Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy

Seokhyun Lee
Korea University
Republic of Korea
seokhyunlee@korea.ac.kr

Sooyoung Cha
Korea University
Republic of Korea
sooyoungcha@korea.ac.kr

Dain Lee
Korea University
Republic of Korea
dain_lee@korea.ac.kr

Hakjoo Oh*
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

ABSTRACT

We present ADAPT, a new white-box testing technique for deep neural networks. As deep neural networks are increasingly used in safety-first applications, testing their behavior systematically has become a critical problem. Accordingly, various testing techniques for deep neural networks have been proposed in recent years. However, neural network testing is still at an early stage and existing techniques are not yet sufficiently effective. In this paper, we aim to advance this field, in particular white-box testing approaches for neural networks, by identifying and addressing a key limitation of existing state-of-the-arts. We observe that the so-called neuron-selection strategy is a critical component of white-box testing and propose a new technique that effectively employs the strategy by continuously adapting it to the ongoing testing process. Experiments with real-world network models and datasets show that ADAPT is remarkably more effective than existing testing techniques in terms of coverage and adversarial inputs found.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Deep neural networks, White-box testing, Online learning

ACM Reference Format:
Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397346

1 INTRODUCTION

Testing deep neural networks is becoming increasingly important. Nowadays, deep neural networks are pervasive in many application domains, including image captioning [25, 36], game playing [28], and safety-critical domains such as medical diagnosis [26] and self-driving cars [2]. Since these applications include deep neural networks as critical components, ensuring that neural networks behave as expected has become a pressing issue. In response, recent years have seen a surge of interest in techniques for testing deep neural networks and identifying their defects before deployment [12, 18, 22, 24, 31, 32, 34]. Furthermore, because traditional coverage metrics (e.g. branch coverage) are not suitable for measuring the effectiveness of neural-network testing [24], developing specialized coverage criteria for deep neural networks continues to be another active research area [15, 18, 24, 27, 32].

Existing testing techniques for deep neural networks are broadly classified into grey-box [22, 34] and white-box approaches [12, 24]. Grey-box testing techniques are based on the idea of coverage-guided fuzzing [37] that has been popular for traditional software, where neural networks are instrumented to trace coverage information and this information is used to generate new test cases that are likely to increase coverage. On the other hand, white-box testing techniques exploit the internals of neural networks more aggressively as follows: (1) they select the internal neurons, (2) calculate the gradients of the outputs of the selected ones (with respect to the input), and (3) generate new test cases by adding the gradients to the original test case in the direction of increasing the output values. Compared to grey-box testing, white-box approaches are therefore likely to achieve higher coverage and find more defects.

A key ingredient of existing white-box approaches is namely a neuron-selection strategy. To determine the direction of the mutation, white-box approaches first select a set of internal neurons and then calculate their gradients. Thus, the ultimate effectiveness of white-box testing depends on the selection of the neurons in the first step. A number of heuristic neuron-selection strategies have been proposed to maximize coverage in a limited time budget. For instance, DeepXplore [24] uses a strategy that randomly selects unactivated neurons. DLFuzz [12] proposed four different neuron-selection strategies for prioritizing neurons covered frequently/rarely, neurons with top weights, and neurons near the activation threshold.

In this paper, we present a new white-box testing technique for deep neural networks. Our technique differs from previous white-box techniques in a crucial way. Existing white-box techniques use the gradients of a select set of internal neurons but the selection is done by a predetermined strategy. However, in this paper, we show that using a fixed neuron-selection strategy is a major limitation of the existing white-box approaches; specifically, we observed that the performance of the existing approaches is not consistent across diverse neural network models and coverage metrics. For instance, although a state-of-the-art tool, DLFuzz, performs well for testing LeNet-5 [17] with Top-k Neuron Coverage (TKNC) [18],

*Corresponding author

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397346

we found that DLFuzz becomes inferior even to a naive random approach for ResNet-50 [13] with TKNC. In this paper, instead of using a predetermined strategy, we present a new testing technique that is able to adaptively determine neuron-selection strategies during the testing process. To do so, we propose a parameterized neuron-selection strategy and provide an online learning algorithm to adjust its parameters effectively.

Experimental results show that our new proposal remarkably improves the effectiveness of white-box testing for neural networks. We implemented the technique in a tool called ADAPT and compared its performance with five existing testing techniques with two coverage metrics, two datasets, and four real-world neural networks. In all experimental settings, ADAPT achieved consistently higher coverage and found more adversarial inputs than existing techniques. On average, ADAPT was 3 times more effective in increasing coverage than other techniques and 6 times more effective in finding various adversarial inputs.

Contributions. Our contributions are as follow:

- We present a new white-box testing approach for deep neural networks. The key novelty is to combine white-box testing with an algorithm that adapts the neuron-selection strategy.
- We demonstrate the effectiveness of our approach with five existing techniques and various experimental settings.
- We provide our implementation as an open-source tool. All experimental results are reproducible.[1]

2 PRELIMINARIES

In this section, we provide background on neural networks, coverage metrics, and existing white-box testing approaches.

2.1 Deep Neural Networks

A deep neural network (*DNN*) is a collection of layers:

$$DNN = \{L_1, L_2, \cdots, L_D\}$$

where D denotes the number of layers called depth. We call L_1 the input layer, L_D the output layer, and $L_2, \cdots, L_{D-1}$ the hidden layers. We define a layer $L_d \in DNN$ to be a set of neurons:

$$L_d = \{n_{d,1}, n_{d,2}, \cdots, n_{d,t_d}\}$$

where t_d indicates the number of neurons in layer L_d. Each neuron in layer $L_d (2 \le d \le D)$ is fully connected[2] to the preceding layer L_{d-1}, where each connection between $n_{d-1,i}$ and $n_{d,j}$ is associated with a weight $w_{d,i,j} \in \mathbb{R}$, where $\mathbb{R}$ denotes real numbers. The weights for neurons in layer $L_d (2 \le d \le D)$ can be represented by a $t_{d-1} \times t_d$-matrix W_d as follows:

$$W_d = \begin{bmatrix} w_{d,1,1} & w_{d,1,2} & \cdots & w_{d,1,t_d} \\ w_{d,2,1} & w_{d,2,2} & \cdots & w_{d,2,t_d} \\ \vdots & & \ddots & \vdots \\ w_{d,t_{d-1},1} & w_{d,t_{d-1},2} & \cdots & w_{d,t_{d-1},t_d} \end{bmatrix}$$

[1] https://github.com/kupl/ADAPT

[2] For presentation simplicity, we focus on fully-connected neural networks but our technique is also applicable to convolutional neural networks.

Assume a t_1-dimensional input vector $I \in \mathbb{R}^{t_1 \times 1}$ is given. We can "execute" the neural network with the input I as follows:

$$V_1 = I, \quad V_d = \mathsf{Layer}_d(V_{d-1}) \ (2 \le d \le D)$$

where $V_d = \langle v_{d,1}, \ldots, v_{d,t_d} \rangle^T$ denotes the output of layer L_d and $v_{d,i}$ the output of neuron $n_{d,i}$. Layer_d is a layer function that takes the output V_{d-1} of the previous layer and produces the output for the layer L_d as follows:

$$\mathsf{Layer}_d(V_{d-1}) = \mathsf{Activation}_d(W_d^T V_{d-1})$$

where $\mathsf{Activation}_d$ is a non-linear activation function for layer L_d. For instance, the dominant ReLU (Rectified Linear Unit) activation function [21] is defined as follows:

$$\mathsf{Activation}_d(\langle x_1, \cdots, x_{t_d} \rangle^T) = \langle \max(x_1, 0), \cdots, \max(x_{t_d}, 0) \rangle^T$$

Given a neural network *DNN* and an input vector $I \in \mathbb{R}^{t_1 \times 1}$, let $\mathsf{Run}(DNN, I)$ be the set of output values of hidden layers in *DNN* when *DNN* is executed with I:

$$\mathsf{Run}(DNN, I) = \{V_2, V_3, \cdots, V_{D-1}\}.$$

Note that V_d denotes the output of all neurons in layer L_d. Thus, $\mathsf{Run}(DNN, I)$ describes the full internal state of *DNN*.

