Correlations between Deep Neural Network Model Coverage Criteria and Model Quality

Shenao Yan
shenao.yan@rutgers.edu
Rutgers University
USA
Juan Zhai
juan.zhai@rutgers.edu
Rutgers University
USA
Guanhong Tao
tao@purdue.edu
Purdue University
USA
Shiqing Ma
shiqing.ma@rutgers.edu
Rutgers University
USA
Xuwei Liu
liu2598@purdue.edu
Purdue University
USA
Lei Xu
xlei@nju.edu.cn
Nanjing University
China
Xiangyu Zhang
xyzhang@cs.purdue.edu
Purdue University
USA

ABSTRACT
Inspired by the great success of using code coverage as guidance in software testing, a lot of neural network coverage criteria have been proposed to guide testing of neural network models (e.g., model accuracy under adversarial attacks). However, while the monotonic relation between code coverage and software quality has been supported by many seminal studies in software engineering, it remains largely unclear whether similar monotonicity exists between neural network model coverage and model quality. This paper sets out to answer this question. Specifically, this paper studies the correlation between DNN model quality and coverage criteria, effects of coverage guided adversarial example generation compared with gradient decent based methods, effectiveness of coverage based retraining compared with existing adversarial training, and the internal relationships among coverage criteria.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging. • Computing methodologies → Neural networks.

KEYWORDS
Software Testing, Deep Neural Networks

ACM Reference Format:

1 INTRODUCTION
Deep Neural Network (DNN) is becoming an integral part of the new generation of software systems, such as self-driving vehicle systems, computer vision systems, and various kinds of bot systems. Just like software testing is a key step in traditional software development life-cycle, many researchers and practitioners believe that DNN model testing is critical to model quality and hence the whole system quality [39, 47, 59, 71]. Software testing can be classified as black-box testing and white-box testing, with the former generating test cases from the specification without looking into the implementation whereas the later generating test cases based on the implementation. Specifically, white-box testing aims to generate test cases to improve code coverage. High code coverage provides more confidence about the subject software’s quality. It is widely used in practice because progress can be easily quantified and test generation is more amenable to automation (compared to black-box testing). The upper half of Figure 1 shows a typical life-cycle of software testing. Given a subject software, various test generation engines, such as random input generation, fuzzing, symbolic execution, search based test generation, can be used to generate test cases. The generated test suite is executed and the code coverage is measured and provided as feed-back to the test generation engine, whose goal is hence to generate more test cases that can improve coverage. The failing test cases are reported to the developers who fix the corresponding faults/bugs/defects, leading to a new version of the subject software.

Inspired by the great success of white-box software testing, researchers have proposed white-box DNN testing to improve model quality [15, 39, 47, 59, 71, 79]. The life-cycle of DNN model testing closely resembles that of software testing, as shown in the lower half of Figure 1. Specifically, the subject becomes a DNN model instead of a program. Model input generation techniques are used to generate inputs. The generation is guided by some coverage criterion just like in software test generation. A failing input example
is the one that causes model mis-classification and can be used to retrain the model. The retraining procedure is analogous to the bug fixing procedure in the software testing life-cycle. It yields a new version of the model.

Code coverage criteria play a critical role in software (white-box) testing. A large number of coverage criteria have been proposed and used. For example, statement coverage measures the percentage of statements that are executed by a test suite; edge coverage measures the percentage of exercised control flow edges; and path coverage measures the percentage of program paths that get executed. Different criteria have different levels of strength (in disclosing bugs) and entail various amount of efforts. Specifically, statement coverage is the simplest and also the weakest, whereas path coverage is one of the most expensive and most powerful. These criteria provide a spectrum of options for developers when they are balancing development cost and product quality. Inspired by these code coverage criteria, researchers have proposed a large set of DNN model coverage criteria [32, 39, 47, 71]. Specifically, a DNN model consists of an input layer, an output layer, and a number of inner layers, each containing a set of neurons. During model inference/prediction, neuron activation values at a layer are computed based on those from the previous layer. During the activation value computation for a neuron, if the value is smaller than 0, it is set to 0 and hence has 0 contribution to the later layer(s). We say the neuron is not activated. Researchers observe the analogy between activating a neuron and covering a software artifact (e.g., statement). Therefore, they propose a number of coverage criteria based on how neurons are activated. For example, neuron coverage [47] measures the percentage of neurons that are activated, analogous to statement coverage; and neuron pattern coverage [39, 71] measures the activation path, analogous to path coverage.

Despite the inspiring correspondence between software testing and DNN model testing, there are a few open questions that need to be answered. Specifically, the effectiveness of white-box software testing is built on a basic assumption for the relation between code coverage and software quality. Specifically, while the developers generate more test cases to achieve higher code coverage, more bugs are disclosed and fixed and hence the software quality is monotonically improved (without considering regression). As such, coverage driven test generation is one of the most popular test generation strategies. While such monotonicity assumption is largely proved (empirically) in software testing, its counter-part in model testing is unclear (as far as we know). In fact, the semantics of DNN models is substantially different from the semantics of programs. The syntactic analogy between software statements and neurons may not directly translate to their semantic analogy. Intuitively, program behaviors are largely discrete whereas model behaviors are continuous. A statement being covered suggests that new functionality is exercised, which is a discrete and modular event. However, a neuron being activated may not have similar implication. It may well be that the semantics of a model lies in the distribution of the entire activation vector instead of a set of discrete events of whether individual neurons are activated.

Therefore, this paper aims to study the following research questions that we believe are important for white-box DNN model testing. First, we want to study if there is monotonic relation between model coverage improvement and model quality: is it true that training sets with increasing model coverage lead to increasing model quality. Second, we want to study if coverage driven test generation is effective in model testing, when compared to existing test generation techniques that are solely based on optimization and leverage continuity and differentiability of model behaviors. Third, we want to study if test cases generated using model coverage criteria have unique advantage in improving model quality (analogous to bug fixing). Last, we aim to understand if there is correlation in the various model coverage criteria.

In our study, we make use of 8 models and 3 datasets. We leverage model coverage based test generation techniques in DeepXplore [47], DeepGauge [39], DeepHunter [71] and SADL [32], and the state-of-the-art optimization based adversarial example generation techniques C&W [11] and PGD [41], to generate test cases that lead to different levels of coverage. These test cases are used to retrain models. Then we measure the quality improvement using a set of well-established metrics and study the aforementioned four research questions. Through our study, we find:

- DNN coverage criteria do not have monotonic relations with model quality (measured by model accuracy in the presence of adversarial examples).
- Although DNN coverage criteria can be used as guidance to find adversarial examples, effective adversarial examples may not lead to higher coverage.
- Existing methods used to generate adversarial examples based on coverage criteria usually add larger (i.e., measured by $\ell_p$ distance) and human visible (i.e., measured by visual similarity) perturbations, compared to existing gradient based methods. Using such inputs in model testing is analogous to having program inputs that may violate the software input preconditions.
- Adversarial examples generated by DNN coverage guided methods can be used to retrain a model to improve model robustness against the adversarial method used to generate the training inputs (e.g., performing semantic preserving operations like changing blurriness to maximize coverage). However, such models are not robust against gradient based attacks (e.g., PGD). On the other hand, PGD based adversarial training can improve the model robustness against PGD attacks but not attacks by using coverage as guidance.
Most existing DNN coverage criteria correlate with each other, some having strong correlations. This helps explain their similar behaviors in all experiments. However, it is unclear whether there exists partial order among them like code coverage criteria.

