Correlations Between Deep Neural Network Model Coverage Criteria and Model Quality

Anonymous Author(s)

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 similar monotonicity exists between neural network model coverage and model quality. This paper sets out to answer this question.

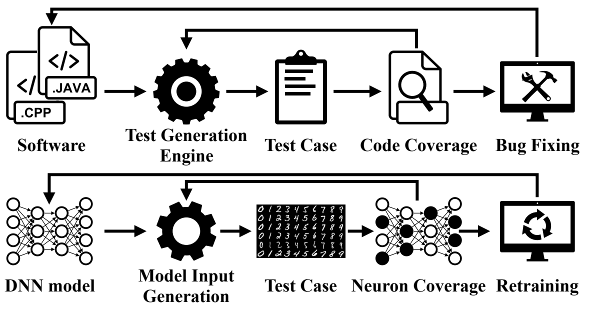


Figure 1: Software Development VS. DNN Development

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 [36, 44, 57, 69]. 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 [13, 36, 44, 57, 69, 76]. 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 [30, 36, 44, 69]. 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* [44] measures the percentage of neurons that are activated, analogous to statement coverage; and *neuron pattern coverage* [36, 69] 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 [44], DeepGauge [36], DeepHunter [69] and SADL [30], and the state-of-the-art optimization based adversarial example generation techniques C&W [10] and PGD [38], 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 ℓ_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 there exists partial order among them like code coverage criteria.

Threat to Validity. First, the results are acquired with a limited set of models, data sets and 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. Second, 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 and datasets from the original papers, and existing original implementations (when available). We also validate our results by cross-checking with those in the original papers.

2 BACKGROUND

2.1 Deep Neural Network

A deep neural network(DNN) is a parameterized function $\mathbf{F}$ that maps an $n-$dimensional input $x \in \mathbb{R}^n$ to one of the k output classes. The output of the DNN $P \in \mathbb{R}^k$ 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 $y = \text{argmax}_{j\in[1,k]} P(x)_j$.

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, $\mathcal{D}_{train} = \{x_i, y_i\}_{i=1}^N$, of N input samples $x = \{x_1, x_2, ..., x_N\} \in \mathbb{R}^n$ and the corresponding ground-truth labels $y = \{y_1, y_2, ..., y_N\} \in [1, k]$. $\mathbf{F}$ is the deep learning model that predicts the corresponding outcomes y' based on the given input x, *i.e.*, $y' = \mathbf{F}(x)$. Within the course of model training, there is a loss function $\mathcal{L} = \sum_{1\le i\le n} ||y'_i - y_i||^2$. So the process of model training can be formalized as:

$$min \sum_{1\le i\le n} ||y'_i - y_i||^2$$

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

$$UCN = \{o \in O \mid \exists \boldsymbol{x} \in T : \phi(\boldsymbol{x}, o) \in (high_o, +\infty)\}$$
$$LCN = \{o \in O \mid \exists \boldsymbol{x} \in T : \phi(\boldsymbol{x}, o) \in (-\infty, low_o)\}$$

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

LIST OF SYMBOLS

$\mathbf{F}$	A DL classifier on k classes, where $\mathbf{F}(x_i) = y_i$
AN	Activated neuron
LCN	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
o	Neuron
P	SoftMax layer output of $\mathbf{F}$, where $\mathbf{F}(x) = \underset{j}{\operatorname{argmax}} P(x)_j$
$P(x)_j$	The j-th probability of $P(x)$, where $j \in \{1, ..., k\}$
UCN	Set of activated neurons that fall in the corner-case regions
x_i^a	An adversarial examples of x_i
x_i	Input samples, $i \in (1, N)$
y_i	The corresponding ground-truth label of x_i, where $y_i = 1, ..., k$

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 δ, a DNN model $\mathbf{F}$ has:

$$\mathbf{F}(x_i^a) = \mathbf{F}(x_i + \delta) = y_t \neq y_i = \mathbf{F}(x_i)$$

Here, we use x_i^a to represent $x_i + \delta$, which is usually referred to as an adversarial example. The perturbation δ added to the input is maliciously manipulated by an adversary and it is commonly bounded by ℓ_p-norm, i.e., $||\delta||_p < \epsilon$. Symbol y_t denotes the predicted label of adversarial example x_i^a, 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 [10]. 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 $\mathbf{F}(\cdot)$ for generating perturbation. The objective function for optimization is the combination of target label ($|\mathbf{F}(x_i^a) - y_t|$) and small perturbation ($||\delta||_p$), which is achieved using optimizer such as Adam.