2.2 Coverage Metrics

The adequacy and effectiveness of software testing are typically measured by coverage metrics (coverage criteria). Recently, various coverage metrics for neural networks have been proposed [15, 18, 24, 32]. In this paper, we evaluate and compare testing techniques with two metrics: Neuron Coverage (NC) [24] and Top-k Neuron Coverage (TKNC) [18].

We define a coverage metric, denoted Cov, as a function from the outcome of Run to a set of coverage identifiers. For example, the Neuron Coverage (NC) metric uses the locations of neurons as identifiers (the location of neuron $n_{d,i}$ is (d, i)). NC identifies the neurons whose output values are greater than a certain threshold θ. The function Cov for NC is defined as follows:

$$\begin{aligned} &\mathsf{Cov}(\{V_2, \cdots, V_{D-1}\}) \\ &\quad = \{(d, i) \mid v_{d,i} > \theta \wedge v_{d,i} \in V_d \wedge d \in \{2, \cdots, D-1\}\}. \end{aligned}$$

The Top-k Neuron Coverage (TKNC) metric is more fine-grained; it collects the k neurons with the highest output values in each layer. The coverage function Cov for TKNC is:

$$\begin{aligned} &\mathsf{Cov}(\{V_2, \cdots, V_{D-1}\}) \\ &\quad = \bigcup_{d \in \{2, \cdots, D-1\}} \{(d, i) \mid v_{d,i} \in \underset{S \subseteq V_d \wedge |S| = k}{\mathrm{argmax}} \sum_{s \in S} s\}. \end{aligned}$$

In both cases, the coverage ratio is calculated by $\frac{|\mathsf{Cov}(\{V_2, \ldots, V_{D-1}\})|}{|L_2 \cup \cdots \cup L_{D-1}|}$. In the rest of this paper, we assume a coverage metric Cov is given.

2.3 White-Box Testing of Neural Networks

One of the main goals of neural network testing is to generate test cases (i.e. input vectors) that maximize a given coverage metric (Cov) in a limited testing budget. Assuming that the budget is given as the number of test cases, denoted n, the objective of testing

Algorithm 1 White-box Testing for Neural Networks

```
1:  procedure TestingDNN(DNN, I, Cov)
2:      C ← Cov(Run(DNN, I))
3:      repeat
4:          W ← {I}
5:          while W ≠ ∅ do
6:              I′ ← Pick an input from W
7:              W ← W \ {I′}
8:              N ← Strategy(DNN)
9:              for t = 1 to η1 do
10:                 I′ ← I′ + λ · ∂(Σ_{n∈N} Neuron(n, I′))/∂I′
11:                 O′ ← Run(DNN, I′)
12:                 if Cov(O′) ⊈ C ∧ Constraint(I, I′) then
13:                     W ← W ∪ {I′}
14:                     C ← C ∪ Cov(O′)
15:     until testing budget expires (e.g. timeout)
16:     return |C|
```

is to find a set of input vectors $T = \{I_1, \ldots, I_n\}$ that collectively maximize the coverage:

$$\text{Find } T = \{I_1, \ldots, I_n\} \text{ that maximizes } \Big| \bigcup_{I_i \in T} \text{Cov}(\text{Run}(DNN, I_i)) \Big|.$$

Existing white-box testing techniques aim to solve this optimization problem with a search algorithm guided by the gradient of internal neurons [8, 12, 24]. Algorithm 1 presents the general architecture that encapsulates the existing white-box techniques. The algorithm takes a neural network (*DNN*), an initial input vector (I), and a coverage metric (Cov). At line 2, the set C is initialized to the set of neurons covered by the initial input. Initially, the worklist W is a singleton set $\{I\}$ (line 4). The algorithm starts by selecting an input I' from the worklist W (line 6). At line 8, the function Strategy takes the *DNN* and then selects a set N of neurons (in experiments, we select 10 neurons). At line 10, the algorithm calculates the gradient of the selected neurons N, and then uses the gradient to adjust the input in the direction of increasing the selected output values; the algorithm generates a new input I' by adding the gradient multiplied by the learning rate λ, where Neuron is a function modeled by the neuron $n_{k,i}$ that takes an input I and calculates the output of the neuron $n_{k,i}$ which can be written as follows:

$$\text{Neuron}(n_{d,i}, I) = \begin{bmatrix} w_{d,1,i} \\ \vdots \\ w_{d,t_{d-1},i} \end{bmatrix}^T (\text{Layer}_{d-1} \circ \cdots \circ \text{Layer}_2)(I)$$

At line 12, the algorithm checks whether the new input I' is able to cover new neurons, and whether it satisfies a given constraint (Constraint) on test cases. For example, $\text{Constraint}(I, I')$ is true if L2 distance between I and I' is less than a certain threshold. If both conditions are met, we add the new input vector I' to the worklist W, and also add the newly covered neurons to the set C (lines 13-14). In the inner loop at lines 9-14, the algorithm repeatedly generates new input vectors I' for η_1 times with the same gradient, where η_1 is typically small (e.g. $\eta_1 = 3$ in our experiments). The procedure described above is repeated until the testing budget expires. Upon termination, the number of covered neurons ($|C|$) is returned.

The existing white-box techniques differ essentially in the choice of the neuron-selection strategy (Strategy). For example, DeepXplore [24] uses a strategy that randomly chooses neurons that have never been activated, and DLFuzz [12] supports four neuron-selection strategies. In this paper, we show that the neuron-selection strategy is a key component of white-box testing and propose a new testing technique that adaptively employs neuron-selection strategies.

3 OUR TECHNIQUE

In this section, we present our white-box testing technique for neural networks. The key feature is continuously learning the neuron-selection strategy while performing Algorithm 1. To do so, we present a parameterized neuron-selection strategy and an online learning algorithm.

3.1 Parameterized Neuron-Selection Strategy

We first parameterize the neuron-selection strategy. The parameterized strategy, denoted Strategy_p, selects the neurons based on the parameter, p, which is a vector of real numbers. We define the strategy as follows:

$$\text{Strategy}_p(DNN) = \underset{S \subseteq \bigcup_{d \in \{2,\cdots,D-1\}} L_d \wedge |S| = m}{\text{argmax}} \Big(\sum_{n \in S} \text{score}_p(n) \Big)$$

Intuitively, the strategy takes as input all neurons in the hidden layers (i.e. $\bigcup_{d \in \{2,\cdots,D-1\}} L_d$), and selects the top-$m$ neurons according to their *scores*. The number of selected neurons (m) is a predetermined hyper-parameter (in experiments, we set m to 10).