2 BACKGROUND

2.1 Deep Neural Network

A deep neural network (DNN) is a parameterized function $F$ that maps an $n$-dimensional input $x \in \mathbb{R}^n$ to one of the $k$ output classes. The output of the DNN $F \in \mathbb{R}^k$ is a probability distribution over the $k$ classes. In particular, $P(x)_j$ is the probability of the input belonging to class $j$. An input $x$ is deemed as class $j$ with the highest probability such that the output class label $y$ has $y = \arg\max_{j \in [1,k]} P(x)_j$.

During training, with the assistance of a training dataset of inputs with known ground-truth labels, the parameters including weights and bias of the DNN model are determined. Specifically, given a learning task, suppose the training dataset is a set, $D_{\text{train}} = \{x_i, y_i\}_{i=1}^N$, of $N$ input samples $x = \{x_1, x_2, ..., x_N\} \in \mathbb{R}^n$ and the corresponding ground-truth labels $y = \{y_1, y_2, ..., y_N\} \in [1,k]$. $F$ is the deep learning model that predicts the corresponding outcomes $y'$ based on the given input $x$, i.e., $y' = F(x)$. Within the course of model training, there is a loss function $L = \sum_{i=1}^{N} ||y'_i - y_i||^2$. So the process of model training can be formalized as:

$$\min_{1 \leq i \leq N} \sum_{1 \leq i \leq N} ||y'_i - y_i||^2$$

For a neuron $o$, if it is activated (i.e., activation value is larger than some threshold value) by some input examples (in a set), it becomes an activated neuron for the set. When we provide the training dataset to the model, the range of the observed neuron activation values is represented as $[\text{low}_o, \text{high}_o]$.

When we provide the test dataset $T$, the neuron activation values may not be limited in $[\text{low}_o, \text{high}_o]$. Instead, the values can also fall in $(-\infty, \text{low}_o)$ or $(\text{high}_o, +\infty)$. We refer to $(-\infty, \text{low}_o) \cup (\text{high}_o, +\infty)$ as the corner case regions (of the neuron). Let $\phi(x, o)$ be the output value of neuron $o$ for input $x$, then $\text{UpperCornerNeuron (UCN)}$ and $\text{LowerCornerNeuron (LCN)}$, which represent the set of neurons that ever fall into the corner case regions, respectively, given some test inputs. Formally,

$$\text{UCN} = \{o \in O \mid \exists x \in T : \phi(x, o) \in (\text{high}_o, +\infty)\}$$

$$\text{LCN} = \{o \in O \mid \exists x \in T : \phi(x, o) \in (-\infty, \text{low}_o)\}$$

The symbols we use in this paper are shown as follows:

**LIST OF SYMBOLS**

- $F$: A DL classifier on $k$ classes, where $F(x_i) = y_i$
- $\text{AN}$: Activated neuron
- $\text{UCN}$: Set of activated neurons that fall in the corner-case regions
- $N$: The number of samples in input dataset
- $O$: Universal set of neurons of a DNN
- $\phi$: SoftMax layer output of $F$, where $F(x) = \arg\max P(x)_j$
- $P(x)_j$: The $j$-th probability of $P(x)$, where $j \in [1,...,k]$
- $\text{PNL}$: Set of activated neurons that fall in the corner-case regions

2.2 Adversarial Examples and DNN Robustness

DNN models are vulnerable to adversarial examples. That is, given an original input $x_i$ and a small adversarial perturbation $\delta$, a DNN model $F$ has:

$$F(x_i') = F(x_i + \delta) = y_i + \delta = F(x_i)$$

Here, we use $x_i'$ to represent $x_i + \delta$, which is usually referred to as an adversarial example. The perturbation $\delta$ added to the input is maliciously manipulated by an adversary and is commonly bounded by $\ell_p$-norm, i.e., $||\delta||_p < \epsilon$. Symbol $y_i$ denotes the predicted label of adversarial example $x_i'$ different from the original prediction $y_i$. There exists a large body of different adversarial attacks. We discuss two widely used attacks in this paper and employ them in our experiment evaluation.

**CW attack** is a set of powerful attacks based on different norm measurements on the magnitude of perturbations introduced by Carlini and Wagner [11]. In particular, CW is formalized as an optimization problem to search for high confidence adversarial examples with small magnitude of perturbations. They leverage the logits $\mathbf{Z}(\cdot)$ (i.e., outputs right before the softmax layer) instead of the final prediction $F(\cdot)$ for generating perturbation. The objective function for optimization is the combination of target label $(|F(x_i') - y_i|)$ and small perturbation $(||\delta||[\ell_p])$, which is achieved using optimizer such as Adam.

**PGD attack** is a first-order universal adversarial attack based on Fast Gradient Sign Method (FGSM) [57]. FGSM performs a single update on the original sample $x$ along the direction of the gradient of a loss function. The loss function is usually defined as the cross-entropy between the output of a network and the true label $y$. PGD [41] is an iterative variant of FGSM, which applies the projected gradient descent algorithm with random starts to FGSM. That is, for each attack iteration, given an input $x_i$, it first adds a small random perturbation within given bound $||\delta||_{\ell_p} < \epsilon$ to the input, i.e., $x_i' = x_i + \epsilon$. Then it performs $\tau$ steps of FGSM and applies the gradient $dx$ to the input, i.e., $x_i'' = x_i' + dx$. The updated sample is subsequently projected to the original bound, i.e., $x_i'' = \text{clip}(x_i', x_i + \epsilon)$. The clip(·) function sets small out-of-bound values to $x_i - \epsilon$ and large ones to $x_i + \epsilon$. The process continues until an adversarial example is generated or time-out.

2.3 Adversarial Training

Adversarial training, introduced by Goodfellow et al. [21] is one of the most effective ways to improve DNN model robustness. The overarching idea of adversarial training is to incorporate adversarial examples for model training. That is, during each training iteration, adversarial examples are first generated against the current state of the model, and then used as training data for optimizing model parameters. Madry et al. [41] leverage the PGD attack with multiple steps for adversarial training. Adversarial training has been shown effective for large scale dataset such as ImageNet [34].
3 COVERAGE BASED DNN TESTING

In this section, we first introduce a number of popular neuron coverage criteria and discuss their intended usage.

