DeepSmartFuzzer: Reward Guided Test Generation For Deep Learning

Samet Demir[1], **Hasan Ferit Eniser**[2*], **Alper Sen**[1]

[1]Department of Computer Engineering, Boğaziçi University, Turkey
[2]Max Planck Institute for Software Systems, Germany
samet.demir1@boun.edu.tr, hfeniser@mpi-sws.org, alper.sen@boun.edu.tr

Abstract

Testing Deep Neural Network (DNN) models has become more important than ever with the increasing usage of DNN models in safety-critical domains such as autonomous cars. The traditional approach of testing DNNs is to create a test set, which is a random subset of the dataset about the problem of interest. This kind of approach is not enough for testing most of the real-world scenarios since these traditional test sets do not include corner cases, while a corner case input is generally considered to introduce erroneous behaviors. Recent works on adversarial input generation, data augmentation, and coverage-guided fuzzing (CGF) have provided new ways to extend traditional test sets. Among those, CGF aims to produce new test inputs by fuzzing existing ones to achieve high coverage on a test adequacy criterion (i.e. coverage criterion). Given that the subject test adequacy criterion is a well-established one, CGF can potentially find error inducing inputs for different underlying reasons. In this paper, we propose a novel CGF solution for structural testing of DNNs. The proposed fuzzer employs Monte Carlo Tree Search to drive the coverage-guided search in the pursuit of achieving high coverage. Our evaluation shows that the inputs generated by our method result in higher coverage than the inputs produced by the previously introduced coverage-guided fuzzing techniques.

Introduction

Given enough amount of data and processing power, training a Deep Neural Network (DNN), also called Deep Learning, is the most popular way for dealing with many hard computational problems such as image classification (Cireşan, Meier, and Schmidhuber 2012), natural language processing (Sutskever, Vinyals, and Le 2014) and speech recognition (Hinton et al. 2012). Impressive achievements in such tasks raised expectations for deploying DNNs in real-world applications, including the ones in safety-critical domains. Examples of safety-critical applications include air traffic control (Julian et al. 2016), medical diagnostics (Litjens et al. 2017) and autonomous vehicles (Bojarski et al. 2016).

Despite the remarkable achievements, recent works (Szegedy et al. 2013; Goodfellow, Shlens, and Szegedy

---

*Most of the work was done when the author was in Boğaziçi University.

2015) have demonstrated that the DNNs are vulnerable to small perturbations on seed inputs, also called adversarial attacks. Considering the catastrophic results that can emerge from erroneous behavior in safety-critical systems, DNNs must be characterized by a high degree of dependability before being deployed in safety-critical systems. Furthermore, it is also suggested that DNN-based systems should be allowed for use in the public domain only after presenting high levels of trustworthiness (Burton, Gauerhof, and Heinzemann 2017).

Testing is the primary practice for analyzing and evaluating the quality of a software system (Jorgensen 2013). It helps in reducing the risk by finding and eliminating erroneous behaviors before deployment of the systems. Moreover, it provides evidence for the required levels of safety of the subject system. One of the most fundamental testing concepts is defining a coverage criterion, also called a test adequacy criterion, for a given test set. A coverage criterion divides the input space into equivalence classes and is satisfied if there exists at least one input for each equivalence class in the given test set. Having a test set that is satisfying a coverage criterion provides a degree of dependability to the system under test.

Recent research in the DNN testing area introduces new DNN-specific coverage criteria such as statement coverage inspired neuron coverage (Pei et al. 2017) and its variants (Ma et al. 2018), MC/DC-inspired criteria (Seshia et al. 2018) or other novel criteria such as surprise adequacy (Kim, Feldt, and Yoo 2019). Previous works (Pei et al. 2017; Ma et al. 2018; Seshia et al. 2018; Kim, Feldt, and Yoo 2019), and future studies on coverage criteria for DNNs could be useful for exposing the defects in DNNs, finding adversarial examples, or forming diverse test sets. On the other hand, satisfying a coverage criterion or at least achieving a high coverage measurement can be difficult without a structured method. Existing works (Xie et al. 2018; Odena and Goodfellow 2018) leverage coverage guided fuzzing (CGF) to achieve high coverage. However, both of these works fuzz inputs randomly. Therefore, their effectiveness is limited, as shown in our experiments.

In this work, we introduce DeepSmartFuzzer, a novel coverage guided fuzzer, for achieving high coverage in DNNs

for all existing criteria in the literature. Ultimately, our goal is to help practitioners to extend their test sets with new inputs so that more cases are covered. We leverage Monte Carlo Tree Search (MCTS) (Bouzy and Helmstetter 2004; Chaslot et al. 2008), a search algorithm for decision processes, in achieving this goal. In our method, given an input, MCTS aims to pick a series of mutations that would result in the best coverage increase.

Contributions of this work are as follows:

- We introduce a novel advanced coverage guided fuzzing technique for testing DNNs, that is designed to work with every coverage metric.

- We show the effectiveness of our method on the most popular coverage criteria and DNNs with changing complexity that are trained on various well-known datasets.

- We compare our method with existing coverage guided fuzzing methods in the literature in terms of various metrics.

- We also show that some of the inputs generated by our method are adversarial examples.

Related Work

Coverage-Guided Fuzzing in Software Fuzzing is a widely used technique for exposing defects in software. Coverage-guided grey-box fuzzing tools, such as AFL and libFuzzer, have been quite successful in detecting thousands of bugs in many software systems.

Testing Deep Neural Networks Several DNN testing techniques have been developed in the literature recently. Among those, there exist works targeting coverage criteria for DNNs. For example, DeepXplore (Pei et al. 2017) proposed neuron coverage (analogous to statement coverage in software). DeepGauge (Ma et al. 2018) proposed a set of fine-grained test coverage criteria. DeepCT (Ma et al. 2019) introduced a combinatorial coverage metric for DNNs. Sun et al. (2018b) presented coverage criteria inspired by MC/DC criteria in software. Kim, Feldt, and Yoo (2019) proposed surprise adequacy criteria based on the amount of surprise caused by the inputs on the neuron activation values.

Li et al. (2019) pointed out the limitation of existing structural coverage criteria for neural networks.

