DeepXplore: Automated Whitebox Testing of Deep Learning Systems

Kexin Pei*, Yinzhi Cao†, Junfeng Yang*, Suman Jana*

*Columbia University, †Lehigh University

ABSTRACT

Deep learning (DL) systems are increasingly deployed in safety- and security-critical domains including 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 whitebox framework for systematically testing real-world DL systems. First, we introduce neuron coverage for systematically 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 including 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%.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132785

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Neural networks**; *Reliability*; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Deep learning testing, differential testing, whitebox testing

ACM Reference Format:
Kexin Pei, Yinzhi Cao, Junfeng Yang, Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '17).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3132747.3132785

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 including image classification [31, 66], speech recognition [83], and playing games such as Go [64]. These advances have led to widespread adoption and deployment of DL in security- and safety-critical systems such as self-driving cars [10], malware detection [88], and aircraft collision avoidance systems [35].

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, overfitting, and underfitting 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 [27]. 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" [73]. Such corner cases were not part of Google's or Tesla's test set and thus never showed up during testing.

Therefore, safety- and security-critical 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 systems 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 [1, 3]. Some DL systems such as Google self-driving cars also use simulation to generate synthetic training data [4]. 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 [26, 49, 72] 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. While 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 or require manual checks. 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 § 6. In essence, the current machine learning testing practices for finding incorrect corner cases are analogous to finding bugs in traditional software by using 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 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 (i.e., the output values are higher than a threshold) 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 a 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 are 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 while the neuron coverage was less than 10%.

(a) Input 1 (b) Input 2 (darker version of 1)

Figure 1: An example 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).

Next, we show how multiple DL systems with similar functionality (e.g., self-driving cars by Google, Tesla, and GM) can be used as cross-referencing oracles to identify erroneous corner cases without manual checks. For example, if one self-driving car decides to turn left while others turn right for the same input, one of them is likely to be incorrect. Such differential testing techniques have been applied successfully in the past for detecting logic bugs without manual specifications in a wide variety of traditional software [6, 11, 14, 15, 45, 86]. 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, the functions approximated by most popular 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 above-mentioned 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

including 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 [26, 49, 72] 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.

Our main contributions are:

- We introduce neuron coverage as the first whitebox testing metric for DL systems that can estimate the amount of DL logic explored by a set of test inputs.
- We demonstrate that the problem of finding a large number of behavioral differences between similar DL systems while maximizing neuron coverage can be formulated as a joint optimization problem. We present a gradient-based algorithm for solving this problem efficiently.
- We implement all of these techniques as part of DeepXplore, the first whitebox DL-testing framework that exposed thousands of incorrect corner case behaviors (e.g., self-driving cars crashing into guard rails as shown in Figure 1) in 15 state-of-the-art DL models with a total of 132,057 neurons trained on five popular datasets containing around 162 GB of data.
- We show that the tests generated by DeepXplore can also be used to retrain the corresponding DL systems to improve classification accuracy by up to 3%.

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 of DNNs (e.g., self-driving car DNNs predicting steering angles without any manual rules) while others may have some DNN components interacting with other traditional software components to produce the final output.

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 automatically from data. The

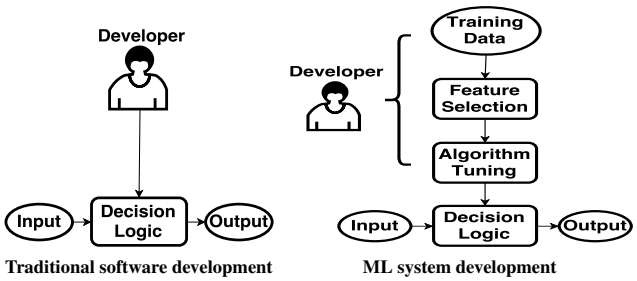


Figure 2: Comparison between traditional and ML system development processes.

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) as shown in Figure 2.

As a DNN's rules are mostly unknown even to its developers, testing and fixing of erroneous behaviors of DNNs are crucial in safety-critical settings. In this paper, we primarily focus on automatically finding inputs that trigger erroneous behaviors in DL systems and provide preliminary evidence about how these inputs can be used to fix the buggy behavior by augmenting or filtering the training data in § 7.3.

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 [20, 38, 47], specialized hardware [34, 50], and efficient training algorithms [31, 39, 66, 71].

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) [48]. 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 sub-functions 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. DNNs can be trained using different training algorithms, but gradient descent using backpropagation is by far the most popular training algorithm for DNNs [60].

Each layer of the network transforms the information contained in its input to a higher-level representation of the data. For example, consider a pre-trained network 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 like edges or colors and feed them to the deeper layers [87]. The last few layers, in turn, extract and assemble the meaningful high-level abstractions like 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 makes 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

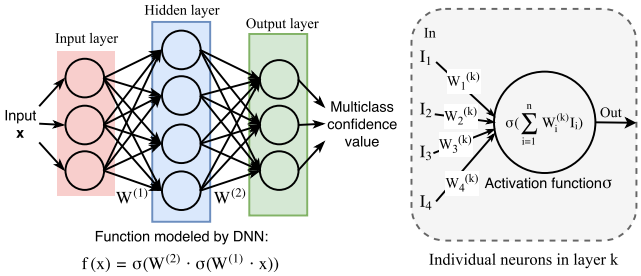
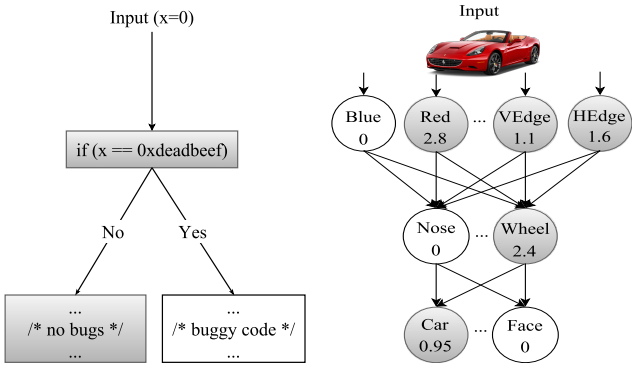


Figure 3: A simple DNN and the computations performed by each of its neurons.



(a) A program with a rare branch **(b) A DNN for detecting cars and faces**

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.

demonstrated the existence of some corner cases where DNN-based image 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 [26, 29, 52, 63, 79, 85]. However, the adversarial inputs, similar to random test inputs, also only cover a small part 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), while 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 is 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 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[1], 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) is 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 while 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 pre-trained DNNs. The gradient computation is efficient because our whitebox approach has access to the pre-trained 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.

[1] 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.