3.1 DeepXplore

Pei et al. [47] introduced the neuron coverage (NC):

\[ NC = \frac{|AN|}{|O|} \]

where \(|AN|\) denotes the number of activated neurons and \(|O|\) means the total number of neurons. Basically, NC measures the percentage of activated neurons (i.e., whose activation value is larger than 0) for a given test suite and DNN model. DeepXplore views NC as the first white-box testing metric for DL systems that can estimate the amount of activated neurons (i.e., whose activation value is larger than 0) for a given test suite and DNN model. DeepXplore tries to generate new test inputs that can maximize NC as well as triggering differential behaviors in multiple DL systems that are designed to have similar functionality. These different DNN models are used as cross-reference to avoid manually labeling datasets. The generation process applies three pre-defined image transformations: adding a single black rectangle, changing all pixel values by a certain degree and adding multiple small black rectangles, with the goal of covering more neurons. After that, DeepXplore mixes these generated examples with benign inputs to re-train the model to improve model accuracy.

3.2 DeepGauge and DeepHunter

Ma et al. extended the coverage concept to different levels (i.e., neuron level, layer level) and proposed many new DNN coverage based testing criteria in DeepGauge [39], and demonstrated that (a test suite with) a higher coverage of these criteria potentially indicates a higher chance to detect the DNN’s defects. Here, a defect is defined as model mispredictions. DeepHunter [71] leverages such coverage metrics as the feedback to fuzz DNN models to produce adversarial samples. These new metrics include:

- **k-multisection Neuron Coverage (KMNC)**. Given a neuron \(o \in O\), the \(k\)-multisection neuron coverage measures how thoroughly the given set of test inputs \(T\) covers the range \([low_o, high_o]\). To quantify KMNC, the range \([low_o, high_o]\) is divided into \(k\) equal sections (i.e., \(k\)-multisections), with \(k > 0\). Also \(S_m^o\) denotes the \(m\)-th section with \(1 \leq m \leq k\). Then \(\phi(x,o) \in S_m^o\) means the \(m\)-th section is covered by at least one input \(x \in T\).

\[ KMNC = \sum_{o \in O} \left( \frac{\left| S_m^o \right| \times \phi(x,o)}{|O|} \right) \]

- **Neuron Boundary Coverage (NBC)**.

\[ NBC = \frac{|UCN| + |LCN|}{2 \times |O|} \]

From the definition, it’s easy to see that \(NBC\) shows how thoroughly the given set of test inputs \(T\) covers the corner-case regions.

- **Strong Neuron Activation Coverage (SNAC)**.

\[ SNAC = \frac{|UCN|}{|O|} \]

Similar to \(NBC\), \(SNAC\) measures the percentage of upper corner-case regions that are covered by the set of test inputs \(T\).

- **Top-k Neuron Coverage (TKNC)**.

\[ TKNC = \frac{\left| \bigcup_{x \in T} U_{1 \leq l \leq L} top_k(x, l) \right|}{|O|} \]

Here, \(L\) denotes layers of a DNN and \(l \in (1, L)\) is the \(l\)-th layer of a DNN. Function \(top_k(x, l)\) denotes the neurons that have the largest \(k\) outputs on layer \(l\) given \(x\). \(TKNC\) measures the percentage of neurons that have ever been the top \(k\) neurons within its layer for a given input set \(T\).

- **Top-k Neuron Patterns (TKNP)**. Given a test input \(x\), the sequence of the top-\(k\) neurons on each layer also forms a pattern. A pattern is an element of \(2^k \times 2^k \times \cdots \times 2^k\), where \(2^k\) is the set of subsets of the neurons on the \(l\)-th layer, for \(1 \leq l \leq L\). Given a test input set \(T\), the number of top-\(k\) neuron patterns for \(T\) is defined as follows.

\[ TKNP = |\{(top_k(x, 1), ..., (top_k(x, L)) | x \in T)\}| \]

Intuitively, \(TKNP\) measures the number of different activation patterns for the most active \(k\) neurons on each layer. It is not a ratio but rather a number.

DeepHunter is a fuzz testing framework for finding potential DNN defects. It mostly leverages the above coverage criteria as feedback to guide the test generation, and to generate new samples. It performs pixel value transformations (i.e., changing image contrast, brightness, blur and noise) and affine transformations (i.e., image translation, scaling, shearing and rotation). To generate images that preserve its original semantics, a \(\epsilon_o\) based threshold is set. If the difference between the original and the generated image is larger than the threshold, the generated image is discarded.

3.3 SADL

Kim et al. introduced surprise adequacy to measure the coverage of **discredited input surprise range** for DL systems [32]. These terms are explained in the following. A test example is “good” if it is sufficiently but not overly surprising comparing with the training data, that is, sufficiently but not overly deviant from the training distribution. Two measurements of surprise were introduced: one is based on **Kernal Density Estimation** (KDE) to approximate the likelihood effect of the system having seen a similar input during training, and the other is based on the distance between the vectors representing the neuron activation traces of the given input and the training data (e.g., **Euclidean distance**). The proposed metrics were also compared with other metrics in DeepGauge and DeepXplore. The results show that they are correlated. Moreover, they were used to guide the retraining of DNN models to improve robustness.

- **Likelihood-based Surprise Adequacy (LSA)**. Let \(a_o(x)\) denote the activation value of a single neuron \(o\) with respect to an input \(x\). For a set of neurons in a layer of the DNN, denoted as \(O' \subseteq O\), \(a_{O'}(x)\) denotes a vector of activation values that represents the Activation Trace (AT) of \(x\) over neurons in \(O'\). For a set of inputs \(X, a_{O'}(X) = (a_{O'}(x) | x \in X)\) denotes the set of activation traces observed for neurons in \(O'\). Given a training set \(T\), a bandwidth matrix \(H\) and a Gaussian kernal function \(K\), the activation trace of a new input \(x\), KDE produces a density function \(f\) as follows.

\[ f(x) = \frac{1}{|A_{O'}(T)|} \sum_{x_i \in T} K_H(a_{O'}(x) - a_{O'}(x_i)) \]
Intuitively, the function measures a normalized distance between the activation values of \( x \) and those of individual inputs in \( T \) (regarding \( O^t \)). Then LSA is defined as follows.

\[
LSA(x) = \frac{1}{\log(\tilde{f}(x))}
\]

**Distance-based Surprise Adequacy (DSA).** Assume a DL system \( F \), which consists of a set of neurons \( O \), is trained for a classification task with a set of classes \( Y \), using a training dataset \( T \). Given the set of activation traces \( A_O(T) \), a new input \( x \), and a predicted class of the new input \( C \in Y \). The closest neighbor of \( x \) that shares the same output class, denoted as \( x_a \), and their distance, are defined as follows.

\[
x_a = \arg\min_{F(x_i)=y} ||a_O(x) - a_O(x_i)||
\]

\[
dist_a = ||a_O(x) - a_O(x_a)||
\]

The closest neighbor in a class other than \( y \), denoted by \( x_b \), and their distance \( dist_b \), are defined as follows.

\[
x_b = \arg\min_{F(x_i)\in Y \setminus\{y\}} ||a_O(x) - a_O(x_i)||
\]

\[
dist_b = ||a_O(x) - a_O(x_b)||
\]