PGD attack is a first-order universal adversary attack based on Fast Gradient Sign Method (***FGSM***) [55]. FGSM performs a single step 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 [38] 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 $||r||_p < \epsilon$ to the input, i.e., $x_i' = x_i + r$. It then performs one step 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., $x_i^a = clip(x_i'', x_i - \epsilon, x_i + \epsilon)$. The $clip(\cdot)$ 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. [19] 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. [38] leverage the PGD attack with multiple steps for adversarial training. Adversarial training has been shown effective for large scale dataset such as ImageNet [32].

3 COVERAGE BASED DNN ROBUSTNESS TESTING

Since our goal is to study the effectiveness of DNN coverage criteria, in this section, we first introduce a number of popular neuron coverage criteria and discuss their intended usage.

3.1 DeepXplore

Pei et al. proposed DeepXplore [44] which introduced the neuron coverage metric (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 DL logic explored by a set of test inputs.* And then, 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 is based on applying 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 [36], 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 [69] 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 $\phi(\boldsymbol{x}, o) \in S_m^o$ means the m−th section is covered by at least one input $\boldsymbol{x} \in T$.

$$KMNC = \frac{\sum_{o \in O} |\{S_m^o \mid \exists \boldsymbol{x} \in T \ : \ \phi(\boldsymbol{x}, o) \ \in \ S_m^o\}|}{k \times |O|}$$

• **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).**

$$TKNC = \frac{|\bigcup_{\boldsymbol{x} \in T}(\bigcup_{1 \leq l \leq L} top_k(\boldsymbol{x}, l))|}{|O|}$$

Here, L denotes layers of a DNN and $l \in (1, L)$ is the l−th layer of a DNN. Function $top_k(\boldsymbol{x}, l)$ denotes the neurons that have the largest k outputs on layer l given $\boldsymbol{x}$. $TKNC$ 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 $\boldsymbol{x}$, the sequence of the top−k neurons on each layer also forms a pattern. A pattern is an element of $2_1^u \times 2_2^u \times \cdots \times 2_l^u$, where 2_l^u 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.

$$TKNP = |\{(top_k(\boldsymbol{x}, 1)), ..., (top_k(\boldsymbol{x}, L)) \mid \boldsymbol{x} \in T|\}|$$

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. DeepHunter 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 help generate images that preserve its original semantics, a ℓ_∞ based threshold is set. If the difference between the original benign image 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 [30]. 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 *Keneral Density Estimation* (KDE) to approximate the likelihood 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 $\alpha_o(x)$ 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 $O' \subseteq O$, $\alpha_{O'}(x)$ denotes a vector of activation values that represents the Activation Trace (AT) of x over neurons in O'. For a set of inputs X, $A_{O'}(X) = \{\alpha_{O'}(x) | x \in X\}$ 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 $\hat{f}$ as follows.

$$\hat{f}(x) = \frac{1}{|A_{O'}(T)|} \sum_{x_i \in T} K_H(\alpha_{O'}(x) - \alpha_{O'}(x_i))$$

Intuitively, the function measures a normalized distance between the activation values of x and those of individual inputs in T (regarding O'). Then LSA is defined as follows.

$$LSA(x) = -log(\hat{f}(x))$$

• **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.

$$x_a = \underset{F(x_i)=y}{\text{argmin}} ||\alpha_O(x) - \alpha_O(x_i)||$$
$$dist_a = ||\alpha_O(x) - \alpha_O(x_a)||$$

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

$$x_b = \underset{F(x_i) \in Y \setminus \{y\}}{\text{argmin}} ||\alpha_O(x) - \alpha_O(x_i)||$$
$$dist_b = ||\alpha_O(x) - \alpha_O(x_b)||$$

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 [5, 14, 15, 28, 31, 33, 39, 40, 42, 65], 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 [36], 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 criteria, we propose the following research questions.

