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

CEUR-WS.org/Vol-2640/paper_19.pdf

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. Traditionally, DNN testing relies on the performance on a dedicated subset of the available data, namely test set. However, DNNs require more thorough testing approaches to exercise corner-case behaviors. Coverage-guided fuzzing (CGF) which is a common practice in software testing aims to produce new test inputs by mutating existing ones to achieve high coverage on a test adequacy criterion. CGF has been an effective method for finding error inducing inputs by satisfying a well-established criterion. In this paper, we propose a novel CGF algorithm for structural testing of DNNs. The proposed algorithm employs Monte Carlo Tree Search to drive the coverage-guided search. In our evaluation, we show that the inputs generated by our method result in higher coverage than the inputs produced by the previously introduced CGF techniques on various criteria in a fixed amount of time.

1 Introduction

Given enough amount of data and processing power, training a Deep Neural Network (DNN) is the most popular way for dealing with many hard computational problems such as image classification [Cireşan *et al.*, 2012], natural language processing [Sutskever *et al.*, 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.

Despite the remarkable achievements, recent works [Szegedy *et al.*, 2013; Goodfellow *et al.*, 2015] have demonstrated that DNNs are vulnerable to small perturbations on seed inputs, also called adversarial attacks. Considering the catastrophic results that can emerge from erroneous behaviors in safety-critical systems, DNNs must be characterized by a high degree of dependability before being deployed in safety-critical systems.

Testing is the primary practice for analyzing and evaluating the quality of a software system [Ammann and Offutt, 2016]. It helps in reducing the risk by finding and eliminating erroneous behaviors before deployment of the systems. One of the most fundamental testing concepts is defining a coverage criterion for a given test set, also called a test adequacy criterion. A coverage criterion measures how much of the system structures are exercised (covered) when test inputs are provided. Having a test set that satisfies a coverage criterion provides a degree of dependability to the system under test.

Recent research in DNN testing introduces new DNN-specific coverage criteria such as neuron coverage [Pei *et al.*, 2017] and its variants [Ma *et al.*, 2018], MC/DC-inspired criterion [Sun *et al.*, 2018b] or other criteria such as surprise adequacy [Kim *et al.*, 2019] and DeepImportance [Gerasimou *et al.*, 2020]. Previous works [Pei *et al.*, 2017; Ma *et al.*, 2018; Kim *et al.*, 2019], and future studies on coverage criteria for DNNs could be useful for exposing 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 methodology. Existing works [Xie *et al.*, 2018; Odena and Goodfellow, 2018] leverage coverage guided fuzzing (CGF) to achieve high coverage for a given criterion. However, both of these works apply mutations on inputs randomly. Therefore, their effectiveness is limited, as shown in our experiments.

In this work, we introduce DeepSmartFuzzer, a novel CGF, for achieving high coverage in DNNs for existing coverage criteria in the literature. Our ultimate goal is to help practitioners extend their test sets with new inputs so that new behaviours are covered. To that end, we leverage Monte Carlo Tree Search (MCTS) [Chaslot *et al.*, 2008], a search algorithm for decision processes. In our method, MCTS is used to determine a series of mutations that would result in the best coverage increase for a given input.

[*]Most of the work was done when the author was in Boğaziçi Uni.
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Contributions of this work are as follows:

- We introduce DeepSmartFuzzer, a novel Coverage Guided Fuzzing (CGF) technique for testing DNNs. DeepSmartFuzzer is applicable to all existing coverage metrics.
- We show the effectiveness of our method for many popular coverage criteria and for many DNNs with different complexities.
- We compare the effectiveness our method with existing CGF methods.

2 Related Work

Recently, several DNN testing techniques have been developed in the literature. Among these techniques, there exist works developing 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. Kim *et al.* [2019] proposed surprise adequacy criteria based on a measure of surprise caused by the inputs and Gerasimou *et al.* [2020] presented a metric for composing a semantically diverse test set. Sun *et al.* [2018a] proposed the first concolic testing approach for DNNs.

We now discuss 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 different from TensorFuzz and DeepHunter where random mutations are applied on inputs whereas we employ Monte Carlo Tree Search (MCTS) for applying mutations on inputs. Also, we apply mutations on a small subset of features of a given input, whereas they apply mutations on all features of a given input. DeepSmartFuzzer is also similar to [Wicker *et al.*, 2018] in that both works employ Monte Carlo Tree Search. However, the objective in [Wicker *et al.*, 2018] is to find the nearest adversarial example, whereas our objective is to increase the value of a given coverage criterion.

