DOI:10.1145/3361566

DeepXplore: Automated Whitebox Testing of Deep Learning Systems

By Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana

Abstract

Deep learning (DL) systems are increasingly deployed in safety- and security-critical domains such as self-driving cars and malware detection, where the correctness and predictability of a system's behavior for corner case inputs are of great importance. Existing DL testing depends heavily on manually labeled data and therefore often fails to expose erroneous behaviors for rare inputs.

We design, implement, and evaluate DeepXplore, the first white-box framework for systematically testing real-world DL systems. First, we introduce neuron coverage for measuring the parts of a DL system exercised by test inputs. Next, we leverage multiple DL systems with similar functionality as cross-referencing oracles to avoid manual checking. Finally, we demonstrate how finding inputs for DL systems that both trigger many differential behaviors and achieve high neuron coverage can be represented as a joint optimization problem and solved efficiently using gradient-based search techniques.

DeepXplore efficiently finds thousands of incorrect corner case behaviors (e.g., self-driving cars crashing into guard rails and malware masquerading as benign software) in state-of-the-art DL models with thousands of neurons trained on five popular datasets such as ImageNet and Udacity self-driving challenge data. For all tested DL models, on average, DeepXplore generated one test input demonstrating incorrect behavior within one second while running only on a commodity laptop. We further show that the test inputs generated by DeepXplore can also be used to retrain the corresponding DL model to improve the model's accuracy by up to 3%.

1. INTRODUCTION

Over the past few years, Deep Learning (DL) has made tremendous progress, achieving or surpassing human-level performance for a diverse set of tasks in many application domains. These advances have led to widespread adoption and deployment of DL in security- and safety-critical systems such as self-driving cars,[1] malware detection,[4] and aircraft collision avoidance systems.[6]

This wide adoption of DL techniques presents new challenges as the predictability and correctness of such systems are of crucial importance. Unfortunately, DL systems, despite their impressive capabilities, often demonstrate unexpected or incorrect behaviors in corner cases for several reasons such as biased training data and overfitting of the models. In safety- and security-critical settings, such incorrect behaviors can lead to disastrous consequences such as a fatal collision of a self-driving car. For example, a Google self-driving car recently crashed into a bus because it expected the bus to yield under a set of rare conditions but the bus did not.[a]

A Tesla car in autopilot crashed into a trailer because the autopilot system failed to recognize the trailer as an obstacle due to its "white color against a brightly lit sky" and the "high ride height".

[b] Such corner cases were not part of Google's or Tesla's test set and thus never showed up during testing.

Therefore, DL systems, just like traditional software, must be tested systematically for different corner cases to detect and fix ideally any potential flaws or undesired behaviors. This presents a new system problem as automated and systematic testing of large-scale, real-world DL systems with thousands of neurons and millions of parameters for all corner cases is extremely challenging.

The standard approach for testing DL systems is to gather and manually label as much real-world test data as possible. Some DL systems such as Google self-driving cars also use simulation to generate synthetic training data. However, such simulation is completely unguided as it does not consider the internals of the target DL system. Therefore, for the large input spaces of real-world DL systems (e.g., all possible road conditions for a self-driving car), none of these approaches can hope to cover more than a tiny fraction (if any at all) of all possible corner cases.

Recent works on adversarial deep learning[3] have demonstrated that carefully crafted synthetic images by adding minimal perturbations to an existing image can fool state-of-the-art DL systems. The key idea is to create synthetic images such that they get classified by DL models differently than the original picture but still look the same to the human eye. Although such adversarial images expose some erroneous behaviors of a DL model, the main restriction of such an approach is that it must limit its perturbations to tiny invisible changes and require ground truth labels. Moreover, just like other forms of existing DL testing, the adversarial images only cover a small part (52.3%) of DL system's logic as shown in Section 5. In essence, the current machine learning testing practices for finding incorrect corner cases are analogous to finding bugs in traditional software by using

[a] http://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report

[b] https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/

The original version of this paper was published in *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China, Oct. 28–31, 2017), 1–18.

test inputs with low code coverage and thus are unlikely to find many erroneous cases.

The key challenges in automated systematic testing of large-scale DL systems are twofold: (1) how to generate inputs that trigger different parts of a DL system's logic and uncover different types of erroneous behaviors, and (2) how to identify erroneous behaviors of a DL system without manual labeling/checking. This paper describes and highlights how we design and build DeepXplore to address both challenges.

First, we introduce the concept of neuron coverage for measuring the parts of a DL system's logic exercised by a set of test inputs based on the number of neurons activated by the inputs. At a high level, neuron coverage of DL systems is similar to code coverage of traditional systems, a standard empirical metric for measuring the amount of code exercised by an input in the traditional software. However, code coverage itself is not a good metric for estimating coverage of DL systems as most rules in DL systems, unlike traditional software, are not written manually by a programmer but rather learned from training data. In fact, we find that for most of the DL systems that we tested, even a single randomly picked test input was able to achieve 100% code coverage, whereas the neuron coverage was less than 10%.

Next, we show how multiple DL systems with similar functionality (e.g., self-driving cars by Google and Tesla, and GM) can be used as cross-referencing oracles to identify erroneous corner cases without providing ground truth labels which require huge manual labeling effort. For example, if one self-driving car decides to turn left whereas others turn right for the same input, one of them is likely to be incorrect. Such techniques have been applied successfully in the past for detecting logic bugs without manual specifications in a wide variety of traditional software.[2] In this paper, we demonstrate how differential testing can be applied to DL systems.

Finally, we demonstrate how the problem of generating test inputs that maximize neuron coverage of a DL system while also exposing as many differential behaviors (i.e., differences between multiple similar DL systems) as possible can be formulated as a joint optimization problem. Unlike traditional programs, Deep Neural Networks (DNNs) used by DL systems are differentiable. Therefore, their gradients with respect to inputs can be calculated accurately given whitebox access to the corresponding model. In this paper, we show how these gradients can be used to efficiently solve the joint optimization problem for large-scale real-world DL systems.