For scoring, we use a simple method based on linear combination. To score each neuron in *DNN*, we first transform each neuron in the network model *DNN* into a feature vector. A single feature is a boolean function on neurons:

$$\mathsf{F_i} : \textit{Neurons} \rightarrow \{0, 1\}$$

where *Neurons* denotes the set of all neurons in the model *DNN*. For instance, a feature may check whether neurons have been activated or not. We designed 29 features describing properties of neurons. With the 29 features, we can convert each neuron n into a 29-dimensional boolean feature vector as follows:

$$\mathsf{F}(n) = \langle \mathsf{F}_1(n), \mathsf{F}_2(n), \ldots, \mathsf{F}_{29}(n) \rangle.$$

After the transformation, we can score each neuron by calculating an inner product of the feature vector $\mathsf{F}(n)$ and the parameter vector p that is also 29-dimensional:

$$\text{score}_p(n) = \mathsf{F}(n) \cdot p.$$

After calculating the score of each neuron, we select the m neurons with the highest scores.

Neuron Features. We designed 29 atomic features that describe characteristics of the neurons in a neural network. We focused on designing features with simplicity and generality. These features in Table 1 are categorized into two classes: 17 constant and 12 variable features. The key difference between constant and variable features is whether the boolean value of the feature changes during neural network testing; the value of constant feature for the same neuron does not change while the value of variable feature may change.

Table 1: Neuron features

#	Description
1	Neuron located in front 25% layers
2	Neuron located in front 25-50% layers
3	Neuron located in front 50-75% layers
4	Neuron located in front 75-100% layers
5	Neuron in a normalization layer
6	Neuron in a pooling layer
7	Neuron in a convolution layer
8	Neuron in a dense layer
9	Neuron in an activation layer
10	Neuron in a layer with multiple input sources
11	Neuron that does not belong to of 5-10 features
12	Neuron with top 10% weights
13	Neuron with weights between top 10% and 20%
14	Neuron with weights between top 20% and 30%
15	Neuron with weights between top 30% and 40%
16	Neuron with weights between top 40% and 50%
17	Neuron with weights in bottom 50%
18	Neuron activated when an adversarial input is found
19	Neuron never activated
20	Neuron with the number of activations (top 10%)
21	Neuron with activation numbers (top 10-20%)
22	Neuron with activation numbers (top 20-30%)
23	Neuron with activation numbers (top 30-40%)
24	Neuron with activation numbers (top 40-50%)
25	Neuron with activation numbers (top 50-60%)
26	Neuron with activation numbers (top 60-70%)
27	Neuron with activation numbers (top 70-80%)
28	Neuron with activation numbers (top 80-90%)
29	Neuron with activation numbers (top 90-100%)

Constant features consist of 11 features describing the layer where the neuron is located and 6 features describing the neuron itself. Specifically, the constant features 1-4 describe the relative positions of layers in the neural network. We designed the features 5-11 describing the layer type to check the importance of different layer types. Likewise, we designed the 6 constant features to evaluate the importance of the weight each neuron has; we did not deliberately divide the neurons having 50% bottom weights that are likely to be redundant. All constant features can be extracted from the given neural network directly without any calculation.

The variable features 18-29 describe the properties of neurons, where these features continuously change over the testing procedure. We designed feature 18 to check whether there are key neurons in generating adversarial inputs. Depending on the number of activations for each neuron, we have designed features 19-29. All the variable features can be easily extracted with little overhead. We also reflected the key insights of existing white-box testing tools. For example, features 12 and 19 came from the strategies used in DLFuzz [12] and DeepXplore [24], respectively.

3.2 White-Box Testing with Online Learning

We now describe our white-box testing technique (Algorithm 2) that adaptively learns and changes neuron-selection strategies based on the data accumulated during neural-network testing. In parameterized neuron-selection strategy, a 29-dimensional parameter vector of real-numbers corresponds to a single neuron-selection strategy. Hence, we consider a set of parameter vectors as a set of neuron-selection strategies in Algorithm 2.

Algorithm 2 Our White-box Testing with Online Learning

```
1:  procedure ADAPT(DNN, I, Cov)
2:      C ← Cov(Run(DNN, I))
3:      P ← {p_1, ··· , p_η2 | p_i ~ U([−1, 1]^29)}
4:      H ← ∅
5:      repeat
6:          for all p ∈ P do
7:              C_p ← ∅
8:              W ← {I}
9:              while W ≠ ∅ do
10:                 I' ← Pick an input from W
11:                 W ← W \ {I'}
12:                 N ← Strategy_p(DNN)
13:                 for t = 1 to η_1 do
14:                     I' ← I' + λ · ∂(Σ_{n∈N} Neuron(n, I'))/∂I'
15:                     O' ← Run(DNN, I')
16:                     if Cov(O') ⊈ C ∧ Constraint(I, I') then
17:                         W ← W ∪ {I'}
18:                         C ← C ∪ Cov(O')
19:                     C_p ← C_p ∪ Cov(O')
20:             H ← H ∪ {(p, C_p)}
21:             H ← Pop(H, η_3)
22:         P ← Learning(H)
23:     until testing budget expires (e.g. timeout)
24:     return |C|
```

Overall Testing Process. Unlike Algorithm 1, Algorithm 2 maintains the set P of neuron selection strategies in the outer loop at lines 6-21, and changes the set P based on the accumulated data H at line 22. The input and output of Algorithm 2 are identical to the ones in Algorithm 1. At line 3, the algorithm initially generates η_2 random neuron-selection strategies, where $\mathcal{U}$ denotes the probability density function of the continuous uniform distribution. At lines 8-18, for each strategy p, Algorithm 2 performs exactly the same as in Algorithm 1; the algorithm selects the set N of neurons (line 12), and generates the new input I' by adding the gradient of the selected neurons N (line 14). Then, algorithm checks whether the new input is able to cover new identifiers and satisfy the distance constraint (line 16). At line 20, unlike Algorithm 1, our technique keeps updating the data H during neural network testing, where each element in H denotes a tuple of the strategy p and the covered identifiers C_p by the strategy. At line 21, to use the latest information accumulated in H as learning data, the algorithm maintains the size of the data H as η_3 by applying the function Pop; the function returns the set H with the most recently accumulated η_3 elements. Then, it newly generates the set P with the Learning function (line 22).

Learning. The idea of Learning is to identify "crucial" neuron-selection strategies from the accumulated data H and then generate

new strategies by combining the features of those crucial strategies. To do so, the learning procedure (Learning) consists of two steps: Extract and Combine.

In the Extract step, the goal is to collect the set S of crucial strategies from H, where the size of S is fixed by a hyper-parameter η_4. Collecting S is done in the following two steps. First, we collect the set S_1 of strategies from H that collectively maximize the number of covered identifiers. Let H^* be all the possible such subsets of H whose sizes are bounded by η_4:

$$H^* = \underset{H' \subseteq H \wedge |H'| \leq \eta_4}{\operatorname{argmax}} \; \Big| \bigcup_{(\_, C_p) \in H'} C_p \Big|$$

where argmax produces the set of all arguments that maximize the given objective, i.e., $\operatorname{argmax}_{g(x)} f(x) = \{x \mid g(x) \wedge \forall y.\ f(y) \leq f(x)\}$.[3] With H^*, we can define the set S_1 of strategies as follows:

$$S_1 = \{p \mid (p, \_) \in \underset{H' \in H^*}{\operatorname{argmin}} |H'|\}$$

where argmin arbitrarily picks a smallest $H' \in H^*$ when it is not unique. Intuitively, the set S_1 denotes the minimum set of strategies that collectively maximize the number of covered identifiers. If the size of the set S_1 is less than η_4, we additionally collect the set S_2 of $\eta_4 - |S_1|$ strategies with the maximal number of covered identifiers. That is, we aim to find the set S_2 defined as follows:

$$S_2 = \{p \mid (p, \_) \in \underset{H' \subseteq H \wedge (|H'| = \eta_4 - |S_1|)}{\operatorname{argmax}} \sum_{(\_, C_p) \in H'} |C_p|\}.$$

Note that, because we collect S_1 and S_2 from the set H, a single strategy p can be an element of both S_1 and S_2. In this case, we assume such a strategy is more promising than the strategies exclusively contained in one of the two sets. Thus, we define the final set S to be the following:

$$S = \{\!\{p \mid p \in S_1\}\!\} \cup \{\!\{p \mid p \in S_2\}\!\}$$

where the notation $\{\!\{\}\!\}$ indicates multisets that allow duplicated elements.