DLFuzz (Guo et al. 2018) introduced the first differential fuzzing framework for deep learning systems. Sun et al. (2018a) proposed the first concolic testing approach for DNNs. DeepCheck (Gopinath et al. 2018) tests neural networks based on symbolic execution. DeepTest (Tian et al. 2018) and DeepRoad (Zhang et al. 2018) proposed testing tools for autonomous driving systems. For more studies on testing neural networks, we refer to the work of Zhang et al. (2019) that surveys testing of machine learning systems.

We now discuss the studies that are close to ours. TensorFuzz (Odena and Goodfellow 2018) proposed the first CGF for neural networks that aims to increase a novel coverage metric. DeepHunter (Xie et al. 2018) is another work exploring CGF for DNNs by leveraging techniques from software fuzzing, such as power scheduling. Our work is distinguished from TensorFuzz and DeepHunter since we use Monte Carlo Tree Search (MCTS) for fuzzing while they do random fuzzing. Moreover, we apply mutations to small regions of a given input, while they apply the mutation to the whole input. Wicker, Huang, and Kwiatkowska (2018) also use MCTS for testing DNNs. Our work has several differences from the work of (Wicker, Huang, and Kwiatkowska 2018). First, their objective is to find the nearest adversarial example, while our objective is to increase a given coverage criterion. Second, while they use feature extraction such as SIFT (Scale Invariant Feature Transform) (Lowe 2004) to ease the selection of pixels to be mutated, we do not need feature extraction because of our formulation of the problem. Third, our method applies image transformations such as brightness change, contrast change, and blur to the images while their approach is based on pixel-level mutations.

Adversarial Deep Learning Szegedy et al. (2013) first discovered the vulnerability of DNNs to adversarial attacks. Since then, multiple white-box and black-box adversarial attacks have been proposed. The most popular white-box attacks include FGSM (Goodfellow, Shlens, and Szegedy 2015), BIM (Kurakin, Goodfellow, and Bengio 2016), DeepFool (Moosavi-Dezfooli, Fawzi, and Frossard 2016), JSMA (Papernot et al. 2016), PGD (Madry et al. 2017) and C&W (Carlini and Wagner 2017).

Attacking to a DNN in a black-box manner is harder than white-box. There exist black-box attacks based on transferability (Papernot, McDaniel, and Goodfellow 2016; Papernot et al. 2017), gradient estimation (Chen et al. 2018; Bhagoji et al. 2018) and local-search (Narodytska and Kasiviswanathan 2017) in the literature.

Background

Coverage Criterion

A coverage criterion is used to measure what percent of the potential behaviors could be tested by a given set of inputs. For this purpose, a coverage criterion divides the input space for a given system into equivalence classes and calculates how many of the equivalence classes have at least one instance input in a given set of inputs. When an equivalence class has at least one instance input in a given set of inputs, the equivalence class is said to be covered. Thus, a coverage criterion calculates how many of its equivalence classes are covered by a set of inputs. If a coverage criterion properly divides the input space for a given system, covering all equivalence classes corresponds to having at least one test sample triggering different behaviors so that all behaviors could be tested.

Coverage Guided Fuzzing

A Coverage Guided Fuzzer (CGF) performs systematic mutations on inputs and produces new test inputs to increase the number of covered cases for a target coverage metric. A typical CGF process starts with selecting a seed input from the seed pool, then continues with mutating the selected seed a certain number of times. After that, the program under test is run with the mutated seed. If a mutated seed creates an

increase in the number of covered cases, CGF keeps the mutated seed in the seed pool. In the meantime, CGF can also keep track of details of execution, such as execution paths and crash reports so that they could be reported if need.

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (Bouzy and Helmstetter 2004; Chaslot et al. 2008) is a search algorithm for decision processes such as game play. It uses game trees to represent a game. Each node of the game tree represents a particular state in the game. On taking an action, one makes a transition from a node to one of its children. MCTS algorithm is used to pick the most promising action on an arbitrary state of a game. Ultimately, the objective is to find the best path (i.e. best sequence of actions) to follow to win the game.

MCTS process can be broken down into the following four steps: **Selection:** Starting from the root node R, successively select child nodes according to their potentials until a leaf node L is reached. The potential of each child node is calculated by using UCT (Upper Confidence Bound applied to Trees) (Kocsis and Szepesvári 2006; Chaslot et al. 2008). UCT is defined as $v + e \times \sqrt{\frac{\ln N}{n}}$ where v refers to the value of the node, n is the visit count of the node, and N is the visit count for the parent of the node. e is a hyperparameter determining exploration-exploitation trade-off. **Expansion:** Unless L is a terminal node (i.e. win/loss/draw), create at least one child node C (i.e. any valid moves from node L) and pick one of them. **Simulation:** play the game from node C by picking moves randomly until reaching a terminal condition. **Backpropagation:** propagates back the result of the play to update values associated with the nodes on the path from C to R. The path containing the nodes with the highest values in each layer would be the optimal strategy in the game.

For practical reasons, after applying the MCTS process on root R for a while, it is useful to pick a new root R' from the child nodes of the current root R and continue the MCTS process on the new root R' such that the MCTS gets one level below and continues searching on a subtree.

Method: DeepSmartFuzzer

Main Idea

The core idea of our approach is to make use of reward-guided exploration and exploitation, in other words reinforcement learning, to extend the test set by mutating inputs in a smarter way so that to achieve a higher coverage score. In reward-guided process, selected mutations are evaluated with the coverage changes they induce. In this way, our approach is able follow the steps that result in coverage increase.

Notations

Let $T = \{[X_1, X_2, ...], [Y_1, Y_2, ...]\}$ be a test set where (X_i, Y_i) is an input-output pair of the ith test sample. Let $\mathcal{I}$ be a set of inputs called a batch. Let $\mathcal{I}', \mathcal{I}'', ..., \mathcal{I}^{(n)}$ be a sequence of mutated batches of the original batch $\mathcal{I}$ such that:

$$\mathcal{I} \xrightarrow{\mathcal{IM}(r',m')} \mathcal{I}' \xrightarrow{\mathcal{IM}(r'',m'')} \mathcal{I}'' ... \xrightarrow{\mathcal{IM}(r^{(n)},m^{(n)})} \mathcal{I}^{(n)} \quad (1)$$

where $\mathcal{IM}(r^{(n)}, m^{(n)})$ is the nth input mutator, $r^{(n)}$ and $m^{(n)}$ are the region and mutation indexes, respectively. Furthermore, let $\mathcal{I}^{best} \in \{\mathcal{I}', \mathcal{I}'', ...\}$ be the best batch, which is the one that creates the greatest amount of coverage increase.