Then the DSA is defined in the following. Intuitively, it measures if \( x \) is closer to the target class or a different class.

\[
DSA(x) = \frac{dist_a}{dist_b}
\]

### 3.4 Research Questions

Testing is a critical step in software development life-cycle to evaluate software quality, find bugs and help developers to improve the software. In traditional Software Engineering, test coverage criteria (including path coverage, basic block coverage etc.) are a set of metrics used to describe the degree to which the subject software is tested, given a test suite. These criteria are usually computed as the number of some software artifacts (e.g., statements) that are executed by at least one test case, divided by the total number of artifacts. An important hypothesis, which has been proven by numerous seminal studies [6, 16, 17, 30, 33, 35, 42, 43, 45, 67], is that higher test coverage suggests better software quality, given the same subject software. This is because a test suite achieving higher coverage is believed to have a better chance of disclosing defects in the subject software. Different software coverage criteria denote the various trade-offs between the difficulty (to achieve high coverage) and the capability of disclosing defects. For example, statement coverage is the least difficult to achieve with the weakest effectiveness in finding bugs, whereas path coverage is much more difficult to achieve but has stronger bug-finding capabilities. DNN coverage metrics were introduced with the goal of serving deep learning model engineering in a way similar to how software coverage criteria have been serving software engineering. For example, in DeepXplore, the authors believe that "neuron coverage is a good metric for DNN testing comprehensiveness". In DeepGauge [39], the paper states that the proposed criteria can "effectively capture the difference between the original test data and adversarial examples, where DNNs could and could not correctly recognize, respectively, demonstrating that a higher coverage of our criteria potentially indicate a higher chance to detect the DNN’s defects". And in SADL, the paper concludes that "SA can provide guidance for more effective retraining against adversarial examples based on our interpretation of the observed trend". In software testing, the correlations between coverage and software quality are well established, which have been driving decades of practice and research. In this paper, we aim to study if the correlations between DNN coverage metrics and the intended objectives can be established.

Based on the above quoted usage of DNN coverage metrics, we propose the following research questions.

#### 3.4.1 RQ: Are DNN coverage metrics correlated with DNN model robustness?

In software testing, the correlation between test coverage and software quality is intuitively established as follows. Given a subject program and an initial test suite, developers generate new test cases to cover software artifacts that have not been covered before (e.g., new statements). These new test cases may disclose defects in the newly covered components. Fixing these defects leads to the improvement of software quality. To validate the correlations between DNN model robustness and coverage criteria, we draw the following analogies: subject model (in DNN testing) vs. subject program (in software testing); training set vs. initial software test suite; adversarial example generation or GAN based example generation vs. software test generation; misclassifications (caused by adversarial examples) vs. software bugs; and adversarial training vs. software bug fixing. With such analogies, we have the following experiment design.

**Experiment Design:** Assume the subject model is \( F_0 \). We use adversarial example generation techniques to generate a large set of adversarial examples. Given a DNN coverage criterion, we perform input selection to select the examples that can lead to the most substantial coverage improvement, and add them to the training suite \( T_0 \) and acquire \( T_1 \). The process repeats to acquire \( T_2 \), \( T_3 \), ..., and \( T_r \) until the coverage is full or cannot improve any more. Here, \( T_2 \) is a superset of \( T_1 \), \( T_3 \) is a superset of \( T_2 \), and so on. This process is analogous to how the developers enhance their test suite overtime to improve coverage. We then perform adversarial training using \( T_1 \), \( T_2 \), ... and \( T_r \), yielding \( F_1 \), \( F_2 \), ... and \( F_r \), respectively, just like fixing software bugs disclosed by new test inputs. Then we use existing methods [37] to measure model robustness and study the correlations between robustness and coverage. Ideally, we would expect to see these models have increasing levels of robustness.

#### 3.4.2 RQ: Is coverage driven test generation effective in disclosing DNN defects? How does it compare to the other commonly used adversarial example generation techniques?

In software testing, an important functionality of code coverage is to guide test generation. A large body of existing software test generation techniques, such as symbolic/concolic execution [9, 13], fuzzing [7, 73], and search based testing [5, 26], make use of code coverage as the guidance. Analogously, researchers of DeepXplore tries to "generate inputs that maximizing neuron coverage" and believe that "neuron coverage helps in increasing the diversity of generated inputs" [47]. DeepHunter [71] "leverages multiple plausible coverage criteria as feedback to guide the test generation from different perspectives". However, while software behaviors are largely discrete, DNN model behaviors are continuous. As such, there exist highly effective input generation techniques built on optimizations (e.g., based on gradients). While these techniques do not explicitly utilize coverage,
optimization algorithms have the implicit capabilities of exploring different activation patterns by following the direction of gradients. As such, we are interested in studying if coverage guided testing has advantages over the popular gradient descent based methods, in disclosing defects (i.e., generating adversarial examples). More over, gradient based test generation often makes use of \( \ell_p \)-norm to bound the scale of perturbation such that adversarial examples do not look substantially different from the original input (in humans’ eyes). In contrast, existing coverage guided test generation uses predefined image transformations such as adding black rectangles, changing image pixels by a certain degree [47], changing image blurriness and rotating images [71]. See details in subsection 3.1 and subsection 3.2. Notice DeepHunter also uses a \( \ell_\infty \) based threshold value to limit the change of the image. However, because the pre-defined operations usually change the image significantly, this threshold value is larger than what is used in gradient based methods. Also, gradient based methods will try to minimize the change (e.g., \( \ell_p \) distance) while DeepHunter only checks if the value is smaller than the threshold. Therefore, we also want to study the quality of generated examples, in comparison with existing gradient descent based techniques.

**Experiment Design.** To answer RQ 2, we utilize the test generation techniques in DeepHunter [71] (coverage criteria based) and PDG [41] (gradient based) to generate test cases, \( D_H \) and \( D_p \), respectively, against the same DNN model. Then, we compare \( D_H \) and \( D_p \) from a few aspects. First, we compare the effectiveness of these methods. Namely, we calculate the percentage of samples in \( D_H \) and \( D_p \) that can lead to discovery of DNN defects (i.e., mis-prediction). Second, we compare the quality of generated samples by calculating their similarity with the original benign image. Third, we calculate the coverage metrics for \( D_p \) and try to see if new adversarial examples lead to the growth of coverage.

3.4.3 RQ: Are the adversarial examples generated by DNN coverage based testing effective in improving model robustness? How are they compared to those generated by popular gradient descent based techniques? In software testing, one aspect to measure effectiveness of test generation techniques is the effectiveness of the generated counter-examples (failing test cases) in bug fixing. Similar objectives are explored in DNN testing. Particularly, DeepXplore and SADL use the generated adversarial examples as part of the new training dataset to retrain the model so that it can achieve better accuracy against adversarial examples (i.e., more robust). For example, the DeepXplore paper states that “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%”, and the SADL paper states that (SADL) “can improve classification accuracy of DL systems against adversarial examples by up to 77.5% via retraining”. Therefore in this research question, we want to study the effectiveness of DNN coverage based test generation in comparison with the existing popular alternatives.