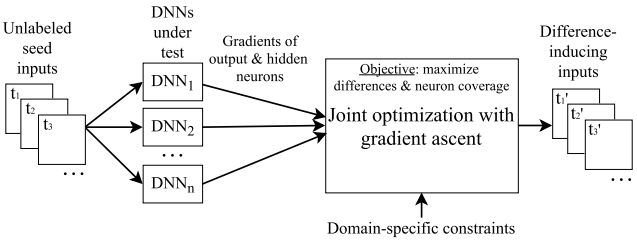
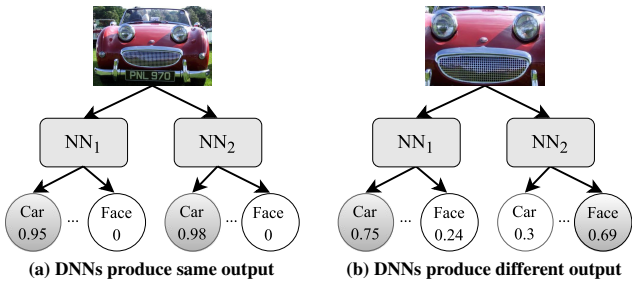


Figure 5: DeepXplore workflow.

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, i.e., 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 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, i.e., 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



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

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

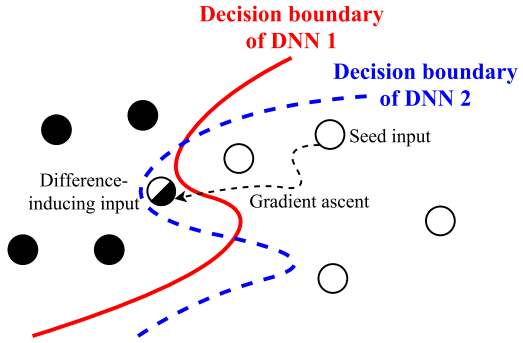


Figure 7: Gradient ascent starting from a seed input and gradually finding the difference-inducing test inputs.

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, e.g., one DNN thinks it is a car while the other thinks it is a face as shown in Figure 6b.

Figure 7 illustrates the basic concept of our technique using gradient ascent. Starting from a seed input, DeepXplore performs the guided search by the gradient in the input space of two similar DNNs supposed to perform the same task such that it finally uncovers the test inputs that lie between the decision boundaries of these DNNs. Such test inputs will be classified differently by the two DNNs. Note that while the gradient provides the rough direction toward reaching the goal (e.g., finding difference-inducing inputs), it does not guarantee the fastest convergence. Thus as shown in Figure 7, the gradient ascent process often does not follow a straight path towards reaching the target.

4 METHODOLOGY

In this section, we provide a detailed technical description of our algorithm. 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. We consider a neuron to be activated if its output is higher than a threshold value (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.

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

To demonstrate how neuron coverage is calculated in practice, consider the DNN showed in Figure 4b. The neuron coverage (with threshold 0) for the input picture of the red car 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 [26, 29, 52, 72] and visualizing/understanding DNNs [44, 65, 87]. We provide a brief definition here for completeness and refer interested readers to [87] 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:

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

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, G can be calculated by utilizing the chain rule in calculus, i.e., 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 input. Note that the dimension of the gradient 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 finding 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 § 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. Algorithm 1 shows the algorithm for generating test inputs by solving this joint optimization problem. 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, i.e., 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 k-th neural network. $\boldsymbol{x}$ represents the input and $\boldsymbol{y}$ represents the output class probability vectors. 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.

Algorithm 1 Test input generation via joint optimization

Input: **seed_set** ← unlabeled inputs as the seeds
dnns ← multiple DNNs under test
λ_1 ← parameter to balance output differences of DNNs (Equation 2)
λ_2 ← parameter to balance coverage and differential behavior
s ← step size in gradient ascent
t ← threshold for determining if a neuron is activated
p ← desired neuron coverage
cov_tracker ← tracks which neurons have been activated

```
1: /* main procedure */
2: gen_test := empty set
3: for cycle(x ∈ seed_set) do  // infinitely cycling through seed_set
4:     /* all dnns should classify the seed input to the same class */
5:     c = dnns[0].predict(x)
6:     d = randomly select one dnn from dnns
7:     while True do
8:         obj1 = COMPUTE_OBJ1(x, d, c, dnns, λ1)
9:         obj2 = COMPUTE_OBJ2(x, dnns, cov_tracker)
10:        obj = obj1 + λ2 · obj2
11:        grad = ∂obj / ∂x
12:        /*apply domain specific constraints to gradient*/
13:        grad = DOMAIN_CONSTRNTS(grad)
14:        x = x + s · grad  //gradient ascent
15:        if d.predict(x) ≠ (dnns-d).predict(x) then
16:            /* dnns predict x differently */
17:            gen_test.add(x)
18:            update cov_tracker
19:            break
20:    if DESIRED_COVERAGE_ACHVD(cov_tracker) then
21:        return gen_test
22: /* utility functions for computing obj1 and obj2 */
23: procedure COMPUTE_OBJ1(x, d, c, dnns, λ1)
24:    rest = dnns - d
25:    loss1 := 0
26:    for dnn in rest do
27:        loss1 += dnn_c(x)  //confidence score of x being in class c
28:    loss2 := d_c(x)  //d's output confidence score of x being in class c
29:    return (loss1 - λ1·loss2)
30: procedure COMPUTE_OBJ2(x, dnns, cov_tracker)
31:    loss := 0
32:    for dnn ∈ dnns do
33:        select a neuron n inactivated so far using cov_tracker
34:        loss += n(x)  //the neuron n's output when x is the dnn's input
35:    return loss
```

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 (Algorithm 1 line 6) and maximize the following objective function:

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

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 easily maximized using gradient ascent by iteratively changing $\boldsymbol{x}$ based on the computed gradient: $\frac{\partial obj_1(\boldsymbol{x})}{\partial \boldsymbol{x}}$ (Algorithm 1 line 8-14 and procedure `COMPUTE_OBJ1`).

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 output of that neuron goes above the neuron activation threshold. Let us assume that we want to maximize the output of a neuron n, i.e., 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 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 (Algorithm 1 line 8-14 and procedure `COMPUTE_OBJ2`).

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

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

where λ_2 is a parameter for balancing between the two objectives of the joint optimization process and n is the inactivated neuron that we randomly pick at each iteration (Algorithm 1 line 33). As all terms of obj_{joint} are differentiable, we jointly maximize them using gradient ascent by modifying $\boldsymbol{x}$ (Algorithm 1 line 14).

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 [63]. In particular, we want to ensure that the changes applied to $\boldsymbol{x_i}$ during the i-th 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 to 255).