For example, suppose that the accumulated data H is the following:

$$H = \{(p_1, \{i_1, i_2, i_3, i_4\}), (p_2, \{i_2, i_4\}), (p_3, \{i_1, i_3\}), \\ (p_4, \{i_3, i_4, i_5\}), (p_5, \{i_1, i_3, i_4\}), (p_6, \{i_5, i_6\})\}$$

where each element in H consists of a tuple of a strategy and the corresponding covered identifiers. When the hyper-parameter η_4 is 3, the set S_1 of crucial strategies is as follows:

$$S_1 = \{p_1, p_6\}$$

where S_1 is the minimum set of the strategies that collectively maximize the number of covered identifiers (i.e., $\{i_1, i_2, i_3, i_4, i_5, i_6\}$). Then, we additionally collect the set S_2 with $\eta_4 - |S_1|$ strategies with the maximal number of covered identifiers, where η_4 is 3 and $|S_1|$ is 2. That is, the set S_2 corresponds to a singleton set $\{p_1\}$ where p_1 has the maximal coverage in H. Finally, we can obtain the set S that includes all the strategies in S_1 and S_2:

$$S = \{\!\{p_1, p_6, p_1\}\!\}.$$

The intuition behind this step is to extract the set S capturing the core knowledge in the accumulated data H.

[3] Elsewhere in this paper, we assume argmax and argmin return a single argument since the result is a singleton set or the choice is unimportant.

In the Combine step, we generate new strategies by combining the features of the strategies in S, where the number of strategies to generate is given as the hyper-parameter η_2. This step has a genetic-algorithm flavor and repeats the next four phases until we have η_2 new strategies.

(1) We randomly sample two strategies from S. For instance, suppose that the strategies, p_1 and p_6, are sampled, where we assume each strategy is represented by a 5-dimensional vector of real-numbers as follows:

$$p_1 = \langle 0.2, 0.6, -0.3, 0.9, -0.4 \rangle \\ p_6 = \langle -0.1, -0.7, 0.2, 0.5, -0.8 \rangle .$$

(2) We generate a new strategy p', a parameter vector, by mixing the two selected strategies. The i-th component of the new vector p' is chosen randomly from the i-th components of the two vectors. For instance, if the first and third components of p' are obtained from p_1, and other components are from p_6, the newly generated strategy p' will be as follows:

$$p' = \langle 0.2, -0.7, -0.3, 0.5, -0.8 \rangle .$$

(3) We add a small random noise to each component of the newly generated strategy p':

$$p' = \langle 0.25, -0.72, -0.3, 0.51, -0.82 \rangle$$

where the noise is sampled from the normal distribution $\mathcal{N}(0, 0.2^2)$. This step aims to enable exploration of the space of possible strategies without changing the new strategy too much; hence, we use a relatively small standard deviation.

(4) Finally, we clip the newly generated strategy p' to ensure that the new one is in the proper range (e.g., $[-1, 1]^{29}$).

As the entire procedure (Algorithm 2) is going on, our technique is able to generate effective neuron-selection strategies based on the accumulated data H, thereby achieving high coverage.

Hyperparameters. Our algorithm involves five hyperparameters m, η_1, η_2, η_3, and η_4. The first hyperparameter m denotes the number of neurons that the strategy (Strategy) selects for the gradient calculation. Assigning the appropriate value to m is important. For instance, with small m (e.g. $m = 3$), we failed to make any notable influence on the inputs; with large m (e.g. $m = 30$), the generated input was often too far from the original one. In the experiments, with trial and error, we set m to 10 (we used the same value for other white-box testing techniques: DLFuzz and DeepXplore). The second hyper-parameter η_1 is the number of times that the set of selected neurons is used for generating new inputs. Note that because Algorithm 2 continuously adds the gradient value η_1 times to new input I' on line 14, the larger the value of η_1, the greater the distance between the original input I and the new one I'. We manually set the η_1 to 3 so that the distance is not too far. The third one η_2 represents the number of new strategies to generate after learning at line 22 in Algorithm 2. Intuitively, as the size of the η_2 increases, the number of learning trials (line 22) during the entire testing budget decreases. That is, η_2 determines how often the algorithm performs the learning procedure. In experiments, we set η_2 to 100 to perform the learning procedure multiple times during the total testing budget. Forth, η_3 denotes the size of the set H to be used as the learning data (line 21 in Algorithm 2). For instance, if η_3 is 10, we only use the most recently accumulated 10

Table 2: Datasets and DNN models

Dataset	Model	# of Neurons	# of Layers	Accuracy
MNIST	LeNet-4	148	9	0.985
	LeNet-5	268	10	0.988
ImageNet	VGG-19	16,168	26	0.713
	ResNet-50	94,123	177	0.764

elements in H as learning data. We set η_3 to 300 in our experiments. Fifth, η_4 denotes the number of effective strategies to select from the learning data H for generating new strategies. Intuitively, if the size of the η_4 is very small (e.g., η_4=2), we will generate new strategies that are similar to the most effective top-2 ones in the learning data. In experiments, we set η_4 to 50. In this work, we manually tuned these hyper-parameters and left automatic tuning as future work.

4 EXPERIMENTS

We implemented our technique in a tool, called ADAPT, using Python 3.6.3, Tensorflow 1.14.0 [1], and Keras 2.2.4 [6] without any modification of the frameworks. We evaluated ADAPT to answer the following research questions:

- **Coverage**: How effectively does ADAPT increase coverage metrics across various DNN models and datasets? How does it compare to existing testing techniques?
- **Adversarial Inputs**: How effectively does ADAPT find adversarial inputs compared to existing techniques? Is there strong correlation between coverage metrics and defects?
- **Learned Insight**: Is there any learned insight that should be considered while testing deep neural networks?

All experiments were done on a machine with two Intel Xeon Processors (E5-2630), 192GB RAM, and a NVIDIA GTX 1080 GPU.

4.1 Experimental Setup

We used two datasets and four neural network models in Table 2, which have been used in prior work [8, 12, 24, 34]. MNIST [17] is a classical dataset of hand-written digits with 10 classes and ImageNet [7] is a huge collection of real-world images with 1,000 class labels. For each dataset, we used two pre-trained models: LeNet-4 and LeNet-5 [17] for MNIST, and VGG-19 [29] and ResNet-50 [13] for ImageNet. In particular, VGG-19 and ResNet-50 are widely used in practice when, for example, extracting the embedding of the images in various tasks such as style transfer [10, 14], image captioning [35], and visual question answering [9].