We design, implement, and evaluate DeepXplore, to the best of our knowledge, the first efficient whitebox testing framework for large-scale DL systems. In addition to maximizing neuron coverage and behavioral differences between DL systems, DeepXplore also supports adding custom constraints by the users for simulating different types of realistic inputs (e.g., different types of lighting and occlusion for images/videos). We demonstrate that DeepXplore efficiently finds thousands of unique incorrect corner case behaviors (e.g., self-driving cars crashing into guard rails) in 15 state-of-the-art DL models trained using five real-world datasets such as Udacity self-driving car challenge data, image data from ImageNet and MNIST, Android malware data from Drebin, and PDF malware data from Contagio/VirusTotal. For all of the tested DL models, on average, DeepXplore generated one test input demonstrating incorrect behavior within one second while running on a commodity laptop. The inputs generated by DeepXplore achieved 34.4 and 33.2% higher neuron coverage on average than the same number of randomly picked inputs and adversarial inputs,[3] respectively. We further show that the test inputs generated by DeepXplore can be used to retrain the corresponding DL model to improve classification accuracy as well as identify potentially polluted training data. We achieve up to 3% improvement in classification accuracy by retraining a DL model on inputs generated by DeepXplore compared to retraining on the same number of random or adversarial inputs.

A number of follow-up papers after DeepXplore have expanded the idea of whitebox testing for domain-specific transformations in self-driving cars[13] and developed exhaustive black box testing techniques for a variety of common transformations.[11] Besides, the metric of neuron coverage has also been extended in TensorFlow as an open-source tool by the Google Brain team.[10] Beyond testing, we have also studied and proposed more rigorous verification techniques leveraging interval arithmetic to certify the robustness of neural networks[15, 16] (Figure 1).

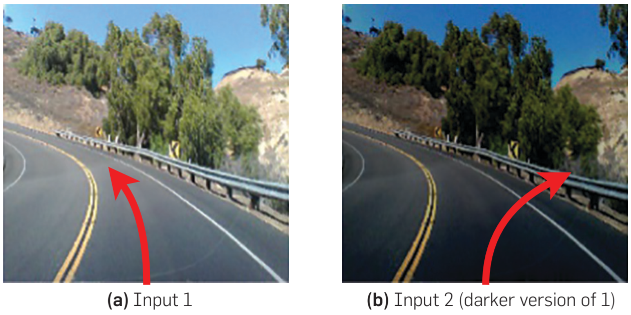
2. BACKGROUND

2.1. DL systems

We define a DL system to be any software system that includes at least one Deep Neural Network (DNN) component. Note that some DL systems might comprise solely DNNs (e.g., self-driving car DNNs predicting steering angles without any manual rules), whereas others may have some DNN components interacting with other traditional software components to produce the final output.

As shown in Figure 2, the development process of the DNN components of a DL system is fundamentally different from traditional software development. Unlike traditional software, where the developers directly specify the logic of the system, the DNN components learn their rules

Figure 1. An example of erroneous behavior found by DeepXplore in Nvidia DAVE-2 self-driving car platform. The DNN-based self-driving car correctly decides to turn left for image (a) but incorrectly decides to turn right and crashes into the guardrail for image (b), a slightly darker version of (a).



(a) Input 1

(b) Input 2 (darker version of 1)

automatically from data. The developers of DNN components can indirectly influence the rules learned by a DNN by modifying the training data, features, and the model's architectural details (e.g., number of layers).

2.2. DNN architecture

DNNs are inspired by human brains with millions of interconnected neurons. They are known for their amazing ability to automatically identify and extract the relevant high-level features from raw inputs without any human guidance besides labeled training data. In recent years, DNNs have surpassed human performance in many application domains due to increasing availability of large datasets, specialized hardware, and efficient training algorithms.

A DNN consists of multiple *layers*, each containing multiple *neurons* as shown in Figure 3. A *neuron* is an individual computing unit inside a DNN that applies an *activation function* on its inputs and passes the result to other connected neurons (see Figure 3). The common activation functions include sigmoid, hyperbolic tangent, or ReLU (Rectified Linear Unit). A DNN usually has at least three (often more) layers: one input, one output, and one or more hidden layers. Each neuron in one layer has directed connections to the neurons in the next layer. The numbers of neurons in each layer and the connections between them vary significantly across DNNs. Overall, a DNN can be defined mathematically as a multi-input, multi-output parametric function *F* composed of many parametric subfunctions representing different neurons.

Each connection between the neurons in a DNN is bound to a *weight* parameter characterizing the strength of the connection between the neurons. For supervised learning, the weights of the connections are learned during training by minimizing a cost function over the training data via gradient descent.

Figure 2. Comparison between traditional and ML system development processes. Developers specify clear logic of the system, whereas DNN learns the logic from training data.

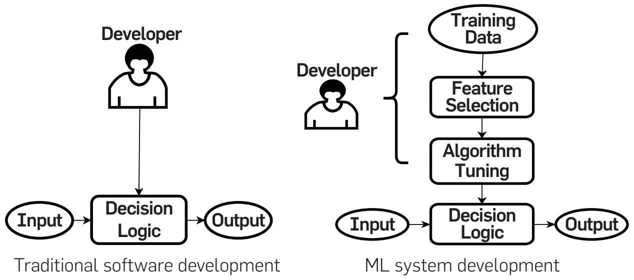
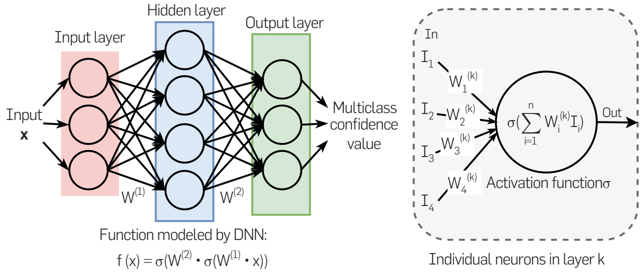


Figure 3. A simple DNN and the computations performed by each neuron.