3.4.1 **RQ1: 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 [34] 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 **RQ2: 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 [8, 12], fuzzing [6, 71], and search based testing [4, 24], 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*" [44]. DeepHunter [69] "*leverages multiple plugable 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 ℓ_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 [44], changing image blurriness and rotating images [69]. See details in subsection 3.1 and subsection 3.2. Notice DeepHunter also uses a ℓ_∞ 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., ℓ_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 [69] (coverage criteria based) and PDG [38] (gradient based) to generate test cases, $\mathcal{D}_H$ and $\mathcal{D}_P$, respectively, against the same DNN model. Then, we compare $\mathcal{D}_H$ and $\mathcal{D}_P$ from a few aspects. First, we compare the effectiveness of these methods. Namely, we calculate the percentage of samples in $\mathcal{D}_H$ and $\mathcal{D}_P$ 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 $\mathcal{D}_P$ and try to see if new adversarial examples lead to the growth of coverage.

3.4.3 **RQ3: 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 [38]. Then, we compare the model effectiveness of the two sets of hardened models.

3.4.4 RQ4: 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 < edge coverage < 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 $C1$ is stronger than $C2$ if 100% $C1$ coverage must imply 100% $C2$ 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 $NC_F = (NC_0^F, NC_1^F, NC_2^F, ..., NC_n^F)$ where n is the total number of added input sample sets. Similarly, we can get $KMNC^F$, NBC^F etc. Then, we perform correlation analysis on these collected data NC^F, $KMNC^F$, NBC^F 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 impreceptiblity* that measures if an adversarial example looks natural, and *adversarial example robustness.* These are the metrics commonly used by adversarial machine learning [34, 36]. Details are explained in the subsections.

4.0.1 *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.

- **Misclassification Ratio (MR).**

$$MR = \frac{1}{N}\sum_{i=1}^{N} count(\mathbf{F}(x_i^a) \neq y_i)$$

Here, N is the number of adversarial samples and $count()$ is a function used to count the number of misclassified samples. Basically, MR calculates the percentage of misclassified input samples in the whole input set. A high quality model has a low MR.

- **Average Confidence of Adversarial Class (ACAC).**

$$ACAC = \frac{1}{n}\sum_{i=1}^{n} P(x_i^a)_{F(x_i^a)}$$

where n $(n \leq N)$ is the total number of adversarial examples that cause misclassification. And $P(x_i^a)_{F(x_i^a)}$ is the prediction confidence towards the incorrect class $F(x_i^a)$. In general, ACAC measures the average prediction confidence towards the incorrect class for adversarial examples.

- **Average Confidence of True Class (ACTC).**

$$ACTC = \frac{1}{n}\sum_{i=1}^{n} P(x_i^a)_{y_i}$$

Here n $(n \leq N)$ is the total number of adversarial examples that cause misclassification. $P(x_i^a)_{y_i}$ is the prediction confidence of true classes for adversarial examples. Just like ACAC, ACTC measures the prediction confidence of true classes for adversarial examples.

4.0.2 *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 (ALD_p).**

$$ALD_p = \frac{1}{n}\sum_{i=1}^{n} \frac{\|x_i^a - x_i\|_p}{\|x_i\|_p}$$

Here, $\|\cdot\|_p$ is the ℓ_p norm distance, which is adopted as distortion metrics for evaluation. Specifically, ℓ_0 calculates the number of pixels changed by the perturbation; ℓ_2 computes the Euclidean distance between original examples and adversarial examples; ℓ_∞ measures the maximum change in all dimensions of adversarial examples. ALD_p measures the average normalized ℓ_p distortion for all adversarial examples that cause misclassification. The smaller the ALD_p, the more imperceptibility the adversarial example has.

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

$$ASS = \frac{1}{n}\sum_{i=1}^{n} SSIM(x_i^a, x_i)$$

Here, *SSIM* is the metric used to quantify the similarity between two images [27]. ASS can measure the average *SSIM* between all adversarial examples that cause misclassification and their corresponding original examples. The larger the *SSIM*, the more imperceptibility the adversarial examples has.

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

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

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 $Sen(R(x_{i,j})) = 1/std(R(x_{i,j}))$, with $std(R(x_{i,j}))$ denoting the standard deviation function. *PSD* evaluates human perception of perturbations. The smaller the *PSD*, the more imperceptibility the adversarial example has.

