7-2023

Seed selection for testing deep neural networks

Yuhan ZHI

Xiaofei XIE

Chao SHEN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Xiaoyu ZHANG

See next page for additional authors

Seed Selection for Testing Deep Neural Networks

YUHAN ZHI, Xi'an Jiaotong University, China
XIAOFEI XIE, Singapore Management University, Singapore
CHAO SHEN*, Xi'an Jiaotong University, China
JUN SUN, Singapore Management University, Singapore
XIAOYU ZHANG, Xi'an Jiaotong University, China
XIAOHONG GUAN, Xi'an Jiaotong University, China

Deep learning (DL) has been applied in many applications. Meanwhile, the quality of DL systems is becoming a big concern. To evaluate the quality of DL systems, a number of DL testing techniques have been proposed. To generate test cases, a set of initial seed inputs are required. Existing testing techniques usually construct seed corpus by randomly selecting inputs from training or test dataset. Till now, there is no study on how initial seed inputs affect the performance of DL testing and how to construct an optimal one. To fill this gap, we conduct the first systematic study to evaluate the impact of seed selection strategies on DL testing. Specifically, considering three popular goals of DL testing (i.e., coverage, failure detection and robustness), we develop five seed selection strategies including three based on single-objective optimization (SOO) and two based on multi-objective optimization (MOO). We evaluate these strategies on 7 testing tools. Our results demonstrate that the selection of initial seed inputs greatly affects the testing performance. SOO-based selection can construct the best seed corpus that can boost DL testing with respect to the specific testing goal. MOO-based selection strategies construct seed corpus that achieve balanced improvement on multiple objectives.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Deep learning testing, Seed selection, Coverage, Robustness

1 INTRODUCTION

Deep learning (DL) [42] has been successfully applied in various applications, such as image processing [13, 63], natural language processing [14] and game playing [58, 71]. However, it has been demonstrated that Deep Neural Networks (DNNs) are vulnerable to adversarial attacks that cause DNNs to make incorrect decisions given slightly perturbed inputs [11, 29, 38, 51, 87]. It poses a threat to the quality of DL systems especially when they are applied in safety-critical applications such as autonomous driving [18, 44], biometrics identification [61, 67] and medical diagnosis [23, 52]. Therefore, the quality of DL systems requires systematic evaluation before deployment.

*Chao Shen is the corresponding author.

Authors' addresses: Yuhan Zhi, zyh1123@stu.xjtu.edu.cn, Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, China, 710049; Xiaofei Xie, Singapore Management University, 81 Victoria Street, Singapore, 188065, xfxie@smu.edu.sg; Chao Shen, Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, China, 710049, chaoshen@mail.xjtu.edu.cn; Jun Sun, Singapore Management University, 81 Victoria Street, Singapore, 188065, junsun@smu.edu.sg; Xiaoyu Zhang, Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, China, 710049, zxy0927@stu.xjtu.edu.cn; Xiaohong Guan, Xi'an Jiaotong University, No.28, Xianning West Road, Xi'an, China, 710049, xhguan@mail.xjtu.edu.cn.

1049-331X/2023/7-ART $15.00
https://doi.org/10.1145/3607190

To evaluate the quality of the DL systems, a number of DL testing techniques [31, 49, 68, 74, 77] have been proposed. Similar to traditional software testing, DL testing aims to generate diverse test cases such that diverse failures (e.g., incorrect prediction) of the DL system can be found. To facilitate the generation of diverse test cases, some novel coverage criteria (e.g., Neuron Coverage [62], Surprise Adequacy [40]) are proposed to guide the testing of DL systems. Inspired by traditional software testing, researchers have developed new coverage-guided testing techniques [31, 59, 62, 77, 88] that have been demonstrated to be effective in finding failures in DL systems.

The basic idea of coverage-guided testing (CGT) is: given a set of initial seed inputs, CGT iteratively generates new test cases by randomly mutating some of the seed inputs that are sampled from the initial seed corpora (called seed sampling). The coverage criteria are used to select and keep "interesting" test cases that can increase coverage (i.e., cover new behaviors of the DNN). CGT reduces to random testing if there is no such coverage feedback. Since all test cases are generated from the initial seed inputs, the quality of the initial seeds greatly affects the testing performance. However, most recent DL testing studies mainly focus on mutation strategies (e.g., adopting advanced data transformation [77]), seed sampling (e.g., random and recency-based strategy [59, 88]) and coverage criteria [40, 49, 74]. Most of the existing testing techniques randomly select seed inputs from the training/test dataset [62, 77, 88]. For example, DeepXplore [62], DeepTest [77] and DeepHunter [88] randomly selected 2000, 100 and 1000 initial seed inputs for evaluation, respectively. There are some existing works proposing different strategies to select initial seeds for testing [3, 20, 65, 95]. However, we still do not have a clear understanding of the impact of these seed selection strategies on testing performance, and whether there is a general method of selecting seeds for a certain testing goal on any testing tool.

In traditional software testing, there have been some studies that note and demonstrate that the quality of seed inputs can greatly impact the performance of CGT such as fuzzing [35, 41, 60, 80], and there are some general ways to select initial seed corpus. Specifically, a common approach in traditional software testing is the corpus minimization technique [2, 35, 64, 91] which selects the smallest subset of seeds such that some redundant inputs that are not worth exploring can be quickly discarded. The quality of seed inputs is also a problem worthy of further discussion in DL testing. Considering the testing budget and time cost, it is impossible to select all possible inputs such as training data, test data or real-world data. Therefore, initial seed selection is a necessary step before DL testing. Informally, the research problem is that, given a large amount of data and the testing budget (i.e., the number of seed inputs), how can we select a set of seed inputs that can boost the performance of the testing techniques? However, due to the fundamental difference between traditional software and DNN, it is still unknown whether the existing seed selection[1] strategies (e.g., corpus minimization) are effective for DL testing. Specifically, ❶ the testing goals are not exactly the same. In addition to the common goals, i.e., coverage and the number of failures, DL testing also focuses on whether the generated test cases can be used to improve robustness [40, 82]. ❷ In terms of failure detection, as it is difficult to measure the potential of a test case in triggering bugs in traditional software, the corpus minimization usually works by finding the minimum set of seeds that still maintains the code coverage. However, there are some metrics (e.g., uncertainty [21, 26, 72], confidence score [53, 92, 94]) that could be used to measure the potential to generate adversarial examples, which can be potentially adopted for seed selection in DL testing. ❸ Differently from traditional software, a DL task can often be accomplished with different models (e.g. different model architectures or the same architecture but with different parameters). One may ask whether high-quality seed corpora selected from one DNN are still useful for testing other DNNs.

To this end, we conduct the first systematic study to understand the seed selection on DL testing. Specifically, we focus on three common testing goals: coverage, failure detection and robustness enhancement. To facilitate an in-depth investigation, we first propose three seed selection strategies aimed at achieving optimal results

[1]Note that we distinguish seed selection (referring to construct the initial seed corpus) from seed sampling (referring to select a seed to mutate during testing).

for different testing goals. To study whether we can select one optimal seed corpus that can work well on all testing goals, we then propose two multi-objective optimization (MOO) based seed selection strategies. Based on different seed selection strategies, we aim to answer the following research questions:

- **RQ1:** For a specific goal (i.e., coverage, failure detection and robustness), is there a seed selection strategy which can boost the testing performance?
- **RQ2:** How useful are the MOO-based seed selection strategies in boosting the testing performance?
- **RQ3:** Can the optimal seed corpus selected from one model be effectively transferred to other models?

By answering these questions, we can better understand the impact of seed selection strategies on DL testing and how it differs from traditional software testing. We conduct more than 14,000 runs of testing. Note that our analysis only considers testing generators that perturb input images at the pixel level. Our results reveal that the SOO-based seed selection strategy designed for a specific goal can significantly help boost the testing performance on that corresponding goal (e.g., coverage or failure detection) compared to random selection. However, it does not work well on other goals. On the other hand, we found that it is difficult to select seed inputs for improving robustness with the existing metrics. The MOO-based seed selection strategies have positive effects on all goals. The results demonstrate the importance and effectiveness of seed selection in boosting testing performance.

In summary, we make the following contributions[2] in this work.

- We conduct the first systematic study to understand the impact of seed selection for DL testing.
- For the three goals of DL testing, we propose three single-objective optimization-based seed selection strategies to boost the testing performance.
- We propose two multi-objective optimization-based seed selection strategies to fulfill all three goals.
- We perform extensive experiments to assess how the different seed selection strategies affect coverage, failure detection and robustness of different models. Our results demonstrate that seed selection can significantly affect the testing performance, and comparisons between testing tools should also take into account suitable seed selection.

2 BACKGROUND

2.1 Coverage-guided Testing

Coverage-guided testing (CGT) is a very popular technique for finding bugs [4]. Fuzzing (e.g., AFL [91]) is a typical CGT technique that has been successfully used to detect thousands of bugs in real-world software [12, 69, 76]. The fuzzing process begins by choosing a seed corpus that contains a set of initial seed inputs. Then at each iteration, it selects one seed from the seed corpus and generates some mutants based on the selected seed. The code coverage information (e.g., branch coverage) is gathered by executing the mutants. The mutants are added into the seed queue if they increase the coverage (i.e., covering new software behaviors). CGT is also adapted to test DNN with the specific coverage criteria defined for DNNs [40, 49, 62]. Many CGT techniques have been proposed for DL testing with different coverage feedback [31, 43, 59, 62, 77, 88].

The initial seed inputs play an important role for CGT. Traditional CGT usually uses the "singleton" corpus (i.e., a single seed), the empty corpus or large corpora including a large number of inputs [35]. Existing works [35, 64] have demonstrated that the corpus minimization technique can boost the performance of CGT more than the above three strategies in most cases. Since singleton and empty corpus are usually not applicable for CGT in DL systems, most DL testing works randomly select a number of inputs from the training or test dataset. However, it is unknown whether and how the seed corpus affects the DL testing.

[2]Our code and data are available on the website [1].

2.2 DL Testing Evaluation

There are three common testing goals for DL testing: the coverage achieved, the number of failures detected and the robustness enhancement based on the generated test cases. In this paper, we mainly study the impact of seed selection on these goals.

Coverage. Coverage is a popular metric used in DL testing. DL coverage metrics follow the design ideas of traditional software testing metrics (i.e., branch coverage, line coverage and so on). The intuition is to measure the diversity of test cases in terms of exploring behaviors of DL systems. Multiple neuron-based metrics for neural networks have been proposed [40, 49, 62, 73].

For example, *Neuron Coverage (NC)* is built on a threshold for the output value of each neuron. Given an input, if the output value exceeds the defined threshold, it is considered to be activated. Otherwise, it is not activated. NC measures the ratio of activated neurons.

k-Multisection Neuron Coverage (KMNC) is a fine-grained coverage metric that partitions the range of values of each neuron into k sections (the values are obtained during the training process) and measures the ratio of all sections covered by the test cases.

Likelihood-based Surprise Coverage (LSC) measures the relative novelty (i.e., surprise) of a given new input with respect to the inputs used for training. It is defined by using bucketing to discretize the space of Likelihood-based Surprise Adequacy (LSA). LSA uses Kernel Density Estimation (KDE) to estimate the probability density of each activation value in the activation traces and obtains the surprise of a new input with respect to the estimated density, which will be detailed in 3.2.3. Given an upper bound U of LSA, and buckets $B = \{b_1, b_2, ..., b_n\}$ that divide $(0, U]$ into n LSA segments, LSC for a set of inputs X is defined as follows:

$$LSC(X) = \frac{\left|\left\{b_i \mid \exists x \in X : SA(x) \in \left(U \cdot \frac{i-1}{n}, U \cdot \frac{i}{n}\right]\right\}\right|}{n} \tag{1}$$

While we use the coverage to name LSC, we have to notice that LSC is different from the traditional structural coverage, such as NC and KMNC. This is because there is no finite set of targets for LSC to cover and LSC does not render itself to a combinatorial set cover problem.

Failure detection. A significant goal of DL testing is to identify incorrect predictions of DL systems (i.e., failures) from a given set of seed inputs. In this paper, we use the number of failures found within the same iterations to compare the performance of different seed selection strategies on failure detection.

Robustness. Recent work RobOT [82] argues that DL testing should be aware of robustness improvement, i.e., the generated test cases should be valuable in improving model robustness after retraining. Specifically, RobOT proposes a metric, named First-Order Stationary Condition (FOSC), which measures the potential of improving the robustness for a test case.

Intuitively, a test case that induces a higher loss is a stronger adversarial example and is more helpful for training robust models. Another intuition is that the loss around a given seed input often first increases and eventually converges as we follow the direction of the gradient to modify the seed. Thus, a test case with better convergence quality corresponds to a higher loss than its neighbors and is more useful for improving model robustness. FOSC provides a measurement of the loss convergence quality of the generated test cases. Let x_0 be a seed input, we assume that a test case x^t is generated in the neighborhood ϵ-sphere around x_0, i.e., $\chi = \left\{x \middle| \|x - x_0\|_p \leq \epsilon\right\}$ (x^t is generated from x_0). Formally, given a seed input x_0, its neighborhood area $\chi = \left\{x \middle| \|x - x_0\|_p \leq \epsilon\right\}$, a test case x^t, and a DL model f (θ is the parameters of f), the FOSC value of x^t is calculated as:

$$c\left(x^t\right) = \epsilon \left\|\nabla_x f\left(\theta, x^t\right)\right\|_1 - \left\langle x^t - x_0, \nabla_x f\left(\theta, x^t\right)\right\rangle \tag{2}$$

The FOSC value represents the first-order loss of a given test case. The loss of a test case converges and reaches its highest value when its FOSC value is zero. Therefore, a lower FOSC value means better convergence quality and a higher loss.

2.3 Seed Selection for DL Testing

Some recent works on search-based DL testing propose initial seed selection strategies from different testing goals. DeepHyperion [95] selects seeds from the feature space that represent meaningful properties of the test scenarios, i.e. discriminative and interpretable properties of the inputs, or behavioral properties manifested by the DL system when exercised by the test inputs. DeepHyperion considers two domains: handwritten digits and autonomous driving. For digit classification, they convert seeds from MNIST to SVG and evaluate the fitness (i.e., the difference between the confidence level associated to the expected class and the maximum confidence level associated to any other class) and feature (i.e., boldness, smoothness, discontinuity, and rotation) values of the seeds. Then with the values, they find the corresponding cells of the feature map. Starting from a randomly selected seed, DeepHyperion greedily selects the most diverse seed sets by computing the pairwise Manhattan distance. DeepMetis [65] constructs the initial population by computing Euclidean distances between bitmaps on MNIST and greedily constructing the set of most diverse seeds, starting from a randomly selected first seed. FITEST [3] aims to select an initial population that includes a diverse and randomly selected set of test input vectors. They use an adaptive random search algorithm that attempts to maximize the Euclidean distance between the input vectors. SEDE [20] measures how much an individual contributes to the diversity of the population by measuring its distance from the closest individual in the population to compute the fitness value, and then selects the individuals with the best fitness. They measure the distance between two individuals based on individuals' chromosome vectors which contain simulator parameter values.

In this paper, we add the seed selection strategies in DeepHyperion and DeepMetis as two baselines. Since FITEST and SEDE both evaluate the diversity by comparing the distance between the input vectors, their customized input vectors that are related to the self-driving domain or in-car sensing task are difficult to transfer to our datasets. Considering the adaptive random search algorithm in FITEST is similar to the search algorithm in DeepHyperion and DeepMetis, and the features in DeepHyperion have been validated as meaningful features by experts on the MNIST dataset, we use DeepHyperion to maximize the distance of features between seed inputs to select the most diverse seed set. For the selection strategy in DeepMetis, since it does not need the manual definition of features, we can extend it to other datasets, so we adopt this approach on four datasets.