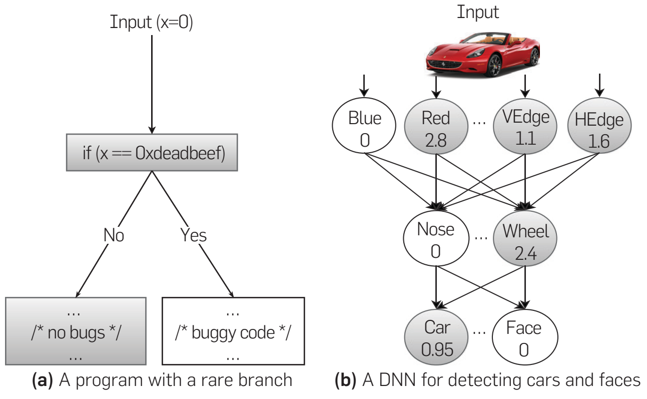
Each layer of the network transforms the information contained in its input to a higher-level representation of the data. For example, consider a pretrained network as shown in Figure 4b for classifying images into two categories: human faces and cars. The first few hidden layers transform the raw pixel values into low-level texture features such as edges or colors and feed them to the deeper layers.[18] The last few layers, in turn, extract and assemble the meaningful high-level abstractions such as noses, eyes, wheels, and headlights to make the classification decision.

2.3. Limitations of existing DNN testing

Expensive labeling effort. Existing DNN testing techniques require prohibitively expensive human effort to provide correct labels/actions for a target task (e.g., self-driving a car, image classification, and malware detection). For complex and high-dimensional real-world inputs, human beings, even domain experts, often have difficulty in efficiently performing a task correctly for a large dataset. For example, consider a DNN designed to identify potentially malicious executable files. Even a security professional will have trouble determining whether an executable is malicious or benign without executing it. However, executing and monitoring a malware inside a sandbox incur significant performance overhead and therefore make manual labeling significantly harder to scale to a large number of inputs.

Low test coverage. None of the existing DNN testing schemes even try to cover different rules of the DNN. Therefore, the test inputs often fail to uncover different erroneous behaviors of a DNN. For example, DNNs are often tested by simply dividing a whole dataset into two random parts—one for training and the other for testing. The testing set in such cases may only exercise a small subset of all rules learned by a DNN. Recent results involving adversarial evasion attacks against DNNs have demonstrated the existence of some corner cases where DNN-based image

Figure 4. Comparison between program flows of a traditional program and a neural network. The nodes in gray denote the corresponding basic blocks or neurons that participated while processing an input.



(a) A program with a rare branch

(b) A DNN for detecting cars and faces classifiers (with state-of-the-art performance on randomly picked testing sets) still incorrectly classify synthetic images generated by adding humanly imperceptible perturbations to a test image.[3] However, the adversarial inputs, similar to random test inputs, also only cover a small part of the rules learned by a DNN as they are not designed to maximize coverage. Moreover, they are also inherently limited to small imperceptible perturbations around a test input as larger perturbations will visually change the input and therefore will require manual inspection to ensure correctness of the DNN's decision.

Problems with low-coverage DNN tests. To better understand the problem of low test coverage of rules learned by a DNN, we provide an analogy to a similar problem in testing traditional software. Figure 4 shows a side-by-side comparison of how a traditional program and a DNN handle inputs and produce outputs. Specifically, the figure shows the *similarity between traditional software and DNNs*: in software program, each statement performs a certain operation to transform the output of previous statement(s) to the input to the following statement(s), whereas in DNN, each neuron transforms the output of previous neuron(s) to the input of the following neuron(s). Of course, unlike traditional software, DNNs do not have explicit branches but a neuron's influence on the downstream neurons decreases as the neuron's output value gets lower. A lower output value indicates less influence and vice versa. When the output value of a neuron becomes zero, the neuron does not have any influence on the downstream neurons.

As demonstrated in Figure 4a, the problem of low coverage in testing traditional software is obvious. In this case, the buggy behavior will never be seen unless the test input is `0xdeadbeef`. The chances of randomly picking such a value are very small. Similarly, low-coverage test inputs will also leave different behaviors of DNNs unexplored. For example, consider a simplified neural network, as shown in Figure 4b, that takes an image as an input and classifies it into two different classes: cars and faces. The text in each neuron (represented as a node) denotes the object or property that the neuron detects,[c] and the number in each neuron is the real value outputted by that neuron. The number indicates how confident the neuron is about its output. Note that randomly picked inputs are highly unlikely to set high output values for the unlikely combination of neurons. Therefore, many incorrect DNN behaviors will remain unexplored even after performing a large number of random tests. For example, if an image causes neurons labeled as "Nose" and "Red" to produce high output values and the DNN misclassifies the input image as a car, such a behavior will never be seen during regular testing as the chances of an image containing a red nose (e.g., a picture of a clown) are very small.

3. OVERVIEW

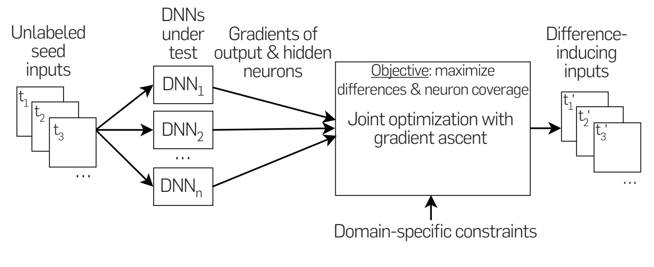
In this section, we provide a general overview of DeepXplore, our whitebox framework for systematically testing DNNs for erroneous corner case behaviors. The main components of DeepXplore are shown in Figure 5. DeepXplore takes unlabeled test inputs as seeds and generates new tests that cover a large number of neurons (i.e., activates them to a value above a customizable threshold) while causing the tested DNNs to behave differently. Specifically, DeepXplore solves a joint optimization problem that maximizes both differential behaviors and neuron coverage. Note that both goals are crucial for thorough testing of DNNs and finding diverse erroneous corner case behaviors. High neuron coverage alone may not induce many erroneous behaviors, whereas just maximizing different behaviors might simply identify different manifestations of the same underlying root cause.