4.0.3 Adversarial Example Robustness. In adversarial machine learning, adversarial example robustness is often measured [32, 38, 59]. 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)**.

$$NTE = \frac{1}{n}\sum_{i=1}^{n}[P(x_i^a)_{\mathbf{F}(x_i^a)} - max\{P(x_i^a)_j\}]$$

Here, $P(x_i^a)_{\mathbf{F}(x_i^a)}$ is the probability of misclassified class, $max\{P(x_i^a)_j$ is the max probability of all other classes, $j \in \{1, ..., k\}$ and $j \neq \mathbf{F}(x_i^a)$. *NTE* can calculate the amount of noise that adversarial examples can tolerate while keeping their misclassified label unchanged. Intuitively, larger *NTE* indicates more robust adversarial examples.

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

$$RGB = \frac{count(\mathbf{F}(\mathbf{GB}(x_i^a)) \neq y_i)}{count(\mathbf{F}(x_i^a) \neq y_i)}$$

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

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

$$RIC = \frac{count(\mathbf{F}(\mathbf{IC}(x_i^a)) \neq y_i}{count(\mathbf{F}(x_i^a) \neq y_i}$$

Here, **IC** is a specific image compression function. Like *RGB*, *RIC* counts how many adversarial examples can maintain their misclassification function after the image compression function. The greater the *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 [30, 36, 44, 69]. MNIST is a handwritten digit recognition dataset, a popular dataset for image classification. The CIFAR-10 dataset is widely used for easy image classification task/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 sasmples 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 [50] and ResNet-20 [25] models. For SVHN, we also use three CNN model, and their model architectures are adopted from previous work [30]. 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 multisections) 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 RQ1: 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, $T_1, ..., T_n$. 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., T_0). 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 T_0. 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,

Table 1: Configurations for RQs

Dataset	Model	NC (threshold)	KMNC (k)	TKNC/TKNP (k)	LSC			DSC	
					Layer	n	ub	n	ub
MNIST	LeNet-1	0.1, 0.3, 0.5, 0.7, 0.9	10	2	conv_2	1000	2000	1000	5
	LeNet-4	0.1, 0.3, 0.5, 0.7, 0.9	10	2	conv_2	1000	2000	1000	5
	LeNet5	0.1, 0.3, 0.5, 0.7, 0.9	10	2	conv_2	1000	2000	1000	5
CIFAR	VGG-16	0.1, 0.3, 0.5, 0.7, 0.9	10	2	conv_2	1000	2000	1000	5
	ResNet-20	0.1, 0.3, 0.5, 0.7, 0.9	10	2	block2_conv1	1000	2000	1000	5
SVHN	SADL-1	0.1, 0.3, 0.5, 0.7, 0.9	10	2	pool_1	1000	2000	1000	5
	SALD-2	0.1, 0.3, 0.5, 0.7, 0.9	10	2	pool_1	1000	2000	1000	5
	SALD-3	0.1, 0.3, 0.5, 0.7, 0.9	10	2	pool_1	1000	2000	1000	5

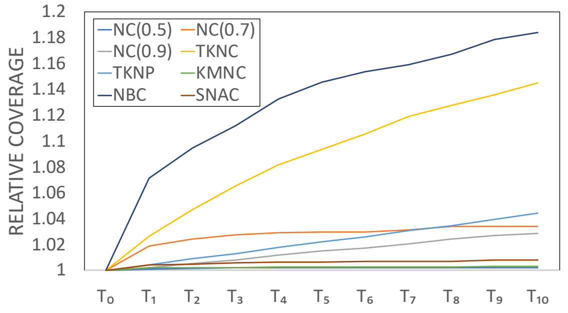


Figure 2: Coverage VS. Training Datasets

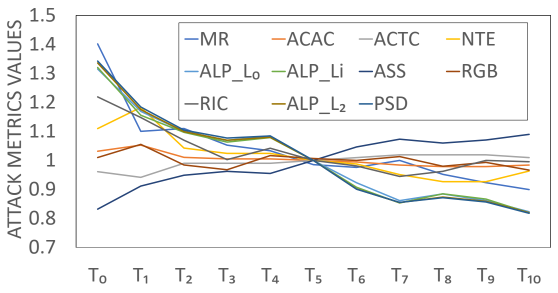


Figure 3: Robustness VS. Training Datasets

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 T_0 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 T_5 (for better visualization). For MR, ACAC, ALD_p 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