While some such constraints can be efficiently embedded into the joint optimization process using the Lagrange Multipliers similar to those used in support vector machines [76], 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 satisfy the constraints by definition, our technique must ensure that after i-th ($i > 0$) iteration of gradient ascent, $\boldsymbol{x_i}$ still satisfies the constraints. Our algorithm ensures this property by modifying the gradient $\boldsymbol{grad}$ (line 13 in Algorithm 1) 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 § 6.2.

Hyperparameters in Algorithm 1. 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 while 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 while 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 while 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 become increasingly harder as t increases.

5 IMPLEMENTATION

We implement DeepXplore using TensorFlow 1.0.1 [5] and Keras 2.0.3 [16] DL frameworks. Our implementation consists of around 7,086 lines of Python code. Our code is built on TensorFlow/Keras but does not require any modifications to these frameworks. We leverage TensorFlow's efficient implementation of gradient computations in our joint optimization process. TensorFlow also supports creating sub-DNNs by marking any arbitrary neuron's output as the sub-DNN's output while keeping the input same as the original DNN's input. We use this feature to intercept and record the output of neurons in the intermediate layers of a DNN and compute the corresponding gradients with respect to the DNN's input. All our experiments were run on a Linux laptop running Ubuntu 16.04 (one Intel i7-6700HQ 2.60GHz processor with 4 cores, 16GB of memory, and a NVIDIA GTX 1070 GPU).

6 EXPERIMENTAL SETUP

6.1 Test datasets and DNNs

We adopt five popular public datasets with different types of data—MNIST, ImageNet, Driving, Contagio/VirusTotal, and Drebin—and then evaluate DeepXplore on three DNNs for each dataset (i.e., a total of fifteen DNNs). We provide a summary of the five datasets and the corresponding DNNs in Table 1. All the evaluated DNNs are either pre-trained (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 as described in Table 1.

MNIST [41] is a large handwritten digit dataset containing 28x28 pixel images with class labels from 0 to 9 The dataset includes 60,000 training samples and 10,000 testing samples. We follow Lecun et al. [40] and construct three different neural networks based on the LeNet family [40], i.e., the LeNet-1, LeNet-4, and LeNet-5.

ImageNet [20] is a large image dataset with over 10,000,000 hand-annotated images that are crowdsourced and labeled manually. We test three well-known pre-trained DNNs: VGG-16 [66], VGG-19 [66], and ResNet50 [31]. All three DNNs achieved competitive performance in the ILSVRC [61] competition.

Driving [75] is the Udacity self-driving car challenge dataset that contains images captured by a camera mounted behind the windshield of a driving car and the simultaneous steering wheel angle applied by the human driver for each image. The dataset has 101,396 training and 5,614 testing samples. We then used three DNNs [8, 18, 78] based on the DAVE-2 self-driving car architecture from Nvidia [10] with slightly different configurations, which are called DAVE-orig, DAVE-norminit, and DAVE-dropout respectively. Specifically, DAVE-orig [8] fully replicates the original architecture from the Nvidia's paper [10]. DAVE-norminit [78] removes the first batch normalization layer [33] and normalizes the randomly initialized network weights. DAVE-dropout [18] simplifies DAVE-orig by cutting down the numbers of convolutional layers and fully connected layers. DAVE-dropout also adds two dropout layer [70] between the final three fully-connected layers. We trained all three implementations with the Udacity self-driving car challenge dataset mentioned above.

Contagio/VirusTotal [19, 77] is a dataset containing different benign and malicious PDF documents. We use 5,000 benign and 12,205 malicious PDF documents from Contagio database as the training set, and then use 5,000 malicious PDFs collected by VirusTotal [77] and 5,000 benign PDFs crawled from Google as the test set. To the best of our knowledge, there is no publicly available DNN-based PDF malware detection system. Therefore, we define and train three different DNNs using 135 static features from PDFrate [54, 68], an online service for PDF malware detection. Specifically, we construct neural networks with one input layer, one softmax output layer, and N fully-connected hidden layers with 200 neurons where N ranges from 2 to 4 for the three tested DNNs. All our models achieve similar performance to the ones reported by a prior work using SVM models on the same dataset [79].

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. [40, 42]	98.3%	98.33%
			MNI_C2	148	LeNet-4, LeCun et al. [40, 42]	98.9%	98.59%
			MNI_C3	268	LeNet-5, LeCun et al. [40, 42]	99.05%	98.96%
Imagenet	General images	State-of-the-art image classifiers from ILSVRC	IMG_C1	14,888	VGG-16, Simonyan et al. [66]	92.6%**	92.6%**
			IMG_C2	16,168	VGG-19, Simonyan et al. [66]	92.7%**	92.7%**
			IMG_C3	94,059	ResNet50, He et al. [31]	96.43%**	96.43%**
Driving	Driving video frames	Nvidia DAVE self-driving systems	DRV_C1	1,560	Dave-orig [8], Bojarski et al. [10]	N/A	99.91%#
			DRV_C2	1,560	Dave-norminit [78]	N/A	99.94%#
			DRV_C3	844	Dave-dropout [18]	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. [29]	98.92%	98.6%
			APP_C2	102	<50, 50>+, Grosse et al. [29]	96.79%	96.82%
			APP_C3	212	<200, 10>+, Grosse et al. [29]	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 and so on
- accuracy using SVM as reported by Šrndic et al. [79]

Drebin [7, 69] is a dataset with 129, 013 Android applications among which 123, 453 are benign and 5, 560 are malicious. There is a total of 545, 333 binary features categorized into eight sets including the features captured from manifest files (e.g., requested permissions and intents) and disassembled code (e.g., restricted API calls and network addresses). We adopt the architecture of 3 out of 36 DNNs constructed by Grosse et al. [29]. As the DNNs' weights are not available, we train these three DNNs with 66% randomly picked Android applications from the dataset and use the rest as the test set.

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

The first constraint restricts image modifications so that DeepXplore can only make the image darker or brighter without changing its content. Specifically, the modification can only increase or decrease all pixel values by the same amount (e.g., $1 * stepsize$ in line 14 of Algorithm 1)—the decision to increase or decrease depends on the value of $mean(G)$ where G denotes the gradients calculated at each iteration of gradient ascent. Note that $mean(G)$ simply denotes the mean of all entries in the multi-dimensional array G. The first and second rows of Figure 8 show some examples of the difference-inducing inputs generated by DeepXplore with these constraints.

The second constraint simulates the effect of the camera lens that may be accidentally or deliberately occluded by a single small rectangle R ($m \times n$ pixels). Specifically, we apply only $G_{i:i+m,j:j+n}$ to the original image (I) where $I_{i:i+m,j:j+n}$ is the location of R. Note that DeepXplore is free to choose any values of i and j to place the rectangle R at any arbitrary position within the image. The third and fourth rows of Figure 8 show some examples DeepXplore generated while running with such occlusion constraints.