**Experiment Design.** To answer RQ 3, we conduct the following experiment. First, we use DeepXplore and SADL to generate adversarial examples, and mix them with original training data to retrain the model. We reuse the parameters (e.g., ratio of new adversarial examples) from the original papers. Second, we train the models with adversarial training. More specifically, we use the PGD based adversarial training. The parameters used in our retraining are adopted from the original paper [41]. Then, we compare the model effectiveness of the two sets of hardened models.

3.4.4 RQ: How are the different DNN coverage criteria correlated? In software testing, there is a semi-lattice for the strength of the different code coverage criteria. For example, statement coverage \( \ll \) edge coverage \( \ll \) path coverage; and condition coverage that aims to cover the true/false values of each comparative expression in a predicate is stronger than statement coverage, weaker than path coverage, and not comparable with edge coverage. We say criterion \( C_1 \) is stronger than \( C_2 \) if 100% \( C_1 \) coverage must imply 100% \( C_2 \) coverage. Such relations are important for choosing the appropriate techniques in software testing. For example, path coverage may be desired for safety critical code despite its high cost. In this research question, we aim to look for correlation or even partial order among the various DNN coverage criteria.

**Experiment Design.** To answer RQ 4, for a given model \( F \), we keep adding new input samples to test the model, and gather the coverage information to get a sequence of values for each test criteria. For example, for \( NC \), we can get a sequence of values \( NC = (NC_0, NC_1, NC_2, ..., NC_n) \) where \( n \) is the total number of added input sample sets. Similarly, we can get \( KMNCF, NBCF \) etc. Then, we perform correlation analysis on these collected data \( NC^F, KMNCF, NBCF \) and so on to see if they have strong correlations or partial order.

### 4 DNN MODEL QUALITY METRICS

In software testing, quality of software is often measured by the number of bugs found within a certain period of time. The quality metrics of DNN models are more diverse. Most DNN testing techniques draw analogy between bugs/defects in software code and adversarial examples in DNN model. An adversarial example is considered manifestation of some undesirable behavior of the model, as it causes misclassification. Just like software quality metrics are based on bugs, model quality metrics are also centered around adversarial examples. They fall into three categories: *model accuracy in the presence of adversarial examples, adversarial example impreceptibility* that measures if an adversarial example looks natural, and *adversarial example robustness*. These are the metrics commonly used by adversarial machine learning [37, 39]. Details are explained in the subsections.

#### 4.1 Model Accuracy for Adversarial Examples

State-of-the-art adversarial example generation techniques, such as C&W and PGD, optimize logits in order to generate inputs. Since logits still have to go through a soft-max layer to produce the final classification outputs, the generated adversarial examples may not yield the intended mis-classification even though the optimizer can successfully reach its objective. Model accuracy in the presence of adversarial examples measures how often the generated adversarial examples lead to correct classification results. High accuracy means that the model is robust. The analogy in software testing is to measure how often the subject software fails when it is stress tested. Specifically, we consider the following metrics that are related to model accuracy.
This metric measures how realistic an adversarial example is in pixels changed by the perturbation; two images $\text{ALD}$ the examples. $\text{ALD}$ measures the maximum change in all dimensions of adversarial

4.2 Adversarial Example Imperceptibility

This metric measures how realistic an adversarial example is in human eyes. It is usually computed by using the original example as a reference. A high quality adversarial example is one that has imperceptible perturbation. We should not say a model is of low quality is it is susceptible to low quality adversarial examples. The analogy in software testing is that the subject software may fail when it is provided with an input that substantially violates the (implicit) input pre-conditions. In such cases, the induced failures cannot be used as evidence of low software quality. Specifically, we use the following metrics.

- **Average $L_p$ Distortion ($\text{ALD}_{p}$).**

  \[
  \text{ALD}_p = \frac{1}{n} \sum_{i=1}^{n} \frac{||x_i^p - x_i||_p}{||x_i||_p}
  \]

  Here, $|| \cdot ||_p$ is the $t_p$ norm distance, which is adopted as distortion metrics for evaluation. Specifically, $t_0$ calculates the number of pixels changed by the perturbation; $t_2$ computes the Euclidean distance between original examples and adversarial examples; $t_\infty$ measures the maximum change in all dimensions of adversarial examples. $\text{ALD}_p$ measures the average normalized $t_p$ distortion for all adversarial examples that cause misclassification. The smaller the $\text{ALD}_p$, the more imperceptibility the adversarial example has.

- **Average Structural Similarity (ASS).**

  \[
  \text{ASS} = \frac{1}{n} \sum_{i=1}^{n} \text{SSIM}(x_i^p, x_i)
  \]

  Here, $\text{SSIM}$ is the metric used to quantify the similarity between two images [29]. ASS can measure the average $\text{SSIM}$ between all adversarial examples that cause misclassification and their corresponding original examples. The larger the $\text{SSIM}$, the more imperceptibility the adversarial examples has.

- **Perturbation Sensitivity Distance (PSD).**

  \[
  \text{PSD} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} \delta_{i,j} \text{Sen}(R(x_{i,j}))
  \]

  Here $m$ is the total number of pixels for an example, $\delta_{i,j}$ denotes the $j$-th pixel of the $i$-th example, $R(x_{i,j})$ represents the surrounding square region of $x_{i,j}$, and $\text{Sen}(R(x_{i,j})) = 1/\text{std}(R(x_{i,j}))$, with $\text{std}(R(x_{i,j}))$ denoting the standard deviation function. $\text{PSD}$ evaluates human perception of perturbations. The smaller the $\text{PSD}$, the more imperceptibility the adversarial example has.

4.3 Adversarial Example Robustness

In adversarial machine learning, adversarial example robustness is often measured [34, 41, 61]. It measures the level of resilience a successful adversarial example (i.e., a example that causes misclassification) has in the presence of (input) perturbation. Intuitively, an adversarial example that is not robust should not be used as evidence of low model quality. The analogy in software testing is that a transient failure (e.g., caused by non-deterministic factors) may not indicate low software quality. In particular, we measure the following.

- **Noise Tolerance Estimation (NTE).**

  \[
  \text{NTE} = \frac{1}{n} \sum_{i=1}^{n} \left[ P(x_i^p | F(x_i^p)) - \max \{ P(x_i^p | F(x_i^p)) \} \right]
  \]

  Here, $P(x_i^p | F(x_i^p))$ is the probability of misclassified class, and $\max \{ P(x_i^p | F(x_i^p)) \}$ is the max probability of all other classes, $j \in \{ 1, ..., k \}$. $\text{NTE}$ measures the amount of noise adversarial examples can tolerate while keeping their misclassified label unchanged. Intuitively, larger $\text{NTE}$ indicates more robust adversarial examples.