Table 2: Comparison of Attack Images

Dataset	Model	MR		ℓ_∞	
		$\mathcal{D}_H$	$\mathcal{D}_P$	$\mathcal{D}_H$	$\mathcal{D}_P$
MNIST	LeNet-1	2.8%	100%	242.8	77
	LeNet-4	4.1%	100%	248.6	77
	LeNet-5	4.3%	99.9%	249.2	77
CIFAR	VGG-16	42.9%	89.8%	106.5	8
	ResNet-20	42.2%	99.9%	92.6	8
SVHN	SADL-1	23.5%	100%	166.2	8.9
	SADL-2	29.6%	99.9%	174.8	8.9
	SADL-3	21.1%	100%	168.8	8.9

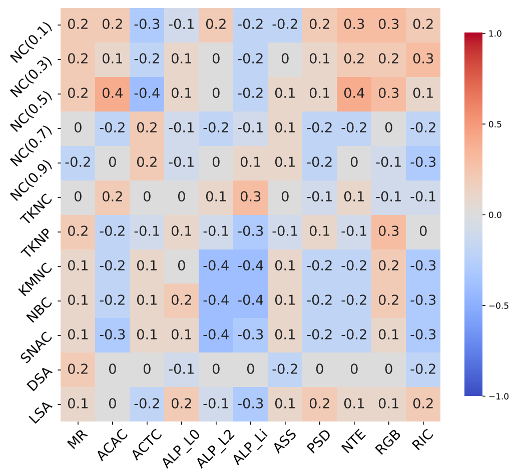
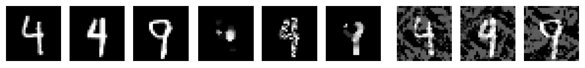


Figure 4: Correlation of Coverage Criteria and Robustness

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 [22], 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 *RQ2: 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 [69] and PDG [38] to generate test cases, $\mathcal{D}_H$ and $\mathcal{D}_P$, respectively, against the same DNN model. For $\mathcal{D}_H$ and $\mathcal{D}_P$, we first compare the attack success rate using MR. For each method and each model, we generate 1,000 images and then test MR. The results are shown in columns 3 and 4 in Table 2. As we can see, PGD achieves almost 100% success rate on all the models and datasets, while DeepHunter achieves significantly lower success rate (i.e., lower MR value, less than 5% for all three LeNet models). It means that during adversarial example generation, DeepHunter wastes a lot of computation resources generating inputs that cannot find DNN defects. Thus PGD, the gradient based method is more effective in finding DNN defects. We also compare the quality of the generated adversarial



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

Figure 5: Generated Adversarial Examples

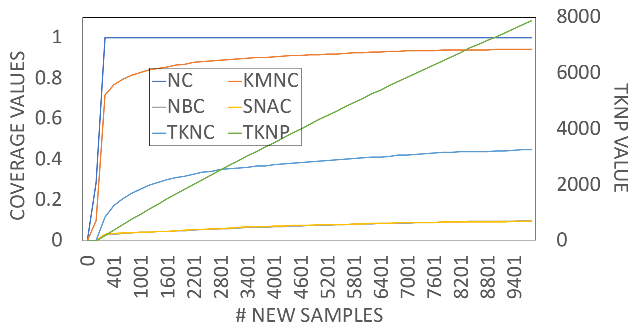


Figure 6: Coverage VS. # Adversarial Examples

examples by calculating the average ℓ_∞ distances of benign inputs and adversarial examples. The results are shown in columns 5 to 6 in Table 2. Here, we choose to use ℓ_∞ distance because it is used by both DeepHunter and PGD as their distance measurement function. The average ℓ_∞ distances for $\mathcal{D}_P$ is 3 to 20 times smaller than that of $\mathcal{D}_H$. Such results indicate that the adversarial image quality generated by PGD is better than DeepHunter. To demonstrate, we also show some generated images in Figure 5. In Figure 5, we show the original images (Figure 5a), attack images by DeepHunter (Figure 5b) and PGD (Figure 5c), respectively. As we can see, many of the images generated by DeepHunter are not recognizable by human any more. It is hence unclear we should call such inputs as manifestations of model bugs. Analogously, we should not say a software is defective if it misbehaves in the presence of inputs that violate the preconditions.