We compared ADAPT with five state-of-the-art testing techniques for DNNs: four white-box and one grey-box approaches. The white-box approaches include DeepXplore [24], two instances of DLFuzz [12], and a random baseline. These techniques mainly differ in the neuron-selection strategy (Strategy) in Algorithm 1. DeepXplore randomly selects unactivated neurons. DeepXplore originally performs differential testing requiring multiple DNNs without labeled data but we used it with a single DNN and labeled data in our experiments. We instantiated DLFuzz with two strategies: $\text{DLFuzz}_{\text{Best}}$ denotes the strategy that performed best in the prior work [12] and $\text{DLFuzz}_{\text{RR}}$ the strategy that combines the three proposed strategies in a round-robin fashion, excluding one strategy that is not compatible with Top-k Neuron Coverage (TKNC) [18]. We included a random strategy Random, which selects neurons randomly. All white-box testing tools select 10 neurons for gradient calculation, as we metioned earlier. We also compare TensorFuzz [22], an available state-of-the art grey-box testing tool.

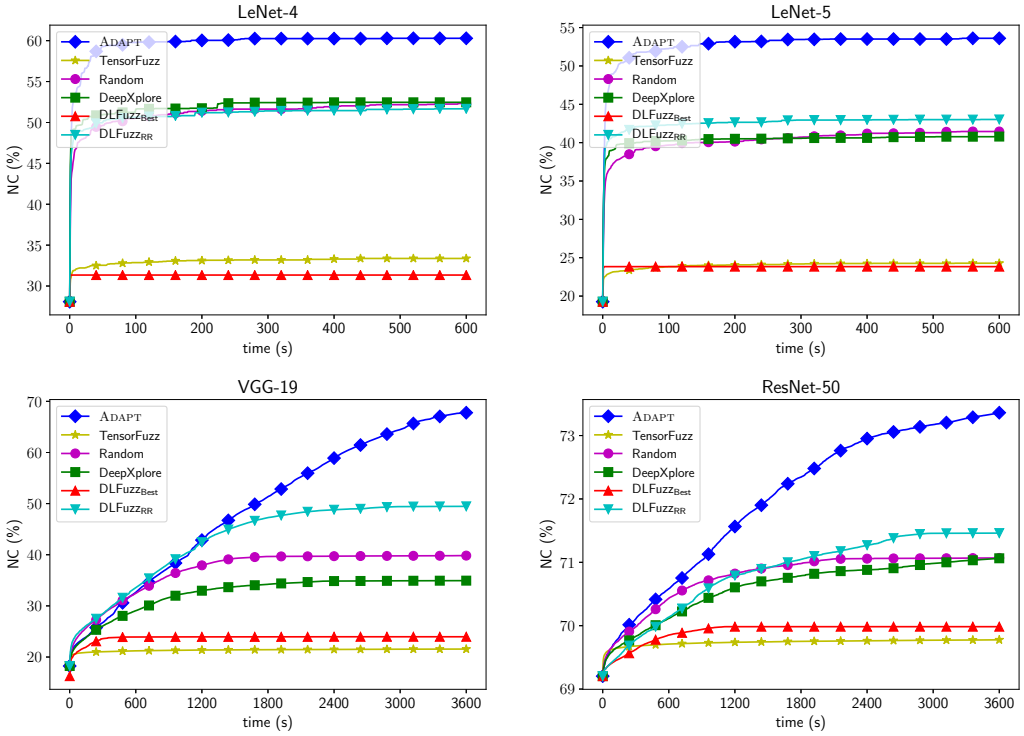
We used two well-known coverage metrics: Neuron Coverage (NC) [24] and Top-k Neuron Coverage (TKNC) [18]. For the former, we set θ to 0.5 and 0.25 for small (LeNet-4, LeNet-5) and large (VGG-19, ResNet-50) models, respectively. For the latter, we set k to 3 and 30 for small and large models, respectively. As an initial input, we used 20 inputs randomly selected from the corresponding testset of datasets for two small models, LeNet-4 and LeNet-5, and 10 randomly chosen images from the 2002 test images provided by DeepXplore [24] for two large models, VGG-19 and ResNet-50. All images are initially correctly classified, which means two models for each dataset classify the images to the same labels. For each input, we allocated 10 minutes and 1 hour as a testing budget for small and large models, respectively. Note that we allocated the same testing budget for ADAPT (Algorithm 2) and other existing tools (Algorithm 1); that is, the learning cost in ADAPT is included in the budget. All the techniques, including ADAPT, performed testing while maintaining the L2-distance between the initial and mutated inputs within 0.05 on average. All numerical values are average over all images tested, except the values with explicit mentions.

4.2 Coverage

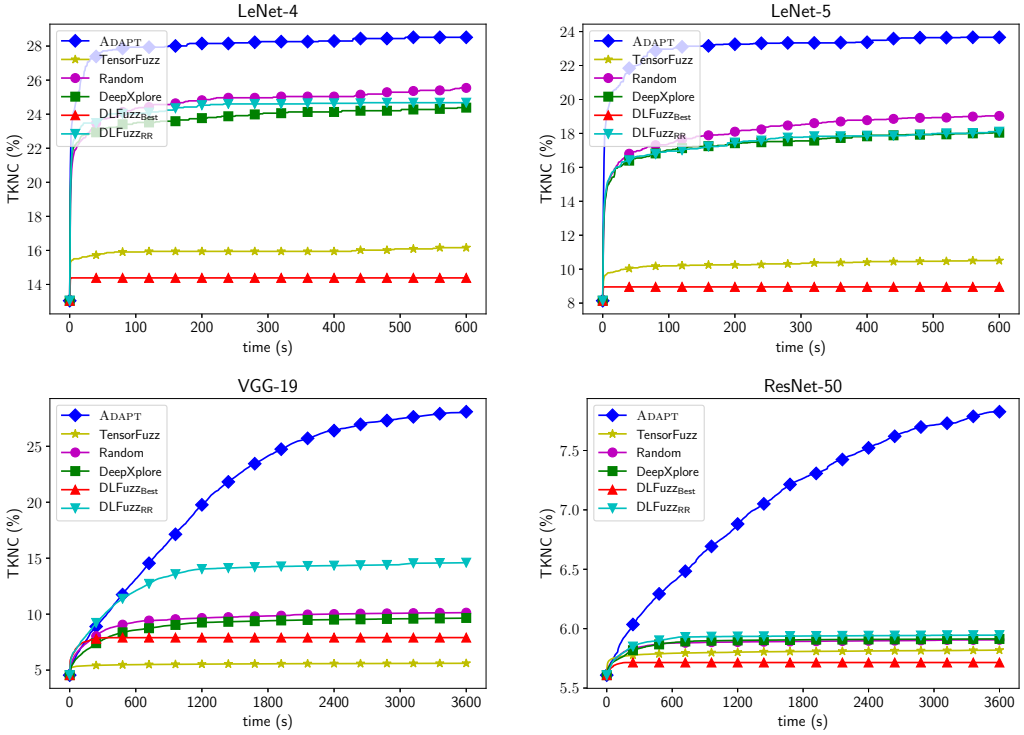
Figure 1 shows that ADAPT can achieve remarkably higher coverage than existing techniques across *all metrics* and *DNN models*. For example, ADAPT was able to achieve 67.8% NC on VGG-19 while the second best technique ($\text{DLFuzz}_{\text{RR}}$) achieved 49.5%; ADAPT covered almost 3,000 more neurons than $\text{DLFuzz}_{\text{RR}}$ during the same time period. Likewise, ADAPT managed to achieve 7.8% TKNC for ResNet-50, while no other existing tools can achieve the higher coverage than 6%, even though the initial coverage is 5.6%. Our tool successfully achieved the highest coverage on small models as well. For instance, ours increased NC by 7.8% and 10.6% more, compared to the second best technique of each model: DeepXplore (Lenet-4) and $\text{DLFuzz}_{\text{RR}}$ (Lenet-5).