3 METHODOLOGY

3.1 Problem Definition and Overview

3.1.1 Problem Definition. Intuitively, seed selection is to select an optimal seed corpus from a given large number of inputs (e.g., training or test dataset) in order to achieve better testing performance. Formally, given a CGT tool T, a DL model f, a large collection of inputs $\mathcal{I}$, and a time budget t, the seed selection (ϕ) is to select an optimal subset $C \subseteq \mathcal{I}$ in terms of a testing goal G such that:

$$C = \arg\max_{C \subseteq \mathcal{I}} G(T(f, C, t)) \tag{3}$$

where $T(f, C, t)$ takes in the target model f, the initial seed inputs C and a time budget t, and outputs a set of new test cases; G is a specific testing goal calculated from the test output. For example, based on the generated test cases, we can calculate the coverage achieved, the number of failures detected or the robustness improvement.

Seed selection is an NP-hard problem [64]. A common approach is to use an approximation algorithm that can approximate an optimal seed corpus. More details can be found in Section 3.2.

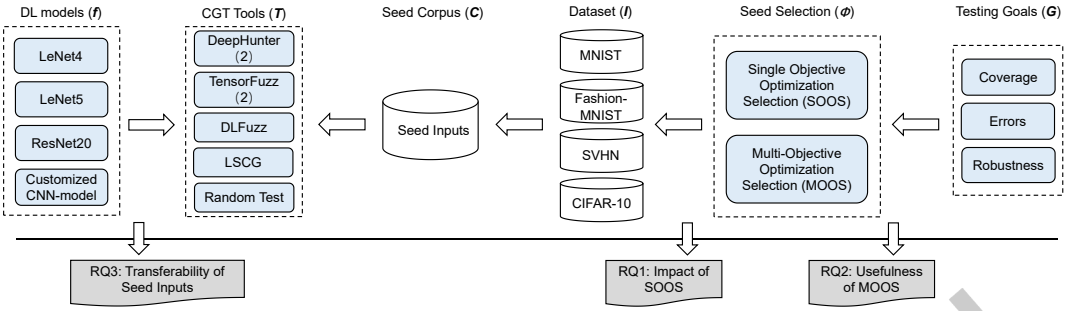


Fig. 1. Overview of our work.

3.1.2 Overview. Figure 1 shows the overview of our study. In general, we aim to study the impact of different seed selection strategies on testing performance. We select 4 widely used datasets (i.e., MNIST, Fashion-MNIST, SVHN, and CIFAR-10) and 8 models (two model architectures for each dataset, i.e., LeNet-4, LeNet-5 for MNIST and Fashion-MNIST, a CNN-model[3], ResNet-20 for SVHN and CIFAR-10). We select 5 different testing techniques including 3 state-of-the-art CGT techniques (i.e., DeepHunter, TensorFuzz, and DLFuzz), 1 proposed Likelihood-based Surprise Coverage Generation Technique (LSCG), and 1 random testing technique. To select seed inputs for the specific goals (i.e., coverage, failure detection, and robustness improvement), we develop multiple seed selection strategies including 3 single-objective optimization-based selection strategies and 2 multi-objective optimization-based selection strategies. With the proposed testing goals and seed selection strategies, we conduct systematic experiments to answer the three research questions introduced in Section 1.

3.2 Seed Selection

In this section, we design different seed selection strategies for DL testing based on the three testing goals.

3.2.1 Metrics used for testing goals. For different testing goals, we select different metrics as the guidance of seed selection:

- **Coverage:** Coverage is a widely-used goal to evaluate the effectiveness of testing. Intuitively, the higher the coverage, the more diverse behaviors might be explored, therefore more different failures of the model are more likely to be found. Based on different coverage goals, we can use the corresponding coverage metric to guide the seed selection. In this paper, we mainly focus on two coverage metrics which are chosen respectively from two typical works proposing coverage criteria: DeepXplore's Neuron Coverage (NC) [62] and DeepGauge's k-Multisection Neuron Coverage (KMNC) [49]. Similarly, our method can be easily generalized to other structural coverage criteria.
- **Failure Detection:** In order to discover failures in the model, we hope to generate more test cases that are mispredicted by the model. We speculate that the number of failures is related to the uncertainty of the seeds. So we select two metrics that indicate the degree of uncertainty, i.e., Prediction Confidence Score (PCS) [94] and Likelihood-based Surprise Adequacy (LSA) [40] to guide the seed selection. More details will be introduced in Section 3.2.3.

[3]We use the CNN model released on https://github.com/tohinz/SVHN-Classifier for SVHN which achieves high accuracy on SVHN, and we use the same model architecture to train a model for CIFAR-10

- **Robustness:** Differently from traditional testing, failures generated in DL testing are usually used to improve the model robustness. To guide the seed selection for robustness improvement, we adopt the gradient-based metric [82] that is used to generate and select test cases for robustness improvement.

3.2.2 Coverage-guided seed selection. Inspired by the corpus minimization technique that is adopted by traditional CGT to select a minimal set of initial seeds [35, 64, 91], we design a coverage-guided optimization strategy to construct initial seeds for DL testing. The basic idea is to select a minimum set of seeds C from a given large dataset $\mathcal{I}$ such that C can maintain the same coverage with $\mathcal{I}$.

Formally, for a DNN f, we define its coverage targets as $f_c = \{c_1, c_2, ..., c_m\}$ (similar to the branches or statements in code) that depend on the coverage criteria. For example, for Neuron Coverage, c_i indicates whether the neuron i is activated and m is the total number of neurons. For k-Multisection Neuron Coverage, c_i indicates a section in a neuron and m is the number of total sections of all neurons (i.e., $k * n$ where n is the number of neurons). Given an input s, we use $cov(c_i, s)$ to represent whether c_i is covered by s, 1 means covered and 0 means not covered.

Given a large set of initial seeds $\mathcal{I} = \{s_1, s_2, ..., s_n\}$, we select the seed corpus C as:

$$\begin{aligned} C = &\arg\min_{C \subseteq \mathcal{I}} \sum_{s \in C} x_s \\ s.t. \quad &\forall c_i \in CT, \exists s_j \in C.cov(c_i, s_j) = 1 \end{aligned} \tag{4}$$

where $CT = \{c | c \in f_c \bigwedge \exists s \in \mathcal{I}\ s.t.\ cov(c, s) = 1\}$ represents all targets that can be covered by $\mathcal{I}$ and

$$x_s = \begin{cases} 1, & select\ s \\ 0, & not\ select\ s \end{cases}$$

We use a greedy strategy to solve the above optimization problem as shown in Algo. 1. We introduce an optional parameter n that controls the number of seeds to be selected in order to avoid selecting too many seeds. The default value n can be the size of $\mathcal{I}$. Specifically, at each iteration, we select a seed that can cover the most uncovered targets (Line 3). Note that there could be multiple seeds that can cover the same number of targets and we randomly select one of them. The algorithm terminates if the number of seeds exceeds the threshold or all targets are covered. If the expected seed number is not reached when all targets are covered, we can run the algorithm again in the remaining seed pool until the expected number is reached.

One thing that needs to be noticed is that the strategy is not suitable for optimizing SC, because a single input yields only a single SA value that does not belong to multiple SA buckets. From a diversity perspective, we can not decide which seed (i.e., which SA value) is better using SC. Hence, we evaluate the value of the seed based on selecting different SA buckets, i.e., higher/lower/uniform SA values (see Section 3.2.3).

3.2.3 Uncertainty-guided seed selection. For the goal with respect to discovering failures, we select two metrics to measure the potential of generating erroneous test cases, including Prediction Confidence Score (PCS) [94] and LSA [40].

PCS measures the probability difference between the two highest softmax outputs. Given a DNN f and an input x, the PCS of the input x on f is calculated as follow:

$$PCS\ (x, f) = \underset{i \in Cls}{max} P\ [i] - \underset{i \in Cls \setminus \{c^*\}}{max} P\ [i] \tag{5}$$

where Cls is the set of classes, $c^* = argmax_{i \in Cls} P\ [i]$ and $P = f\ (x)$ is the output probability vector of the input x over all classes.

PCS provides a quantitative metric to estimate the uncertainty of the model on the input. Intuitively, the smaller the PCS, the closer input is to the decision boundary (between the two classes with the highest probability). Therefore the input with the smaller PCS has greater uncertainty and is more likely to help failure detection.

Algorithm 1: Coverage-guided seed selection

Input: f: the DNN, $\mathcal{I}$: the large seed pool, $n \leq |\mathcal{I}|$: the number of seeds to be selected
Output: C: the selected seed corpus

```
1 C = ∅;
2 while |C| < n ∧ f_c ≠ ∅ do
3     s = arg max_{s∈I} |{c ∈ f_c | cov(c, s) = 1}|;
4     C = C ⋃ {s};
5     I = I \ {s};
6     f_c = f_c \ {c ∈ f_c | cov(c, s) = 1};
7 end
8 return C;
```

LSA measures the relative novelty (i.e., surprise) of a given input with respect to the training inputs. $A_{N_L}(\mathbf{X})$ is a set of activation traces observed over certain neurons for a set of inputs X: $A_{N_L}(X) = \{\alpha_N(x) | x \in X\}$. Given a training set $\mathbf{T}$, LSA uses Kernel Density Estimation (KDE) to estimate the probability density of each activation value in $A_{N_L}(\mathbf{T})$. With a bandwidth matrix H and Gaussian kernel function K, the activation trace of the new input x, and $x_i \in \mathbf{T}$, KDE produces density function $\hat{f}$ as follows:

$$\hat{f}(x) = \frac{1}{\left|A_{N_L}(\mathbf{T})\right|} \sum_{x_i \in \mathbf{T}} K_H \left(\alpha_{N_L}(x) - \alpha_{N_L}(x_i)\right) \tag{6}$$

LSA is defined to be the negative of the log of density, making it negatively correlated with the probability density:

$$LSA(x) = -log(\hat{f}(x)) \tag{7}$$

The study in [40] has demonstrated that inputs with higher LSA are harder to correctly classify, so we wonder whether seeds with higher LSA will also induce more failures to be generated.

Differently from the coverage-guided selection where the coverage target (see f_c in Section 3.2.2) is defined as discrete variables (e.g., a neuron/section is covered or not), PCS and LSA are continuous variables. Hence, we cannot adopt the same strategy with the coverage-guided selection. Instead, we sort the value of these metrics and select the seeds based on their values. In order to verify whether PCS and LSA are related to failure detection and whether seeds with lower PCS and higher LSA are really helpful for failure detection, we set up three ways of selecting seeds, each set of seeds has a different range of PCS/LSA values. We sort the METRIC (i.e., PCS and LSA) value of all seeds $\mathcal{I}$ and use the following strategies in our study:

- *Low-Value of METRIC:* We select the top n seeds with the lowest METRIC value.
- *High-Value of METRIC:* We select the top n seeds with the highest METRIC value.
- *K-Division-Value (KDV) of METRIC:* Inspired by Surprise Coverage (SC) which uses bucketing to discretise the space of surprise, we propose K-Division-Value (KDV) algorithm to investigate whether the degree of diversity is important in this strategy. We divide the continuous METRIC into equal intervals and sample the seeds uniformly that can cover different intervals, which has been detailed in Algo. 2. We first divide the range of METRIC (the smallest METRIC and the largest METRIC among $\mathcal{I}$) into k intervals (Line 2). Then we iteratively select one seed from each interval until n seeds are selected (Line 5-6). *next_interval* defines the next interval. If it is the last interval (i.e., P_k), then the first interval (P_1) is returned. Note that the seed sets in some intervals (i,e., *cur_seeds* at Line 5) may be empty.

Algorithm 2: K-Division-Value (KDV) algorithm

Input: f: the DNN, k: number of intervals, $\mathcal{I}$: the large seed pool, $n \leq |\mathcal{I}|$: the number of seeds to be selected

Output: C: the selected seed corpus

$C = \emptyset$;
Divide METRIC range $[min_{s\in\mathcal{I}} METRIC(s, f), max_{s\in\mathcal{I}} METRIC(s, f)]$ into k equal intervals $[P_1, P_2, ..., P_k]$;
$cur\_itv = P_1$;
while $|C| < n$ **do**
    $cur\_seeds = \{s | s \in \mathcal{I} \bigwedge METRIC(s, f) \in cur\_itv\}$;
    $s = rand\_selectone(cur\_seeds)$;
    $C = C \bigcup \{s\}$;
    $\mathcal{I} = \mathcal{I} \backslash \{s\}$;
    $cur\_itv = next\_interval(cur\_itv)$;
end
return C;

3.2.4 Gradient-guided seed selection. As a data-driven model, the failures discovered are usually used to mitigate the failures via retraining (e.g., adversarial retraining). Hence, another goal is to select seeds that can generate failures for better robustness enhancement. Inspired by the robustness-oriented testing [82], the loss can be used to guide the test case selection for improving the robustness. Intuitively, the test cases with higher loss are selected because they could lead to stronger adversarial examples that are more helpful in training robust models.

For the robustness measurement, we have two steps for the selection: 1) the seed selection that selects seeds for the testing and 2) the failure selection that selects failures (from the discovered failures) for the retraining. Following the work [82], for the first step, we use the gradient of loss to select the seeds. For the second step, we adopt the FOSC (refer to Section 2.2 for more details) to select valuable failures for the retraining.

Formally, for each input $s \in \mathcal{I}$ and the DL model f, we calculate the entropy loss of s on f as:

$$e_{(s)} = crossEntropy\left(f\left(s\right), y\right) \tag{8}$$

where y is the ground-truth label of s. Then the gradient can be calculated as:

$$gradient = \nabla e_{(s)} = \frac{\partial e}{\partial s} \tag{9}$$

Like the selection strategy in [82], we use the seeds with a high gradient of loss, which makes it easier to generate test cases. We sort the seeds $\mathcal{I}$ based on the gradient and select the top n seeds with the highest gradient.

3.2.5 MOO-based seed selection. In the previous sections, we mainly introduce the seed selection based on a single objective metric (i.e., coverage, uncertainty, and gradient). However, our evaluation results show that the seeds selected based on one metric do not work well on other metrics. Considering that we expect to achieve optimized results on multiple goals, we design two multi-objective optimization (MOO) based seed selection strategies.

The non-dominated sorting is usually adopted [46] in MOO because there does not exist a solution that minimizes all objective functions simultaneously. A sample s_1 is said to dominate another one s_2 if there is at least one objective of s_1 better than that objective of s_2 and there is no objective of s_1 worse than that objective of s_2 [8]. s_1 and s_2 are considered equally good if they are not dominated by each other.

To use a non-dominated sorting, we need to define the dominance relation with respect to each metric (i.e., coverage, uncertainty, gradient) between any two seeds s_0 and s_1. For uncertainty, we choose PCS as the ranking metric in MOO-based seed selection, since it finds more failures than LSA. For PCS and gradient, we can directly compare their values. For example, s_0 dominates s_1 in terms of PCS if $PCS(s_0, f) < PCS(s_1, f)$. However, it is difficult to define the dominance for coverage (i.e., which seed gets better coverage) because different seeds often cover different targets. To this end, we propose two different strategies for MOO-based selection.

New Coverage Increase (NCI). Given a DNN f and a set of seeds S that have been selected, then we define the *new coverage increase* from a seed s as:

$$\begin{aligned} CT_s &= \{c | c \in f_c \wedge cov(c, s) = 1\} \\ CT_S &= \{c | c \in f_c \wedge \exists s' \in S \; s.t. \; cov(c, s') = 1\} \\ NCI(f, S, s) &= CT_s \backslash CT_S \end{aligned} \tag{10}$$

Intuitively, NCI measures the new targets that can be covered by s over the existing targets covered by S. Note that S is empty at the beginning. Given two seeds s_0 and s_1, we say s_0 dominates s_1 in terms of coverage if $NCI(f, S, s_0) > NCI(f, S, s_1)$.