The third constraint restricts the modifications so that DeepXplore only selects a tiny $m \times m$ size patch, $G_{i:i+m,j:j+m}$, from G with upper-left corner at (i, j) during each iteration of the gradient ascent. If the average value $mean(G_{i:i+m,j:j+m})$ of this patch is greater than 0, we set $G_{i:i+m,j:j+m} = 0$, i.e., we only allow the pixel values to be decreased. Unlike the second constraint described above, here DeepXplore will pick multiple positions (i.e., multiple (i, j) pairs) to place the black rectangles simulating pieces of dirt on the camera lens. The fifth and sixth rows of Figure 8 show some generated examples with these constraints.

Other constraints (Drebin and Contagio/VirusTotal). For Drebin dataset, DeepXplore enforces a constraint that only

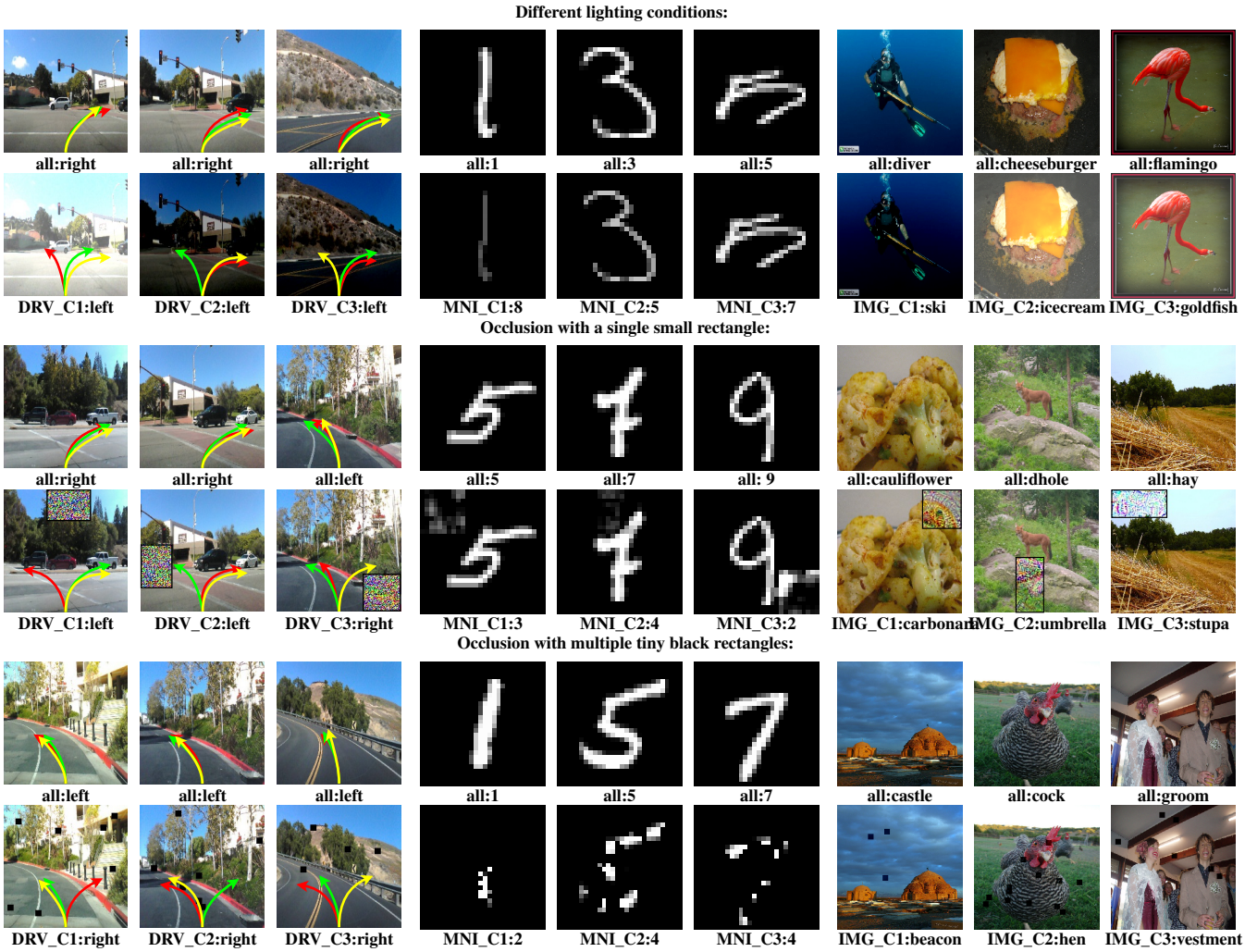


Figure 8: Odd rows show the seed test inputs and even rows show the difference-inducing test inputs generated by DeepXplore. The left three columns show inputs for self-driving car, the middle three are for MNIST, and the right three are for ImageNet.

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 do 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, DeepXplore follows the restrictions on each feature as described by Šrndic et al. [79].

7 RESULTS

Summary. DeepXplore found thousands of erroneous behaviors in all the tested DNNs. Table 2 summarizes the numbers of erroneous behaviors found by DeepXplore for each tested DNN while using 2,000 randomly selected seed inputs from the corresponding test sets. Note that as the testing set has similar number of samples for each class, these randomly-chosen 2,000 samples also follow that distribution. The hyper-parameter values for these experiments, as shown in Table 2, are empirically chosen to maximize both the rate of finding difference-inputs as well as the neuron coverage achieved by these inputs.

For the experimental results shown in Figure 8, we apply three domain-specific constraints (lighting effects, occlusion by a single rectangle, and occlusion by multiple rectangles) as described in § 6.2. 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

Table 2: Number of difference-inducing inputs found by DeepXplore for each tested DNN obtained by randomly selecting 2,000 seeds from the corresponding test set for each run.

DNN name	Hyperparams (Algorithm 1)				# Differences Found
	λ_1	λ_2	s	t	
MNI_C1	1	0.1	10	0	1,073
MNI_C2					1,968
MNI_C3					827
IMG_C1	1	0.1	10	0	1,969
IMG_C2					1,976
IMG_C3					1,996
DRV_C1	1	0.1	10	0	1,720
DRV_C2					1,866
DRV_C3					1,930
PDF_C1	2	0.1	0.1	0	1,103
PDF_C2					789
PDF_C3					1,253
APP_C1	1	0.5	N/A	0	2,000
APP_C2					2,000
APP_C3					2,000

Table 3: The features added to the manifest file by DeepXplore for generating two sample malware inputs which 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 which 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

in § 6.2. We use the hyperparameter values listed in Table 2 in all the experiments unless otherwise specified.

Figure 8 shows some difference-inducing inputs generated by DeepXplore (with different domain-specific constraints) 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.

