networks are vulnerable to adversarial samples [39], many attacking methods have been developed to generate adversarial samples efficiently with minimal perturbation. Some examples are the FGSM method [39], the Jacobian-based saliency map attack [47], and C&W [8]. Lastly, this work is remotely related to recent papers which proposed different coverage criteria for evaluating the effectiveness of a test set, along with different methods to generate test cases to improve the coverage criteria. For instance, DeepXplore [48] proposed the first testing criterion for DNN models, i.e., Neuron Coverage (NC), which calculates the percentage of activated neurons (w.r.t. an activation function) among all neurons. Unlike the above-mentioned work, this project focuses on verification of neural networks.

VI. CONCLUSION

In this work, we aim to develop a unified platform for neural network verification. Towards our goal, we make three technical contributions. First, we propose a unified JSON format for capturing a variety of neural network models as well as an assertion language for specifying neural network properties. We further build a repository of 12347 verification tasks, which serves as a comprehensive benchmark for evaluating neural network verification techniques. Second, we develop two novel algorithms for tackling the verification problems for a variety of models and properties. The experiment results show that these two algorithms complement existing approaches. Lastly, we devote non-trivial amount of engineering effort to make our project a useful open source platform.

We are continuously developing our platform further as the following activities: integrating all existing verification algorithms, extending them with further optimizations (such as abstraction refinement for existing abstraction interpretation based approaches) and developing new algorithms (such as probabilistic model checking for neural networks).

REFERENCES

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[2] X. Yin and X. Liu, "Multi-task convolutional neural network for pose-invariant face recognition," *IEEE Transactions on Image Processing*, vol. 27, no. 2, pp. 964–975, 2017.

[3] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[4] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, 2015, pp. 1422–1432.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[7] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[8] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 39–57. [Online]. Available: https://doi.org/10.1109/SP.2017.49

[9] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and D. Ting, "White-box fairness testing through adversarial sampling," 2020.

[10] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[11] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1599–1614.

[12] ——, "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*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 6369–6379. [Online]. Available: http://papers.nips.cc/paper/7873-efficient-formal-safety-analysis-of-neural-networks

[13] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[14] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 802–10 813.

[15] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 41, 2019.

[16] Y. Jacoby, C. W. Barrett, and G. Katz, "Verifying recurrent neural networks using invariant inference," *CoRR*, vol. abs/2004.02462, 2020. [Online]. Available: https://arxiv.org/abs/2004.02462

[17] W. Ryou, J. Chen, M. Balunovic, G. Singh, A. M. Dan, and M. T. Vechev, "Fast and effective robustness certification for recurrent neural networks," *CoRR*, vol. abs/2005.13300, 2020. [Online]. Available: https://arxiv.org/abs/2005.13300

[18] "Onnx," https://onnx.ai.

[19] "Nnef," https://www.khronos.org/nnef/.

[20] "Socrates," https://github.com/longph1989/Socrates.

[21] E. M. Clarke and P. Zuliani, "Statistical model checking for cyber-physical systems," in *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, ser. Lecture Notes in Computer Science, T. Bultan and P. Hsiung, Eds., vol. 6996. Springer, 2011, pp. 1–12. [Online]. Available: https://doi.org/10.1007/978-3-642-24372-1_1

[22] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "Boosting robustness certification of neural networks," in *International Conference on Learning Representations*, 2018.

[23] G. Singh, R. Ganvir, M. Püschel, and M. Vechev, "Beyond the single neuron convex barrier for neural network certification," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 098–15 109.

[24] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[25] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[26] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2002, pp. 55–74.

[27] J. M. Spivey and J. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[28] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[29] M. Joseph, M. J. Kearns, J. H. Morgenstern, and A. Roth, "Fairness in learning: Classic and contextual bandits," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 325–333. [Online]. Available: http://papers.nips.cc/paper/6355-fairness-in-learning-classic-and-contextual-bandits

[30] "Benchmark for Socrates," https://figshare.com/s/f2c4959b59cf32da4891.

[31] "MNIST Adversarial Examples Challenge," https://github.com/MadryLab/mnist_challenge.

[32] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A limited memory algorithm for bound constrained optimization," *SIAM Journal on scientific computing*, vol. 16, no. 5, pp. 1190–1208, 1995.

[33] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 23, no. 4, pp. 550–560, 1997.

[34] "Census Income Dataset," https://archive.ics.uci.edu/ml/datasets/adult.

[35] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 28, no. 1, pp. 1–39, 2018.

[36] A. Wald, "Sequential tests of statistical hypotheses," *The annals of mathematical statistics*, vol. 16, no. 2, pp. 117–186, 1945.

[37] M. Chen and B. W. Schmeiser, "General hit-and-run monte carlo sampling for evaluating multidimensional integrals," *Oper. Res. Lett.*, vol. 19, no. 4, pp. 161–169, 1996. [Online]. Available: https://doi.org/10.1016/0167-6377(96)00030-2

[38] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of sat solutions for testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 549–559.

[39] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[40] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[41] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2017, pp. 269–286.

[42] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel, "Towards fast computation of certified robustness for relu networks," *arXiv preprint arXiv:1804.09699*, 2018.

[43] E. Wong and J. Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," *arXiv preprint arXiv:1711.00851*, 2017.

[44] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: Automatically testing software for discrimination," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 871–875.

[45] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 498–510.

[46] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 98–108.

[47] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 2016, pp. 372–387. [Online]. Available: https://doi.org/10.1109/EuroSP.2016.36

[48] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, 2017, pp. 1–18. [Online]. Available: https://doi.org/10.1145/3132747.3132785