After the dominance relations of the three metrics are defined, we adopt the fast non-dominated sorting algorithm [15] to solve this optimization problem, i.e., select the best n seeds. Note that, if the number of the best non-dominated set exceeds the total number we need to select, we then randomly select a certain number of seeds from the set. We denote this strategy as MOO_{NCI}.

Coverage First (CF). A drawback of NCI strategy is that it depends on the seeds S that have been selected. The coverage dominance relation between two seeds may change when different seeds (S) are selected. Hence, we provide another heuristics strategy that first performs an SOO-based selection on coverage and then performs a MOO-based selection on PCS and gradient. Specifically, to select n seeds, we first adopt Algo. 1 to select m ($m < n$) seeds that have better coverage. Then MOO (on PCS and gradient) is used to select the remaining $n - m$ seeds. We denote this strategy as MOO_{CF}.

4 EVALUATION

4.1 Experiment Settings

4.1.1 Test Case Generation. We choose 7 testing tools to evaluate the impact of initial seed selection on test case generation. Specifically, we select 2 kinds of general CGT techniques, i.e., DeepHunter [88] and TensorFuzz [59]. We choose DeepHunter because it is a relatively comprehensive, systematic and effective testing technique among other techniques. Compared with DeepHunter, TensorFuzz has different mutation constraints and seed prioritization strategies (i.e., how to select a seed to mutate during testing), which makes generation tools more diverse. Specifically, DeepHunter adopts the probabilistic seed prioritization strategy, TensorFuzz randomly selects a seed from the reservoir which contains one seed randomly selected from the whole queue and other five seeds picked from the rear of the queue. For DeepHunter and TensorFuzz, we select two widely used metrics, i.e., NC and KMNC, as the coverage feedback, respectively.

Considering both NC and KMNC are based on neuron-based coverage, we also select the metric Likelihood-based Surprise Coverage (LSC) [40] as the coverage feedback. Since there is no existing tool that integrates LSC, we implemented an LSC-guided generation tool, named Likelihood-based Surprise Coverage Generation Technique (LSCG). We calculate the LSA between each generated test case and the training set to obtain the LSC of the test case. If the new test case increases the overall LSC, it will be retained in the queue. Note that LSC calculation is quite slow, and we only measure the LSC results on the LSCG instead of all generators.

DLFuzz [31] is a representative of search-based fuzzing testing techniques, it mutates the input to maximize the neuron coverage and the prediction difference between the original input and the mutated input. We adopt DLFuzz as a test generator, and the mutation process is completed by solving a joint optimization problem of both maximizing NC and the possibility of causing incorrect prediction, which is different from the random-based mutations adopted by DeepHunter and TensorFuzz.

In addition, we also select random testing as a testing tool, it iteratively generates new test cases by randomly selecting which seed in the queue to be mutated and randomly selecting which mutated seeds to retain in the queue without coverage guidance.

Finally, we configure 7 testing tools, i.e., DeepHunter with NC and KMNC, TensorFuzz with NC and KMNC, LSCG, DLFuzz, and random testing.

4.1.2 Research Questions. For RQ1 and RQ2, with the different numbers of seed inputs, we investigate whether and how the SOO-based and MOO-based seed selection strategies boost the testing performance. For RQ3, since we can design different models for a task, we study whether the initial seeds selected from one model can still be useful for testing another model in the same task. The detailed settings of each research question are described later.

4.1.3 Configuration. In order to ensure the fairness of the comparison, given the seed inputs, we run each testing tool with the same number of iterations (i.e., 5,000) and use the same metamorphic mutation strategies (i.e., $\alpha = 0.02, \beta = 0.2$ for DeepHunter [88], TensorFuzz [59] and LSCG, α and β limit the number of changed pixels and the value that a pixel can change respectively). To mitigate the randomness during the testing, we run each configuration 5 times and average the results (for Random selection, we randomly select 5 different seed sets, run each set one time and average the results). For DLFuzz, we use the configuration reported by the author to achieve optimal performance. All the experiments were conducted on a server with the Ubuntu 20.04.1 system with 64-core 2.90GHz Xeon CPU and 125GiB of RAM.

We conduct extensive experiments (more than 14,000 runs of testing) to evaluate the impact of seed selection. Due to the space limit, we only show partial results. More detailed results that show a similar trend can be found on our website [1].

4.2 RQ1: Results of SOO-based Selection

Setup. We evaluate the effect of the 3 SOO-based selection strategies on the DL testing performance with regard to different testing goals. For each strategy, we respectively select 2%, 3%, and 5% of the original seed set size as the expected number. For the robustness improvement, most of the existing work on evaluating robustness reflects robustness by constructing a test set and calculating the accuracy of the retrained model on the test set. Some works construct the test set by combining the mutants generated by mutation operators [27, 28, 34, 40, 45, 70], and some test sets are composed of adversarial examples generated by adversarial attacks [22, 40, 82, 90]. We also use these two methods to construct the test sets to evaluate model robustness. The first test set consists of test failures that are generated by the mutation operators. It represents a test set with the same distribution as the retraining data and we name it $Test_{ID}$. The second one is an adversarial example dataset with a different distribution from the retraining data and we name it $Test_{OOD}$. Evaluating the accuracy of the retrained model on these two datasets can give us a more comprehensive understanding of the improvement in model robustness. $Test_{ID}$ consists of test failures that are not selected as the retraining data. We randomly select $n/(k * m)$ failures from each configuration of each testing tool (n is the total number of failures to be selected, k is the number of testing tools, and m is the number of configurations). $Test_{OOD}$ are the new failures generated by the existing adversarial attacks, i.e., FGSM (Step size = 0.01 for MNIST and CIFAR-10, Step size = 0.05 for Fashion-MNIST and SVHN) and PGD (Step size = 0.01 for MNIST and CIFAR-10, Step size = 0.05 for Fashion-MNIST and SVHN, Steps

= 10). We apply FGSM and PGD to generate one adversarial example for each test data, respectively. We randomly select half of the adversarial examples from each of the two adversarial example sets to build up $Test_{OOD}$. As for the retraining data, after testing under a specific configuration, we respectively add a certain number of test failures (i.e., 1%, 2%, 4%, 6%, 8%, 10% of the original training dataset size[4]) to the training dataset to retrain the model. We select the test failures by adopting the test case selection strategy proposed in [82] (i.e., form retraining dataset by equally combining test cases with small and large FOSC values).

Table 1 shows detailed results related to the coverage (shown in Row NC and KMNC) and the number of failures (shown in Row #Failure). The optimization results that are guided by NC and KMNC are shown in Columns CGS-NC (CGS refers to Coverage-Guided Selection) and CGS-KMNC. Row #Seeds shows the number of seeds selected by different strategies. For instance, we select 200, 300, and 500 seeds for the MNIST dataset. Limited by the space, we only show the testing results with selecting 2% of the original seed set size and one model (LeNet-5 for MNIST and Fashion-MNIST, CNN for SVHN and ResNet-20 for CIFAR-10) here, the rest of the results that show a similar trend are put on our website [1]. The bold numbers show the best results obtained under the same experimental settings across different selection strategies.

The results of two baselines, DeepMetis and Random selection, are put in Table 1. However, DeepHyperion is a domain-specific work, the features designed for MNIST can not be transferred to other datasets. In particular, constructing the feature dimensions in a new domain (e.g., CIFAR) highly relies on the domain knowledge of experts. Hence, we only compare our methods with DeepHyperion on MNIST. The results of DeepHyperion are put in Table 7. Since DeepHyperion randomly selects the first seed, causing that the optimized seed set is not the same at each selection. Therefore, we select five sets of seeds and use them as the initial seeds separately, namely DHp1, DHp2, DHp3, DHp4, and DHp5 in Table 7.

Coverage. We optimize the seeds by using coverage information. We set the threshold of NC to 0.75 which is generally adopted by some popular test generators such as DeepXplore and DeepHunter. For KMNC, we set k to 5 to control the computation, i.e., to keep the number of neuron sections within an acceptable amount of computation. We find that a large proportion of coverage targets can be covered when we use KMNC as the coverage metric, except on SVHN dataset. Further analysis of the results from the SVHN dataset showed that the seeds in the SVHN test dataset mostly cover the same set of neuron sections. This is because KMNC delimits the upper and lower bounds based on the neuron output of the training dataset, while the SVHN training set has some inputs that cause the neurons to output extreme values (i.e., large upper bound or small lower bound), resulting in a large neuron output range. After dividing the neuron output range into k sections (we set $k = 1000$ during test generation which follows the setting in DeepHunter), the range of each section is very large, thus the neuron outputs of most seeds fall into the same range, which causes KMNC to be low for SVHN.

Considering the coverage results in Table 1, we can observe that the coverage-guided selection (or *MOO* strategies) can always select the seeds that achieve the best coverage results than others (i.e., the random selection and uncertainty/gradient-guided selection). The seeds optimized using KMNC (Column CGS-KMNC) can achieve the highest KMNC values than other seed sets in most cases, and so do CGS-NC (*MOO* sometimes achieve the highest NC values on DLFuzz, since they considered coverage during selection). For instance, compared to the random seed selection, KMNC-guided selection can help DeepHunter increase the KMNC value by 9.8%, 14.3%, and 13.2% on MNIST, Fashion-MNIST, and CIFAR-10, respectively. Compared to DeepMetis, KMNC-guided selection can help DeepHunter increase the KMNC value by 9.0%, 10.2%, and 11.5% on MNIST, Fashion-MNIST, and CIFAR-10, respectively. Considering the randomness of test generation, we calculate the p-value and effect size between CGS-NC/KMNC and random selection. In most cases, the p-value is less than 0.05, and the effect size is large. We report the results in Table 3. The p-values less than 0.05 are bolded. We also calculate the p-value and effect size between CGS-NC/KMNC and DeepMetis, the results are shown in Table 4. In most cases, the p-value

[4]We show the average results in Table 6 and detailed results are put on the website [1].

Table 1. Testing results of different seed selection strategies (DH refers to DeepHunter, TF refers to TensorFuzz, RT refers to Random Test, M refers to MNIST, F refers to Fashion-MNIST, S refers to SVHN, C refers to CIFAR-10).

Settings			Optimization Strategies									
Data	Metric	Testing Tool	CGS-NC	CGS-KMNC	PCS-low	PCS-high	PCS-KDV	Grad-high	MOO_{NCI}	MOO_{CF}	DeepMetis	Random
M	#Seeds		200	200	200	200	200	200	200	200	200	200
	NC	DH-NC	**70.9**	68.5	68.6	62.4	67.6	68.1	68.8	69.7	70.5	68.8
		TF-NC	**69.6**	68.1	67.1	60.5	66.9	66.6	67.6	68.5	69.2	67.2
		DLFuzz	70.7	71.6	70.7	40.1	72.0	72.4	**72.6**	71.3	69.9	67.8
		RT	**68.1**	64.7	63.0	48.5	63.3	64.3	64.5	65.8	64.5	59.5
	KMNC	DH-KMNC	69.3	**80.1**	61.8	62.3	64.2	62.1	70.2	76.2	71.1	70.3
		TF-KMNC	46.7	**49.6**	43.3	42.7	43.1	41.7	45.5	47.4	47.4	45.8
		RT	55.4	**60.8**	50.7	49.2	51.7	49.4	54.7	58.1	56.5	55.5
	#Failure	DH-NC	4,133.2	3,373.0	8,455.2	2,255.6	7,665.0	**8,582.6**	7,297.6	5,891.2	3,554.6	2,424.4
		TF-NC	3,211.2	5,358.6	**5,779.8**	3,379.2	4,189.0	4,714.4	4,414.6	3,322.8	2,102.4	2,959.6
		DH-KMNC	2,633.4	1,767.0	**6,883.2**	632.0	6,156.0	6,450.6	5,136.4	3,748.2	2,019.8	1,136.2
		TF-KMNC	302.0	323.2	**986.6**	58.4	862.4	929.4	820.6	473.4	870.8	186.4
		DLFuzz	309.0	204.3	**967.0**	1.0	847.0	915.3	741.7	502.3	98.6	97.0
		RT	638.0	453.0	**2,213.8**	128.6	1,716.6	2,080.6	1,469.0	989.2	530.2	245.6
F	#Seeds		200	200	200	200	200	200	200	200	200	200
	NC	DH-NC	62.6	**62.8**	60.9	60.9	62.0	62.3	62.4	62.4	62.7	62.1
		TF-NC	62.4	**62.8**	60.9	59.7	61.6	61.9	62.0	62.4	62.4	61.9
		DLFuzz	**64.7**	63.8	63.4	63.4	63.1	63.8	63.7	63.9	64.6	64.4
		RT	**62.6**	62.0	58.1	57.8	61.1	60.5	60.8	62.0	61.4	61.6
	KMNC	DH-KMNC	56.3	**71.5**	45.2	58.8	50.4	48.4	58.3	68.0	61.3	57.2
		TF-KMNC	39.5	**47.9**	32.7	44.5	33.8	36.3	40.0	46.3	45.9	40.5
		RT	48.1	**59.3**	37.9	50.5	40.5	41.2	50.5	55.3	51.6	47.7
	#Failure	DH-NC	8,015.0	7,877.8	**19,848.2**	7,369.4	12,468.8	11,408.2	16,449.4	12,748.8	8,758.8	7,794.0
		TF-NC	2,811.6	4,722.6	**15,443.4**	9,256.2	10,118.4	12,066.4	8,815.8	8,582.2	7,676.4	11,215.0
		DH-KMNC	6,729.0	5,694.6	**15,838.8**	3,147.2	11,642.8	12,209.2	14,633.0	8,906.4	6,591.4	5,704.6
		TF-KMNC	1,277.2	954.8	**2,432.0**	466.6	1,911.8	2,312.4	2,362.6	1,605.0	3,843.8	1,118.4
		DLFuzz	884.7	797.7	982.3	707.7	955.0	**987.7**	957.7	882.3	877.4	841.7
		RT	1,584.8	1,405.0	**4,758.4**	930.8	3,709.2	3,264.0	3,884.4	2,303.4	2,251.0	1,863.4
S	#Seeds		500	500	500	500	500	500	500	500	500	500
	NC	DH-NC	**64.7**	60.4	56.1	57.0	56.3	55.4	55.5	62.2	57.4	58.5
		TF-NC	**63.8**	56.4	53.8	54.3	53.9	52.8	52.1	61.6	53.8	54.7
		DLFuzz	73.7	71.2	71.0	71.5	71.6	71.7	71.9	**73.8**	70.5	71.5
		RT	**57.6**	47.0	44.4	46.1	46.8	44.5	44.2	56.6	45.5	46.9
	KMNC	DH-KMNC	0.85	**1.01**	0.69	0.90	0.84	0.66	0.65	0.78	1.00	0.85
		TF-KMNC	0.77	**0.93**	0.62	0.81	0.77	0.60	0.58	0.69	0.92	0.77
		RT	0.69	**0.83**	0.54	0.73	0.67	0.52	0.51	0.62	0.82	0.69
	#Failure	DH-NC	5,447.8	4,523.2	**19,590.0**	3,157.6	4,908.0	12,875.6	18,299.8	15,185.8	4,487.0	4,863.0
		TF-NC	2,880.2	2,805.0	**13,893.2**	2,697.4	3,311.4	8,697.4	13,152.6	8,198.2	3,031.8	3,763.0
		DH-KMNC	3,031.0	2,029.8	**13,270.2**	1,710.8	2,717.0	9,078.8	13,188.0	8,102.5	2,005.2	2,634.6
		TF-KMNC	1,912.0	1,229.6	7,390.0	985.6	1,698.0	5,601.8	**7,955.0**	4,991.4	1,076.2	1,357.2
		DLFuzz	2,282.0	1,971.3	**2,490.0**	2,056.0	2,283.7	2,477.0	2,489.0	2,430.0	2,000.0	2,258.7
		RT	1,392.6	690.2	8,316.6	422.4	1,146.4	5,453.2	**8,451.0**	5,254.4	850.8	1,120.4
C	#Seeds		200	200	200	200	200	200	200	200	200	200
	NC	DH-NC	**28.3**	25.0	19.1	21.0	22.5	18.8	22.3	24.5	17.4	20.8
		TF-NC	**29.4**	25.4	21.3	21.5	22.3	18.1	22.5	24.5	18.5	19.1
		DLFuzz	70.7	**71.2**	70.3	69.8	70.1	69.5	70.6	70.7	70.2	69.8
		RT	**18.5**	15.9	13.8	12.5	14.1	12.6	15.4	14.7	12.3	13.8
	KMNC	DH-KMNC	77.7	**85.5**	68.9	74.3	69.5	68.2	80.4	82.5	74.0	72.3
		TF-KMNC	55.9	**61.8**	49.9	54.8	52.1	50.2	54.9	58.3	54.5	52.1
		RT	68.4	**73.3**	60.0	63.1	59.6	58.8	67.0	69.6	64.4	61.7
	#Failure	DH-NC	10,450.0	9,361.4	**20,386.8**	7,635.4	15,993.4	10,216.0	14,350.6	13,008.6	9,493.4	9,325.0
		TF-NC	15,968.8	11,903.2	17,710.4	**21,734.6**	17,356.2	14,815.4	14,012.4	3,272.8	9,187.4	14,777.0
		DH-KMNC	6,622.4	5,572.6	**18,510.8**	3,037.8	13,608.0	5,810.6	9,321.2	7,799.2	6,128.0	6,401.4
		TF-KMNC	2,953.6	2,785.8	**5,632.2**	1,303.4	5,120.6	2,088.6	4,056.8	3,695.0	2,656.6	2,457.2
		DLFuzz	919.7	889.7	**929.3**	898.7	930.3	940.0	951.0	901.0	876.0	909.0
		RT	3,772.6	2,720.0	**9,610.0**	1,284.6	7,182.2	3,120.2	5,221.8	4,365.4	5,916.0	3,459.6