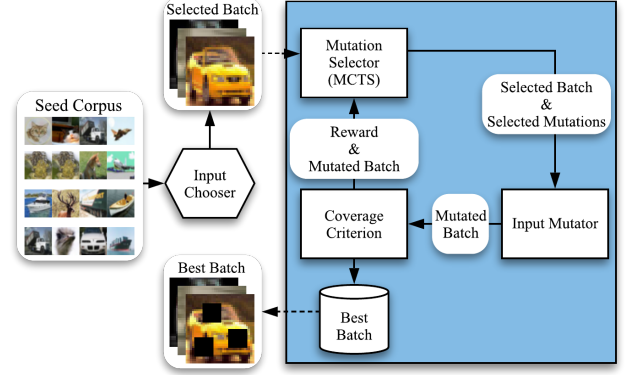
Figure 1: Workflow of DeepSmartFuzzer for an iteration

Overview

DeepSmartFuzzer is a MCTS-driven coverage-guided fuzzer for DNNs. It can be classified as a grey-box testing method since it uses coverage information which is related to internal states of a DNN model. The method is designed to generate inputs that increase the current level of coverage formed by the initial test set. It is composed of input chooser, coverage criterion, input mutator, and mutation selector, which is the most important part. The coverage criterion is a hyperparameter for our method. For each iteration, the input chooser chooses a batch, which is a set of inputs $\mathcal{I}$. After that, the mutation selector determines which mutation (r', m') to be applied to the inputs. The batch $\mathcal{I}$ and selected mutation (r', m') then goes to input mutator which applies the selected mutation to the batch of inputs so that the mutated inputs $\mathcal{I}'$ are formed ($\mathcal{I} \xrightarrow{\mathcal{IM}(r',m')} \mathcal{I}'$). The mutated inputs are then given to a coverage criterion to calculate the coverage of the mutated inputs union the test set. The coverage and mutated inputs are given to the mutation selector such that it could make use of the coverage and continue working with $\mathcal{I}'$ so that new mutated inputs $\mathcal{I}''$ are generated ($\mathcal{I}' \xrightarrow{\mathcal{IM}(r'',m'')} \mathcal{I}''$). This process continues until a termination condition such as exploration limit or mutation limit is reached. The best set of mutated inputs $\mathcal{I}^{best}$ is stored and updated in the meantime. If there is an increase in coverage because of the mutated inputs $\mathcal{I}^{best}$, they are added to the test set. This concludes the iteration for the batch $\mathcal{I}$. We continue iterating with different batches until a termination condition such as a target number of new inputs or timeout is reached. The workflow of the proposed method is illustrated in Figure 1.

Input Chooser

We use two types of input chooser for selecting inputs to make mutations on. These are random input chooser and clustered random input chooser. The random input chooser randomly samples a batch of inputs from the uniform random distribution. The clustered random input chooser aims to sample similar inputs together. It applies an off-the-shelf clustering algorithm. After clustering, it selects a random cluster using the uniform random distribution. Finally, it samples a random batch of inputs from the selected cluster. We use sampling without replacement to avoid multiple same inputs in a batch since we apply the same mutations to all inputs in the batch. In this work, we use k-means clustering as the clustering algorithm.

Mutation Selector

The mutation selector takes a batch of inputs and sequentially selects parameters region index r and mutation index m. The selected mutations are sequentially applied to the selected regions by the input mutator and a sequence of mutated batches $\mathcal{I}', \mathcal{I}'', ..., \mathcal{I}^{(n-1)}, \mathcal{I}^{(n)}$ is generated. Note that $\mathcal{I}^{(n)}$ contains all the mutations up to that point.

Modelling the mutation selection as a two-player game Our proposed mutation selector is a two-player game such that Player I selects the region to be mutated, and Player II selects the mutation to be made on the chosen region. Since regions and mutations are enumerated, these are just integer selection problems. The game continues as Players I and II play iteratively so that multiple local mutations could be applied.

Region selection and mutation selection are considered as separate actions. We call a tuple of actions taken by Players I and II together as a complete action (r, m).

Reward A naive reward for our problem is the coverage increase for each action. We use this reward to guide the search for mutations. In this study, coverage increase corresponds to the difference between the coverage led by current state of the test set and the coverage obtained by adding a new batch to the test set. The purpose of the mutation selector is to find the best mutations that could result in the greatest amount of coverage increase.

End of the game In order to avoid creating unrealistic inputs, we put some constraints to limit the mutations. Generally, L_p norms are used for this purpose. These are defined as $||X_i' - X_i||_p < \epsilon$ where X_i' is the mutated input such that the distance between mutated inputs and original inputs are limited. In general form, let $d(X_i', X_i)$ be a distance metric, the game is over when $d(X_i', X_i) < \epsilon$ constraint is violated, where d and ϵ are hyperparameters.

Searching We use Monte Carlo Tree Search (MCTS) in order to search the game tree for the best mutations. The nodes in our game tree correspond to region and mutation selections. We continuously update the batch of inputs $\mathcal{I}^{best}$ that creates the best coverage increase so that the batch is added to the test set when our MCTS is finished. Figure 3 shows the steps of our MCTS where edges on odd levels of

the game tree correspond to region selections, and edges on even levels of the game tree correspond to mutation selections.

Input Mutator

The input mutator mutates the input according to the region index and mutation index selected by the mutation selector. The availability of so many mutations could potentially harden the job of Mutation Selector. Therefore, by addressing that problem, we come up with a general input mutator for images. It divides an image into local regions and provides general image mutations as mutation options for each region. These general mutations include, but are not limited to, brightness change, contrast change, and blur. When a region r and a mutation m are selected, it applies the selected mutation to the selected region. The number of regions and the set of available mutations for regions are hyperparameters for the input mutator. With appropriate settings, we can obtain either a pixel-level mutator or an image-level mutator or something in-between, which we think is the best for practical reasons. We enumerate the regions and mutations so that the mutation selector identifies them by indexes. Figure 2 shows an example for the division of input into regions. Our proposed input mutator induces a bias towards locality since it applies mutations to regions of an image. Therefore, it is a natural fit for image problems and convolutional neural networks.
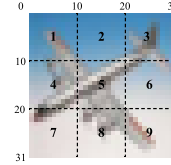