7.1 Benefits of neuron coverage

In this subsection, we evaluate how effective, neuron coverage, our new metric, is in measuring the comprehensiveness of DNN testing. 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 [58, 87]. Essentially, each neuron tends to learn a different set of rules than others. 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 § 6.1. 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, i.e., a neuron is considered covered only if its output is greater than 0.75 for at least one input.

Note that for the DNNs where the outputs of intermediate layers produce values in a different range than those of the final layers, we scale the neuron outputs to be within $[0, 1]$ by computing $(\boldsymbol{out} - min(\boldsymbol{out})/(max(\boldsymbol{out}) - min(\boldsymbol{out}))$ where $\boldsymbol{out}$ is the vector denoting the output of all neurons of a given layer.

The results, as shown in Table 6, 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 while 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. By contrast, the neuron coverage for the complete Contagio/Virustotal test set reaches 100%.

Effect of neuron coverage on the difference-inducing inputs found by DeepXplore. The primary goal behind maximizing neuron coverage as one of the objectives during the joint optimization process is to generate *diverse* difference-inducing inputs as discussed in § 3. In this experiment, we evaluate the effectiveness of neuron coverage at achieving this goal.

Table 5: The increase in *diversity* (L1-distance) in the difference-inducing inputs found by DeepXplore while using neuron coverage as part of the optimization goal (Equation 2). This experiment uses 2,000 randomly picked seed inputs from the MNIST dataset. Higher values denote larger diversity. NC denotes the neuron coverage (with $t = 0.25$) achieved under each setting.

Exp. #	$\lambda_2 = 0$ (w/o neuron coverage)			$\lambda_2 = 1$ (with neuron coverage)		
	Avg. diversity	*NC*	*# Diffs*	*Avg. diversity*	*NC*	*# Diffs*
1	237.9	69.4%	871	283.3	70.6%	776
2	194.6	66.7%	789	253.2	67.8%	680
3	170.8	68.9%	734	182.7	70.2%	658

Table 6: Comparison of code coverage and neuron coverage for 10 randomly selected inputs from the original test set of each DNN.

Dataset	Code Coverage			Neuron Coverage		
	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%

We randomly pick 2,000 seed inputs from MNIST test dataset and use DeepXplore to generate difference-inducing inputs with and without neuron coverage by setting λ_2 in Equation 2 to 1 and 0 respectively. We measure the *diversity* of the generated difference-inducing inputs in terms of averaged L1 distance between all difference-inducing inputs generated from the same seed and the original seed. The L1-distance calculates the sum of absolute differences of each pixel values between the generated image and the original one. Table 5 shows the results of three such experiments. The results clearly show that neuron coverage helps in increasing the diversity of generated inputs.

Note that even though the absolute value of the increase in neuron coverage achieved by setting $\lambda_2 = 1$ instead of $\lambda_2 = 0$ may seem small (e.g., 1-2 percentage points), it has a significant effect on increasing the diversity of the generated difference-inducing images as shown in Table 5. These results show that increasing neuron coverage, similar to code coverage, becomes increasingly harder for higher values but even small increases in neuron coverage can improve the test diversity significantly. Also, the numbers of difference-inducing inputs generated with $\lambda_2 = 1$ are less than those for $\lambda_2 = 0$ as setting $\lambda_2 = 1$ causes DeepXplore to focus on finding diverse differences rather than simply increasing the number of differences with the same underlying root cause. In general, the number of difference-inducing inputs alone is a not a good metric for measuring the quality of the generated tests for vision-related tasks as one can create a large number of difference-inducing images with the same root cause by making tiny changes to an existing difference-inducing image.

Table 7: 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

Activation of neurons for different classes of inputs. In this experiment, 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 7 shows the results, which confirm 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.

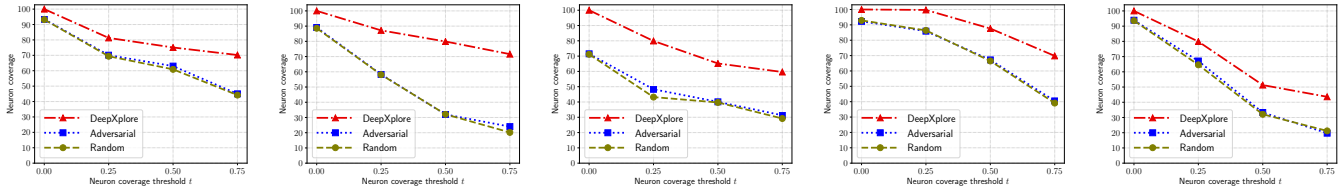
7.2 Performance

We evaluate DeepXplore's performance using two metrics: neuron coverage of the generated tests and execution time for generating difference-inducing inputs.

Neuron coverage. In this experiment, we compare the neuron coverage achieved by the same number of tests generated by three different approaches: (1) DeepXplore, (2) adversarial testing [26], and (3) random selection from the original test set. The results are shown in Table 8 and Figure 9.

We can make two key observations from the results. First, DeepXplore, on average, covers 34.4% and 33.2% more neurons than random testing and adversarial testing as demonstrated in Figure 9. Second, the neuron coverage threshold t (defined in § 4), which decides when a neuron has been activated, greatly affects the achieved neuron coverage. As the threshold t increases, all three approaches cover fewer neurons. This is intuitive as a higher value of t makes it increasingly harder to activate neurons using simple modifications.

Execution time and number of seed inputs. For this experiment, we measure the execution time of DeepXplore to generate difference-inducing inputs with 100% neuron coverage for all the tested DNNs. We note that some neurons in fully-connected layers of DNNs on MNIST, ImageNet and Driving are very hard to activate, we thus only consider neuron coverage on layers except fully-connected layers. Table 8 shows the results, which indicate that DeepXplore is very efficient in terms of finding difference-inducing inputs as well as increasing neuron coverage.



(a) MNIST (b) ImageNet (c) Driving (d) VirusTotal (e) Drebin

Figure 9: The neuron coverage achieved by the same number of inputs (1% of the original test set) produced by DeepXplore, adversarial testing [26], and random selection from the original test set. The plots show the changes in neuron coverage for all three methods as the threshold t (defined in § 4) increases. DeepXplore, on average, covers 34.4% and 33.2% more neurons than random testing and adversarial testing.

Table 8: Total time taken by DeepXplore to achieve 100% neuron coverage for different DNNs averaged over 10 runs. The last column shows the number of seed inputs.