is less than 0.05, and the effect size is large, except for some results of TensorFuzz. This is because the variance of results on TensorFuzz is very large, resulting in poor p-value results. To display the effect of our strategy more intuitively, we plot the coverage-increasing trends on four datasets under the criteria KMNC and NC in Figure 2, and the trends of Random Selection are drawn as a reference. The solid line represents testing with the seeds optimized by coverage. The dashed legend with the suffix '-r' means testing with randomly selected seeds. Obviously, testing with seeds selected by coverage-guided optimization strategy always achieves better coverage than random selection throughout the 5,000 iterations. To make the results more clear, we also give the AUC of coverage trends in Figure 2, the results are shown in Table 2. We conduct a statistical analysis of the results, the AUC values that are significantly better than the compared ones are underlined. We can find that the AUC values of our strategies are always higher and significantly better than the AUC values of random selection. In Figure 2, we do not plot the coverage trend of DLFuzz, because DLFuzz cannot set the number of iterations. Different initial seeds will lead to different iterations, and comparing the coverage trend under different iterations is not reliable. So we only compare the final coverage in Table 1. Due to the space limit, the comparisons between coverage-guided strategy and DeepMetis (trends and AUC) are shown on the website[1], and the results show that our strategies are still effective, especially on large datasets.

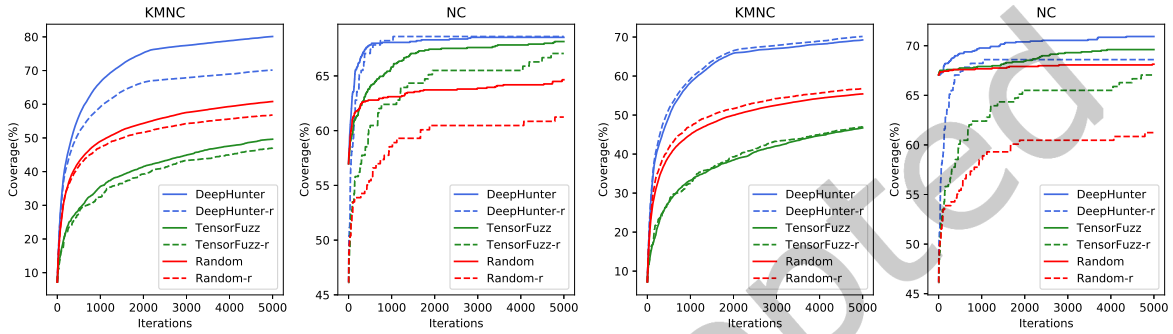
Table 2. AUC of coverage trends.

Dataset	Metrics	DeepHunter	DeepHunter-r	TensorFuzz	TensorFuzz-r	Random	Random-r
MNIST	KMNC	0.8943	0.7894	0.5105	0.4663	0.6653	0.6144
	NC	0.9919	0.9600	0.9705	0.9029	0.9582	0.8271
Fashion-MNIST	KMNC	0.9029	0.7182	0.8507	0.7077	0.7951	0.6584
	NC	0.9985	0.9912	0.9969	0.9797	0.9989	0.9765
SVHN	KMNC	0.9388	0.7742	0.8507	0.7077	0.7951	0.6584
	NC	0.9638	0.8403	0.9510	0.7817	0.8839	0.6986
CIFAR-10	KMNC	0.9416	0.7965	0.6499	0.5709	0.7960	0.6939
	NC	0.9240	0.6820	0.8866	0.6261	0.6097	0.4519

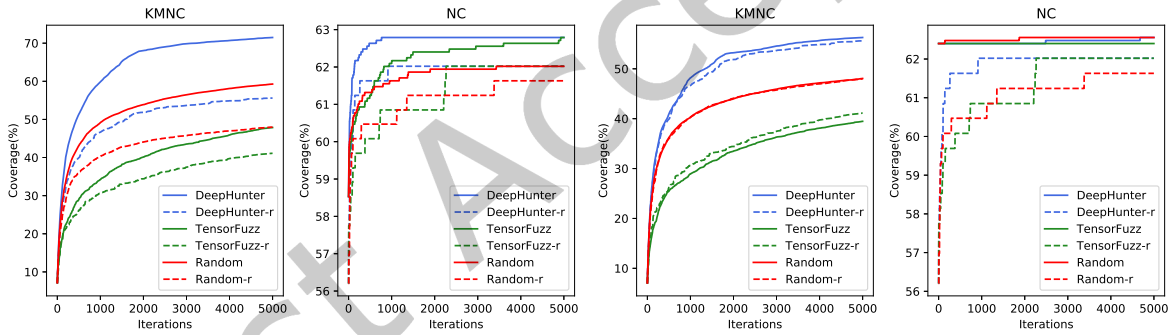
Failures. For each dataset, we select three sets of initial seeds with low-value, high-value, and K-division-value (KDV) of PCS, i.e., PCS-low, PCS-high, and PCS-KDV, respectively. Overall, we can observe that PCS-low discovers more failures in most cases than others because it tends to select seeds that are closer to the decision boundary. Conversely, PCS-high discovers fewer failures as it tends to select more robust seed inputs. For example, on SVHN, DeepHunter with PCS-low seed set detects 19,590 failures under NC-guidance while PCS-high and PCS-KDV only uncover 3,158 and 4,908 failures. In most cases, under the same configuration, PCS-low can find the maximum number of failures compared with other seed sets (including baselines). For instance, on MNIST, Random Test with PCS-low seed set detects 2,213 failures while DeepMetis and Random selection only uncover 530 and 245 failures. Table 3 shows the p-value and effect size between PCS-low/Grad-high and random selection. Table 4 shows the p-value and effect size between PCS-low/Grad-high and DeepMeits. In most cases, the p-value is less than 0.05, and the effect size is large, especially for the results of PCS-low. This demonstrates the effectiveness of our method.

Since the LSCG requires a lot of time (e.g., 12 hours on SVHN) to run the same iterations (i.e., 5000 iterations) as other testing tools (e.g., more than three times the time required for DLFuzz, and more than 25 times the time required for DeepHunter and TensorFuzz), we do not run other seed sets on the LCSG. We report the results of LSA-based optimization strategies on the LCSG in Table 5. To provide more information about the effects of seeds selected by LSA, we use LSA-low, LSA-high, and LSA-KDV as initial seeds for DeepHunter, TensorFuzz, and

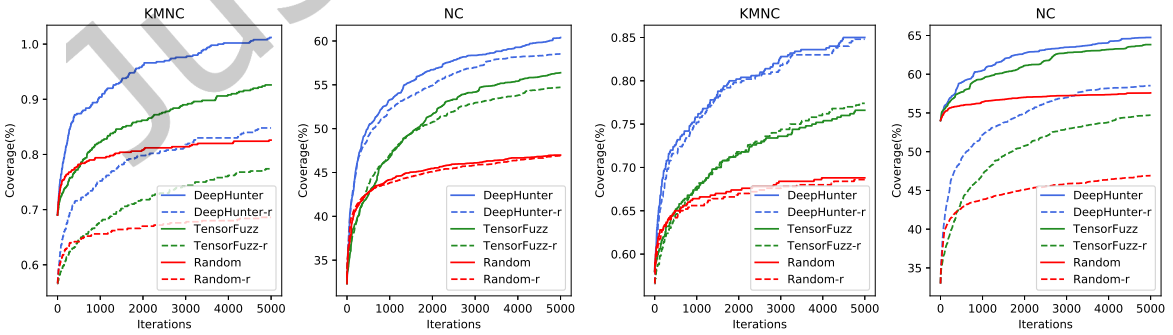
Random Test (we do not show DLFuzz as it takes too long to run), and the results are shown in Table 5. We find that LSA-high finds more failures than LSA-low, LSA-KDV and Random selection on MNIST and Fashion-MNIST in most cases. However, this trend seems to be not evident on SVHN and CIFAR-10. So actually LSA-high is not very useful for failure detection when it expands to larger datasets. Figure 3 shows the trends in terms of failure detected by DeepHunter on four datasets using different seed sets. The results show that, under the same number of seed inputs and iterations, the PCS-low strategy can detect more failures than random selection and other strategies. LSA does not have a good effect on failure detection.



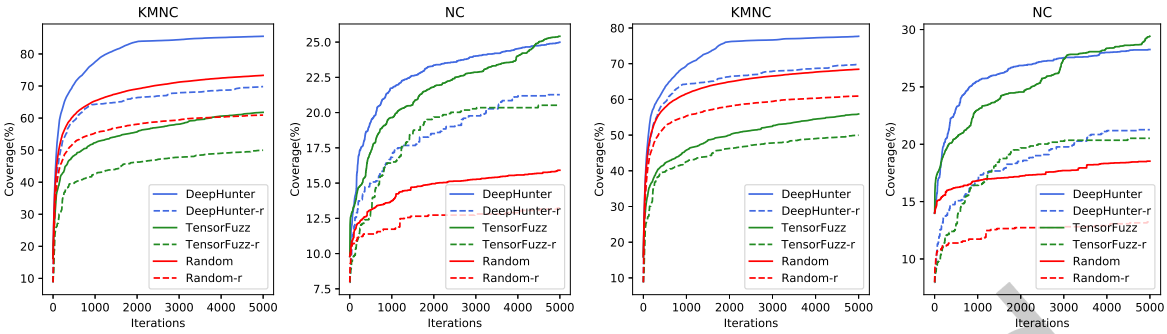
(a) MNIST (CGS-KMNC) (b) MNIST (CGS-NC)



(c) Fashion-MNIST (CGS-KMNC) (d) Fashion-MNIST (CGS-NC)

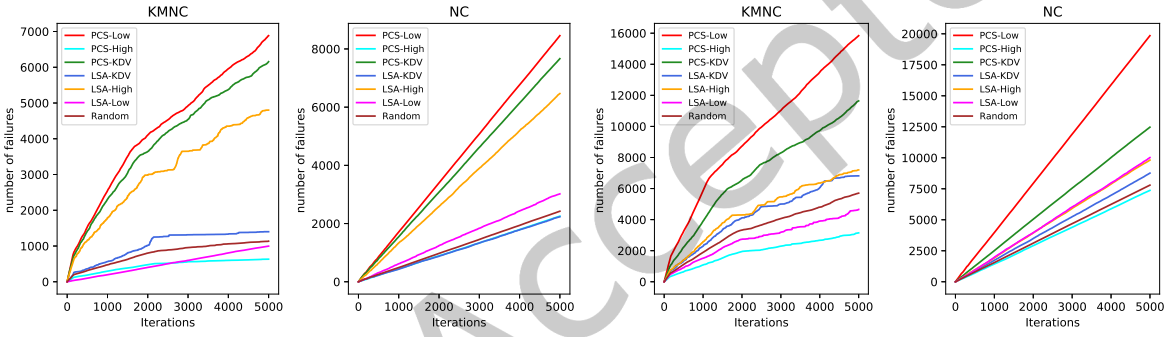


(e) SVHN (CGS-KMNC) (f) SVHN (CGS-NC)

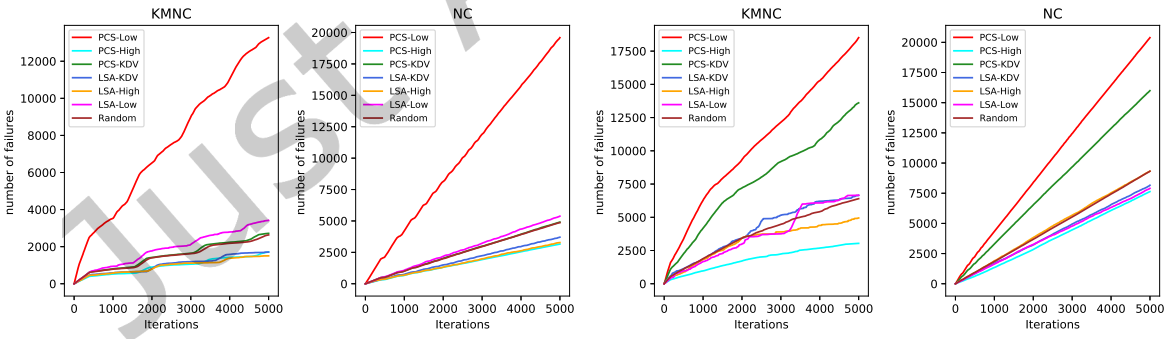


(g) CIFAR-10 (CGS-KMNC) (h) CIFAR-10 (CGS-NC)

Fig. 2. Average coverage achieved on four datasets (using different coverage guidance).



(a) MNIST (b) Fashion-MNIST



(c) SVHN (d) CIFAR-10

Fig. 3. Average number of failures detected by DeepHunter on four datasets.

Another interesting finding is that we expected LSA-KDV to perform well on coverage, but it did not show a significant advantage in coverage compared to other methods. We consider that the selection process of LSA-KDV

Table 3. P-value and effect size when compared with Random Selection (DH refers to DeepHunter, TF refers to TensorFuzz, RT refers to Random Test).