Compared to the larger models (VGG-19 and ResNet-50), the smaller models (LeNet-4 and LeNet-5) converged remarkably fast. This is because a small model requires much less time for its execution than the larger model. For instance, we observed that our tool generated about 3 times more inputs when testing LeNet-4 with NC than when testing ResNet-50 with NC, even though we allocated 6 times more time budget to larger models than smaller ones.

Note that, except for ADAPT, existing testing tools have unstable performance. For example, $\text{DLFuzz}_{\text{RR}}$ achieved the highest neuron coverage for LeNet-5, but DeepXplore outperforms other existing techniques for LeNet-4. Regarding TKNC, all existing techniques are inferior even to the random baseline (Random) for LeNet-4 and LeNet-5. Figure 1 also shows that TensorFuzz, a grey-box technique, is overall less effective than white-box techniques.



(a) Average neuron coverage (NC) achieved by each technique on four models and two datasets

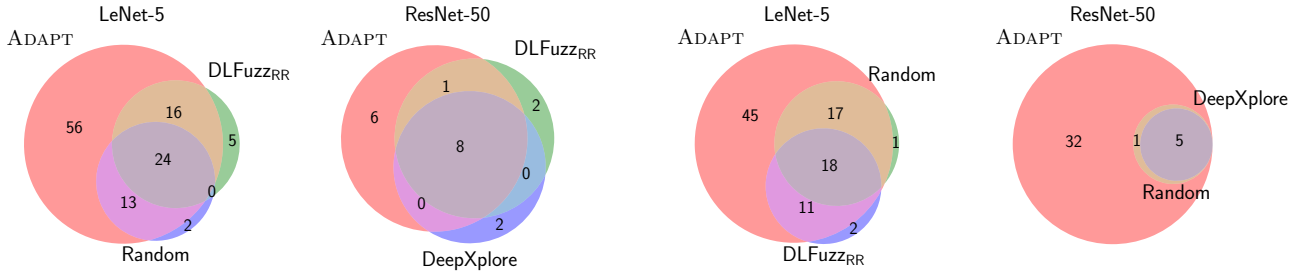


(b) Average Top-k neuron coverage (TKNC) achieved by each technique on four models and two datasets

Figure 1: Effectiveness for increasing NC and TKNC metrics

Table 3: Effectiveness for finding adversarial inputs

Dataset	Model	Technique	With Neuron Coverage: Mutations	Adv. Inputs	Labels	Seeds	With Top-k Neuron Coverage: Mutations	Adv. Inputs	Labels	Seeds
MNIST	LeNet-4	ADAPT	31888.9	**950.9**	**73**	**20/20**	38320.4	955.8	**48**	**16/20**
		TensorFuzz	74881.0	0.0	0	0/20	73515.8	0.0	0	0/20
		Random	34988.6	192.3	38	17/20	32394.8	178.8	33	14/20
		DeepXplore	38191.6	30.2	26	14/20	38083.0	11.1	15	7/20
		$\text{DLFuzz}_{\text{Best}}$	33895.5	542.2	3	3/20	32857.7	**1079.7**	1	1/20
		$\text{DLFuzz}_{\text{RR}}$	34223.8	423.9	33	16/20	32934.0	41.1	20	12/20
	LeNet-5	ADAPT	35601.6	**3974.5**	**109**	**20/20**	36483.6	**3202.9**	**91**	**20/20**
		TensorFuzz	89290.4	0.0	0	0/20	90024.1	0.0	0	0/20
		Random	30686.7	225.3	39	17/20	32994.7	215.8	36	16/20
		DeepXplore	36188.1	39.0	37	17/20	34952.9	88.9	23	12/20
		$\text{DLFuzz}_{\text{Best}}$	31659.3	2.0	4	4/20	33006.6	0.2	2	2/20
		$\text{DLFuzz}_{\text{RR}}$	32742.3	183.8	45	18/20	33776.3	182.6	31	15/20
ImageNet	VGG-19	ADAPT	12028.6	**5301.0**	**265**	**10/10**	11926.1	3322.7	**300**	**10/10**
		TensorFuzz	15173.1	0.0	0	0/10	12911.8	0.0	0	0/10
		Random	12778.1	1135.0	56	6/10	13005.6	126.2	10	4/10
		DeepXplore	10210.6	695.4	25	5/10	9077.9	163.6	15	4/10
		$\text{DLFuzz}_{\text{Best}}$	12888.5	4713.2	9	8/10	12246.3	**5091.5**	6	5/10
		$\text{DLFuzz}_{\text{RR}}$	11844.7	2131.0	100	9/10	12139.4	673.2	65	7/10
	ResNet-50	ADAPT	8469.3	**1925.1**	**15**	5/10	8302.2	**1721.2**	**38**	**6/10**
		TensorFuzz	9237.0	0.0	0	0/10	9464.0	0.0	0	0/10
		Random	9256.5	713.0	10	5/10	8916.4	137.1	5	3/10
		DeepXplore	6541.3	658.7	10	5/10	6832.9	146.3	6	4/10
		$\text{DLFuzz}_{\text{Best}}$	8357.8	597.6	8	5/10	9087.0	61.1	3	3/10
		$\text{DLFuzz}_{\text{RR}}$	8435.0	1095.2	11	**6/10**	9026.4	167.5	3	3/10



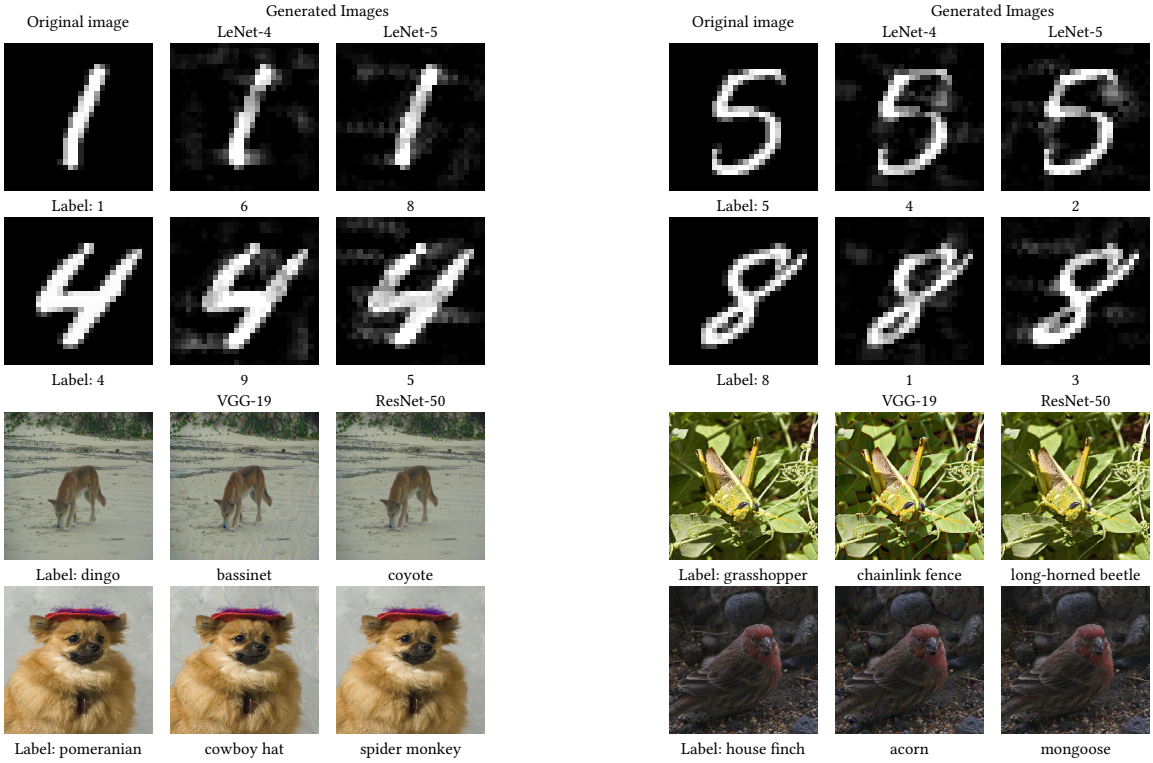
(a) Distinguishable labels found with NC