- **Robustness to Gaussian Blur (RGB).**

  \[
  \text{RGB} = \frac{\text{count}(\text{F}(\text{GB}(x_i^p)) \neq y_i)}{\text{count}(\text{F}(x_i^p) \neq y_i)}
  \]

  Where $\text{GB}$ denotes the Gaussian blur function, an algorithm that can reduce noises in images and $\text{count()}$ is used to count the number of specific samples. $\text{RGB}$ counts how many adversarial examples can maintain their misclassification function after Gaussian blur. The greater the $\text{RGB}$ is, the more robust adversarial examples are.

- **Robustness to Image Compression (RIC).**

  \[
  \text{RIC} = \frac{\text{count}(\text{F}(\text{IC}(x_i^p)) \neq y_i)}{\text{count}(\text{F}(x_i^p) \neq y_i)}
  \]

  Here, $\text{IC}$ is a specific image compression function. Like $\text{RGB}$, $\text{RIC}$ counts how many adversarial examples can maintain their misclassification function after the image compression function. The greater the $\text{RIC}$, the more robust the adversarial examples.
5 EXPERIMENTS AND RESULTS

5.1 Setup

Datasets and Models. We use MNIST [32], CIFAR-10 [48] and SVHN [1] as our datasets. These are popular datasets used in the literature of model coverage [32, 39, 47, 71]. MNIST is a handwritten digit recognition benchmark in the research community. It contains 60,000 32 × 32 color images in 10 different classes. The Street View House Numbers (SVHN) dataset is obtained from house numbers in Google Street View images. It consists of 73,257 training samples and 26,032 testing samples. For MNIST, we use three pre-trained LeNet family models, i.e., LeNet-1, LeNet-4, and LeNet-5 as the baseline model. For CIFAR-10, we use VGG-16 [32] and ResNet-20 [27] models. For SVHN, we also use three CNN model, and their model architectures are adopted from previous work [32]. We directly use pre-trained models if possible. Our trained models are also available online [1].

Configurations. The configurations for coverage criteria is shown in Table 1. For all research questions, we set the threshold for NC to be 0.1, 0.3, 0.5, 0.7 and 0.9. For KMNC, the k value (i.e., number of multisectins) we use is 10. For TKNP and TKNC, the k value (i.e., the top-k neuron coverage) is 2. For LSC and DSC, the layers we analyze are shown in column 6 in Table 1, the numbers of buckets for LSC and DSC are shown in columns 7 and 9, respectively, and the upper bounds of SA are shown in columns 8 and 10, respectively. These are the same settings published in the original papers or published in their open source repository. For C&W attacks, we use the implementation from CleverHans and use their default parameters. For PDG attack and PDG based adversarial training, we also use the default parameters used in their paper and repository.

5.2 Results and Analysis

5.2.1 RQ: Are DNN coverage metrics correlated with DNN model robustness? As mentioned in subsection 3.4, to answer RQ1, we obtain a set of new training suites, T₁, ..., T₉. Each one has more training data that can enlarge the coverage metrics. In our experiment, we obtain 11 training datasets in total for each model (i.e., n = 10) including the original one (i.e., T₀). In each step, we add 250 new images to the dataset, thus in total we add 2,500 new training images to the training dataset. Compared with DeepXplore and SADL, it introduces more new training samples.

Figure 2 shows the coverage metric value changes w.r.t. different training datasets. In this graph, the y-axis is the relative coverage value and the base value we use is the one obtained on T₀. This is because the results are in wide value ranges. For example, TKNP is an absolute number whose value is larger than a few thousand, and many others are ratios whose value range is [0, 1]. For NC, we only show lines with threshold values being 0.5, 0.7 and 0.9. This is because for 0.1 and 0.3, the original dataset T₀ can achieve 100% coverage. For some metrics, the original coverage values are already very high (over 98%), and hence in the graph, it does not show significant growth. Overall, we can see that with new training samples, all coverage metrics are growing.

Figure 3 shows the changes of attack metrics (i.e., model robustness) w.r.t. different training datasets. Similarly, the values are also normalized based on the values obtained in T₀ (for better visualization). For MR, ACAC, ALD₀ and PSD, larger y values indicate less robust. While for ACTC, ASS, NTE, RGB and RIC, a larger y value always indicates a more robust model. From the graph, we observe that none of them is monotonous. It means from the attack’s point of view, adding new samples to improve coverage does not always lead to the improvement of model robustness.

Due to the space limit, Figure 2 and Figure 3 show the results for the MNIST dataset on the LeNet-1 model. Other models and datasets
do have a similar pattern. Our generated datasets, trained models, and original results are available in GitHub [1]. To have a better understanding of whether coverage criteria are correlated with robustness, we also perform a correlation analysis on all trained models and datasets using the Kendall’s τ method, which is a standard statistical method used to measure the linear and non-linear relationships between two different variables. The result is shown in Figure 4. In this figure, each label in x-axis represents one attack criterion and each label in y-axis represents one coverage criterion. Each cell represents the correlation between criteria. We use blue color to represent negative correlation (i.e., variables change in opposite directions) and red labels to represent positive correlation (i.e., both variables change in the same direction). According to the definition of correlation in Guildford scale [24], if the correlation is less than 0.4, the positive or negative correlation is low; values in [0.4, 0.7] indicate that the correlation is moderate; and high correlation values (i.e., 0.7~0.9 or above 0.9) represent strong correlation. As we can see, most cells are in light colors and have low correlation values, indicating neutral correlations. In other words, there is no clear relationship between these variables.

5.2.2 RQ: Is coverage driven test generation effective in disclosing DNN defects? How does it compare to the other commonly used adversarial example generation techniques? We utilize DeepHunter [71] and PGD [41] to generate test cases, \(D_H\) and \(D_P\), respectively, against the same DNN model. For \(D_H\) and \(D_P\), we first compare the attack success rate using MR. For each method and each model, we generate 1,000 images and then test MR. For both methods, we limit the \(\epsilon_0\) by using the default value provided by DeepHunter [71] (i.e., 0.2 ÷ 255). The results are shown in columns 3 and 4 in Table 2. As we can see, PGD achieves almost 100% success rate on all models and datasets, while DeepHunter achieves lower success rate (i.e., lower MR value, about 60% for SADL-1 and SADL-3 model and about 90% for other models). We also compare the quality of the generated adversarial examples by calculating the average \(\epsilon_0\) distances of benign inputs and adversarial examples. The results are shown in columns 5 to 6 in Table 2. The average \(\epsilon_0\) distances for PGD (\(D_P\)) is 3 to 100 times smaller than that of DeepHunter (\(D_H\)), indicating that the adversarial image quality generated by PGD is better than DeepHunter. In Figure 5, we show the original images (Figure 5a), attack images by DeepHunter (Figure 5b) and PGD (Figure 5c), respectively. The reason why DeepHunter has larger \(\epsilon_0\) perturbation is because of the affine transformations it uses, which results in larger \(\epsilon_0\) values but maintains the semantics. In this sense, \(\epsilon_0\) distance is not an ideal metric to use, and how to choose a better metric is out of the scope. Here, we use it because it is used by both PGD and DeepHunter to determine if the generated example is a valid input.