Optimization Strategies	Metric	Testing Tool	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
			p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
CGS-NC	NC	DH-NC	**<0.001**	4.28	**<0.03**	1.66	**<0.001**	5.16	**<0.001**	8.53
		TF-NC	**<0.001**	3.53	**<0.001**	42.43	**<0.001**	12.38	**<0.001**	4.95
		DLFuzz	**<0.05**	1.36	0.30	0.70	**<0.001**	62.23	**<0.001**	19.88
		RT	**<0.001**	20.80	**<0.001**	6.88	**<0.001**	17.82	**<0.001**	4.47
CGS-KMNC	KMNC	DH-KMNC	**<0.001**	17.62	**<0.001**	15.44	**<0.001**	15.71	**<0.001**	15.89
		TF-KMNC	**<0.001**	2.61	**<0.001**	7.13	**<0.001**	15.69	**<0.001**	14.94
		RT	**<0.001**	3.20	**<0.001**	14.55	**<0.001**	13.73	**<0.001**	17.89
PCS-low	#Failure	DH-NC	**<0.001**	64.49	**0.00**	32.22	**<0.001**	62.32	**<0.001**	22.96
		TF-NC	**<0.001**	4.22	**<0.03**	1.82	**<0.001**	8.46	0.07	1.30
		DH-KMNC	**<0.001**	27.20	**<0.001**	14.88	**<0.001**	53.21	**<0.001**	27.24
		TF-KMNC	**<0.001**	6.08	**<0.001**	3.95	**<0.001**	12.53	**<0.001**	5.13
		DLFuzz	**<0.001**	6.53	0.08	1.23	0.45	0.49	0.88	0.09
		RT	**<0.001**	7.86	**<0.001**	7.51	**<0.001**	10.99	**<0.001**	6.48
Grad-high	#Failure	DH-NC	**<0.001**	92.21	**<0.001**	18.07	**<0.001**	54.14	**<0.02**	2.10
		TF-NC	**<0.001**	5.86	0.45	0.50	**<0.001**	5.58	0.20	0.88
		DH-KMNC	**<0.001**	22.57	**<0.001**	9.95	**<0.001**	47.36	0.20	-0.87
		TF-KMNC	**<0.001**	5.19	**<0.01**	2.27	**<0.001**	2.80	0.11	-1.13
		DLFuzz	**<0.001**	5.70	0.07	1.27	0.48	0.47	0.83	0.14
		RT	**<0.001**	9.89	**<0.001**	8.00	**<0.001**	7.34	**<0.02**	-1.95
MOO(CF)	NC	DH-NC	**<0.03**	1.73	0.14	1.04	**<0.001**	3.32	**<0.002**	3.16
		TF-NC	**<0.02**	2.01	**<0.001**	23.57	**<0.001**	5.83	**<0.002**	2.91
		DLFuzz	**<0.03**	1.65	0.10	-1.18	**<0.001**	65.05	**<0.001**	18.77
		RT	0.14	7.87	0.15	1.01	**<0.001**	20.35	0.23	0.82
	KMNC	DH-KMNC	**<0.001**	10.90	**<0.001**	11.91	**<0.001**	-5.15	**<0.001**	11.39
		TF-KMNC	**<0.03**	0.78	**<0.001**	4.63	**<0.001**	-8.36	**<0.001**	5.31
		RT	**<0.02**	1.04	**<0.001**	3.20	**<0.001**	-5.91	**<0.002**	2.95
	#Failure	DH-NC	**<0.001**	47.61	**<0.001**	17.91	**<0.001**	71.69	**<0.001**	6.50
		TF-NC	**<0.01**	2.16	0.88	-0.10	**<0.001**	3.23	0.23	0.83
		DH-KMNC	**<0.001**	14.98	**<0.001**	4.57	**<0.001**	31.02	**<0.002**	2.95
		TF-KMNC	**<0.03**	1.75	0.40	-0.56	**<0.001**	5.84	0.16	0.99
		DLFuzz	**<0.002**	2.82	0.58	0.36	0.58	0.36	0.97	-0.03
		RT	**<0.001**	6.64	**<0.003**	2.80	**<0.001**	11.54	0.48	-0.47

is a strategy to make the seed set more diverse, which will achieve a high LSC and may have some help in achieving higher coverage. But this result does not cater to our conjecture, indicating that this strategy is not as stable as coverage-guided seed selection. Using the KDV algorithm can only ensure that the seeds we choose are relatively diverse, but it is difficult to determine whether they are globally optimal.

Robustness Enhancement. Table 6 shows the robustness evaluation of the newly trained models on failures and adversarial examples. For uncertainty-guided selection, we choose seed sets selected by PCS for comparison because they perform better than those selected by LSA on failure detection, the seed sets selected by LSA are slow in generating failures. However, retraining requires a lot of failures. Considering the time cost, we only use the seeds selected by PCS here. For baselines, we select Random Selection for comparison because DeepMetis is also not good at generating failures. For each dataset, we retrain 6 models using different numbers of failures, and we put the average accuracy here. More details can be found on the website [1]. We can observe that there is no single best strategy that works well on all cases (even including the gradient-guided selection strategy), which is unexpected although we have selected retraining data guided by the metrics [82] that are proposed for

Table 4. P-value and effect size when compared with DeepMetis (DH refers to DeepHunter, TF refers to TensorFuzz, RT refers to Random Test).

Optimization Strategies	Metric	Testing Tool	MNIST		Fashion-MNIST		SVHN		CIFAR-10	
			p-value	effect size	p-value	effect size	p-value	effect size	p-value	effect size
CGS-NC	NC	DH-NC	**<0.003**	2.81	**<0.05**	-1.45	**<0.001**	7.03	**<0.001**	15.58
		TF-NC	0.19	0.90	1.00	0.00	**<0.001**	13.55	**<0.001**	6.19
		DLFuzz	**<0.01**	1.88	**<0.005**	2.36	**<0.001**	90.51	**<0.001**	11.04
		RT	**<0.001**	6.66	**<0.001**	3.25	**<0.001**	12.12	**<0.001**	8.95
CGS-KMNC	KMNC	DH-KMNC	**<0.001**	18.57	**<0.001**	14.40	**<0.01**	2.04	**<0.001**	29.45
		TF-KMNC	0.07	1.33	0.10	1.19	0.14	1.04	**<0.001**	3.95
		RT	**<0.002**	3.09	**<0.001**	6.77	0.12	1.09	**<0.001**	8.48
PCS-low	#Failure	DH-NC	**<0.001**	58.85	**<0.001**	41.58	**<0.001**	21.37	**<0.001**	65.61
		TF-NC	**<0.001**	3.87	**<0.03**	1.67	**<0.03**	1.71	**<0.001**	10.68
		DH-KMNC	**<0.001**	22.60	**<0.001**	17.53	**<0.001**	37.65	**<0.001**	8.49
		TF-KMNC	0.11	1.14	**<0.04**	-1.57	**<0.003**	2.67	**<0.001**	11.96
		DLFuzz	**<0.001**	16.73	**<0.001**	24.41	**<0.001**	34.82	0.28	0.72
		RT	**<0.001**	6.37	**<0.001**	3.82	**<0.001**	4.65	**<0.001**	13.24
Grad-high	#Failure	DH-NC	**<0.001**	94.84	**<0.001**	41.37	**<0.001**	54.14	0.28	-0.73
		TF-NC	**<0.001**	4.99	0.47	0.48	**<0.001**	13.58	**<0.05**	1.48
		DH-KMNC	**<0.001**	18.50	**<0.001**	11.97	**<0.001**	47.36	0.20	-0.52
		TF-KMNC	0.24	0.80	**<0.03**	-1.67	**<0.001**	26.80	0.46	-0.49
		DLFuzz	**<0.001**	10.94	**<0.001**	25.57	**<0.001**	33.90	0.34	0.64
		RT	**<0.001**	7.35	**<0.001**	4.06	**<0.001**	7.34	**<0.02**	-2.09
MOO(CF)	NC	DH-NC	**<0.001**	-4.17	**<0.001**	-35.87	**<0.001**	4.35	**<0.002**	8.70
		TF-NC	0.06	-1.41	1.00	0.00	**<0.001**	6.61	**<0.002**	4.65
		DLFuzz	**<0.001**	25.35	**<0.001**	-13.73	**<0.001**	93.30	**<0.001**	9.94
		RT	**<0.04**	1.56	0.09	1.22	**<0.001**	11.97	**<0.003**	2.74
	KMNC	DH-KMNC	**<0.001**	10.75	**<0.001**	9.66	**<0.001**	-35.31	**<0.001**	16.49
		TF-KMNC	0.95	0.04	0.69	0.26	**<0.001**	-25.84	**<0.02**	1.96
		RT	0.34	0.65	**<0.05**	1.50	**<0.001**	-19.13	**<0.005**	2.45
	#Failure	DH-NC	**<0.001**	39.64	**<0.001**	29.58	**<0.001**	79.21	**<0.001**	3.86
		TF-NC	**<0.03**	1.77	0.89	-0.09	**<0.001**	4.12	0.07	1.33
		DH-KMNC	**<0.001**	9.85	**<0.001**	4.87	**<0.001**	43.78	**<0.001**	3.75
		TF-KMNC	**<0.002**	-3.11	**<0.005**	-2.49	**<0.001**	5.93	0.25	0.79
		DLFuzz	**<0.001**	5.42	0.22	0.83	**<0.001**	30.58	0.855	0.12
		RT	**<0.002**	2.87	0.25	-0.79	**<0.001**	12.12	**<0.03**	-1.68

robustness improvement. It indicates that how to select the optimal initial seeds for robustness enhancement remains to be further investigated. The accuracy on failures is higher than the accuracy on adversarial examples because the new models are trained with some of the failures. Although the retraining data and the test data do not coincide, they may have similar distributions. The average accuracy on four datasets shows that seed inputs selected with coverage and MOO guidance can achieve relatively better results on failures (51.82%, 52.60%, 52.54% and 52.86%). PCS-low achieves the highest accuracy on adversarial examples.

Comparison with DeepMetis. In Table 1, we report the testing results of seed selection strategies in DeepMetis. We find that DeepMetis can achieve good coverage on small datasets (i.e., MNIST and Fashion-MNIST), which is not difficult to understand, as the selection strategy of DeepMetis is to choose more diverse seeds. And we can observe that it still does not have the higher coverage achieved by our method. However, when using this selection strategy on larger datasets (i.e., SVHN and CIFAR-10), the effect is not good. This is to be expected, after all, on complex images, simply calculating the distance between pixels can not reflect the distance between the

Table 5. Testing results of LSA-based seed selection strategies (DH refers to DeepHunter, TF refers to TensorFuzz, RT refers to Random Test).

Dataset	Metric	Testing Tool	LSA-low	LSA-high	LSA-KDV	Random	Dataset	Metric	Testing Tool	LSA-low	LSA-high	LSA-KDV	Random
MNIST	#Seeds		200	200	200	200	Fashion -MNIST	#Seeds		200	200	200	200
	LSC (%)	LSCG	96.9	95.8	**99.2**	98.3		LSC (%)	LSCG	99	**99.7**	99.6	99.6
	NC (%)	DH-NC	40.3	69.4	**69.8**	68.8		NC (%)	DH-NC	44.96	62.4	**62.4**	62.1
		TF-NC	28.27	**68.6**	68.5	67.2			TF-NC	43.8	62.1	**62.5**	61.9
		RT	30.61	**64.2**	61.5	59.5			RT	36.05	61.4	61.4	**61.6**
	KMNC (%)	DH-KMNC	36.6	65.7	**70.5**	70.3		KMNC (%)	DH-KMNC	36.8	**57.6**	57.1	57.2
		TF-KMNC	41.47	44.4	**47.0**	45.8			TF-KMNC	28.79	39.9	39.3	**40.5**
		RT	27.52	53.3	53.2	**55.5**			RT	29.74	48.0	**48.6**	47.7
	#Failure	LSCG	2870.1	**5,906.2**	2,036.3	3,856.4		#Failure	LSCG	6840.2	**9,938.2**	8,499.3	7,167.6
		DH-NC	3020.2	**6,466.0**	2,234.7	2,424.4			DH-NC	**10017.5**	9,815.2	8,764.6	7,794.0
		TF-NC	**5271.2**	3,128.9	1,346.0	2,959.6			TF-NC	**19093.2**	7,461.3	6,497.0	11,215.0
		DH-KMNC	997.5	**4,802.0**	1,400.1	1,136.2			DH-KMNC	4660.4	**7,198.1**	6,817.7	5,704.6
		TF-KMNC	164.1	**565.7**	208.7	186.4			TF-KMNC	682.9	**1,302.3**	1,247.3	1,118.4
		RT	123.2	**1,169.3**	333.0	245.6			RT	886.1	**2,323.7**	2,241.3	1,863.4
Dataset	Metric	Testing Tool	LSA-low	LSA-high	LSA-KDV	Random	Dataset	Metric	Testing Tool	LSA-low	LSA-high	LSA-KDV	Random
SVHN	#Seeds		500	500	500	500	CIFAR-10	#Seeds		200	200	200	200
	LSC (%)	LSCG	**99.9**	99.6	**99.9**	99.8		LSC (%)	LSCG	7.09	**7.1**	6.2	5.9
	NC (%)	DH-NC	56.37	51.0	54.9	**58.5**		NC (%)	DH-NC	17.92	**23.6**	19.9	20.8
		TF-NC	54.61	46.6	52.5	**54.7**			TF-NC	19.6	**22.1**	20.3	19.1
		RT	44.05	38.4	43.9	**46.9**			RT	12.31	**15.0**	14.0	13.8
	KMNC (%)	DH-KMNC	0.65	**1.0**	1.0	0.8		KMNC (%)	DH-KMNC	62.56	77.6	**78.0**	72.3
		TF-KMNC	0.59	**0.9**	0.9	0.8			TF-KMNC	49.92	54.5	**54.9**	52.1
		RT	0.5	**0.8**	0.8	0.7			RT	58.6	**67.7**	66.1	61.7
	#Failure	LSCG	**4864.4**	2,805.6	3,366.2	4,476.2		#Failure	LSCG	6819.9	5,799.1	**7,461.2**	6,064.8
		DH-NC	**5381.7**	3,278.2	3,708.3	4,863.0			DH-NC	7904.2	9,322.9	8,160.5	**9,325.0**
		TF-NC	**5462.4**	2,614.3	2,442.7	3,763.0			TF-NC	5445.5	10,540.0	12,905.0	**14,777.0**
		DH-KMNC	**3419**	1,504.1	1,709.6	2,634.6			DH-KMNC	**6659.1**	4,948.5	6,646.3	6,401.4
		TF-KMNC	**3411.2**	742.0	803.0	1,357.2			TF-KMNC	3332	**3,355.3**	1,893.0	2,457.2
		RT	**2120.7**	684.3	590.3	1,120.4			RT	4541.3	**9,322.0**	3,269.7	3,459.6

Table 6. Robustness improvement (%) of models (retrained using test cases generated from different seed input sets) against test failures ($Test_{ID}$) and adversarial examples ($Test_{OOD}$).

Dataset	Test set	Cov-NC	Cov-KMNC	PCS-low	PCS-high	PCS-KDV	Grad-high	MOO_{NCI}	MOO_{CF}	Random
MNIST	$Test_{ID}$	54.66	58.31	57.48	48.58	56.26	53.74	62.81	**63.31**	52.43
	$Test_{OOD}$	18.69	19.47	**23.57**	21.40	21.91	19.66	19.80	20.47	21.74
Fashion-MNIST	$Test_{ID}$	51.41	**52.62**	39.89	46.69	43.93	41.38	47.40	50.95	49.05
	$Test_{OOD}$	12.12	11.31	12.01	11.57	11.15	11.70	11.86	**12.44**	10.93
SVHN	$Test_{ID}$	50.25	48.86	41.37	50.54	**51.16**	43.46	44.96	44.38	48.68
	$Test_{OOD}$	5.78	**6.64**	5.55	5.34	6.28	4.98	5.14	5.72	5.80
CIFAR-10	$Test_{ID}$	50.96	50.60	47.64	52.45	48.22	51.61	**54.99**	52.82	48.71
	$Test_{OOD}$	22.95	21.58	**23.17**	22.73	22.98	22.52	21.69	21.07	21.21
Average	$Test_{ID}$	51.82	52.60	46.59	49.57	49.89	47.55	52.54	**52.86**	49.71
	$Test_{OOD}$	14.88	14.75	**16.08**	15.26	15.58	14.71	14.62	14.93	14.92

two images on the model feature space. Compared to DeepMetis, our method performs well on larger datasets and finds more failures than DeepMetis, demonstrating the efficiency of our strategies.