	C1	C2	C3	# seeds
MNIST	6.6 s	6.8 s	7.6 s	9
ImageNet	43.6 s	45.3 s	42.7 s	35
Driving	11.7 s	12.3 s	9.8 s	12
VirusTotal	31.1 s	29.7 s	23.2 s	6
Drebin	180.2 s	196.4 s	152.9 s	16

Table 9: The variation in DeepXplore runtime (in seconds) while generating the first difference-inducing input for the tested DNNs with different step size choice (s for gradient ascent shown in Algorithm 1 line 14). All numbers averaged over 10 runs. The fastest times for each dataset is highlighted in gray.

	s=0.01	*s=0.1*	*s=1*	*s=10*	*s=100*
MNIST	0.19 s	0.26 s	0.65 s	0.62 s	0.65 s
ImageNet	9.3 s	4.78 s	1.7 s	1.06 s	6.66 s
Driving	0.51 s	0.5 s	0.53 s	0.55 s	0.54 s
VirusTotal	0.17 s	0.16 s	0.21 s	0.21 s	0.22 s
Drebin	7.65 s	7.65 s	7.65 s	7.65 s	7.65 s

Different choices of hyperparameters. We further evaluate how the choices of different hyperparameters of DeepXplore (s, λ_1, λ_2, and t as described in § 4.2) influence DeepXplore's performance. The effects of changing neuron activation threshold t was shown in Figure 9 as described earlier. Tables 9, 10, and 11 show the variations in DeepXplore runtime with changes in s, λ_1, and λ_2 respectively. Our results show that the optimal values of s and λ_1 vary across the DNNs and datasets, while $\lambda_2 = 0.5$ tend to be optimal for all the datasets.

We use the time taken by DeepXplore to find the first difference-inducing input as the metric for comparing different choices of hyperparameters in Tables 9, 10, and 11. We choose this metric as we observe that finding the first difference-inducing input for a given seed tend to be significantly harder than increasing the number of difference-inducing inputs.

Testing very similar models with DeepXplore. Note that while DeepXplore's gradient-guided test generation process works very well in practice, it may fail to find any difference-inducing inputs within a reasonable time for some cases especially for DNNs with very similar decision boundaries. To estimate how similar two DNNs have to be in order to make DeepXplore to fail in practice, we control three types of differences between two DNNs and measure the changes in iterations required to generate the first difference-inducing inputs in each case.

We use MNIST training set (60,000 samples) and LeNet-1 trained with 10 epochs as the control group. We change the (1) number of training samples, (2) number of filters per convolutional layer, and (3) number of training epochs respectively to create variants of LeNet-1. Table 12 summarizes the averaged number of iterations (over 100 seed inputs) needed by DeepXplore to find the first difference inducing inputs between these LeNet-1 variants and the original version. Overall, we

Table 10: The variation in DeepXplore runtime (in seconds) while generating the first difference-inducing input for the tested DNNs with different λ_1, a parameter in Equation 2. Higher λ_1 values indicate prioritization of minimizing a DNNs' outputs over maximizing the outputs of other DNNs showing differential behavior. The fastest times for each dataset is highlighted in gray.

	$\lambda_1 = 0.5$	$\lambda_1 = 1$	$\lambda_1 = 2$	$\lambda_1 = 3$
MNIST	0.28 s	0.25 s	0.2 s	0.17 s
ImageNet	1.38 s	1.26 s	1.21 s	1.72 s
Driving	0.62 s	0.59 s	0.57 s	0.58 s
VirusTotal	0.13 s	0.12 s	0.05 s	0.09 s
Drebin	6.4 s	5.8 s	6.12 s	7.5 s

Table 11: The variation in DeepXplore runtime (in seconds) while generating the first difference-inducing input for the tested DNNs with different λ_2, a parameter in Equation 3. Higher λ_2 values indicate higher priority for increasing coverage. All numbers averaged over 10 runs. The fastest times for each dataset is highlighted in gray.

	$\lambda_2 = 0.5$	$\lambda_2 = 1$	$\lambda_2 = 2$	$\lambda_2 = 3$
MNIST	0.19 s	0.22 s	0.23 s	0.26 s
ImageNet	1.55 s	1.6 s	1.62 s	1.85 s
Driving	0.56 s	0.61 s	0.67 s	0.74 s
VirusTotal	0.09 s	0.15 s	0.21 s	0.25 s
Drebin	6.14 s	6.75 s	6.82 s	6.83 s

Table 12: Changes in the number of iterations DeepXplore takes, on average, to find the first difference inducing inputs as the type and numbers of differences between the test DNNs increase.

Training Samples	*# diff*	*0*	1	*100*	*1000*	*10000*
	# iter	-*	-*	616.1	503.7	256.9
Neurons per layer	*# diff*	*0*	1	*2*	*3*	*4*
	# iter	-*	69.6	53.9	33.1	18.7
Training Epochs	*# diff*	*0*	5	*10*	*20*	*40*
	# iter	-*	453.8	433.9	348.7	210

*- indicates timeout after 1000 iterations

find that DeepXplore is very good at finding differences even between DNNs with minute variations (only failing once as shown in Table 12). As the number of differences goes down, the number of iterations to find a difference-inducing input goes up, i.e., it gets increasingly harder to find difference-inducing tests between DNNs with smaller differences.

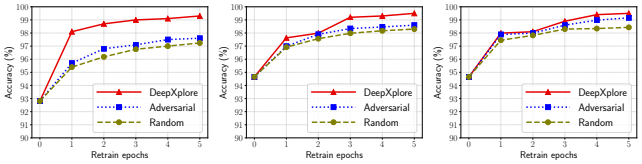
7.3 Improving DNNs with DeepXplore

In this section, we demonstrate two additional applications of the error-inducing inputs generated by DeepXplore: augmenting training set and then improve DNN's accuracy and detecting potentially corrupted training data.

Augmenting training data to improve accuracy. We augment the original training data of a DNN with the error-inducing inputs generated by DeepXplore for retraining the DNN to fix the erroneous behaviors and therefore improve its accuracy. Note that such strategy has also been adopted for fixing a DNN's behavior for adversarial inputs [26]—but the key difference is that adversarial testing requires manual labeling while DeepXplore can adopt majority voting [23] to automatically generate labels for the generated test inputs. Note that the underlying assumption is that the decision made by the majority of the DNNs are more likely to be correct.

To evaluate this approach, we train LeNet-1, LeNet-4, and LeNet-5 as shown in Table 1 with 60,000 original samples. We further augment the training data by adding 100 new error-inducing samples and retrain the DNNs by 5 epochs. Our experiment results—comparing three approaches, i.e., random selection ("random"), adversarial testing ("adversarial") and DeepXplore—are shown in Figure 10. The results show that DeepXplore achieved 1–3% more average accuracy improvement over adversarial and random augmentation.