Figure 2: Regions for an example input

Algorithm

We describe our method in Algorithm 1. The while loop in line 2 refers to iterating until a termination condition (tc_0) that is a timeout or reaching a target number of new inputs. In line 3, a batch of inputs $\mathcal{I}$ is sampled using the input chooser. The root node R is created in line 4, and variables to store best mutated batch $\mathcal{I}^{best}$ are initialized in line 5. The while loop in line 6 refers to looping until a termination condition (tc_1) that determines at most how many levels in the search tree the MCTS can go down. Next, the while loop in line 7 refers to iterating until a termination condition (tc_2) that determines at most how many times a MCTS node can be explored. In line 8, MCTS Selection is made and it results in a leaf node L. Then, in line 9, MCTS Expansion is applied and it creates a new child node C as a child of the leaf node L. We assume our MCTS Selection and MCTS Expansion functions mutate the original batch $\mathcal{I}$ according to the game tree and store the batch corresponding to each $node$ as a property of the node, which is denoted as $node.\mathcal{I}$. Note that $node.\mathcal{I}$ is created by applying the mutations corresponding to the edges that are on the path from the root
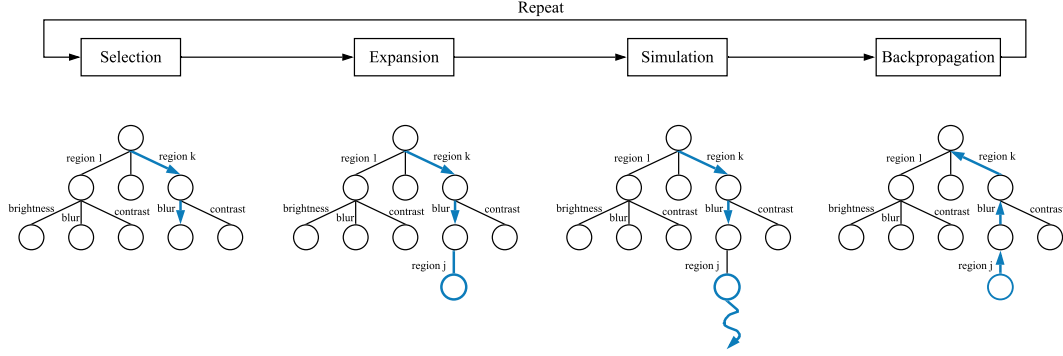
Figure 3: Scheme of a Monte-Carlo Tree Search.

Algorithm 1 Algorithmic description of our method: DeepSmartFuzzer

1: **procedure** DEEPSMARTFUZZER(T, Input_Chooser, Coverage_Criterion, tc_0, tc_1, tc_2, tc_3)
2:     **while** not tc_0 **do**
3:         $\mathcal{I}$ = Input_Chooser(T)
4:         R = MCTS_Node($\mathcal{I}$)
5:         best_coverage, $\mathcal{I}^{best}$ = 0, $\mathcal{I}$
6:         **while** not tc_1 **do**
7:             **while** not tc_2 **do**
8:                 L = MCTS_Selection(R)
9:                 C = MCTS_Expansion(L)
10:                 $\mathcal{I}^{(n-1)}$ = $C.\mathcal{I}$ $\triangleright \mathcal{I} \xrightarrow{MCTS\ Selection\ and\ Expansion} \mathcal{I}^{(n-1)}$
11:                 $r^{(n)}, m^{(n)}$ = MCTS_Simulation(C)
12:                 $\mathcal{I}^{(n)}$ = Input_Mutator($\mathcal{I}^{(n-1)}$, $r^{(n)}$, $m^{(n)}$) $\triangleright \mathcal{I}^{(n-1)} \xrightarrow{\mathcal{IM}(r^{(n)}, m^{(n)})} \mathcal{I}^{(n)}$
13:                 **if** not tc_3 **then** $\triangleright$ the game is not finished: $d(\mathcal{I}^{(n)}, \mathcal{I}) < \epsilon$
14:                     coverage_increase = Coverage_Criterion(T $\cup$ $\mathcal{I}^{(n)}$) - Coverage_Criterion(T)
15:                     **if** coverage_increase > best_coverage **then**
16:                         best_coverage, $\mathcal{I}^{best}$ = coverage_increase, $\mathcal{I}^{(n)}$
17:                     MCTS_Backpropagation(C, coverage_increase)
18:         R = select_child(R) $\triangleright$ selects the child that has the greatest value in terms of coverage increase
19:         **if** best_coverage > 0 **then**
20:             T = T $\cup$ $\mathcal{I}^{best}$
21:     **return** T

---

node R to the given $node$. In line 10, the mutated batch that is the result of MCTS Selection and Expansion is referred as $\mathcal{I}^{(n-1)}$. Then, the MCTS Simulation plays the game until a complete action so that $r^{(n)}$ and $m^{(n)}$ are assigned to a region index and a mutation index, respectively (line 11). The input mutator then mutates the batch $\mathcal{I}^{(n-1)}$ according to a region index $r^{(n)}$ and a mutation index $m^{(n)}$ so that a new batch $\mathcal{I}^{(n)}$ is created (line 12). Termination condition (tc_3) limits distance between the mutated batch $\mathcal{I}^{(n)}$ and the original batch $\mathcal{I}$. If this new batch $\mathcal{I}^{(n)}$ does not violate the termination condition (tc_3), then the mutated batch $\mathcal{I}^{(n)}$ is considered as a candidate batch of test inputs (line 13-14). In line 15, coverage increase is calculated from the difference between coverage of test set T union mutated batch $\mathcal{I}^{(n)}$ and just test set T. If this is the greatest coverage increase for this batch $\mathcal{I}$, the mutated batch $\mathcal{I}^{(n)}$ is stored as the best mutated batch $\mathcal{I}^{best}$ (line 15-16). MCTS Backpropagation is applied from the new child node C with coverage

increase as reward (line 17). This concludes one iteration of the while loop with tc_2, and the algorithm continues looping this way to explore the root node until tc_2. When termination condition tc_2 is reached, it sets the best child R' of root R as the new root node (line 18). The best child R' is the node with the greatest value, which is average coverage increase (reward) found on the paths (sequences of mutations) that contain the node. After setting a child as the new root, an iteration of the while loop with tc_1 is completed, and the while loop continues iterating by working on the subtree of the previous game tree. After the while loop is completely finished, the best batch found $\mathcal{I}^{best}$ is added to the test set if it creates a coverage increase (line 19-20). This concludes a complete iteration of MCTS on the batch $\mathcal{I}$ and the algorithm continue iterating with new batches until termination condition tc_0 is reached. When it is reached, the final test set which includes the mutated inputs found up to that point is returned (line 21). Figure 4 illustrates the algorithm in ac-