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 add the corresponding seed input as the new sample. The results is shown in Figure 7. As we can see, adding benign samples and adding adversarial samples have almost the same effects, indicating that this criteria cannot really distinguish the differences of benign samples and adversarial samples. In other words, it is almost equivalent to the count of samples.

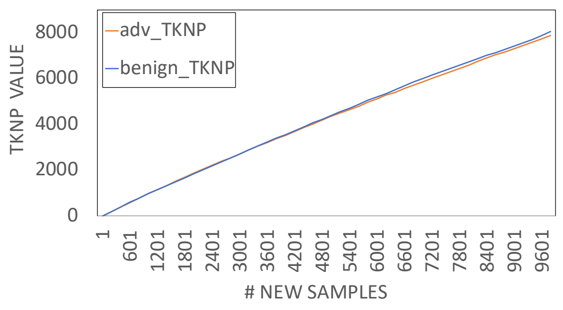


Figure 7: TKNP VS. # New Inputs

Table 3: Model Accuracy under Different Scenarios

Dataset	Model	Benign	D_H	PGD
MNIST	LeNet-1	98.7%(+0.09%)	90.3%(+2.37%)	0%(+0%)
	LeNet-4	98.7%(+0.07%)	90.1%(+2.3%)	0%(+0%)
	LeNet-5	98.71%(-0.26%)	91%(+1.1%)	0%(+0%)
	LeNet-1 Adv.	97.6%(-1.07%)	91.3%(+3.4%)	19.2%(+19.2%)
	LeNet-4 Adv.	96.9%(-1.72%)	88.9%(+1.1%)	9.6%(+9.6%)
	LeNet-5 Adv.	97%(-1.97%)	90.1%(+0.2%)	30.5%(30.3%)
CIFAR	VGG-16	10%(-82.8%)	9.89%(-47.41%)	9.99%(+8.8%)
	ResNet-20	75.4%(-16.4%)	60.5%(+1.4%)	0.13%(+0.13%)
	VGG-16 Adv.	87.8%(-5%)	55.9%(-1.4%)	40.9%(+39.7%)
	ResNet-20 Adv.	86.7%(-5.04%)	57.5%(-1.7%)	36.6%(+36.6%)
SVHN	SADL-1	93.97%(+4.27%)	82.95%(+6.9%)	0%(+0%)
	SADL-2	91.55%(+3.83%)	77.65%(+5.9%)	0.1%(+0.1%)
	SADL-3	94.28%(+1.71%)	84.47%(+5.1%)	1.1%(+1.1%)
	SADL-1 Adv.	85.8%(-3.9%)	71.25%(-4.8%)	46.6%(+46.6%)
	SADL-2 Adv.	81.7%(-6.12%)	66.73%(-4.8%)	44.6%(+44.6%)
	SADL-3 Adv.	87.7%(-4.87%)	74.13%(-5.23%)	50.6%(+50.6%)

5.2.3 *RQ3: 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?*

To answer RQ3, we retrain the models using two different approaches: for one set, we use the adversarial examples generated by coverage guided testing, and for the other set, we use PGD based adversarial training. The results are shown in Table 3. The first column shows the datasets. The second column presents the models including those retrained with coverage guided adversarial examples (rows 2-4, 8-9, 12-14) and those retrained by PGD (rows 5-7, 10-11, 15-17). For each model, we show the accuracy on benign samples, on adversarial examples generated by coverage guidance and under PGD attack. In each cell, we show the model accuracy as well as the difference compared with the original model (without retraining). Numbers with + means that the accuracy is higher than the original model and numbers with − means that the model accuracy is lower than the original one. From the table, we can see that most models with the same architecture have similar accuracy on benign testing datasets (which is also similar to the original model accuracy). In some cases, PGD based adversarial training gets lower accuracy on benign testing datasets. This is mainly because adversarial training is hard to converge [64]. 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 DeepHunter have significantly large perturbations, and the re-training process is very difficult to converge.

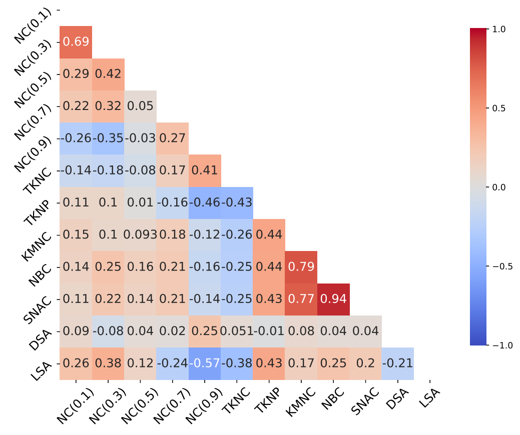


Figure 8: Correlation between Coverage Criteria

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 *RQ4: 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 [30, 36].