(b) Distinguishable labels found with TKNC

Figure 2: The number of distinguishable labels found

4.3 Adversarial Inputs

While conducting the experiments in terms of coverage in Section 4.2, we found that ADAPT is highly effective in finding adversarial inputs as well. Table 3 reports the average number of mutations per image (Mutations), the average number of adversarial inputs found per image (Adv. Inputs), the total number of incorrectly classified labels (Labels), and the number of initial inputs (Seeds) from which adversarial inputs were found. An incorrectly classified label consists of an original image and a found label. That is, we consider two adversarial inputs differently, which classified into the same label but came from different input source. Overall, ADAPT outperforms existing techniques in almost all metrics; on average, ADAPT succeeded to find 4.9 times more adversarial inputs, 6.4 times more incorrectly classified labels, and 2.0 times more seed images than existing techniques. Out of 24 metrics, ADAPT ranked at the first place in 21 metrics. In terms of the number of the labels found, no existing techniques could beat ADAPT. ADAPT is also able to find adversarial inputs from 86% of the given inputs on average, while the best technique ($\text{DLFuzz}_{\text{RR}}$) among existing techniques can found adversarial inputs from only 69%. $\text{DLFuzz}_{\text{Best}}$



(a) Images found during testing with NC

(b) Images found during testing with TKNC

Figure 3: Images with incorrectly classified labels found exclusively by ADAPT.

oddly found a lots of adversarial inputs, while the number of labels and the number of seed inputs from which adversarial inputs are found are relatively small. This is becuase $\text{DLFuzz}_{\text{Best}}$ selects most activated neurons, which are fixed as the testing procedure goes by, which results in generating a huge amount of similar inputs without any variety. Although the grey-box testing tool, TensorFuzz, which does not calculate the gradients of the outputs of internal neurons, is on average 1.7 times faster than ADAPT, it failed to generated any adversarial inputs across all experiments.

Figure 2 shows that ADAPT successfully found the various incorrect labels compared to existing techniques. The Venn diagram represents the relationship between first (red), second (green), and third (blue) ranked tools in terms of variety of labels. With NC, ADAPT could find 69 more incorrect labels while testing LeNet-5 compared to $\text{DLFuzz}_{\text{RR}}$, which is second best technique among existing techniques. For ResNet-50, ADAPT found all the labels that second best (DeepXplore) and third best (Random) found and 32 labels more. In total, ADAPT found 611 more labels that no other techniques could find. Figure 3 shows the some examples with incorrectly classified labels found exclusively by ADAPT (all existing testing techniques totally failed to find those incorrect labels). In particular, the adversarial images found for VGG-19 and ResNet-50 were visually indistinguishable from the original ones.

Correlation between Coverage and Adversarial Inputs. From Figure 1 and Table 3, we can notice that coverage and variety of labels are highly correlated, while coverage and the number of adversarial inputs are not correlated strongly. In testing VGG-19 with NC, the order of coverage value of each technique exactly matches the order of the number of the labels found by each technique.

In Figure 2, we can find that the effectiveness of coverage metrics is correlated with the size of the model to test. NC is more helpful than TKNC in finding diverse inputs for small models, like LeNet-4 and LeNet-5, TKNC is more useful in finding diverse inputs for larger models, like VGG-19 and ResNet-50. For instance, ADAPT could find 109 incorrect labels from LeNet-5 with NC, but only 91 labels with TKNC. On the other hand, ADAPT only could find 15 labels from ResNet-50 with NC, but found 38 labels with TKNC.

4.4 Learned Insights

Figure 4 shows the changes of the weights of the top-3 (red) and bottom-3 (blue) features of the strategies that our algorithm collected while testing each model with NC, where f_i indicates the i-th element of the parameterized neuron-selection strategies. The top features mean the properties of neurons which should be selected to increase coverage, and the bottom features are characteristics that neurons that should not be selected have. The solid lines represent the medium value of the feature and the colored areas represent the value from the top 20% to 80% of the collected strategies. As testing goes by, the solid lines were being skewed toward one direction, and the colored areas were becoming narrower, which means that

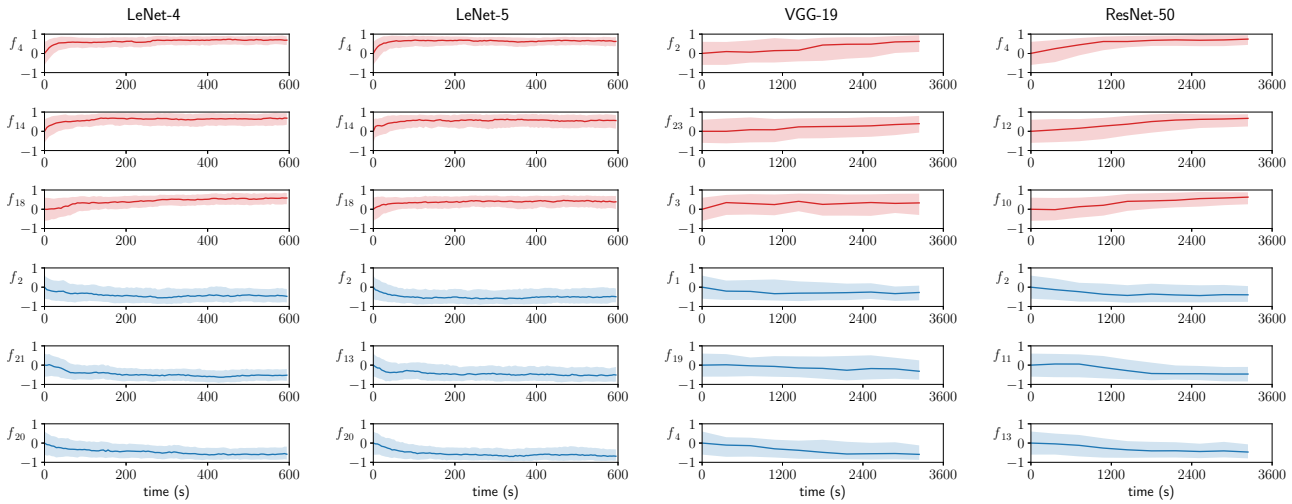


Figure 4: Top-3 and bottom-3 features learned by our algorithm during testing each model with NC

the learning algorithm extracts the characteristics of strategies, which increase coverage, well.

f_4, which indicates the layers located in the back 25% of the network, is learned as the most important feature that neurons to select should have, and f_2, which indicates the layers located relatively front of the network, is included in bottom-3 features in most cases (LeNet-4, LeNet-5, and ResNet-50). This trend indicates that neurons with higher expressive ability, which means that neurons are located in deeper convolutional layers [38], should be selected while testing the deep neural networks. This can explain the results from VGG-19, which has a few fully-connected layers at the back of the network; neurons with f_2 and f_3, which represent the convolutional layers that located at relatively back, should be considered, and neurons with f_4, which includes the last fully-connected layers, should be discarded.

f_{10}, which indicates the layers with multiple input sources and no other networks have, is one of the top-3 features that should be selected for testing ResNet-50, while no other features that related to the type of the layer did not show any trend across all experiments. This implies that the layers with multiple input sources should be considered when testing a deep neural network that contains them as their components; the neurons that located in the layers with residual connection in ResNet-50 are examples of the neuron with multiple input sources.