DeepXplore also supports enforcing of custom domain-specific constraints as part of the joint optimization process. For example, the value of an image pixel has to be between 0 and 255. Such domain-specific constraints can be specified by the users of DeepXplore to ensure that the generated test inputs are valid and realistic.

We designed an algorithm for efficiently solving the joint optimization problem mentioned above using gradient ascent. First, we compute the gradient of the *outputs* of the neurons in both the output and hidden layers with the *input value* as a variable and the *weight parameter* as a constant. Such gradients can be computed efficiently for most DNNs. Note that DeepXplore is designed to operate on pretrained DNNs. The gradient computation is efficient because our whitebox approach has access to the pretrained DNNs' weights and the intermediate neuron values. Next, we iteratively perform gradient ascent to modify the test input toward maximizing the objective function of the joint optimization problem described above. Essentially, we perform a gradient-guided local search starting from the seed inputs and find new inputs that maximize the desired goals. Note that, at a high level, our gradient computation is similar to the backpropagation performed during the training of a DNN, but the key difference is that, unlike our algorithm, backpropagation treats the *input value* as a constant and the *weight parameter* as a variable.

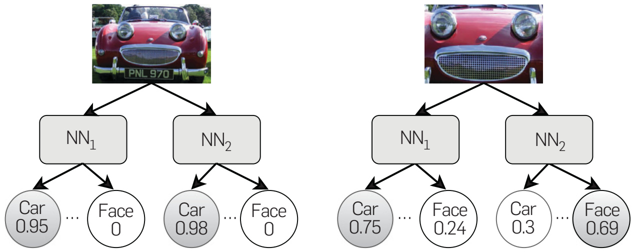
A working example. We use Figure 6 as an example to show how DeepXplore generates test inputs. Consider that we have two DNNs to test—both perform similar tasks, that is, classifying images into cars or faces, as shown in Figure 6, but they are trained independently with different datasets and parameters. Therefore, the DNNs will learn similar but slightly different classification rules. Let us also assume that

Figure 5. DeepXplore workflow.


Unlabeled seed inputs
t_1
t_2
t_3
...
DNNs under test
DNN_1
DNN_2
...
DNN_n
Gradients of output & hidden neurons
Objective: maximize differences & neuron coverage
Joint optimization with gradient ascent
Domain-specific constraints
Difference-inducing inputs
t_1'
t_2'
t_3'
...

[c] Note that one cannot always map each neuron to a particular task, i.e., detecting specific objects/properties. Figure 4b simply highlights that different neurons often tend to detect different features.

Figure 6. Inputs inducing different behaviors in two similar DNNs.



(a) DNNs produce same output **(b)** DNNs produce different output

we have a seed test input, the image of a red car, which both DNNs identify as a car as shown in Figure 6a.

DeepXplore tries to maximize the chances of finding differential behavior by modifying the input, that is, the image of the red car, towards maximizing its probability of being classified as a car by one DNN but minimizing corresponding probability of the other DNN. DeepXplore also tries to cover as many neurons as possible by activating (i.e., causing a neuron's output to have a value greater than a threshold) inactive neurons in the hidden layer. We further add domain-specific constraints (e.g., ensure the pixel values are integers within 0 and 255 for image input) to make sure that the modified inputs still represent real-world images. The joint optimization algorithm will iteratively perform a gradient ascent to find a modified input that satisfies all of the goals described above. DeepXplore will eventually generate a set of test inputs where the DNNs' outputs differ, for example, one DNN thinks it is a car, whereas the other thinks it is a face as shown in Figure 6b.

4. METHODOLOGY

In this section, we provide a brief technical description of our algorithm. The details can be found in the original paper. First, we define and explain the concepts of neuron coverage and gradient for DNNs. Next, we describe how the testing problem can be formulated as a joint optimization problem. Finally, we provide the gradient-based algorithm for solving the joint optimization problem.

4.1. Definitions

Neuron coverage. We define neuron coverage of a set of test inputs as the ratio of the number of unique activated neurons for all test inputs and the total number of neurons in the DNN.[d] We consider a neuron activated if its output is greater than a threshold (e.g., 0).

More formally, let us assume that all neurons of a DNN are represented by the set $N = \{n_1, n_2, ...\}$, all test inputs are represented by the set $T = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ...\}$, and $out(n, \boldsymbol{x})$ is a function that returns the output value of neuron n in the DNN for a given test input $\boldsymbol{x}$. Note that the bold $\boldsymbol{x}$ signifies that $\boldsymbol{x}$ is a vector. Let t represent the threshold for considering a neuron to be activated. In this setting, neuron coverage can be defined as follows.

$$NCov(T, \boldsymbol{x}) = \frac{\left|\{n \mid \forall \boldsymbol{x} \in T, out(n, \boldsymbol{x}) > t\}\right|}{|N|}$$

To demonstrate how neuron coverage is calculated in practice, consider the DNN as shown in Figure 4b. The neuron coverage (with threshold 0) for the input picture of the red car as shown in Figure 4b will be $5/8 = 0.625$.

Gradient. The gradients or forward derivatives of the outputs of neurons of a DNN with respect to the input are well known in deep learning literature. They have been extensively used both for crafting adversarial examples and visualizing/understanding DNNs.[18] We provide a brief definition here for completeness and refer interested readers to[18] for more details.