To understand how PGD generated adversarial examples correlate with coverage criteria, we also calculate the coverage metric value change by adding 10 images in each single step. The result is shown in Figure 6. The primary y-axis shows the percentage for ratio values (i.e., NC, KMNC, NBC, SNAC, TKNC) and the second y-axis shows the value for TKNP. From the graph we can see that the lines for NC, NBC, KMNC, SNAC and TKNC show a similar pattern: they grow rapidly at the beginning and then plateau afterwards. This tells us that new adversarial examples do increase the coverage. However, adversarial examples may not necessarily increase coverage. To future demonstrate this, we use adaptive gradient based attacks to limit the coverage change during optimization while generating the adversarial examples. The results show that it can still generate adversarial examples with almost 100% success rate on all models and datasets.

On the other hand, TKNP shows a almost linear relationship with the number of new samples. To further study this, we design another experiment, which is to calculate the growth of these coverage metrics when we add benign samples instead of adversarial sample. In our case, we fix the setting (i.e., \(k = 2\)), and then we

![Figure 5: Generated Adversarial Examples](image1)

![Figure 6: Coverage VS. # Adversarial Examples](image2)

### Table 1: Configurations for RQs

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>NC (threshold)</th>
<th>KMNC</th>
<th>TKNC/TKNP</th>
<th>LSC</th>
<th>DSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>LeNet-4</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>conv_2</td>
<td>1000</td>
</tr>
<tr>
<td>CIFAR</td>
<td>VGG-16</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>block_2</td>
<td>1000</td>
</tr>
<tr>
<td>CIFAR</td>
<td>ResNet-20</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>conv_2</td>
<td>1000</td>
</tr>
<tr>
<td>SVHN</td>
<td>SADL-1</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>pool_1</td>
<td>1000</td>
</tr>
<tr>
<td>SVHN</td>
<td>SADL-2</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>pool_1</td>
<td>1000</td>
</tr>
<tr>
<td>SVHN</td>
<td>SADL-3</td>
<td>0.1, 0.3, 0.5, 0.7, 0.9</td>
<td>10</td>
<td>2</td>
<td>pool_1</td>
<td>1000</td>
</tr>
</tbody>
</table>

### Table 2: Comparison of Attack Images

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>MR</th>
<th>(\epsilon_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR</td>
<td>VGG-16</td>
<td>94.4%</td>
<td>84.7%</td>
</tr>
<tr>
<td>CIFAR</td>
<td>ResNet-20</td>
<td>94.4%</td>
<td>89.8%</td>
</tr>
<tr>
<td>MNIST</td>
<td>LeNet-4</td>
<td>92.5%</td>
<td>89.6%</td>
</tr>
<tr>
<td>MNIST</td>
<td>LeNet-5</td>
<td>84.3%</td>
<td>90.6%</td>
</tr>
</tbody>
</table>

![Image 1](image1)

(a) Original Images (b) DeepHunter (c) PGD

![Image 2](image2)

Figure 5: Generated Adversarial Examples

Figure 6: Coverage VS. # Adversarial Examples
Figure 7: TKNP VS. # New Inputs

Table 3: Model Accuracy under Different Scenarios

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>Benign</th>
<th>(D_H)</th>
<th>PGD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>LeNet-1</td>
<td>98.7% (+0.9%)</td>
<td>90.3% (2.3%)</td>
<td>0% (+0.5%)</td>
</tr>
<tr>
<td></td>
<td>LeNet-4</td>
<td>98.7% (+0.9%)</td>
<td>90.3% (2.3%)</td>
<td>0% (+0.5%)</td>
</tr>
<tr>
<td></td>
<td>LeNet-5</td>
<td>98.7% (+0.9%)</td>
<td>90.3% (2.3%)</td>
<td>0% (+0.5%)</td>
</tr>
<tr>
<td></td>
<td>LeNet-1 Adv</td>
<td>97.6% (+1.0%)</td>
<td>91.3% (3.1%)</td>
<td>19% (19.2%)</td>
</tr>
<tr>
<td></td>
<td>LeNet-4 Adv</td>
<td>96.9% (+1.2%)</td>
<td>88.9% (2.6%)</td>
<td>9.6% (+9.6%)</td>
</tr>
<tr>
<td></td>
<td>LeNet-5 Adv</td>
<td>97.4% (+1.7%)</td>
<td>90.4% (+2.5%)</td>
<td>30.5% (+30.3%)</td>
</tr>
<tr>
<td>CIFAR</td>
<td>VGG-16</td>
<td>10% (+82.8%)</td>
<td>9.8% (47.4%)</td>
<td>9.9% (+8.8%)</td>
</tr>
<tr>
<td></td>
<td>ResNet-20</td>
<td>75.4% (+15.4%)</td>
<td>60.5% (+1.4%)</td>
<td>1.3% (+1.3%)</td>
</tr>
<tr>
<td></td>
<td>VGG-16 Adv</td>
<td>87.8% (+5.5%)</td>
<td>55.9% (+1.4%)</td>
<td>40% (+39.7%)</td>
</tr>
<tr>
<td></td>
<td>ResNet-20 Adv</td>
<td>86.7% (+5.0%)</td>
<td>57.1% (+1.7%)</td>
<td>36.6% (+36.6%)</td>
</tr>
<tr>
<td>SVHN</td>
<td>SADL-1</td>
<td>93.9% (+4.2%)</td>
<td>82.9% (+6.9%)</td>
<td>0% (+0.5%)</td>
</tr>
<tr>
<td></td>
<td>SADL-2</td>
<td>91.5% (+3.8%)</td>
<td>77.6% (+5.9%)</td>
<td>0.1% (+0.1%)</td>
</tr>
<tr>
<td></td>
<td>SADL-3</td>
<td>94.2% (+4.1%)</td>
<td>84.7% (+5.1%)</td>
<td>1.1% (+1.1%)</td>
</tr>
<tr>
<td></td>
<td>SADL-1 Adv</td>
<td>85.8% (+3.3%)</td>
<td>71.2% (+4.3%)</td>
<td>46.6% (+46.6%)</td>
</tr>
<tr>
<td></td>
<td>SADL-2 Adv</td>
<td>81.7% (+6.1%)</td>
<td>66.7% (+4.3%)</td>
<td>44.6% (+44.6%)</td>
</tr>
<tr>
<td></td>
<td>SADL-3 Adv</td>
<td>87.5% (+4.8%)</td>
<td>74.1% (+5.2%)</td>
<td>50.6% (+50.6%)</td>
</tr>
</tbody>
</table>

Figure 8: Correlation between Coverage Criteria

Threat to Validity. First, most existing coverage criteria focuses on DNN based image classifications, and so is this work. Second, the results are acquired with a limited set of models, data sets and accuracy on benign testing datasets. This is mainly because adversarial training is hard to converge [66]. We trained several versions and got similar results. A similar issue happened for VGG-16 model on the CIFAR dataset. We found that images generated by Deep-Hunter have significantly large perturbations, and the re-training process is difficult to converge.