6 RELATED WORK

DNN Testing and Validation. Besides coverage criteria we discussed in section 3. A large body of testing methods was proposed for testing machine learning models, such as fuzzing [16, 23, 41, 60, 61, 66, 69, 77], symbolic execution [2, 21, 49, 53], runtime validation [52, 62], fairness testing [2, 46, 60], etc. DeepTest [57] 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 [74] leverages generative adversarial networks (GANs) to generate test cases simulating different weather conditions. DeepBillboard [76] manipulates contents on billboards to cause wrong steering angles of autonomous driving systems. DeepImportance [18] 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 [13], text classification [60], image classification [53, 58], machine translation [26, 54]. More related works can be found in this survey [73].

To validate safety of DNN models, researchers leverage verification techniques to provide formal guarantees [17, 20, 29, 43, 45, 47, 63, 66]. Reluplex [29] 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 [20] was built on Reluplex and cannot scale to large models. ReluVal [63] leverages interval arithmetic to verify robustness of DNNs, which is 200 times faster than Reluplex. AI^2 [47] uses abstract interpretation to approximate the data region after ReLU activation function. Due to its over-approximation nature, AI^2 may fail to verify an input which has certain property. DeepPoly [51] combines floating point polyhedron with intervals, which can scale to large models. DISSECTOR [62] 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 [7, 9, 11, 19, 35, 37, 55, 56, 67, 75]. 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 [7], back-door attacks [35], etc. A lot of defense mechanisms were proposed to mitigate the threat of adversarial examples such as input transformations [68, 70], differential certificate [48], adversarial training [32, 38, 59], model internal checking [37, 56], etc. More details regarding attacks and defenses can be found in these papers [3, 72].

DNN testing techniques have also been applied on generating counterexamples. The proposed neuron coverage based metrics were utilized to guide the search [30, 36, 44]. 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**).

7 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.

REFERENCES

[1] [n.d.]. DNNTesting/CovTesting. https://github.com/DNNTesting/CovTesting. (Accessed on 03/05/2020).

[2] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 625–635.

[3] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. In *International Conference on Machine Learning*. 274–283.

[4] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. 2011. Symbolic search-based testing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 53–62.

[5] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[7] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. In *International Conference on Learning Representations (ICLR)*.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 209–224.

[9] Nicholas Carlini and David Wagner. 2017. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. 3–14.

[10] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy (S&P)*. 39–57.

[11] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Notices* 46, 3 (2011), 265–278.

[13] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Deepstellar: model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 477–487.

[14] Irwin S Dunietz, Willa K Ehrlich, BD Szablak, Colin L Mallows, and Anthony Iannino. 1997. Applying design of experiments to software testing: experience report. In *Proceedings of the 19th international conference on Software engineering*. 205–215.

[15] Gordon Fraser and Andrea Arcuri. 2012. Sound empirical evidence in software testing. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 178–188.

[16] Xiang Gao, Ripon Saha, Mukul Prasad, and Roychoudhury Abhik. 2020. Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[17] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18.

[18] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. 2020. Importance-Driven Deep Learning System Testing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations (ICLR)*.

[20] Divya Gopinath, Guy Katz, Corina S Pasareanu, and Clark Barrett. 2017. Deepsafe: A data-driven approach for checking adversarial robustness in neural networks. *arXiv preprint arXiv:1710.00486* (2017).

[21] Divya Gopinath, Corina S Pasareanu, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. 2019. Symbolic execution for attribution and attack synthesis in neural networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 282–283.

[22] Joy Paul Guilford. 1950. Fundamental statistics in psychology and education. (1950).

[23] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

[24] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2 (2009), 226–247.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-Invariant Testing for Machine Translation. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[27] Alain Hore and Djemel Ziou. 2010. Image quality metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition*. IEEE, 2366–2369.

[28] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[29] 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 *International Conference on Computer Aided Verification*. Springer, 97–117.