Let $\boldsymbol{\theta}$ and $\boldsymbol{x}$ represent the parameters and the test input of a DNN, respectively. The parametric function performed by a neuron can be represented as $y = f(\boldsymbol{\theta}, \boldsymbol{x})$ where f is a function that takes $\boldsymbol{\theta}$ and $\boldsymbol{x}$ as input and output y. Note that y can be the output of any neuron defined in the DNN (e.g., neuron from output layer or intermediate layers). The gradient of $f(\boldsymbol{\theta}, \boldsymbol{x})$ with respect to input $\boldsymbol{x}$ can be defined as:

$$\boldsymbol{G} = \nabla_x f(\boldsymbol{\theta}, \boldsymbol{x}) = \partial y / \partial \boldsymbol{x} \qquad (1)$$

The computation inside f is essentially a sequence of stacked functions that compute the input from previous layers and forward the output to next layers. Thus, $\boldsymbol{G}$ can be calculated by utilizing the chain rule in calculus, that is, by computing the layer-wise derivatives starting from the layer of the neuron that outputs y until reaching the input layer that takes $\boldsymbol{x}$ as the input. Note that the dimension of the gradient $\boldsymbol{G}$ is identical to that of the input $\boldsymbol{x}$.

4.2. DeepXplore algorithm

The main advantage of the test input generation process for a DNN over traditional software is that the test generation process, once defined as an optimization problem, can be solved efficiently using gradient ascent. In this section, we describe the details of the formulation and find solutions to the optimization problem. Note that solutions to the optimization problem can be efficiently found for DNNs as the gradients of the objective functions of DNNs, unlike traditional software, can be easily computed.

As discussed earlier in Section 3, the objective of the test generation process is to maximize both the number of observed differential behaviors and the neuron coverage while preserving domain-specific constraints provided by the users. Below, we define the objectives of our joint optimization problem formally and explain the details of the algorithm for solving it.

Maximizing differential behaviors. The first objective of the optimization problem is to generate test inputs that can induce different behaviors in the tested DNNs, that is, different DNNs will classify the same input into different classes. Suppose we have n DNNs $F_{k \in 1..n}: \boldsymbol{x} \rightarrow \boldsymbol{y}$, where F_k is the function modeled by the kth neural network. $\boldsymbol{x}$ represents the input and $\boldsymbol{y}$ represents the output class probability vectors.

[d] Neuron coverage can be defined in many different ways other than that defined in this paper. We refer readers to other follow-up papers for details on different definitions.

Given an arbitrary $\boldsymbol{x}$ as seed that gets classified to the same class by all DNNs, our goal is to modify $\boldsymbol{x}$ such that the modified input $\boldsymbol{x'}$ will be classified differently by at least one of the n DNNs.

Let $F_k(\boldsymbol{x})[c]$ be the class probability that F_k predicts $\boldsymbol{x}$ to be c. We randomly select one neural network F_j and maximize the following objective function:

$$obj_1(\boldsymbol{x}) = \sum_{k \neq j} F_k(\boldsymbol{x})[c] - \lambda_1 \cdot F_j(\boldsymbol{x})[c] \qquad (2)$$

where λ_1 is a parameter to balance the objective terms between the DNNs' $F_{k \neq j}$ that maintain the same class outputs as before and the DNN F_j that produce different class outputs. As all of $F_{k \in 1..n}$ are differentiable, Equation 2 can be maximized using gradient ascent by iteratively changing $\boldsymbol{x}$ based on the computed gradient: $\frac{\partial obj_1(\boldsymbol{x})}{\partial \boldsymbol{x}}$.

Maximizing neuron coverage. The second objective is to generate inputs that maximize neuron coverage. We achieve this goal by iteratively picking inactivated neurons and modifying the input such that the output of that neuron goes above the neuron activation threshold. Let us assume that we want to maximize the output of a neuron n, that is, we want to maximize $obj_2(\boldsymbol{x}) = f_n(\boldsymbol{x})$ such that $f_n(\boldsymbol{x}) > t$, where t is the neuron activation threshold, and we write $f_n(\boldsymbol{x})$ as the function modeled by neuron n that takes $\boldsymbol{x}$ (the original input to the DNN) as the input and produce the output of neuron n (as defined in Equation 1). We can again leverage the gradient ascent mechanism as $f_n(\boldsymbol{x})$ is a differentiable function whose gradient is $\frac{\partial f_n(\boldsymbol{x})}{\partial \boldsymbol{x}}$.

Note that we can also potentially jointly maximize multiple neurons simultaneously, but we choose to activate one neuron at a time in this algorithm for clarity.

Joint optimization. We jointly maximize obj_1 and f_n described above and maximize the following function:

$$obj_{joint} = \left(\sum_{i \neq j} F_i(\boldsymbol{x})[c] - \lambda_1 F_j(\boldsymbol{x})[c]\right) + \lambda_2 \cdot f_n(\boldsymbol{x}) \qquad (3)$$

where λ_2 is a parameter for balancing between the two objectives and n is the inactivated neuron that we randomly pick at each iteration. As all terms of obj_{joint} are differentiable, we jointly maximize them using gradient ascent by modifying $\boldsymbol{x}$.

Domain-specific constraints. One important aspect of the optimization process is that the generated test inputs need to satisfy several domain-specific constraints to be physically realistic. In particular, we want to ensure that the changes applied to $\boldsymbol{x}_i$ during the ith iteration of gradient ascent process satisfy all the domain-specific constraints for all i. For example, for a generated test image $\boldsymbol{x}$, the pixel values must be within a certain range (e.g., 0–255).

Although some such constraints can be efficiently embedded into the joint optimization process using the Lagrange Multipliers similar to those used in support vector machines, we found that the majority of them cannot be easily handled by the optimization algorithm. Therefore, we designed a simple rule-based method to ensure that the generated tests satisfy the custom domain-specific constraints. As the seed input $\boldsymbol{x}_{seed} = \boldsymbol{x}_0$ always satisfies the constraints by definition, our technique must ensure that after the ith ($i > 0$) iteration of gradient ascent, $\boldsymbol{x}_i$ still satisfies the constraints. Our algorithm ensures this property by modifying the gradient ***grad*** such that $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + s \cdot \boldsymbol{grad}$ still satisfies the constraints (s is the step size in the gradient ascent).