LeNet-4 and LeNet-5 showed very similar results; their top-3 features and two of bottom-3 features exactly match. From these results, we can infer that neurons with moderately high weights (f_{14}) and neurons that activated when the given objective is satisfied (f_{18}) should be considered carefully while testing small models, like LeNet-4 and LeNet-5. In addition, neurons that activated a lot (f_{20} and f_{21}) should not be selected. These are not consistent with DLFuzz's selection strategies [12] which selects the neurons that activated a lot or neurons with biggest weights; on the other hand, the other strategy of DLFuzz (f_{12}, neurons with top 10% weights) is included in top-3 strategies in the results of ResNet-50.

The overall trends of the features of the collected strategies during test procedure with TKNC are similar with those with NC. For instance, neurons with higher expressiveness are important in both cases. This represents that there are some common characteristics that two metrics share.

5 RELATED WORK

DNN Testing. Unlike existing gradient-based white-box techniques, ADAPT uses an adaptive neuron-selection strategy. DeepXplore [24] proposed the first white-box differential testing algorithm to generate inputs which can cause inconsistencies between the set of DNNs. The tool uses gradient ascent as an input generation algorithm, which uses random selection as a neuron selection strategy. The following approach, DLFuzz [12], enabled testing with a single DNN. They use gradient ascent like the former, using four fixed heuristics to select neurons. DeepFault [8] presented a new fault localization-based testing approach by using a neuron-selection strategy based on suspiciousness metric.

Another white-box approach, DeepConcolic [31], tests DNN using concolic testing, which has proven to be effective in small neural networks. However, its applicability to real-world sized networks needs to be examined. DeepMutation [19] analyzes the robustness of the model by mutating the dataset and the model with its self-designed heuristics. It chooses the layers for the heuristics while ADAPT chooses neurons for the gradient calculation. They also have not been confirmed as applicable to real scale models such as ImageNet models. MODE [20] focuses on a different problem; it aims to pick the mis-trained layer and create the input that generates the bug during the learning process.

Grey-box testing techniques are largely based on coverage-guided fuzzing. DeepTest [33] presented a testing method for detecting erroneous behaviors of autonomous car models. They mimic what would happen in the physical world and generate input by applying a set of natural image transformations randomly. DeepHunter [34] performed misbehavior detection of DNNs as well as model quality

evaluation and defect detection in quantization settings based on multiple pluggable coverage criteria feedback. TensorFuzz [22] debugged neural networks by using logit-based coverage and adding additive random noise.

Existing coverage metrics are largely divided into the coverage applied to the weights of the trained model and the coverage applied based on the changes of the weights during the training. In the former case, in addition to NC [24] and TKNC [18], there is a Top-k Neuron Pattern (TKNP) [18] that measures the different kinds of activated scenarios by recording the patterns of top-k neurons per layer. There is also a coverage metric inspired from the classical MC/DC [30]. The latter case includes k-multisection Neuron Coverage (KMNC) [18], Neuron Boundary Coverage (NBC) [18], Strong Neuron Activation Coverage (SNAC) [18], and Surprise Coverage (SC) [15]. In this paper, we used only the former cases using the pre-trained weights of real-world models. However, our approach is in principle applicable to other coverage metrics as well.

Using Gradients to Attack DNNs. Gradients, which can also be used to increase the probability of a particular class, have been used for generating inputs that fool neural networks by generating adversarial examples [3, 11, 16, 23]. They are similar to white-box testing in that they calculate a gradient through an objective function and attempt to find a malfunctioning input. However, they differ from the testing approaches as they aim to find common misbehaving inputs and do not take into account the logic inside the model. The emphasis of testing techniques is on systematically examining the logic of the model and explore various internal states.

Software Testing with Learning. Combining software testing and learning is not new. For example, Cha et al. [4, 5] used learning to generate search heuristics of concolic testing automatically. To our knowledge, however, existing works target traditional software and ADAPT is the first tool that uses learning for DNN testing.

6 CONCLUSION

Since deep neural networks are used in safety-critical applications, testing safety properties of deep neural networks is important. Although many testing techniques have been introduced recently, there is no technique that is sufficiently effective across different models and coverage metrics. In this paper, we present a new white-box technique, called ADAPT, that performs well regardless of models and metrics, via parameterizing the neuron-selection strategy and learning appropriate parameters online. Experimentally, we demonstrated that ADAPT is significantly more effective than existing white-box and grey-box techniques in increasing coverage and finding adversarial inputs. As future work, we plan to apply our technique to various models, domains, and coverage metrics.

ACKNOWLEDGMENTS

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair) and Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

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

[3] N. Carlini and D. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy (S&P'17)*. 39–57.

[4] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 1244–1254.

[5] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. 235âĂŞ245.

[6] François Chollet et al. 2015. Keras. https://keras.io.

[7] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.

[8] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering (FASE'19)*. 171–191.

[9] Akira Fukui, Dong Huk Park, Daylen Yang, Anna Rohrbach, Trevor Darrell, and Marcus Rohrbach. 2016. Multimodal Compact Bilinear Pooling for Visual Question Answering and Visual Grounding. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP'16)*. 457–468.

[10] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2016. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR'16)*. 2414–2423.

[11] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations (ICLR'15)*.

[12] 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 (ESEC/FSE'18)*. 739–743.

[13] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778.

[14] Xun Huang and Serge Belongie. 2017. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'17)*. 1501–1510.

[15] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 1039–1049.

[16] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *The International Conference on Learning Representations (ICLR'17)*.

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998), 2278–2324.

[18] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 120–131.

[19] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei Xu, Chao Xie, Li Li, Jianjun Zhao Yang Liu, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. *CoRR* abs/1805.05206 (2018).

[20] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection.

[21] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. 807–814.

[22] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*. 4901–4911.

[23] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *2016 IEEE European Symposium on Security and Privacy*. 372–387.

[24] 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 (SOSP'17)*. 1–18.

[25] 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. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 2360–2368.

[26] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. 2017. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225* (2017).

[27] J. Sekhon and C. Fleming. 2019. Towards Improved Testing For Deep Learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE'19)*. 85–88.

[28] 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 (2017), 354.

[29] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations (ICLR'15)*.

[30] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *CoRR* abs/1803.04792 (2018).

[31] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. DeepConcolic: Testing and Debugging Deep Neural Networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 111–114.

[32] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Structural Test Coverage Criteria for Deep Neural Networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 320–321.

[33] 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 (ICSE'18)*. 303–314.

[34] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. 146–157.

[35] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning (ICML'15)*. 2048–2057.

[36] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo. 2016. Image Captioning with Semantic Attention. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4651–4659.

[37] MichaÅĆ Zalewski. 2007. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

[38] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *European conference on computer vision (ECCV'14)*. 818–833.