Comparison with DeepHyperion. In Table 7, we report the testing results of seed selection strategies in DeepHyperion (DHp). Considering the randomness in selecting the first seed of DHp, we select 5 sets of seeds. We show the average results of 5 seed sets selected by DeepHyperion in Column Avg_{DHp}. To compare clearly, we

also give the results of MOO_{NCI} in Column MOO_{NCI} and give the results of Random selection (another baseline) in Column Random. For a comparison with other strategies, please refer to the results in Table 1.

We find that our strategies can achieve better coverage and find more failures than DHp and MOO also performs better than Avg_{DHp} on each metric, which shows the efficiency of our strategies.

Table 7. Testing results of seed selection strategies in DeepHyperion (DHp refers to DeepHyperion, DH refers to DeepHunter, TF refers to TensorFuzz, RT refers to Random Test)

Settings		Optimization Strategies							
Metric	Testing Tool	DHp1	DHp2	DHp3	DHp4	DHp5	Avg_{DHp}	MOO_{NCI}	Random
#Seeds		200	200	200	200	200	200	200	200
NC	DH-NC	68.3	69.1	68.9	69.8	67.2	68.7	68.8	68.8
	TF-NC	66.8	68.1	67.5	68.8	66.0	67.4	67.6	67.2
	DLFuzz	70.4	70.2	69.4	68.4	68.2	69.3	72.6	67.8
	RT	62.4	63.0	62.5	62.3	61.4	62.3	64.5	59.5
KMNC	DH-KMNC	66.3	68.6	67.3	68.9	67.0	67.6	70.2	70.3
	TF-KMNC	44.7	43.7	44.5	46.8	43.7	44.7	45.5	45.8
	RT	53.3	53.7	53.5	54.6	52.3	53.5	54.7	55.5
#Failure	DH-NC	2,668.0	2,782.0	2,537.4	2,396.8	2,653.8	2,607.6	7,297.6	2,424.4
	TF-NC	1,576.8	3,896.6	1,948.4	2,108.8	2,097.2	2,325.6	4,414.6	2,959.6
	DH-KMNC	1,423.2	1,277.8	1,144.8	1,239.0	1,248.2	1,266.6	5,136.4	1,136.2
	TF-KMNC	169.0	149.0	175.8	175.2	232.8	180.4	820.6	186.4
	DLFuzz	106.7	88.2	95.4	83.9	66.6	88.2	741.7	97.0
	RT	330.6	447.4	272.2	418.2	345.6	362.8	1,469.0	245.6

Comparison across different selection strategies. By comparing the results of different seed selection strategies, we find that both PCS-low and Grad-high have good effects on failure detection, although we expected Grad-high to have a good effect on improving model robustness. We also observe that the selection strategy proposed for a specific testing goal cannot work well on another testing goal. For example, coverage-guided selection can significantly improve coverage but cannot discover more failures than uncertainty-guided selection. Even if both strategies are CGS, different coverage metrics may not necessarily achieve consistent results. For example, the seeds optimized by CGS-NC do not perform as well as randomly selected seeds on KNMC in a few cases. That is because NC is not as fine-grained as KMNC, the seeds which can achieve better NC are not necessarily good on KMNC. Conversely, the seeds which are optimized by KMNC may not perform well on NC. Since the threshold of NC is 0.75, the value range of some neuron sections may be less than 0.75. Seeds with higher KMNC do not mean they would have higher NC. These results suggest that when choosing a seed selection strategy, one should start with the desired testing objectives and choose the corresponding strategy, the desired testing goal should be considered first, and then the corresponding strategy could be selected. If one wants to select a diverse seed set, then the CGS can be chosen. According to different needs, the coverage metric in CGS can be replaced with any metric that can characterize diversity in a similar form. If one wants to generate more failures in a short period of time, then the PCS-low should be used as a favorable seed selection strategy before testing, which can help you save a lot of time. Furthermore, when comparing different testing tools, for the sake of fairness, suitable seed selection should also be taken into account.

Answer to RQ1: With regards to a testing goal such as coverage and the number of failures, the corresponding seed selection strategy can improve the testing performance and make DL testing more efficient. However, the

selection strategy proposed for a testing goal does not work well on another goal. For the robustness goal, none of the selected metrics work well.

4.3 RQ2: Results of MOO-based Selection

Setup. In RQ1, we can find that a single metric guided optimization strategy only performs better on the specific goal, and none of those strategies can make the testing performance better than random selection for all goals. These results may indicate that we need a multi-objective optimization-based selection for achieving better results on multiple goals. We evaluate the effect of the two MOO-based seed selection strategies. We also select 2%, 3% and 5% of the original seed set size as the expected number. Because KMNC is more fine-grained than NC, we use the information of KMNC as the basis for ranking the coverage (except svhn, which performs poorly on KMNC). For MOO_{CF}, m is determined by the optimization results in Algo. 1 (i.e., 154 for MNIST, 132 for Fashion-MNIST, 186 for SVHN and 135 for CIFAR-10).

Table 1 shows the testing performance with 200 input seeds that are selected by MOO-based selection strategies (500 for SVHN). Other results can be found on our website [1]. Overall, we can observe that although the two MOO-based selection strategies do not perform the best on every testing goal, they both achieve a good balance across multiple goals. For the coverage results, we find that the results of MOO_{CF} are very close to the results of coverage-guided selection and they achieve the highest coverage in each strategy in some cases since the seed inputs include the whole first seed set selected by coverage guidance. The coverage achieved by MOO_{NCI} is not as good as MOO_{CF}, but is better than PCS and gradient-guided selection in most cases. This is not surprising since we have anticipated that ranking coverage is not an effective method to select high coverage seeds. For the number of failures, the results are fewer than PCS-low because we selected fewer seeds from PCS-low. However, MOO_{CF} is superior to random selection in coverage and failure in most cases. Table 3 shows the p-value and effect size between MOO_{CF} and random selection.In most cases, the p-value is less than 0.05, and the effect size is large. However, for KMNC of MOO_{CF} on SVHN, the effect size is a negative number. This is because KMNC overall performs poorly on SVHN and we have analyzed the reasons in Section 4.2. Table 4 shows the p-value and effect size between MOO_{CF} and DeepMetis. We can find that in most cases, MOO results are better than DeepMetis results.

Table 6 shows the robustness results for MOO-based strategies (retrained using failures generated from 200 initial seed inputs). We can see that MOO-based selection strategies have better effects of improving model robustness on the failures. On the other hand, the performance on adversarial examples are similar with other SOO-based selection strategies.

Answer to RQ2: Overall, MOO-based seed selection strategies are useful in boosting testing performance, and MOO_{CF} performs better than random selection on all goals. Compared with SOO-based selection strategies, they perform a balance on multiple goals although they do not outperform the corresponding goal. Moreover, MOO-based selection achieves promising results in robustness enhancement.

4.4 RQ3: Transferability of Seed Selection

Setup. In this section, we are concerned about whether the optimal seed corpus selected from one model can be transferred to other models. Parameters (e.g., the weights and biases) and architecture (e.g., LeNet-5 and ResNet-20) are two important factors that make up a model. To evaluate how much they affect the transferability of seed corpus, we consider two different cases to study this problem: the model uses the same architecture but with different parameters and the model uses different architectures. LeNet-4 and LeNet-5 are chosen as the two

different model architectures for MNIST and Fashion-MNIST, Resnet-20 and CNN-model[3] are chosen for SVHN and CIFAR-10 (since we have trained on these models for our dataset in the previous RQs). Then we trained another new model that has the same architecture (LeNet-5 for MNIST and Fashion-MNIST, CNN-model for SVHN and Resnet-20 for CIFAR-10) but different parameters for each dataset. To obtain models with different parameters, we use the original training dataset to retrain the model ten times to fine-tune the model to ensure that the accuracy of the model does not change significantly. We use the seed corpus selected from the original model to be the seed inputs of the new model for testing. We mainly consider the goals for the coverage and failure detection since the robustness improvement has no obvious effect on the original model.

Table 8 and table 9 show the results of the transferability. We select the seed set with the best effect under each metric for transfer. For CGS, we choose CGS-NC for SVHN and choose CGS-KMNC for other datasets. We can observe that after transferring to new models, the testing results of seed sets selected by our methods are still better than random selection in most cases. The coverage-guided strategy usually achieves higher coverage compared to random selection. For the number of failures, PCS-low or Grad-high are still more effective than the coverage-guided selection and random selection. Overall, the seed selection has some transferability between different models in the same task.

Answer to RQ3: The seed selection strategy is still effective when we transfer to other model. The seed corpus selected on a model works on other models on the same dataset.

5 DISCUSSION AND THREATS TO VALIDITY

5.1 Discussion

While DL testing is widely studied and many techniques [40, 59, 62, 77, 88] have been proposed, there is still a lack of study on the impact of seed inputs. The findings in this paper can provide valuable information for the following research:

- *Seed inputs have a large impact.* Our results show that we can select the optimal seed corpus that can largely boost the testing performance for the specific testing goals. In the future research, for a fair comparison of the performance of different testing tools, the selection of seed inputs should be well considered according to the testing goals.
- *Time cost of seed selection.* The time cost of our strategies is worth discussing. The time cost of each strategy is shown in Table 10. We note that the time cost of random selection is lower than seed selection. However, considering that seed selection can enable coverage and the number of failures to reach higher values earlier in the subsequent test generation process, and achieve a higher upper limit of the final value of metrics, the time for seed selection is acceptable. For example, when we select 200 seeds, the KMNC of CGS-KMNC on DH-KMNC takes 75s during the testing process to achieve 57.2% which is the highest KMNC achieved by the Random seed set on Fashion-MNIST. Adding the time that seed selection takes (e.g., 400s), a total of 475s is required, while the Random seed set requires 550s to achieve this coverage. And CGS-KMNC eventually reaches a KMNC value of 71.5%, which is not achievable with randomly selected seeds. On CIFAR-10, the selection of CGS-NC needs 400s, and the NC of CGS-NC on TF-NC requires 75s to reach 19.1% which is the highest NC of the Random seed set. To reach the same coverage, the total time needed by CGS-NC is 475s while the Random seed set needs 1400s. As for the number of failures, on MNIST, the Random seed set finds 97 failures in 2700s, while PCS-low only needs 115s (95s for test generation and 20s for seed selection) to find the same number of failures. On SVHN, within 2000s (sum of the time for test generation and seed selection), Random set finds 750 failures while PCS-low finds 9900 failures. The randomly selected seed set is much slower than the optimized seed set to achieve the same coverage or to find the same number of failures. And randomly selected seeds can

Table 8. The transferability of optimized seed corpus (with different parameters).

Settings			Optimization Strategies					
Dataset	Metric	Testing Tool	CGS	PCS-low	Grad-high	MOO_{NCI}	MOO_{CF}	Random
MNIST	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	69.0	**69.8**	**69.8**	69.6	**69.8**	67.2
		TF-NC	67.4	67.4	67.8	67.4	**68.8**	65.4
		DLFuzz	69.5	69.6	70.3	70.8	**71.2**	62.9
		RT	60.5	59.9	60.5	**62.4**	61.6	57.3
	KMNC (%)	DH-KMNC	**80.6**	64.6	65.0	71.7	78.3	71.8
		TF-KMNC	**52.0**	42.7	43.4	48.2	50.5	48.4
		RT	**62.4**	51.4	51.7	57.1	58.0	57.8
	#Failure	DH-NC	2,874.8	5,791.8	**6,101.6**	4,388.8	5,206.6	4,433.6
		TF-NC	1,734.0	**3,653.8**	3,403.2	3,097.8	3,145.2	2,897.0
		DH-KMNC	1,425.4	4,234.6	**4,505.4**	2,675.8	3,827.4	2,763.8
		TF-KMNC	198.6	545.2	**652.8**	247.0	427.4	325.6
		DLFuzz	137.8	600.1	**635.0**	497.0	381.4	70.3
		RT	349.8	1,192.6	**1,382.2**	673.8	1,202.2	670.8
Fashion-MNIST	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	65.2	65.0	**65.9**	65.1	65.3	65.1
		TF-NC	64.2	64.0	**65.0**	64.1	64.8	64.4
		DLFuzz	67.7	67.2	67.7	67.7	**68.3**	67.7
		RT	63.7	62.6	63.1	63.1	**64.5**	64.2
	KMNC (%)	DH-KMNC	**69.6**	46.1	49.1	56.2	65.6	55.5
		TF-KMNC	**46.7**	33.8	33.5	39.3	43.2	38.3
		RT	**56.4**	38.2	40.6	46.9	53.5	46.6
	#Failure	DH-NC	7,331.2	**14,371.0**	12,649.0	12,721.0	10,198.2	8,166.4
		TF-NC	7,155.8	**20,712.8**	12,715.6	12,318.6	11,537.8	8,474.0
		DH-KMNC	4,893.0	**13,756.2**	13,563.0	13,387.4	8,821.0	6,771.2
		TF-KMNC	920.4	2,340.6	**2,539.0**	1,870.2	1,862.2	1,201.2
		DLFuzz	819.3	**998.0**	997.7	975.5	896.0	868.9
		RT	1,230.8	**4,103.8**	3,951.4	3,822.8	2,923.2	2,511.0
SVHN	#Seeds		500	500	500	500	500	500
	NC (%)	DH-NC	**60.4**	59.5	58.7	58.8	58.8	56.8
		TF-NC	**60.7**	53.9	56.1	54.2	54.0	54.6
		DLFuzz	**73.8**	72.0	72.1	71.9	73.4	73.4
		RT	**51.4**	44.4	44.9	44.3	44.5	45.9
	KMNC (%)	DH-KMNC	**0.9**	0.8	0.7	0.7	0.8	**0.9**
		TF-KMNC	**0.8**	0.7	0.6	0.6	0.7	**0.8**
		RT	**0.7**	0.6	0.6	0.6	0.6	**0.7**
	#Failure	DH-NC	6,121.6	13,767.6	12,850.2	**14,045.0**	13,887.6	5,284.0
		TF-NC	4,795.0	10,258.2	10,745.2	**12,143.0**	10,341.6	4,302.6
		DH-KMNC	2,949.8	8,622.0	9,088.8	**9,254.0**	8,847.4	2,598.4
		TF-KMNC	1,763.6	5,135.8	**5,723.8**	5,691.8	4,995.8	1,422.0
		DLFuzz	2,297.2	**2,490.3**	2,479.8	2,488.1	2,426.0	3,607.4
		RT	1,620.8	6,008.8	5,130.6	**6,615.4**	6,237.4	1,336.2
CIFAR-10	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	21.7	18.1	19.8	**23.8**	23.8	18.7
		TF-NC	21.9	18.8	19.6	21.1	**21.9**	20.2
		DLFuzz	**70.5**	69.0	68.5	69.7	69.7	69.2
		RT	13.6	12.8	11.6	13.4	**13.7**	12.0
	KMNC (%)	DH-KMNC	**84.8**	69.4	69.3	80.5	82.0	72.1
		TF-KMNC	**59.6**	51.3	50.3	57.4	57.8	52.8
		RT	**72.1**	59.8	58.3	68.2	69.6	61.6
	#Failure	DH-NC	8,513.2	**14,368.0**	10,415.8	11,196.2	10,528.0	10,584.2
		TF-NC	9,298.6	**20,189.2**	15,485.4	15,342.0	19,245.2	11,756.8
		DH-KMNC	5,959.2	**13,480.2**	8,352.6	8,196.2	7,701.8	6,853.4
		TF-KMNC	2,966.8	**5,449.8**	2,834.0	3,570.2	3,925.6	3,254.0
		DLFuzz	899.1	970.7	**973.6**	971.5	938.2	951.0
		RT	3,362.8	**7,705.0**	4,986.8	5,785.8	4,389.4	5,048.8

Table 9. The transferability of optimized seed corpus (with different architectures).