Detecting training data pollution attack. As another application of DeepXplore, we demonstrate how it can be used to detect training data pollution attacks with an experiment on two LeNet-5 DNNs: one trained on 60,000 hand-written digits from MNIST dataset and the other trained on an artificially polluted version of the same dataset where 30% of the images originally labeled as digit 9 are mislabeled as 1. We use DeepXplore to generate error-inducing inputs that are classified as the digit 9 and 1 by the unpolluted and polluted



(a) LeNet-1 (b) LeNet-4 (c) LeNet-5

Figure 10: Improvement in accuracy of three LeNet DNNs when the training set is augmented with the same number of inputs generated by random selection ("random"), adversarial testing ("adversarial") [26], and DeepXplore.

versions of the LeNet-5 DNN respectively. We then search for samples in the training set that are closest to the inputs generated by DeepXplore in terms of structural similarity [80] and identify them as polluted data. Using this process, we are able to correctly identify 95.6% of the polluted samples.

8 DISCUSSION

Causes of differences between DNNs. The underlying root cause behind prediction differences between two DNNs for the same input is differences in their decision logic/boundaries. As described in § 2.1, a DNN's decision logic is determined by multiple factors including training data, the DNN architecture, hyperparameters, etc. Therefore, any differences in the choice of these factors will result in subtle changes in the decision logic of the resulting DNN. As we empirically demonstrated in Table 12, the more similar the decision boundaries of two DNNs are, the harder it is to find difference-inducing inputs. However, all the real-world DNNs that we tested tend to have significant differences and therefore DeepXplore can efficiently find erroneous behaviors in all of the tested DNNs.

Overhead of training vs. testing DNNs. There is a significant performance asymmetry between the prediction/gradient computation and training of large real-world DNNs. For example, training a state-of-the-art DNN like VGG-16 [66] (one of the tested DNNs in this paper) on 1.2 million images in ImageNet dataset [61] competitions) can take up to 7 days on a single GTX 1080 Ti GPU. By contrast, the prediction and gradient computations on the same GPU take around 120 milliseconds in total per image. Such massive performance difference between training and prediction for large DNNs make DeepXplore especially suitable for testing large, pre-trained DNNs.

Limitations. DeepXplore adopts the technique of differential testing from software analysis and thus inherits the limitations of differential testing. We summarize them briefly below.

First, differential testing requires at least two different DNNs with the same functionality. Further, if two DNNs only differ slightly (i.e., by a few neurons), DeepXplore will take longer to find difference-inducing inputs than if the DNNs

were significantly different from each other as shown in Table 12. However, our evaluation shows that in most cases multiple different DNNs, for a given problem, are easily available as developers often define and train their own DNNs for customization and improved accuracy.

Second, differential testing can only detect an erroneous behavior if at least one DNN produces different results than other DNNs. If all the tested DNNs make the same mistake, DeepXplore cannot generate the corresponding test case. However, we found this to be not a significant issue in practice as most DNNs are independently constructed and trained, the odds of all of them making the same mistake is low.

9 RELATED WORK

Adversarial deep learning. Recently, the security and privacy aspects of machine learning have drawn significant attention from the researchers in both machine learning [26, 49, 72] and security [12, 21, 22, 55, 63, 74, 82] communities. Many of these works have demonstrated that a DNN can be fooled by applying minute perturbations to an input image, which was originally classified correctly by the DNN, even though the modified image looks visibly indistinguishable from the original image to the human eye.

Adversarial images demonstrate a particular type of erroneous behaviors of DNNs. However, they suffer from two major limitations: (1) they have low neuron coverage (similar to the randomly selected test inputs as shown in Figure 9) and therefore, unlike DeepXplore, can not expose different types of erroneous behaviors; and (2) the adversarial image generation process is inherently limited to only use the tiny, undetectable perturbations as any visible change will require manual inspection. DeepXplore bypasses this issue by using differential testing and therefore can perturb inputs to create many realistic visible differences (e.g., different lighting, occlusion, etc.) and automatically detect erroneous behaviors of DNNs under these circumstances.

Testing and verification of DNNs. Traditional practices in evaluating machine learning systems primarily measure their accuracy on randomly drawn test inputs from manually labeled datasets [81]. Some machine learning systems like autonomous vehicles leverage ad hoc unguided simulations [2, 4]. However, without the knowledge of the model's internals, such blackbox testing paradigms are not able to find different corner cases that induce erroneous behaviors [25]. This observation has inspired several researchers to try to improve the robustness and reliability of DNNs [9, 13, 17, 30, 32, 46, 51, 53, 62, 84, 89, 90]. However, all of these projects only focus on adversarial inputs and rely on the ground truth labels provided manually. By contrast, our technique can systematically test the robustness and reliability of DL systems for a broad range of flaws in a fully automated manner without any manual labeling.

Another recent line of work has explored the possibility of formally verifying DNNs against different safety properties [32, 37, 57]. None of these techniques scale well to find violations of interesting safety properties for real-world DNNs. By contrast, DeepXplore can find interesting erroneous behaviors in large, state-of-the-art DNNs but cannot provide any guarantee about whether a specific DNN satisfies a given safety property.

Other applications of DNN gradients. Gradients have been used in the past for visualizing activation of different intermediate layers of a DNN for tasks like object segmentation [44, 66], artistic style transfer between two images [24, 43, 59], etc. By contrast, in this paper, we apply gradient ascent for solving the joint optimization problem that maximizes both neuron coverage and the number of differential behaviors among tested DNNs.

Differential testing of traditional software. Differential testing has been widely used for successfully testing various types of traditional software including JVMs [14], C compilers [45, 86], SSL/TLS certification validation logic [11, 15, 56, 67], PDF viewers [56], space flight software [28], mobile applications [36], and Web application firewalls [6].

The key advantage of applying differential testing to DNNs over traditional software is that the problem of finding a large number of difference-inducing inputs while simultaneously maximizing neuron coverage can be expressed as a well defined joint optimization problem. Moreover, the gradient of a DNN with respect to the input can be utilized for efficiently solving the optimization problem using gradient ascent.

10 CONCLUSION

We designed and implemented DeepXplore, the first whitebox system for systematically testing DL systems and automatically identify erroneous behaviors without manual labels. 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 problem that maximizes both neuron coverage and the number of potentially erroneous behaviors. DeepXplore was able to find thousands of erroneous behaviors in fifteen state-of-the-art DNNs trained on five real-world datasets.

ACKNOWLEDGMENTS

We would like to thank Byung-Gon Chun (our shepherd), Yoav Hollander, Daniel Hsu, Murat Demirbas, Dave Evans, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CNS-16-17670, CNS-15-63843, and CNS-15-64055; ONR grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788; and a Google Faculty Fellowship.

REFERENCES

[1] 2010. ImageNet crowdsourcing, benchmarking & other cool things. http://www.image-net.org/papers/ImageNet$_2$010.pdf. (2010).