Most models also have a higher accuracy (compared with the original models) on the adversarial examples generated by coverage guided testing. VGG-16 on CIFAR-10 does not converge and that is why the model accuracy is low. We find that adversarial training does not necessarily improve the model accuracy against such images. For example, all three models on SVHN end up with a lower accuracy on this dataset. One more finding is that models retrained with coverage guided testing has nearly 0% accuracy under PGD attack, while PGD based adversarial training increases the model accuracy by 30% for most cases. This indicates that coverage criteria guided retraining can improve model robustness under attacks that use the same image perturbation strategy, but does not increase the model robustness under gradient based attacks. Similarly, PGD based retraining can improve model accuracy against PGD attacks but cannot improve its accuracy against adversarial inputs generated by coverage guided methods.

5.2.4 RQ: How are the different DNN coverage criteria correlated? If the coverage criteria have partial order relationship, they ought to be correlated. Thus, we first analyze the correlation among all the coverage criteria. The results are shown in Figure 8. The meaning of the graph is the same as Figure 4. From the graph, we can see that NBC and SNAC are correlated with each other with high strength, whereas they have weak or no correlation with the other metrics. NC, KNC, TKNC, LSA and DSA are also correlated with each other to a certain degree, but they do not show very strong correlations. Such results are also consistent with existing work [14, 32, 39].

6  THREAT TO VALIDITY AND DISCUSSION
specific training hyper parameter settings. They may not be representative for other models or settings. To mitigate the threat, we publish all the setup details, implementation, data (e.g., the generated tests) at [1] for reproduction. Third, our study is largely based on existing model coverage criteria and test generation techniques. A possible threat is that we may not faithfully reproduce these existing works. To mitigate the threat, we use the models, datasets and implementations (when available) from the original papers. We also validate our results by cross-checking with those in the original papers.

Discussion and suggestions. According to results from RQ2 and RQ3, we know that it is relatively easy to generate adversarial even when the model has achieve over 100% coverage. This is different from traditional software testing where software are generally less vulnerable after achieving high coverage. This brings the doubt about the usefulness of existing coverage criteria in DNN testing. However, we are uncertain if there will be other coverage criteria that shows a very positive correlation with DNN robustness.

One important message from this study is that our community shall establish a set of standards for evaluating future proposals. They shall include testing various model properties (e.g., robustness, accuracy and fairness), and cover various model architectures and tasks so that the results can be comparable. As an initial step, we have create a GitHub repository [2] for this purpose, and strongly encourage our readers to contribute.

7 RELATED WORK

DNN Testing and Validation. The relationship between coverage and DNN testing has been studied by others as well. Dong et al. [14] show that there is limited correlation between coverage and robustness for DNNs. Li et al. [36] argues that previously reported fault-detection “capabilities” conjectured from high coverage testing are more likely due to the adversary-oriented search but not the real “high” coverage itself.

Besides coverage criteria, A large body of testing methods was proposed for testing machine learning models, such as fuzzing [18, 25, 44, 62, 63, 68, 71, 80], symbolic execution [3, 23, 51, 55], runtime validation [54, 64], fairness testing [3, 62, 77], etc. DeepTest [59] utilizes nine types of realistic image transformations for generating test images, which discovered more than 1,000 erroneous behaviors of DNNs used in autonomous driving systems. DeepRoad [76] leverages generative adversarial networks (GANs) to generate test cases simulating different weather conditions. DeepBillboard [79] manipulates contents on billboards to cause wrong steering angles of autonomous driving systems. DeepImportance [20] selects important neurons from pre-trained models and clusters those neurons with respect to their value regions. The coverage of those important neurons is then used for testing model behaviors. Model testing has also been applied on other domains such as automatic speech recognition [15], text classification [62], image classification [55, 60], machine translation [28, 56]. More related works can be found in this survey [75].

To validate safety of DNN models, researchers leverage verification techniques to provide formal guarantees [19, 22, 31, 46, 48, 49, 65, 68]. Reluplex [31] transforms DNN models to numerical constraints and utilizes SMT-solver to verify robustness of DNNs. Models evaluated in Reluplex were small with 8 layers and 300 ReLU nodes. DeepSafe [22] was built on Reluplex and cannot scale to large models. ReluVal [65] leverages interval arithmetic to verify robustness of DNNs, which is 200 times faster than Reluplex. AI² [49] uses abstract interpretation to approximate the data region after ReLU activation function. Due to its over-approximation nature, AI² may fail to verify an input which has certain property. DeepPoly [53] combines floating point polyhedron with intervals, which can scale to large models. DISSECTOR [64] generates sub-models for intermediate layers of original models and validates given inputs by measuring prediction probabilities from sub-models.

Adversarial Machine Learning. The vulnerability of machine learning models has been an extensively discussed topic [8, 10, 12, 21, 38, 40, 57, 58, 69, 78]. As we elaborate in subsection 2.2 that an attacker can use gradient to perturb original inputs to induce misclassification of DNNs. There also exist other types of adversaries such as black-box attacks [8], back-door attacks [38], etc. A lot of defense mechanisms were proposed to mitigate the threat of adversarial examples such as input transformations [70, 72], differential certificate [50], adversarial training [34, 41, 61], model internal checking [40, 58], etc. More details regarding attacks and defenses can be found in these papers [4, 74].

DNN testing techniques have also been applied on generating counterexamples. The proposed neuron coverage based metrics were utilized to guide the search [32, 39, 47]. In this paper, we study the relationship between those coverage based metrics and adversarial examples in RQ2. We also leverage adversarial training approach to study the robustness of DNN models trained with coverage based examples and gradient based ones, respectively (see details in RQ3).

8 CONCLUSION

In this paper, we study existing neural network coverage criteria, and find that they are not correlated with model robustness. Although they can be used for adversarial example generation and improve model robustness in limited scenarios (i.e., attackers use the same perturbation strategy), they tend to generate adversarial examples that have substantial perturbations and hence perceptible by humans. In contrast, existing optimization based adversarial example generation techniques can generate less perceptible examples. There are correlations between existing coverage criteria. However, it remains unclear if they have partial order relations as code coverage criteria. Our experiments and data are public for reproduction and validation.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive comments, feedbacks and suggestions. We also thank the authors of tools used in this paper for their open source efforts, without which this research is impossible. This research was supported, in part by NSF China No.61802166, DARPA FA8650-15-C-7562, NSF 1748764, NSF 1901242, NSF 1910300, ONR N000141410468, ONR N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions made in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.
REFERENCES

Correlations between Deep Neural Network Model Coverage Criteria and Model Quality


[55] Jonathan Uesato, Ananya Kumar, Csaba Szepesvari, Tom Erez, Avraham Rud-


[59] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoming Ma, and Jian Lu. 2020. DIS-


[63] Weizhu Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. 2018. Mit-


[65] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jin Sun, Guoliang Dong, Xinyu Wang, Xingen Zhao, Jin Song Dong, and Dai Ting. 2020. White-box fairness testing through directed fairness testing. In Proceedings of the 25nd Annual International Conference on Software Engineering (ICSE). IEEE.