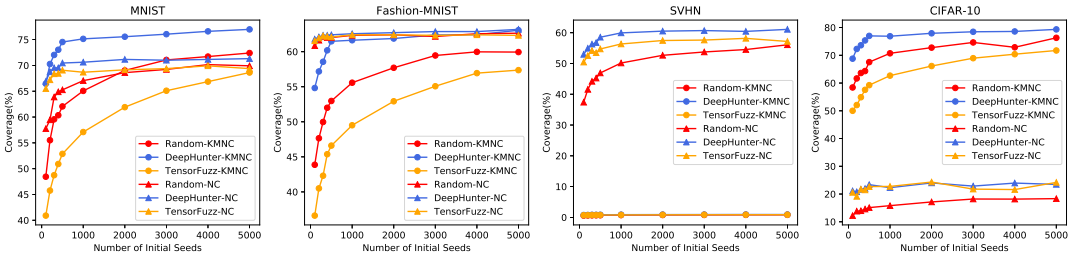
Settings			Optimization Strategies					
Dataset	Metric	Testing Tool	CGS	PCS-low	Grad-high	MOO_{NCI}	MOO_{CF}	Random
MNIST	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	59.4	**62.0**	60.1	60.1	60.1	60.6
		TF-NC	59.6	**61.2**	59.6	60.1	60.1	60.4
		DLFuzz	60.1	61.1	62.0	**62.1**	60.1	58.2
		RT	58.6	57.7	58.4	59.0	**59.6**	59.3
	KMNC (%)	DH-KMNC	**74.1**	66.1	66.3	69.5	72.9	67.4
		TF-KMNC	**50.7**	45.6	45.8	49.7	50.5	47.8
		RT	**59.9**	54.6	54.1	55.9	59.7	54.1
	#Failure	DH-NC	3,456.6	**8,403.6**	8,364.2	7,151.2	5,701.8	2,616.2
		TF-NC	3,957.6	8,322.4	**10,405.2**	6,012.2	5,696.0	4,511.6
		DH-KMNC	2,041.0	6,364.8	**6,412.6**	5,319.0	4,020.0	1,251.6
		TF-KMNC	287.0	**1,134.0**	1,006.2	824.4	503.0	226.0
		DLFuzz	119.0	**416.3**	394.7	309.0	229.2	68.7
		RT	669.8	1,912.0	**1,918.2**	1,363.0	1,122.8	314.4
Fashion-MNIST	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	46.2	42.5	44.3	**48.8**	47.7	45.5
		TF-NC	44.8	41.0	43.8	**46.8**	45.8	44.8
		DLFuzz	48.6	41.5	47.1	47.8	**49.3**	45.9
		RT	41.4	36.2	41.3	41.7	**43.3**	40.6
	KMNC (%)	DH-KMNC	**66.4**	48.2	55.1	57.1	64.1	59.1
		TF-KMNC	44.0	35.5	39.5	40.3	**44.4**	43.1
		RT	**51.6**	38.6	43.9	45.2	49.9	48.4
	#Failure	DH-NC	6,535.8	**11,088.8**	10,135.4	9,940.4	8,473.2	7,199.6
		TF-NC	5,846.0	7,555.8	**7,833.6**	6,971.6	6,530.0	5,896.2
		DH-KMNC	3,631.0	**8,552.2**	7,783.2	7,123.2	5,484.6	4,514.4
		TF-KMNC	437.2	1,229.8	**1,399.8**	1,073.4	788.2	632.2
		DLFuzz	241.3	**461.7**	449.3	430.7	343.3	245.7
		RT	953.8	**2,793.6**	2,575.4	1,790.6	1,820.2	1,138.4
SVHN	#Seeds		500	500	500	500	500	500
	NC (%)	DH-NC	**14.7**	16.9	14.5	15.7	16.8	14.2
		TF-NC	14.0	16.4	15.1	15.9	**16.7**	15.1
		DLFuzz	72.5	**72.6**	72.2	72.4	72.5	72.2
		RT	**11.2**	10.8	10.3	11.0	10.6	10.7
	KMNC (%)	DH-KMNC	5.0	4.7	4.6	4.6	4.7	**5.1**
		TF-KMNC	4.6	4.3	4.3	4.2	4.3	**4.7**
		RT	4.5	4.3	4.2	4.2	4.3	**4.5**
	#Failure	DH-NC	6,634.0	13,461.2	13,616.2	**14,981.6**	14,820.5	6,059.6
		TF-NC	8,278.4	10,206.6	10,761.8	**10,897.8**	19,081.0	6,550.0
		DH-KMNC	5,182.2	11,755.0	12,130.8	**12,324.0**	12,373.0	4,557.4
		TF-KMNC	2,880.2	4,069.6	**4,550.8**	4,665.0	4,532.0	2,914.8
		DLFuzz	1,972.3	2,368.8	2,377.5	**2,400.3**	2,272.0	2,368.0
		RT	1,995.0	5,946.6	6,158.4	**6,494.8**	6,157.5	1,926.2
CIFAR-10	#Seeds		200	200	200	200	200	200
	NC (%)	DH-NC	**59.0**	48.9	47.0	55.9	55.6	53.3
		TF-NC	**55.5**	45.7	45.1	51.6	53.1	51.0
		DLFuzz	**57.9**	53.1	51.4	55.4	56.5	54.1
		RT	**43.2**	34.0	33.1	38.3	39.9	39.0
	KMNC (%)	DH-KMNC	**68.9**	52.0	55.1	61.6	65.5	59.2
		TF-KMNC	**42.2**	34.5	35.2	38.8	39.3	38.0
		RT	**55.3**	42.1	44.8	46.9	51.4	49.0
	#Failure	DH-NC	6,126.6	**11,657.8**	8,905.4	8,883.0	7,633.0	7,856.8
		TF-NC	6,246.6	9,738.4	**9,878.4**	7,587.8	6,625.0	7,048.8
		DH-KMNC	4,898.8	**10,428.0**	6,576.4	6,455.2	6,765.8	6,049.4
		TF-KMNC	1,227.8	**4,336.0**	2,852.0	2,338.6	2,327.4	2,492.2
		DLFuzz	964.2	983.5	**984.3**	977.0	966.9	977.3
		RT	2,644.8	**4,808.0**	4,697.2	3,309.4	3,208.8	3,168.8

never achieve the highest coverage that can be achieved by optimized seeds, which can be found in Figure 2. In the initial seed selection step, the seed selection strategies cost more than random selection, but using seed selection strategies can save much more time and resources in the rest of the testing process.

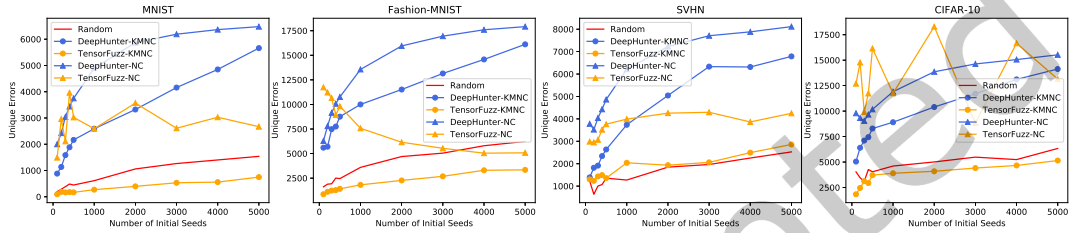
Table 10. The time cost of each strategy.

Settings		Optimization Strategies							
Dataset	Model	CGS	PCS	LSA	Gradient	MOO_{NCI}	MOO_{CF}	DeepHyperion	Random
MNIST	LeNet-5	100s	20s	20s	1s	5000s	120s	400s	1s
Fashion-MNIST	LeNet-5	100s	20s	20s	1s	5000s	120s	\	1s
SVHN	CNN	2700s	90s	90s	40s	140000s	2900s	\	4s
CIFAR-10	ResNet-20	400s	120s	120s	20s	20000s	550s	\	2s

- *The scalability and generalisability of the results.* The scalability of our method is acceptable, as it only affects the seed selection phase before testing. However, for DL testing, scalability and time costs are largely determined by the testing generators employed. The time cost of our seed selection strategies is almost directly proportional to the input and model sizes, and the increase in time cost with increasing size is acceptable. Furthermore, our approach is generalizable and not limited to specific datasets or models. We demonstrate the efficacy of our method on four commonly used datasets, as discussed in the paper, and our results exhibit good generalizability across different datasets. Due to the significant amount of experimentation under various settings and repetitions done in the literature, we did not conduct large-scale experiments, such as ImageNet. In addition, testing tools are quite slow when it comes to testing these large models.
- *Size of the seed inputs.* We notice that existing DL testing works choose different numbers of seed inputs, so we also study the impact of the number of seeds on the testing performance in terms of coverage and the number of failures. Figure 4 shows the results on four datasets. The results show that the coverage is not impacted when the number of seed inputs reaches a certain point, but the number of failures is impacted when the number of seed inputs is changed. This is determined by the characteristics of DL-based software. However, the input space of a DNN could be so large that we cannot test all the inputs given limited computation resource. So seed selection is important and necessary when testing DNNs in reality. We observe that as shown in Table 1, given a testing budget (e.g., selecting limited seed inputs or running testing within a time limit), a better seed selection strategy (e.g., PCS-low) can boost the performance of testing, e.g., discover failures more effectively and efficiently. On the other hand, we may need to consider whether the number of failures makes sense because it is easy to increase the number by selecting more seed inputs. Like traditional software testing, we may find a lot of failed test cases, but we are more concerned with the unique root causes. A big challenge is how to localize the root cause of these failures in DL testing. Some failures discovered from different seeds may have the same root cause. We need another meaningful metric such as the number of unique root causes of failures, not just the number of failures.
- *Decrease of the number of failures generated by TensorFuzz-NC on Fashion-MNIST.* In Figure 4.b, we find that the number of failures generated by TensorFuzz-NC decreases with the increase of the number of seeds on Fashion-MNIST. Our analysis indicates that the test selection strategy of TensorFuzz is limited in the Fashion-MNIST task. Specifically, if a new test case improves the coverage, TensorFuzz will add it into the tail of the test queue. At each fuzzing iteration, TensorFuzz selects test cases to mutate by constructing a reservoir that contains one test randomly selected from the whole queue and the other 5 test cases picked from the end of the queue. Then it randomly selects a seed from the reservoir for mutation. Compared to other datasets, we observe that the coverage of TensorFuzz-NC on Fashion-MNIST reaches high NC faster. When the number



(a) Coverage increasing by each technique with different number of initial seeds on four datasets



(b) Number of failures increasing by each technique with different number of initial seeds on four datasets

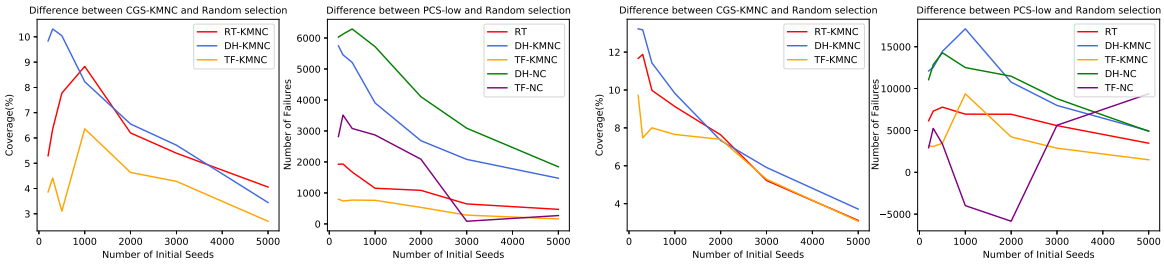
Fig. 4. Results of coverage and the number of failures from each testing tool with different number of initial seeds.

of initial seeds is larger, the initial coverage will be higher, and there will be fewer new test cases that can increase the coverage. Therefore, fewer test cases can be added at the end, and the number of optional test cases for mutation will reduce, which will affect the number of failures generated.

- *The variation of strategy effectiveness with the size of the seed set.* We have observed in Figure 4 that the testing performance can be affected by the size of the randomly selected seed inputs, then the size of the selected seed inputs can also affect the testing performance. For the relationship between the two, our intuition is that with larger seed sizes, the difference in testing performance between our strategy and random selection should be smaller. To verify this hypothesis, we use CGS-KMNC and PCS-low to select additional 1000, 2000, 3000, and 5000 seeds on MNIST and CIFAR for testing. Under the same number of seeds, the differences between the results of optimized seeds and the results of randomly selected seeds are plotted and displayed in Figure 5. We can observe that after reaching a certain number, as the number of seeds increases, the gap between our strategy and random selection becomes smaller and smaller (except for TF-NC, from Figure 4, we can find that the failures detected by TF-NC are not related to the seed size). Considering that we cannot test too many inputs with limited computation and time resources, this result demonstrates the effectiveness and efficiency of our strategies in selecting a limited number of seeds.
- *Designing seed selection strategy for robustness improvement.* As a data-driven software, improving robustness is one of the important testing goals for DL systems. However, our findings show that the state-of-the-art metrics (e.g., FOSC, DeepGini) do not perform well in the seed selection in terms of robustness improvement because improving robustness is a really complex task. However, the MOO-based selection shows some promising results that reveal the potential of combining multiple information (e.g., coverage, uncertainty, and gradient) to design a better metric.

5.2 Threats to Validity

External Validity: The choice of datasets and DNN models are the threats to our results. To mitigate this threat, we use four well-studied datasets and select popular pre-trained DNN models with excellent prediction accuracy



(a) MNIST (b) CIFAR-10

Fig. 5. The variation of strategy effectiveness with the size of the seed set.

on the training dataset. The datasets and models are widely used in the existing works. However, considering the time consumption and the amount of our experiment, we just conduct the experiments on small and simple image datasets such as MNIST, Fashion-MNIST, SVHN, and CIFAR-10. Some larger and complex datasets such as CIFAR-100 or the ImageNet dataset are not evaluated, which may affect the generalisability of the results. We plan to evaluate a wider range of DL models and datasets in the future. Another threat is that some coverage criteria such as Input Distribution Coverage [16] are not evaluated, we plan to evaluate more coverage criteria in the future.

Internal Validity: DL Testing is different from traditional software testing, it is difficult to define the failure of a DNN or to attribute a failure to any one cause. We mutated the inputs at the pixel level to find test cases that make mispredictions on the model. The validity of the generated test cases may be a threat. We can not guarantee that the generated test cases are recognizable to the human eyes, nor can we determine whether the generated test cases belong to different failure types or are generated for different root causes. And we distinguish failures by comparing the hash of two images, which could be a threat. This is mainly reflected in two aspects: the first one is collisions where different images may be reported as the same. The second one is sensitivity, as any changes to the image may result in different hashes. This may allow failure-inducing images modified by one or a few "irrelevant" pixels to be reported as different failures, thereby inflating the evaluation results.

Construct Validity: The coverage criteria used in our study (NC, KMNC, and LSC) could be a threat. We chose widely-used metrics to measure the diversity of inputs for the image classification task, however, whether the values of neuron outputs can represent the diversity of an input exactly is still a problem worth discussing.

To measure the uncertainty, we use the probability difference between the two highest softmax outputs (i.e., PCS) and the relative novelty of a given input with respect to the training inputs (i.e., LSA). They have been proven to be related to uncertainty. They rely on the probability values of the layer output.

For the robustness improvement, most of the existing work on evaluating robustness reflects robustness by constructing a test set composed of mutants generated by mutation operators or adversarial examples generated by adversarial attacks, and calculating the accuracy of the retrained model on the test set, we also use this approach to evaluate model robustness. Using this method to evaluate robustness may be a threat to our results and it is also an important task worth discussing. During the process of retraining, we integrate the measurement method proposed in [82] which is the state-of-the-art work of model robustness improvement. However, the metric is not specifically designed for the selection of initial seeds. That may be a threat to our results. The training process and the seed selection for the robustness goal are still open problems.