For discrete features, we round the gradient to an integer. For DNNs handling visual input (e.g., images), we add different spatial restrictions such that only part of the input images is modified. A detailed description of the domain-specific constraints that we implemented can be found in Section 5.2.

Hyperparameters. To summarize, there are four major hyperparameters that control different aspects of DeepXplore as described below. (1) λ_1 balances the objectives between minimizing one DNN's prediction for a certain label and maximizing the rest of DNNs' predictions for the same label. Larger λ_1 puts higher priority on lowering the prediction value/confidence of a particular DNN, whereas smaller λ_1 puts more weight on maintaining the other DNNs' predictions. (2) λ_2 provides balance between finding differential behaviors and neuron coverage. Larger λ_2 focuses more on covering different neurons, whereas smaller λ_2 generates more difference-inducing test inputs. (3) s controls the step size used during iterative gradient ascent. Larger s may lead to oscillation around the local optimum, whereas smaller s may need more iterations to reach the objective. (4) t is the threshold to determine whether each individual neuron is activated or not. Finding inputs that activate a neuron becomes increasingly harder as t increases.

5. EXPERIMENTAL SETUP

5.1. Test datasets and DNNs

We adopt 5 popular public datasets with different types of data—MNIST, ImageNet, Driving, Contagio/VirusTotal, and Drebin—and then evaluate DeepXplore on 3 DNNs for each dataset (i.e., a total of 15 DNNs). We provide a summary of the five datasets and the corresponding DNNs in Table 1. The detailed description can be found in the full paper. All the evaluated DNNs are either pretrained (i.e., we use public weights reported by previous researchers) or trained by us using public real-world architectures to achieve comparable performance to that of the state-of-the-art models for the corresponding dataset. For each dataset, we used DeepXplore to test three DNNs with different architectures.

5.2. Domain-specific constraints

As discussed earlier, to be useful in practice, we need to ensure that the generated tests are valid and realistic by applying domain-specific constraints. For example, generated images should be physically producible by a camera. Similarly, generated PDFs need to follow the PDF specification to ensure that a PDF viewer can open the test file. Below we describe two major types of domain-specific constraints (i.e., image and file constraints) that we use in this paper. **Image constraints (MNIST, ImageNet, and Driving).** DeepXplore used three different types of constraints for simulating different environmental conditions of images: (1) lighting effects for simulating different intensities of lights, (2) occlusion by a single small rectangle for simulating an attacker potentially blocking some parts of a camera,

Table 1. Details of the DNNs and datasets used to evaluate DeepXplore.

Dataset	Dataset description	DNN description	DNN name	# of neurons	Architecture	Reported Acc.	Our Acc.
MNIST	Hand-written digits	LeNet variations	MNI_C1	52	LeNet-1, LeCun et al. [8]	98.3%	98.33%
			MNI_C2	148	LeNet-4, LeCun et al. [8]	98.9%	98.59%
			MNI_C3	268	LeNet-5, LeCun et al. [8]	99.05%	98.96%
Imagenet	General images	State-of-the-art image classifiers from ILSVRC	IMG_C1	14,888	VGG-16, Simonyan et al. [12]	92.6%**	92.6%**
			IMG_C2	16,168	VGG-19, Simonyan et al. [12]	92.7%**	92.7%**
			IMG_C3	94,059	ResNet50, He et al. [5]	96.43%**	96.43%**
Driving	Driving video frames	Nvidia DAVE self-driving systems	DRV_C1	1,560	Dave-orig [1]	N/A	99.91%#
			DRV_C2	1,560	Dave-norminit##	N/A	99.94%#
			DRV_C3	844	Dave-dropout++	N/A	99.96%#
Contagio/ Virustotal	PDFs	PDF malware detectors	PDF_C1	402	<200, 200>+	98.5%–	96.15%
			PDF_C2	602	<200, 200, 200>+	98.5%–	96.25%
			PDF_C3	802	<200, 200, 200, 200>+	98.5%–	96.47%
Drebin	Android apps	Android app malware detectors	APP_C1	402	<200, 200>+, Grosse et al. [4]	98.92%	98.6%
			APP_C2	102	<50, 50>+, Grosse et al. [4]	96.79%	96.82%
			APP_C3	212	<200, 10>+, Grosse et al. [4]	92.97%	92.66%

** Top-5 test accuracy; we exactly match the reported performance as we use the pretrained networks.
# We report 1-MSE (mean squared error) as the accuracy because steering angle is a continuous value.
+ <x,y,...> denotes three hidden layers with *x* neurons in first layer, *y* neurons in second layer, etc.
– Accuracy using SVM as reported by Šrndic et al. [14].
## https://github.com/jacobgil/keras-steering-angle-visualizations.
++ https://github.com/navoshta/behavioral-cloning.

and (3) occlusion by multiple tiny black rectangles for simulating effects of dirt on camera lens.

Other constraints (Drebin and Contagio/VirusTotal). For Drebin dataset, DeepXplore enforces a constraint that only allows modifying features related to the Android manifest file and thus ensures that the application code is unaffected. Moreover, DeepXplore only allows adding features (changing from zero to one) but does not allow deleting features (changing from one to zero) from the manifest files to ensure that no application functionality is changed due to insufficient permissions. Thus, after computing the gradient, DeepXplore only modifies the manifest features whose corresponding gradients are greater than zero. For Contagio/VirusTotal dataset, we follow the restrictions on each feature as described by Šrndic and Laskkov.[14]

6. RESULTS

6.1. Summary

Table 2 summarizes the numbers of erroneous behaviors found by DeepXplore for each tested DNN while using 2000 randomly selected seed inputs from the corresponding test sets. Note that as the testing set has a similar number of samples for each class, these randomly-chosen 2000 samples also follow that distribution. The hyperparameters for these experiments, as shown in Table 2, are empirically chosen to maximize both the rate of finding difference-inducing inputs as well as the neuron coverage.