[30] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

[31] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. 2004. Software fault interactions and implications for software testing. *IEEE transactions on software engineering* 30, 6 (2004), 418–421.

[32] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial Machine Learning at Scale. In *International Conference on Learning Representations (ICLR)*.

[33] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*. IEEE, 60–69.

[34] Xiang Ling, Shouling Ji, Jiaxu Zou, Jiannan Wang, Chunming Wu, Bo Li, and Ting Wang. 2019. Deepsec: A uniform platform for security analysis of deep learning model. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 673–690.

[35] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning Attack on Neural Networks. In *Proceedings of the 25nd Annual Network and Distributed System Security Symposium (NDSS)*.

[36] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[37] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. 2019. NIC: Detecting Adversarial Samples with Neural Network Invariant Checking.. In *Proceedings of the 25nd Annual Network and Distributed System Security Symposium (NDSS)*.

[38] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*.

[39] David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. 2007. 'Good'organisational reasons for'Bad'software testing: An ethnographic study of testing in a small software company. In *29th international conference on software engineering (ICSE'07)*. IEEE, 602–611.

[40] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[41] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *International Conference on Machine Learning*. 4901–4911.

[42] A Jefferson Offutt. 1992. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1, 1 (1992), 5–20.

[43] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: Differential Verification of Deep Neural Networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[44] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[45] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785* (2017).

[46] Jun Sun Guoliang Dong Xinyu Wang Xingen Wang Jin Song Dong Dai Ting Peixin Zhang, Jingyi Wang. 2020. White-box Fairness Testing through Adversarial Sampling. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[47] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*. Springer, 243–257.

[48] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. 2018. Certified defenses against adversarial examples. In *International Conference on Learning Representations*.

[49] Arvind Ramanathan, Laura L Pullum, Faraz Hussain, Dwaipayan Chakrabarty, and Sumit Kumar Jha. 2016. Integrating symbolic and statistical methods for

testing intelligent systems: Applications to machine learning and computer vision. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 786–791.

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

[51] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[52] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour Prediction for Autonomous Driving Systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[53] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 109–119.

[54] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic Testing and Improvement of Machine Translation. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[55] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing Properties of Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[56] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. 2018. Attacks meet interpretability: Attribute-steered detection of adversarial samples. In *Advances in Neural Information Processing Systems*. 7717–7728.

[57] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[58] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2020. Testing DNN Image Classifier for Confusion & Bias Errors. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[59] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2018. Ensemble adversarial training: Attacks and defenses. In *International Conference on Learning Representations*.

[60] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 98–108.

[61] Jonathan Uesato, Ananya Kumar, Csaba Szepesvari, Tom Erez, Avraham Ruderman, Keith Anderson, Nicolas Heess, Pushmeet Kohli, et al. 2018. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. *arXiv preprint arXiv:1812.01647* (2018).

[62] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. DISSECTOR: Input Validation for Deep Learning Applications by Crossing-layer Dissection. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[63] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1599–1614.

[64] Yisen Wang, Xingjun Ma, James Bailey, Jinfeng Yi, Bowen Zhou, and Quanquan Gu. 2019. On the convergence and robustness of adversarial training. In *International Conference on Machine Learning*. 6586–6595.

[65] James A Whittaker and Michael G Thomason. 1994. A Markov chain model for statistical software testing. *IEEE Transactions on Software engineering* 20, 10 (1994), 812–824.

[66] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-guided black-box safety testing of deep neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 408–426.

[67] Chaowei Xiao, Jun Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. 2018. Spatially transformed adversarial examples. In *6th International Conference on Learning Representations, ICLR 2018*.

[68] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. 2018. Mitigating Adversarial Effects Through Randomization. In *International Conference on Learning Representations*.

[69] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.

[70] Weilin Xu, David Evans, and Yanjun Qi. 2018. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of the 25nd Annual Network and Distributed System Security Symposium (NDSS)*.

[71] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 712–723.

[72] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2019. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems* 30, 9 (2019), 2805–2824.

[73] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[74] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[75] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards Characterizing Adversarial Defects of Deep Learning Software from the Lens of Uncertainty. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE.

[76] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Lingming Zhang, Bei Yu, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *2020 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE.

[77] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic testing of driverless cars. *Commun. ACM* 62, 3 (2019), 61–67.