Conclusion Validity: There is some randomness during the testing process, which could be a threat. To mitigate this threat, we repeated multiple times for each configuration and averaged the results. In addition, we also ran statistical tests (i.e., p-value and effect size) to assess the significance of our results.

6 RELATED WORK

DL testing. Deep learning testing has been widely studied. DeepXplore [62] proposes the first white-box testing technique guided by coverage metric NC. After that, DeepGauge [49] extends NC and proposes a set of more fine-grained coverage metric including KMNC. Inspired by these works, many DL testing works focus on designing coverage metrics, such as DeepCover [75], DeepCT [48], DeepMutation [50], DeepPath [79], Surprise Coverage [40] and coverage at the layer-level [68]. Based on these metrics, there are some testing techniques designed to generate test cases aimed to increase the coverage, which are called CGT techniques [17, 77].

In this paper, we select three exsiting testing techniques. Specifically, TensorFuzz [59] designed the approximate nearest neighbors algorithms to calculate coverage. DeepHunter [88] proposed some seed sampling strategies and integrated the coverage criteria from DeepGauge [49]. DLFuzz [31] is the first differential fuzzing testing framework, which mutates the input to maximize the neuron coverage and the prediction difference between the original input and the mutated input at the same time. DeepJanus [66] characterizes the frontier of DNN misbehaviours by identifying pairs of inputs that are close to each other, with one input leading to a correct DNN output and the other to a DNN failure. SINVAD [39] is a search-based input space navigation, it uses Variational AutoencoDers (VAEs) to construct a plausible input space that resembles the true training distribution. SINVAD navigates in the space and finds it is a valid way of searching for images that meet desiderata while remaining plausible. DeepHyperion [95] defines specific feature space for DL systems and resorts to Illumination Search to find the highest performing test cases through the map cells which represent the feature space. [19] leverages generative machine learning to generate fresh test inputs that vary in high-level features (e.g., object shape, location, texture, and color). They can detect failures that other existing methods cannot. DeepHyperion-CS [96] enhances DeepHyperion by promoting the inputs that contributed more to feature space exploration during the previous search iterations. We notice that there are extensive DL testing works that focus on designing coverage metrics or test case generation algorithms [86, 89, 93] to detect the vulnerabilities of DL systems. To the best of our knowledge, there is little work evaluating the impact of selecting different seed inputs.

Seed selection in search-based software testing. Search-based techniques have been shown to be a promising approach to tackle software testing tasks [33, 54], for example in the case of test case generation for object-oriented software [78]. Since the outcome of test cases has to be manually verified in most cases, the test suite needs to be small enough for software engineers to control within a feasible time [24]. Seed selection is one such factor that may strongly influence efficiency, thus some seed selection techniques for search-based software testing have been studied.

The most common case of seed selection in the context of search-based software testing regards the case when testing targets (e.g., branches to cover) are sought one at a time [85]. The control dependency graph can be used to choose the order in which to find targets, so when we try to cover a dependent target, we can reuse input data from previous runs. McMinn et al. [56] proposed obtaining seed values from source code and documentation in order to reduce the cost of manual oracle. Fraser and Zeller [25] use common objects for seeding in search to reduce manual oracle costs and improve the readability of generated test cases. Miraz et al. [57] create the initial seed set by selecting the best individuals from a larger seed pool which contains randomly generated individuals. When the code of the SUT is analyzed, Alshraideh and Bottaci [6] proposed a seed selection technique that extracts string constants to use as a starting point for the generation of string inputs, McMinn et al. [55] investigated a strategy that extracts candidates input strings from web queries on search engines based on SUT information. Alshahwan and Harman [5] proposed Dynamically Mined Value strategy for testing web applications, that is, the

HTML pages generated as the output of the test cases are then used as the source of string input for the new test cases in the search.

Seed selection in traditional fuzzing. Fuzzing is a popular technique for finding bugs in traditional software. The most important goal of fuzzing is generating test cases that cause the program crash. There are two kinds of fuzzers that differ in the way they generate test cases: generation-based (e.g., QuickFuzz [30] and CodeAlchemist [32]) and mutation-based (e.g., AFL [91], libFuzzer [69] and honggfuzz [76]).

In traditional fuzzing, seed input collection comes in different ways, e.g., some unreported seeds [36], some manually-constructed seeds [7] and randomly selected seeds [37, 47, 81, 83]. Empty seed is also a popular strategy used for fuzzing [9, 10]. Recently, to improve the fuzzing efficiency, many works start to evaluate the impact of different seed selection strategies, where one of the most effective technique is *corpus minimization* (sometimes called distillation). For instance, [73] first formalized this problem as *Minimum Set Cover Problem* (MSCP) and used a greedy algorithm to solve it. [64] proposed six corpus minimization techniques and found *Unweighted Minset* perform best. *afl-cmin* [91] may be the most popular corpus minimization tool that uses AFL's own notion of edge coverage to minimize seed inputs. *OptiMin* [35] optimizes the corpus minimization by encoding the problem as a maximum satisfiability problem (MaxSAT).

Motivated by these works, we study the impact of different seed selection strategies with regard to different testing goals on DL testing. Due to the fundamental differences between traditional software and DL models, we found that the selection strategies in DL testing are different from traditional software testing (e.g., empty seed is not good in DL testing).

Seed sampling and test selection. Seed sampling is to select a seed from the seed queue during each iteration of fuzzing. The seed is selected to generate new mutants. There are three popular seed sampling strategies: random sampling [59] that randomly selects a seed from the queue, recency-aware seed sampling [59, 77] that prefers to sample seed at the tail of the queue, and frequency-aware seed sampling that sample the seed based on the number of times that the seed was selected. The seed sampling is a different problem with the seed selection in this work.

In order to improve the model robustness, many test selection strategies [22, 40, 45, 82, 84] are also proposed to prioritize the test cases. Without loss of generality, these metrics can be considered as our testing objectives (similar with coverage, PCS). In this work, we select KMNC [49], NC [62], PCS [94], LSA [40], and FOSC [82] to measure the potential of improving coverage, failure detection and robustness. In future work, we plan to add more test selection metrics in our MOO-based selection strategy, which can potentially achieve better testing performance.

7 CONCLUSION

In this paper, we conduct the first study on the impact of seed selection strategies for DL testing. We suppose that seed selection is an important step before DL testing. Our results provide ample evidence for this supposition and demonstrate that testing performance is affected by the initial seed selection. We propose three SOO-based seed selection strategies that can improve testing performance for different goals. We also propose two MOO-based seed selection strategies that can lead to good performance on multiple goals. This study makes the first step along this direction towards considering the importance of seed selection on DL testing. We hope our work draws the attention of researchers working on DL testing, altogether to facilitate further studies on constructing safe and reliable DL systems.

REFERENCES

[1] 2022. *Seed Selection.* https://sites.google.com/view/seedselection

[2] Humberto Abdelnur, Obes Jorge Lucangeli, Olivier Festor, et al. 2010. *Spectral Fuzzing: Evaluation & Feedback.* Ph. D. Dissertation. INRIA.

[3] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 143–154.
[4] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Testing Blog* (2016).
[5] Nadia Alshahwan and Mark Harman. 2011. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 3–12.
[6] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability* 16, 3 (2006), 175–203.
[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.
[8] Chunteng Bao, Lihong Xu, Erik D Goodman, and Leilei Cao. 2017. A novel non-dominated sorting algorithm for evolutionary multi-objective optimization. *Journal of Computational Science* 23 (2017), 31–43.
[9] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724.
[10] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.
[11] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*. IEEE, 39–57.
[12] Oliver Chang, Abhishek Arya, Kostya Serebryany, and Josh Armour. 2017. OSS-Fuzz: Five months later, and rewarding projects.
[13] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 3642–3649.
[14] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of machine learning research* 12, ARTICLE (2011), 2493–2537.
[15] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*. Springer, 849–858.
[16] Swaroopa Dola, Matthew B Dwyer, and Mary Lou Soffa. 2022. Input Distribution Coverage: Measuring Feature Interaction Adequacy in Neural Network Testing. *ACM Transactions on Software Engineering and Methodology* (2022).
[17] 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.
[18] Ranjie Duan, Xingjun Ma, Yisen Wang, James Bailey, A Kai Qin, and Yun Yang. 2020. Adversarial camouflage: Hiding physical-world attacks with natural styles. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 1000–1008.
[19] Isaac Dunn, Hadrien Pouget, Daniel Kroening, and Tom Melham. 2021. Exposing previously undetectable faults in deep neural networks. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 56–66.
[20] Hazem Fahmy, Fabrizio Pastore, Lionel Briand, and Thomas Stifter. 2022. Simulator-based explanation and debugging of hazard-triggering events in DNN-based safety-critical systems. *ACM Transactions on Software Engineering and Methodology* (2022).
[21] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. 2017. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410* (2017).
[22] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–188.
[23] Samuel G Finlayson, John D Bowers, Joichi Ito, Jonathan L Zittrain, Andrew L Beam, and Isaac S Kohane. 2019. Adversarial attacks on medical machine learning. *Science* 363, 6433 (2019), 1287–1289.
[24] Gordon Fraser and Andrea Arcuri. 2012. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE fifth international conference on software testing, verification and validation*. IEEE, 121–130.
[25] Gordon Fraser and Andreas Zeller. 2011. Exploiting common object usage in test case generation. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 80–89.
[26] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. PMLR, 1050–1059.
[27] Xinyu Gao, Yang Feng, Yining Yin, Zixi Liu, Zhenyu Chen, and Baowen Xu. 2022. Adaptive test selection for deep neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 73–85.
[28] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. 2020. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *Proceedings of the acm/ieee 42nd international conference on software engineering*. 1147–1158.

[29] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).
[30] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. 2017. QuickFuzz testing for fun and profit. *Journal of Systems and Software* 134 (2017), 340–354.
[31] 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.
[32] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.. In *NDSS*.
[33] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.
[34] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking neural network robustness to common corruptions and perturbations. *arXiv preprint arXiv:1903.12261* (2019).
[35] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.
[36] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.
[37] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. {FuZZan}: Efficient Sanitizer Metadata Design for Fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 249–263.
[38] Linxi Jiang, Xingjun Ma, Shaoxiang Chen, James Bailey, and Yu-Gang Jiang. 2019. Black-box adversarial attacks on video recognition models. In *Proceedings of the 27th ACM International Conference on Multimedia*. 864–872.
[39] Sungmin Kang, Robert Feldt, and Shin Yoo. 2020. Sinvad: Search-based image space navigation for dnn image classifier test input generation. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 521–528.
[40] 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.
[41] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.
[42] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[43] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[44] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. 2011. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE intelligent vehicles symposium (IV)*. IEEE, 163–168.
[45] Zixi Liu, Yang Feng, Yining Yin, and Zhenyu Chen. 2022. DeepState: selecting test suites to enhance the robustness of recurrent neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 598–609.
[46] Qiang Long, Xue Wu, and Changzhi Wu. 2021. Non-dominated sorting methods for multi-objective optimization: review and numerical comparison. *Journal of Industrial & Management Optimization* 17, 2 (2021), 1001.
[47] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.
[48] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–618.
[49] 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.
[50] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.
[51] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Grant Schoenebeck, Dawn Song, Michael E Houle, and James Bailey. 2018. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613* (2018).
[52] Xingjun Ma, Yuhao Niu, Lin Gu, Yisen Wang, Yitian Zhao, James Bailey, and Feng Lu. 2021. Understanding adversarial attacks on deep learning based medical image analysis systems. *Pattern Recognition* 110 (2021), 107332.
[53] Amit Mandelbaum and Daphna Weinshall. 2017. Distance-based confidence score for neural network classifiers. *arXiv preprint arXiv:1709.09844* (2017).
[54] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[55] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. 2012. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 141–150.
[56] Phil McMinn, Mark Stevenson, and Mark Harman. 2010. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*. 1–4.
[57] Matteo Miraz, Pier Luca Lanzi, and Luciano Baresi. 2010. Improving evolutionary testing by means of efficiency enhancement techniques. In *IEEE congress on evolutionary computation*. IEEE, 1–8.
[58] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
[59] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*. PMLR, 4901–4911.
[60] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. {MoonShine}: Optimizing {OS} Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.
[61] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep face recognition. (2015).
[62] 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.
[63] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. 2016. Variational autoencoder for deep learning of images, labels and captions. *Advances in neural information processing systems* 29 (2016).
[64] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.
[65] Vincenzo Riccio, Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. Deepmetis: Augmenting a deep learning test set to increase its mutation score. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 355–367.
[66] Vincenzo Riccio and Paolo Tonella. 2020. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 876–888.
[67] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.
[68] Jasmine Sekhon and Cody Fleming. 2019. Towards improved testing for deep learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 85–88.
[69] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.
[70] Weijun Shen, Yanhui Li, Lin Chen, Yuanlei Han, Yuming Zhou, and Baowen Xu. 2020. Multiple-boundary clustering and prioritization to promote neural network retraining. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 410–422.
[71] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.
[72] Lewis Smith and Yarin Gal. 2018. Understanding measures of uncertainty for adversarial example detection. *arXiv preprint arXiv:1803.08533* (2018).
[73] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, and Rob Ashmore. 2019. Structural Test Coverage Criteria for Deep Neural Networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.
[74] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Deepconcolic: testing and debugging deep neural networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 111–114.
[75] 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.
[76] Robert Swiecki. 2016. Honggfuzz. *Available online at: http://code. google.com/p/honggfuzz* (2016).
[77] 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.
[78] Paolo Tonella. 2004. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 119–128.
[79] Dong Wang, Ziyuan Wang, Chunrong Fang, Yanshan Chen, and Zhenyu Chen. 2019. DeepPath: Path-driven testing criteria for deep neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 119–120.
[80] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.
[81] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[82] Jingyi Wang, Jialuo Chen, Youcheng Sun, Xingjun Ma, Dongxia Wang, Jun Sun, and Peng Cheng. 2021. Robot: robustness-oriented testing for deep learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 300–311.
[83] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.
[84] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 397–409.
[85] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and software technology* 43, 14 (2001), 841–854.
[86] 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.
[87] Dongxian Wu, Yisen Wang, Shu-Tao Xia, James Bailey, and Xingjun Ma. 2020. Skip connections matter: On the transferability of adversarial examples generated with resnets. *arXiv preprint arXiv:2002.05990* (2020).
[88] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Hongxu Chen, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. 2018. Deephunter: Hunting deep neural network defects via coverage-guided fuzzing. *arXiv preprint arXiv:1809.01266* (2018).
[89] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. DiffChaser: Detecting Disagreements for Deep Neural Networks.. In *IJCAI*. 5772–5778.
[90] Shenao Yan, Guanhong Tao, Xuwei Liu, Juan Zhai, Shiqing Ma, Lei Xu, and Xiangyu Zhang. 2020. Correlations between deep neural network model coverage criteria and model quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 775–787.
[91] Michal Zalewski. 2015. American fuzzy lop (2017). *URL http://lcamtuf. coredump. cx/afl* 14 (2015), 28.
[92] Hugo Zaragoza and Florence d'Alché Buc. 1998. Confidence measures for neural network classifiers. In *Proceedings of the Seventh Int. Conf. Information Processing and Management of Uncertainty in Knowlegde Based Systems*, Vol. 9. Citeseer.
[93] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 949–960.
[94] 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, 739–751.
[95] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. Deephyperion: exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 79–90.
[96] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2022. Efficient and effective feature space exploration for testing deep learning systems. *ACM Transactions on Software Engineering and Methodology* (2022).