(a) The game tree

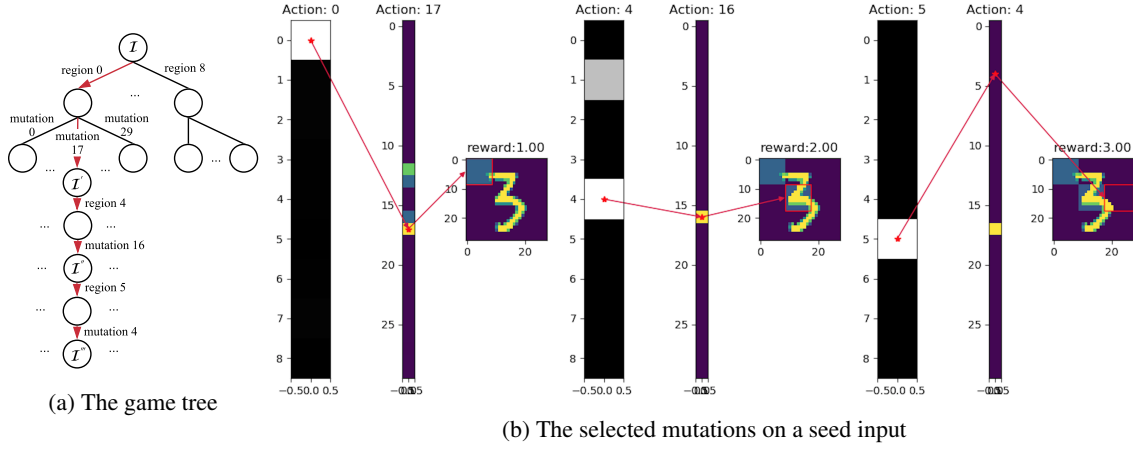(b) The selected mutations on a seed input

Figure 4: Visualization of a snapshot when our method searching the mutation space for TFC and MNIST-LeNet5 where action columns represents the potentials (the more brighter the more potential) of each enumerated action on the search tree

tion as follows: it selects a region (action) and a mutation (action) so that the input mutator applies the mutation to the region, and then this process is continued repeatedly while MCTS is searching the game tree.

Experiments[1]

Setup

Datasets and DL Systems We evaluate DeepSmartFuzzer on two popular publicly available datasets namely MNIST (LeCun 1998) and CIFAR10 (Krizhevsky and Hinton 2009) (referred to as CIFAR from now on). MNIST is a handwritten digit dataset with 60000 training and 10000 testing inputs. Each input is a 28x28 pixel white and black image with a class label from 0 to 9. CIFAR is a 3-channel colored image dataset with 50000 training and 10000 testing samples. Each input is a 3x32x32 image in ten different classes (e.g., plane, ship, car). For the MNIST dataset, we study LeNet1, LeNet4, and LeNet5 (LeCun et al. 1998) DNN architectures, which are the three well-known and popular models in the literature. For the CIFAR dataset, we make use of a suggested convolutional neural network (CNN) architecture with proven success to some degree. The subject DNN architectures are selected from different complexities, and they achieve competitive accuracy in the respective test sets. More details can be seen in Table 1.

Compared Techniques and Coverage Criteria Benchmarks We evaluate our tool by comparing its performance with two existing CGF frameworks for deep learning systems. The first tool, namely DeepHunter (Xie et al. 2018), aims to achieve high coverage by randomly selecting a batch of inputs and applying random mutations on them. Deep-Hunter also leverages various fuzzing techniques from software testing, such as power scheduling. However, the tool is not publicly available. Therefore we use our implementation of DeepHunter in evaluation. The second tool, namely

---

[1]The code for our method and experiments can be found on https://github.com/hasanferit/DeepSmartFuzzer.

Table 1: Datasets and DL Systems used in our experiments.

Dataset	DL System	Parameters	Accuracy
MNIST	LeNet1	7206	98.33%
	LeNet4	69362	98.59%
	LeNet5	107786	98.96%
CIFAR	20 layer CNN with max-pooling and dropout layers	952234	77.68%

Tensorfuzz (Odena and Goodfellow 2018), uses the guidance of coverage to debug DNNs. For example, it finds numerical errors and disagreements between neural networks and quantized versions of those networks. Tensorfuzz code is publicly available, and we integrate it into our framework.

For an unbiased evaluation of DeepSmartFuzzer, we test our tool on various coverage criteria from the literature. We use DeepXplore's (Pei et al. 2017) neuron coverage (NC), DeepGauge's (Ma et al. 2018) k-multisection neuron coverage (KMN), neuron boundary coverage (NBC), strong neuron activation coverage (SNAC) and Tensorfuzz's coverage (TFC). Neuron Coverage is defined as the ratio of neurons whose activation value is greater than a threshold to all neurons given a set of inputs. KMN, NBC, and SNAC are derived from Neuron Coverage. KMN divides the space of activation values into k sections. Given a set of inputs, it finds the sections that the activation values of each neuron fall into. Those sections are said to be covered. Then it calculates the ratio of the covered sections to all sections. NBC finds the boundaries (maximum and minimum) of activation values of each neuron given the set of training inputs. Neurons whose activation values are out of boundary values are said to cover the boundaries. The ratio of covered boundaries to all boundaries is the coverage value. TFC uses the output activation vector of the penultimate layer of the DNN. Different inputs can generate different activation vectors. It

Model - Coverage / CGF	MNIST LeNet1 (NC)	MNIST LeNet1 (KMN)	MNIST LeNet1 (NBC)	MNIST LeNet1 (SNAC)	MNIST LeNet1 (TFC)	MNIST LeNet4 (NC)	MNIST LeNet4 (KMN)	MNIST LeNet4 (NBC)	MNIST LeNet4 (SNAC)	MNIST LeNet4 (TFC)
DeepHunter	0	2.34 ± 0.03 %	35.42 ± 2.76 %	41.67 ± 4.17 %	29.00 ± 3.61	0	1.91 ± 0.04 %	**13.15 ± 2.34 %**	16.67 ± 0.81 %	20.00 ± 2.00
TensorFuzz	0	1.83 ± 0.23 %	0	0	0.33 ± 0.58	0	1.26 ± 0.05 %	0	0	0
DeepSmartFuzzer	0	**2.91 ± 0.11 %**	**41.67 ± 4.77 %**	**42.36 ± 6.36 %**	**204.67 ± 8.50**	**1.41 ± 0.00 %**	**2.07 ± 0.14 %**	11.50 ± 1.13 %	**16.90 ± 3.07 %**	**64.33 ± 6.03**
DeepSmartFuzzer(clustered)	0	2.88 ± 0.04 %	38.54 ± 0.00 %	39.58 ± 7.51 %	111.00 ± 14.53	**1.41 ± 0.00 %**	2.02 ± 0.07 %	11.50 ± 0.54 %	15.02 ± 2.15 %	53.33 ± 8.39

Table 2: Coverage increase achieved by each CGF for MNIST-LeNet1 and MNIST-LeNet4 models.