For the experimental results shown in Figure 7, we apply three domain-specific constraints (lighting effects, occlusion by a single rectangle, and occlusion by multiple rectangles) as described in Section 5.2. For all other experiments involving vision-related tasks, we only use the lighting effects as the domain-specific constraints. For all malware-related experiments, we apply all the relevant domain-specific constraints described in Section 5.2. We use the hyperparameters listed in Table 2 in all the experiments unless otherwise specified. Figure 7 shows some difference-inducing inputs generated by DeepXplore for MNIST, ImageNet, and Driving dataset along with the corresponding erroneous behaviors. Table 3 (Drebin) and Table 4 (Contagio/VirusTotal) show two sample difference-inducing inputs generated by DeepXplore that caused erroneous behaviors in the tested DNNs. We highlight the differences between the seed input features and the features modified by DeepXplore. Note that we only list the top three modified features due to space limitations.

Table 2. Number of difference-inducing inputs found by DeepXplore for each tested DNN obtained by randomly selecting 2000 seeds from the corresponding test set for each run.

	Hyperparams				
DNN name	λ_1	λ_2	*s*	*t*	# Differences found
MNI_C1	1	0.1	10	0	1073
MNI_C2					1968
MNI_C3					827
IMG_C1	1	0.1	10	0	1969
IMG_C2					1976
IMG_C3					1996
DRV_C1	1	0.1	10	0	1720
DRV_C2					1866
DRV_C3					1930
PDF_C1	2	0.1	0.1	0	1103
PDF_C2					789
PDF_C3					1253
APP_C1	1	0.5	N/A	0	2000
APP_C2					2000
APP_C3					2000

6.2. Benefits of neuron coverage

In this subsection, we evaluate how effective neuron coverage is in measuring the comprehensiveness of DNN testing.

Figure 7. The first row shows the seed test inputs and the second row shows the difference-inducing test inputs generated by DeepXplore. The left three columns show results under different lighting effects, the middle three are using single occlusion box, and the right three are using black rectangles as the transformation constraints. For each type of transformation (three pairs of images), the images from left to right are from self-driving car, MNIST, and ImageNet.

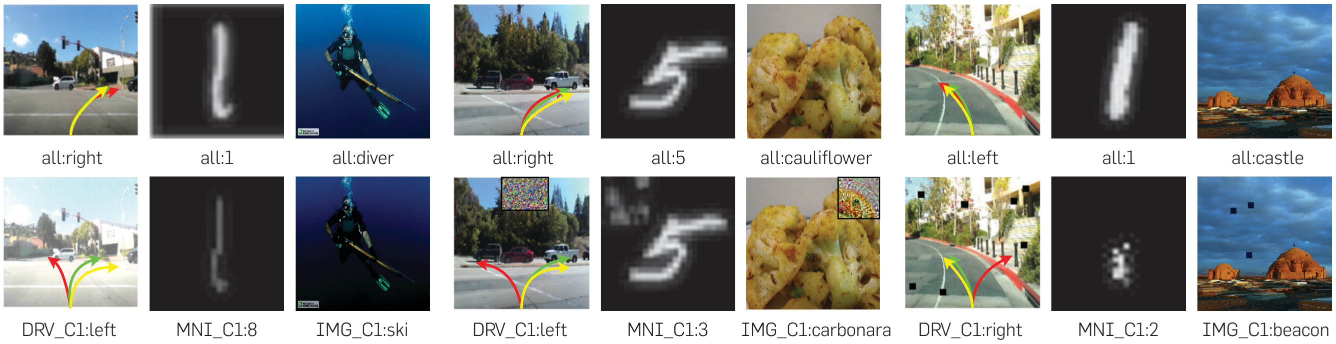


Table 3. The features added to the manifest file for generating two malware inputs that Android app classifiers (Drebin) incorrectly mark as benign.

input 1	**feature**	*feature:: bluetooth*	*activity:: .SmartAlertTerms*	**service_ receiver:: .rrltpsi**
	before	0	0	0
	after	1	1	1
input 2	**feature**	*provider:: xclockprovider*	*permission:: CALL_PHONE*	*provider:: contentprovider*
	before	0	0	0
	after	1	1	1

Table 4. The top-3 most in(de)cremented features for generating two sample malware inputs that PDF classifiers incorrectly mark as benign.

input 1	**feature**	*size*	*count_action*	*count_endobj*
	before	1	0	1
	after	34	21	20
input 2	**feature**	*size*	*count_font*	*author_num*
	before	1	0	10
	after	27	15	5

Table 5. Comparison of code coverage and neuron coverage for 10 randomly selected inputs from the original test set of each DNN.

	Code coverage			Neuron coverage		
Dataset	***C1***	***C2***	***C3***	***C1***	***C2***	***C3***
MNIST	100%	100%	100%	32.7%	33.1%	25.7%
ImageNet	100%	100%	100%	1.5%	1.1%	0.3%
Driving	100%	100%	100%	2.5%	3.1%	3.9%
VirusTotal	100%	100%	100%	19.8%	17.3%	17.3%
Drebin	100%	100%	100%	16.8%	10%	28.6%

It has recently been shown that each neuron in a DNN tends to independently extract a specific feature of the input instead of collaborating with other neurons for feature extraction.[18] This finding intuitively explains why neuron coverage is a good metric for DNN testing comprehensiveness. To empirically confirm this observation, we perform two different experiments as described below.

First, we show that neuron coverage is a significantly better metric than code coverage for measuring comprehensiveness of the DNN test inputs. More specifically, we find that a small number of test inputs can achieve 100% code coverage for all DNNs where neuron coverage is actually less than 34%. Second, we evaluate neuron activations for test inputs from different classes. Our results show that inputs from different classes tend to activate more unique neurons than inputs from the same class. Both findings confirm that neuron coverage provides a good estimation of the numbers and types of DNN rules exercised by an input.