[2] 2016. Google auto Waymo disengagement report for autonomous driving. https://www.dmv.ca.gov/portal/wcm/connect/946b3502-c959-4e3b-b119-91319c27788f/GoogleAutoWaymo$_d$isengage$_r$eport$_2$016.pdf?MOD=AJPERES. (2016).

[3] 2016. Report on autonomous mode disengagements for waymo self-driving vehicles in california. https://www.dmv.ca.gov/portal/wcm/connect/946b3502-c959-4e3b-b119-91319c27788f/GoogleAutoWaymo$_d$isengage$_r$eport$_2$016.pdf?MOD=AJPERES. (2016).

[4] 2017. Inside Waymo's secret world for training self-driving cars. https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/. (2017).

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*.

[6] George Argyros, Ioannis Stais, Suman Jana, Angelos D Keromytis, and Aggelos Kiayias. 2016. SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*.

[8] autopilot:dave 2016. Nvidia-Autopilot-Keras. https://github.com/0bserver07/Nvidia-Autopilot-Keras. (2016).

[9] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring neural net robustness with constraints. In *Proceedings of the 29th Advances in Neural Information Processing Systems*.

[10] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[11] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[12] Yinzhi Cao and Junfeng Yang. 2015. Towards Making Systems Forget with Machine Unlearning. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[13] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*.

[14] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[15] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*.

[16] François Chollet. 2015. Keras. https://github.com/fchollet/keras. (2015).

[17] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. 2017. Parseval networks: Improving robustness to adversarial examples. In *Proceedings of the 34th International Conference on Machine Learning*.

[18] clone:dave 2016. Behavioral cloning: end-to-end learning for self-driving cars. https://github.com/navoshta/behavioral-cloning. (2016).

[19] contagio 2010. Contagio, PDF malware dump. http://contagiodump.blogspot.de/2010/08/malicious-documents-archive-for.html. (2010).

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 22nd IEEE Conference on Computer Vision and Pattern Recognition*.

[21] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[22] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing.. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*.

[23] Yoav Freund and Robert E Schapire. 1995. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*.

[24] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2015. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576* (2015).

[25] Ian Goodfellow and Nicolas Papernot. 2017. The challenge of verification and testing of machine learning. http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html. (2017).

[26] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of the 3rd International Conference on Learning Representations*. http://arxiv.org/abs/1412.6572

[27] google-accident 2016. A Google self-driving car caused a crash for the first time. http://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report. (2016).

[28] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th international conference on Software Engineering*.

[29] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2016. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435* (2016).

[30] Shixiang Gu and Luca Rigazio. 2015. Towards deep neural network architectures robust to adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations*.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[32] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *Proceedings of the 29th International Conference on Computer Aided Verification*.

[33] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

[35] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 35th IEEE/AIAA Digital Avionics Systems Conference*.

[36] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*.

[37] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proceedings of the 29th International Conference On Computer Aided Verification*.

[38] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[41] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. (1998).

[42] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[43] Chuan Li and Michael Wand. 2016. Combining markov random fields and convolutional neural networks for image synthesis. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*.

[44] Aravindh Mahendran and Andrea Vedaldi. 2015. Understanding deep image representations by inverting them. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition*.

[45] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* (1998).

[46] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. 2017. On detecting adversarial perturbations. In *Proceedings of the 6th International Conference on Learning Representations*.

[47] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* (1995).

[48] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*. 807–814.

[49] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition*.

[50] Nvidia. 2008. CUDA Programming guide. (2008).

[51] Nicolas Papernot and Patrick McDaniel. 2017. Extending defensive distillation. *arXiv preprint arXiv:1705.05264* (2017).

[52] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Proceedings of the 37th IEEE European Symposium on Security and Privacy*.

[53] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*.

[54] pdfrate 2012. PDFRate, A machine learning based classifier operating on document metadata and structure. http://pdfrate.com/. (2012).

[55] Roberto Perdisci, David Dagon, Wenke Lee, P. Fogla, and M. Sharif. 2006. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*.

[56] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*.

[57] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification*.

[58] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. 2017. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444* (2017).

[59] Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. 2016. Artistic style transfer for videos. In *German Conference on Pattern Recognition*.

[60] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1988. Learning representations by back-propagating errors. *Cognitive modeling* (1988).

[61] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* (2015).

[62] Uri Shaham, Yutaro Yamada, and Sahand Negahban. 2015. Understanding adversarial training: Increasing local stability of neural nets through robust optimization. *arXiv preprint arXiv:1511.05432* (2015).

[63] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*.

[64] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* (2016).

[65] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[66] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[67] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. 2017. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*. San Jose, CA.

[68] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*.

[69] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*.

[70] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* (2014).

[71] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition*.

[72] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *Proceedings of the 2nd International Conference on Learning Representations*.

[73] tesla-accident 2016. Understanding the fatal Tesla accident on Autopilot and the NHTSA probe. https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/. (2016).

[74] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction APIs. In *Proceedings of the 25th USENIX Security Symposium*.

[75] udacity-challenge 2016. Using Deep Learning to Predict Steering Angles. https://github.com/udacity/self-driving-car. (2016).

[76] Vladimir Naumovich Vapnik. 1998. *Statistical learning theory*.

[77] virustotal 2004. VirusTotal, a free service that analyzes suspicious files and URLs and facilitates the quick detection of viruses, worms, trojans, and all kinds of malware. https://www.virustotal.com/. (2004).

[78] visualize:dave 2016. Visualizations for understanding the regressed wheel steering angle for self driving cars. https://github.com/jacobgil/keras-steering-angle-visualizations. (2016).

[79] Nedim Šrndic and Pavel Laskov. 2014. Practical evasion of a learning-based classifier: a case study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*.

[80] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* (2004).

[81] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[82] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F Naughton. 2016. A Methodology for Formalizing Model-Inversion Attacks. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium*.

[83] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2016. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256* (2016).

[84] Weilin Xu, David Evans, and Yanjun Qi. 2017. Feature squeezing: detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[85] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically evading classifiers. In *Proceedings of the 23rd Network and Distributed Systems Symposium*.

[86] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*.

[87] Jason Yosinski, Jeff Clune, Thomas Fuchs, and Hod Lipson. 2015. Understanding neural networks through deep visualization. In *2015 ICML Workshop on Deep Learning*.

[88] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*.

[89] Yuqian Zhang, Cun Mu, Han-Wen Kuo, and John Wright. 2013. Toward guaranteed illumination models for non-convex objects. In *Proceedings of the 26th IEEE International Conference on Computer Vision*. 937–944.

[90] Stephan Zheng, Yang Song, Thomas Leung, and Ian Goodfellow. 2016. Improving the robustness of deep neural networks via stability training. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*. 4480–4488.