Model - Coverage / CGF	MNIST LeNet1 (NC)	MNIST LeNet1 (KMN)	MNIST LeNet1 (NBC)	MNIST LeNet1 (SNAC)	MNIST LeNet1 (TFC)	MNIST LeNet4 (NC)	MNIST LeNet4 (KMN)	MNIST LeNet4 (NBC)	MNIST LeNet4 (SNAC)	MNIST LeNet4 (TFC)
DeepHunter	0*	1051.00 ± 4.00	847.00 ± 159.74*	724.67 ± 180.17*	1029.67 ± 29.48	0*	1051.00 ± 4.00	1036.00 ± 12.49	1033.67 ± 27.50	1026.67 ± 33.50
TensorFuzz	0*	1023.33 ± 1.15	0*	0*	0.33 ± 0.58*	0*	768.00 ± 0.00*	0*	0*	0*
DeepSmartFuzzer	0*	1024.00 ± 0.00	1002.67 ± 36.95	533.33 ± 73.90*	1024.00 ± 0.00	128.00 ± 0.00*	981.33 ± 73.90†	1024.00 ± 0.00†	789.33 ± 195.52*	1024.00 ± 0.00
DeepSmartFuzzer(clustered)	0*	1024.00 ± 0.00	896.00 ± 128.00*	469.33 ± 97.76*	1024.00 ± 0.00	128.00 ± 0.00*	1024.00 ± 0.00†	1024.00 ± 0.00†	725.33 ± 97.76*	1024.00 ± 0.00

Table 3: Number of new inputs produced by each CGF for MNIST-LeNet1 and MNIST-LeNet4 models. *2 hours timeout †Extended experiments with 6 hours limit for timeout

differentiates between the activation vectors by using a distance threshold. If a given input creates an activation vector that is distant from the previous inputs, the given input is said to cover a new case, so the coverage is increased by one.

Hyperparamters We set neuron activation threshold to 0.75 in NC and the number of sections k to 10000 in KMN, respectively. For NBC and SNAC, we set as lower (upper) bound the minimum (maximum) activation value encountered in the training set, respectively. These are the recommended settings in the original studies. On the other hand, we observed that the distance threshold used in the original TensorFuzz study was too small for MNIST and CIFAR models such that every little mutation could increase the coverage. Therefore, we tune the threshold of TFC for LeNet1, LeNet4, LeNet5 and CIFAR CNN as 30^2, 13^2, 11^2 and 3^2, respectively.

The number of regions, the set of mutations, and termination conditions (tc_1, tc_2, tc_3) constitute the hyperparameters of DeepSmartFuzzer. The number of regions is selected as 9, which corresponds to 3×3 division of an image. The set of mutations is selected as contrast change, brightness change, and blur. The first termination conditions (tc_1) is chosen to limit MCTS from going down more than 8 levels deep in the game tree. The second termination condition (tc_2) limits the number of iterations on each root to 25. For the last termination condition (tc_3), we are using the limitations that DeepHunter (Xie et al. 2018) puts on the distance between mutated and seed inputs to avoid unrealistic mutated inputs.

Results

Summary We aim to show that DeepSmartFuzzer is able to generate good test inputs. First, we compare DeepSmartFuzzer with DeepHunter and TensorFuzz by comparing the coverage increases created by approximately 1000 new inputs for each method in combination with different DNN models and coverage criteria. Experimental results show that the inputs generated by our method result in the greatest amount of coverage increase for all (DNN model, coverage criterion) pairs except for a few. This suggests that DeepSmartFuzzer creates better test inputs than DeepHunter and TensorFuzz with regards to the coverage measurements.

Second, we calculate the percentage of adversarial, in order words error inducing, test inputs created by our method. The results indicate that there are many adversarial inputs, and they constitute an important part of all generated inputs. Therefore, we conclude that DeepSmartFuzzer is better than the other coverage-guided fuzzers for DNNs, and it is able to generate adversarial inputs. Figure 5 shows two mutated MNIST and two mutated CIFAR inputs created by DeepSmartFuzzer.
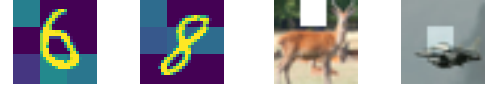

Figure 5: Example inputs generated by DeepSmartFuzzer

Comparison to DeepHunter and Tensorfuzz We focus on the inputs generated by DeepSmartFuzzer, DeepHunter, and Tensorfuzz. For experimental integrity, we make each method generate approximately 1000 input samples. Only the inputs which induce coverage increase are taken into account. We also put a time limit in order to avoid unending cases resulting from being unable to find any coverage increase for some (DNN model, coverage criteria) pairs. When a method could not produce the target amount of inputs in time, yet it creates some coverage increase such that it shows more potential to be explored, the timeout limit is extended so that they could reach 1000 inputs. This condition is not applied to TensorFuzz since it generates inputs one by one, and therefore, it could practically take days to reach 1000 inputs for some cases. The timeout is set to be 2 hours initially. It is then gradually increased to 6, 12, and 24 hours to explore the full potential. Tables 2 and 4 show the amounts of coverage increase produced by approximately 1000 generated input samples from each method with divergent set of coverage criteria and DNN models for MNIST and CIFAR datasets. In order to provide complete results, Tables 3 and 5 indicate exactly how many inputs are generated for each case. All of these results are given as mean and standard deviation of the population resulting from running the same experiment three times with different random seeds.