Neuron coverage vs. code coverage. We compare both code and neuron coverages achieved by the same number of inputs by evaluating the test DNNs on ten randomly picked testing samples as described in Section 5.1. We measure a DNN's code coverage in terms of the line coverage of the Python code used in the training and testing process. We set the threshold t in neuron coverage 0.75, that is, a neuron is considered covered only if its output is greater than 0.75 for at least one input.

The results, as shown in Table 5, clearly demonstrate that neuron coverage is a significantly better metric than code coverage for measuring DNN testing comprehensiveness. Even 10 randomly picked inputs result in 100% code coverage for all DNNs, whereas the neuron coverage never goes above 34% for any of the DNNs. Moreover, neuron coverage changes significantly based on the tested DNNs and the test inputs. For example, the neuron coverage for the complete MNIST testing set (i.e., 10,000 testing samples) only reaches 57.7, 76.4, and 83.6% for C1, C2, and C3, respectively. In contrast, the neuron coverage for the complete Contagio/Virustotal test set reaches 100%.

Activation of neurons for different classes of inputs. We measure the number of active neurons that are common across the LeNet-5 DNN running on pairs of MNIST inputs of the same and different classes, respectively. In particular, we randomly select 200 input pairs where 100 pairs have the same label (e.g., labeled as 8) and 100 pairs have different labels (e.g., labeled as 8 and 4). Then, we calculate the number of common (overlapped) active neurons for these input pairs. Table 6 confirms our hypothesis that inputs coming from the same class share more activated neurons than those coming from different classes. As inputs from different classes tend to get detected through matching of different DNN rules, our result also confirms that neuron coverage can effectively estimate the numbers of different rules activated during DNN testing.

7. LIMITATIONS AND FUTURE WORKS

Although our results are very encouraging, several other obstacles must be solved to make ML systems more reliable.

First, DeepXplore only considers a small subset of transformations to test the corresponding properties. Although they are arguably more realistic than adversarial perturbations, they still do not fully capture all real-world input distortions. Tian et al. have recently developed a testing tool for autonomous vehicles[13] that considers a wider range of transformations and uses neuron coverage to guide the search for errors. However, testing complex realistic transformations such as simulating shadows from other objects still remains an open problem.

Next, it is challenging to *efficiently* search for error-inducing test cases for arbitrary transformations. DeepXplore efficiently finds error-inducing inputs leveraging the input gradients. However, there are many realistic transformations for which such input gradient information cannot be computed accurately. For example, it is difficult to compute gradients directly to emulate different weather conditions (e.g., snow or rain) for testing self-driving vehicles. There is an emerging area of research that leverages the generative adversarial networks (GANs) to learn differentiable representations of such complex transformations to enable gradient-based search for error-inducing inputs.[9]

Finally, a key limitation of our gradient-based local search is that it does not provide any guarantee about the absence of errors. There has been recent progress on two complementary directions that can provide stronger guarantees than DeepXplore. First, Pei et al. considered a specific subset of transformations where the output space is polynomial in the input image size.[11] Therefore, it is feasible for these transformations to exhaustively enumerate the transformed inputs to verify the absence of errors. Second, several recent works have explored new formal verification techniques for NNs[7,15–17], that can either ensure the absence of adversarial inputs or provide a concrete counterexample for a given network and a test input. However, scaling these techniques to larger networks remains a major challenge.

Table 6. Average number of overlaps among activated neurons for a pair of inputs of the same class and different classes. Inputs of different classes tend to activate different neurons.

	Total neurons	*Avg. no. of activated neurons*	*Avg. overlap*
Diff. class	268	83.6	45.9
Same class	268	84.1	74.2

8. CONCLUSION

We designed and implemented DeepXplore, the first whitebox system for systematically testing DL systems. We introduced a new metric, neuron coverage, for measuring how many rules in a DNN are exercised by a set of inputs. DeepXplore performs gradient ascent to solve a joint optimization that maximizes both neuron coverage and the number of potentially erroneous behaviors. DeepXplore was able to find thousands of erroneous behaviors in 15 state-of-the-art DNNs trained on five real-world datasets. We hope DeepXplore's results and its limitations can encourage and motivate other researchers to work on this challenging but critical and exciting area.

References

1. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).
2. Brubaker, C., Jana, S., Ray, B., Khurshid, S., Shmatikov V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).
3. Goodfellow, I., Shlens, J., Szegedy, C. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations* (2015).
4. Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security* (2017).
5. He, K., Zhang, X., Ren, S., Sun, J. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (2016).
6. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th IEEE/AIAA Digital Avionics Systems Conference* (2016).
7. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J. Reluplex: An efficient smt solver for verifying deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification* (2017).
8. LeCun, Y., Cortes, C., Burges, C.J. MNIST handwritten digit database. 2010.
9. Liu, M.-Y., Breuel, T., Kautz, J. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems* (2017).
10. Odena, A., Goodfellow, I. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875* (2018).
11. Pei, K., Cao, Y., Yang, J., Jana, S. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785* (2017).
12. Simonyan, K., Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations* (2015).
13. Tian, Y., Pei, K., Jana, S., Ray, B. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ACM (2018), 303–314
14. Šrndic, N., Laskov, P. Practical evasion of a learning-based classifier: a case study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).
15. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems* (2018).
16. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium* (2018).
17. Wong, E., Kolter, Z. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning* (2018).
18. Yosinski, J., Clune, J., Fuchs, T., Lipson, H. Understanding neural networks through deep visualization. In *2015 ICML Workshop on Deep Learning* (2015).

Kexin Pei, Junfeng Yang, and Suman Jana ({kpei,junfeng,suman}@cs.columbia.edu), Columbia University, USA.

Yinzhi Cao (yinzhi.cao@jhu.edu), Johns Hopkins University, USA.

Copyright held by author/owner. Publication rights licensed by ACM.

Watch the authors discuss their work in this exclusive *Communications* video. https://cacm.acm.org/videos/deepxplore