For most of the cases, DeepSmartFuzzer is better than the other two. Especially for the case of TFC, DeepSmartFuzzer

Model - Coverage / CGF	MNIST LeNet5 (NC)	MNIST LeNet5 (KMN)	MNIST LeNet5 (NBC)	MNIST LeNet5 (SNAC)	MNIST LeNet5 (TFC)	CIFAR CNN (NC)	CIFAR CNN (KMN)	CIFAR CNN (NBC)	CIFAR CNN (SNAC)	CIFAR CNN (TFC)
DeepHunter	0.51 ± 0.58 %	1.77 ± 0.03 %	6.23 ± 0.55 %	8.40 ± 0.66 %	19.00 ± 1.73	1.99 ± 0.19 %	0.98 ± 0.03 %	2.39 ± 0.64 %	4.48 ± 0.90 %	16.00 ± 2.65
Tensorfuzz	0.13 ± 0.22 %	0.75 ± 0.06 %	0.13 ± 0.22 %	0	1.33 ± 0.58	0.93 ± 0.15 %	0.13 ± 0.01 %	1.54 ± 0.19 %	2.92 ± 0.34 %	0
DeepSmartFuzzer	2.16 ± 0.44 %	**1.99 ± 0.01 %**	7.82 ± 1.06 %	**9.03 ± 1.10 %**	**76.33 ± 5.69**	**3.51 ± 0.48 %**	**1.38 ± 0.09 %**	2.39 ± 1.23 %	4.91 ± 2.51	42.33 ± 4.51
DeepSmartFuzzer(clustered)	**2.29 ± 0.38 %**	1.92 ± 0.08 %	**7.89 ± 0.72 %**	8.40 ± 1.91 %	76.00 ± 8.89	3.51 ± 0.37 %	1.33 ± 0.06 %	**3.83 ± 2.66 %**	**8.80 ± 7.11**	**48.67 ± 7.02**

Table 4: Coverage increase achieved by each CGF for MNIST-LeNet5 and CIFAR-CNN models.

Model - Coverage / CGF	MNIST LeNet5 (NC)	MNIST LeNet5 (KMN)	MNIST LeNet5 (NBC)	MNIST LeNet5 (SNAC)	MNIST LeNet5 (TFC)	CIFAR CNN (NC)	CIFAR CNN (KMN)	CIFAR CNN (NBC)	CIFAR CNN (SNAC)	CIFAR CNN (TFC)
DeepHunter	86.33 ± 96.81*	1051.00 ± 4.00	1021.00 ± 10.54	1021.67 ± 19.66	1034.00 ± 17.52	1047.00 ± 6.24	1035.67 ± 13.58	1031.67 ± 12.34	1049.00 ± 10.54	1042.67 ± 5.51
Tensorfuzz	0.33 ± 0.58*	448.00 ± 0.00*	0.67 ± 1.15*	0*	1.33 ± 0.58*	7.33 ± 1.15*	192.00 ± 0.00	21.00 ± 2.65	20.67 ± 3.21	0*
DeepSmartFuzzer	362.67 ± 73.90*	1024.00 ± 0.00†	1024.00 ± 0.00†	725.33 ± 36.95*	1024.00 ± 0.00	1024.00 ± 0.00	1024.00 ± 0.00†	320.00 ± 0.00	341.33 ± 36.95	1024.00 ± 0.00
DeepSmartFuzzer(clustered)	362.67 ± 73.90*	1024.00 ± 0.00†	1024.00 ± 0.00†	682.67 ± 36.95*	1024.00 ± 0.00	1024.00 ± 0.00	1024.00 ± 0.00†	341.33 ± 36.95	341.33 ± 36.95	1024.00 ± 0.00

Table 5: Number of new inputs produced by each CGF for MNIST-LeNet5 and CIFAR-CNN models. *2 hours timeout †Extended experiments with 6, 12, 24 hours limits for timeout

provides a substantial improvement over DeepHunter and TensorFuzz. This might be related to TFC being a layer-level coverage criterion, while the others are neuron-level coverage criteria. Our solution gets better when model complexity is increased. This is suggested by the increasing performance gap between our method and the others. Furthermore, DeepSmartFuzzer with clustering tends to be better than naive DeepSmartFuzzer when the complexity of the model is increased.

On the other hand, for a few cases, our approach fails to provide an improvement. For example, in neuron coverage (NC) with LeNet1 model case, we observe that all fuzzers fail to generate any coverage-increasing input. This is because when we cannot find any reward (i.e. coverage increase), our MCTS solution is similar to a random search. However, we believe this problem can be avoided with a well-designed reward shaping, and this is left to future work. Also, for the case of LeNet4 in combination with NBC, DeepHunter seems to be better than ours. This may indicate a need for further hyperparameter tuning since it conflicts with the general trend. **Overall, we conclude that DeepSmartFuzzer provides a significant improvement over existing coverage-guided fuzzers for DNNs.**

Model - Coverage / Generated Inputs	MNIST LeNet1 (KMN)	MNIST LeNet1 (NBC)	MNIST LeNet1 (SNAC)	MNIST LeNet1 (TFC)
# Adversarial	160.00 ± 22.11	59.00 ± 3.00	32.00 ± 8.54	183.67 ± 21.78
# Total	917.33 ± 73.90	1002.67 ± 36.95	533.33 ± 73.90	1024.00 ± 0.00
Percent %	**17.43 ± 1.73 %**	**5.89 ± 0.37 %**	**5.94 ± 0.92 %**	**17.94 ± 2.13 %**

Table 6: Statistics on adversarial inputs generated by DeepSmartFuzzer on MNIST dataset.

Adversarial Input Generation We further experiment with our proposed method in order to check how good it is at finding error-inducing inputs, which is one of the ultimate purposes of testing. The results are provided in Table 6. The percentage of adversarial inputs that are generated by DeepSmartFuzzer is dependent on the used coverage criteria as expected. DeepSmartFuzzer with KMN or TFC provides approximately 17% adversarial inputs while DeepSmartFuzzer with NBC or SNAC provides approximately 5% adversarial inputs. Although one may think that these percentages are low, we note that adversarial example generation is not the main goal of DeepSmartFuzzer and the coverage criteria used in this paper. Thus, we can say that DeepSmartFuzzer is able to generate error-inducing (adversarial) input samples.

Conclusion & Future Work

In this study, we introduce an advanced coverage guided fuzzer for DNNs that uses Monte Carlo Tree Search (MCTS) to explore and exploit the coverage increase patterns. We experimentally show that our method is better than the previous coverage guided fuzzers for DNNs. Our results also show the potential of reinforcement learning methods for DNN testing. We use naive coverage increase as reward. Therefore, experimentation with reward shaping and different reinforcement learning methods for this problem are left to future studies. We also show that an important portion of the test inputs generated by our method is adversarial. This suggests that our method is successful in terms of testing DNNs. Finally, we share the code for our experiments online in order to provide a base for future studies.

References

American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 13-05-2019.

Bhagoji, A. N.; He, W.; Li, B.; and Song, D. 2018. Practical black-box attacks on deep neural networks using efficient query mechanisms. In *European Conference on Computer Vision*, 158–174. Springer.

Bojarski, M.; Del Testa, D.; Dworakowski, D.; Firner, B.; et al. 2016. End to end learning for self-driving cars.

Bouzy, B., and Helmstetter, B. 2004. *Monte-Carlo Go Developments*. Boston, MA: Springer US. 159–174.

Burton, S.; Gauerhof, L.; and Heinzemann, C. 2017. Making the case for safety of machine learning in highly automated driving. In *Computer Safety, Reliability, and Security*, 5–16.

Carlini, N., and Wagner, D. 2017. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (S&P)*, 39–57.

Chaslot, G. M. J.; Winands, M. H.; HERIK, H. J. V. D.; Uiterwijk, J. W.; and Bouzy, B. 2008. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation* 4(03):343–357.

Chen, J.; Song, L.; Wainwright, M. J.; and Jordan, M. I. 2018. Learning to explain: An information-theoretic perspective on model interpretation. *arXiv preprint arXiv:1802.07814*.

Cireşan, D.; Meier, U.; and Schmidhuber, J. 2012. Multi-column deep neural networks for image classification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 3642–3649.

Goodfellow, I.; Shlens, J.; and Szegedy, C. 2015. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*.

Gopinath, D.; Wang, K.; Zhang, M.; Pasareanu, C. S.; and Khurshid, S. 2018. Symbolic execution for deep neural networks. *arXiv preprint arXiv:1807.10439*.

Guo, J.; Jiang, Y.; Zhao, Y.; Chen, Q.; and Sun, J. 2018. DLFuzz: Differential fuzzing testing of deep learning systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 739–743.

Hinton, G.; Deng, L.; Yu, D.; Dahl, G. E.; et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29(6):82–97.

Jorgensen, P. C. 2013. *Software testing: a craftsman's approach*. Auerbach Publications.

Julian, K. D.; Lopez, J.; Brush, J. S.; Owen, M. P.; and Kochenderfer, M. J. 2016. Policy compression for aircraft collision avoidance systems. In *IEEE Digital Avionics Systems Conference (DASC)*, 1–10.

Kim, J.; Feldt, R.; and Yoo, S. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41th International Conference on Software Engineering*, ICSE 2019.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, 282–293. Berlin, Heidelberg: Springer-Verlag.

Krizhevsky, A., and Hinton, G. 2009. Learning multiple layers of features from tiny images. Technical report, Citeseer.

Kurakin, A.; Goodfellow, I.; and Bengio, S. 2016. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*.

LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P.; et al. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11):2278–2324.

LeCun, Y. 1998. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist*.

Li, Z.; Ma, X.; Xu, C.; and Cao, C. 2019. Structural coverage criteria for neural networks could be misleading. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, 89–92. IEEE Press.

Libfuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed: 13-05-2019.

Litjens, G.; Kooi, T.; Bejnordi, B. E.; et al. 2017. A survey on deep learning in medical image analysis. *Medical Image Analysis* 42:60–88.

Lowe, D. G. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60(2):91–110.

Ma, L.; Juefei-Xu, F.; Zhang, F.; Sun, J.; et al. 2018. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Ma, L.; Juefei-Xu, F.; Xue, M.; Li, B.; Li, L.; Liu, Y.; and Zhao, J. 2019. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 614–618. IEEE.

Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; and Vladu, A. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*.

Moosavi-Dezfooli, S.-M.; Fawzi, A.; and Frossard, P. 2016. Deepfool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2574–2582.

Narodytska, N., and Kasiviswanathan, S. 2017. Simple black-box adversarial attacks on deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 1310–1318. IEEE.

Odena, A., and Goodfellow, I. 2018. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *arXiv preprint arXiv:1807.10875*.

Papernot, N.; McDaniel, P.; Jha, S.; Fredrikson, M.; et al. 2016. The limitations of deep learning in adversarial settings. In *International Symposium on Security and Privacy (S&P)*, 372–387.

Papernot, N.; McDaniel, P.; Goodfellow, I.; Jha, S.; Celik, Z. B.; and Swami, A. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 506–519. ACM.

Papernot, N.; McDaniel, P.; and Goodfellow, I. 2016. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*.

Pei, K.; Cao, Y.; Yang, J.; and Jana, S. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *Symposium on Operating Systems Principles (SOSP)*, 1–18.

Seshia, S. A.; Desai, A.; Dreossi, T.; Fremont, D. J.; Ghosh, S.; Kim, E.; Shivakumar, S.; Vazquez-Chanlatte, M.; and Yue, X. 2018. Formal specification for deep neural networks. In *Technical Report, University of California at Berkeley*.

Sun, Y.; Wu, M.; Ruan, W.; Huang, X.; et al. 2018a. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 109–119.

Sun, Y.; Huang, X.; Kroening, D.; Sharp, J.; Hill, M.; and Ashmore, R. 2018b. Testing deep neural networks.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *International Conference on Neural Information Processing Systems*, 3104–3112.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; et al. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Tian, Y.; Pei, K.; Jana, S.; and Ray, B. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering (ICSE)*, 303–314.

Wicker, M.; Huang, X.; and Kwiatkowska, M. 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 (TACAS)*, 408–426.

Xie, X.; Ma, L.; Juefei-Xu, F.; Chen, H.; et al. 2018. Deephunter: Hunting deep neural network defects via coverage-guided fuzzing. In *arXiv preprint arXiv:1809.01266*.

Zhang, M.; Zhang, Y.; Zhang, L.; Liu, C.; and Khurshid, S. 2018. Deeproad: Gan-based metamorphic autonomous driving system testing. In *arXiv preprint arXiv:1802.02295*.

Zhang, J. M.; Harman, M.; Ma, L.; and Liu, Y. 2019. Machine learning testing: Survey, landscapes and horizons. *arXiv preprint arXiv:1906.10742*.