Szegedy *et al.* [2013] first discovered the vulnerability of DNNs to adversarial attacks. Since then, numerous white-box and black-box adversarial attacks have been proposed. The most popular attacks include FGSM [Goodfellow *et al.*, 2015], JSMA [Papernot *et al.*, 2016], PGD [Madry *et al.*, 2017], and C&W [Carlini and Wagner, 2017].

3 Background

Coverage Criterion in Software A coverage criterion partitions the input space into equivalence classes [Ammann and Offutt, 2016] and it is measured by dividing the number of equivalence classes that are sampled by at least one input in the test set to the number of all equivalence classes in the test set. For example, statement coverage in software measures the percentage of the statements in the program that are executed with the given set of test inputs.

Coverage Guided Fuzzing Coverage Guided Fuzzing (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.

Monte Carlo Tree Search (MCTS) Monte Carlo Tree Search [Chaslot *et al.*, 2008] is a search algorithm for decision processes such as game play. It represents games as trees where each node has children for each possible action to be taken. 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 aims to find the most promising action in an arbitrary state of a game. In other words, given an initial state, the objective is to find the best sequence of actions 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 action from node L) and take one of them. **Simulation:** play the game from node C by picking moves randomly until reaching a terminal condition. **Backpropagation:** propagate 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.

4 Method: DeepSmartFuzzer

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 i^{th} 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 n^{th} input mutation, $r^{(n)}$ and $m^{(n)}$ are the region and mutation indexes, respectively. Also, let $\mathcal{I}^{best} \in \{\mathcal{I}', \mathcal{I}'', ...\}$ be the best batch that creates the greatest amount of coverage increase.

4.1 Overview

DeepSmartFuzzer is an MCTS-based coverage-guided fuzzer for DNNs. It can be classified as a grey-box testing method since it uses coverage information which is related to the internal states of a DNN model. Our method generates inputs that increase the current level of coverage formed by

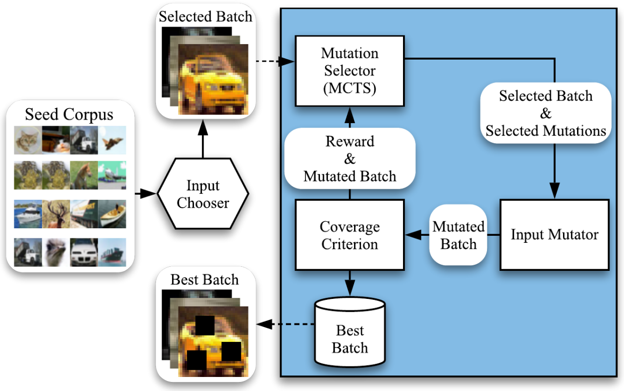


Figure 1: Workflow of DeepSmartFuzzer for an iteration

the initial test set. The core idea of our approach is to employ reward-guided exploration and exploitation in order to achieve high coverage scores. The workflow of the proposed method is illustrated in Figure 1. It is composed of an input chooser, a coverage criterion, an input mutator, and a mutation selector, which is the essential part.

For each iteration, the input chooser chooses a batch, which is a set of inputs $\mathcal{I}$. Then, the mutation selector determines the mutation (r', m') to be applied to the inputs. Note that, applying one kind of mutation to all input features may be too coarse because, for a given mutation, a subset of input features may increase coverage while others may decrease coverage. Our fuzzer takes a finer approach and applies mutations to a subset of input features (i.e. *regions*) at each step. The input mutator takes the selected batch $\mathcal{I}$ and the selected mutation (r', m'), where the selected mutation is applied to the selected batch of inputs such that the mutated inputs $\mathcal{I}'$ are formed ($\mathcal{I} \xrightarrow{\mathcal{IM}(r',m')} \mathcal{I}'$). The mutated inputs are then given to the coverage criterion to calculate the coverage of the mutated inputs together with the test set. The coverage and mutated inputs are given to the mutation selector such that it could use 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. Now, we detail each component.

4.2 Input Chooser

We use two types of input choosers for selecting inputs. These are random input chooser and clustered random input chooser. The random input chooser randomly samples a batch of inputs using the uniform random distribution. The clustered random input chooser samples 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 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.

4.3 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. This formulation of the problem has a sequential nature. Therefore, we decided to model the problem as a game, which consists of a sequence of actions and rewards related to the actions. We use a two-player game model since two selections (one for region and one for mutation) are made for each mutation.

Formulating 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 mutations could be applied on top of each other and a sequence of mutated batches is generated as described above. 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, the coverage increase corresponds to the difference between the coverage for the current test set and the coverage obtained by adding a new batch to the test set. Overall, the purpose of the mutation selector is to find the best sequence of mutations that could result in the greatest amount of coverage increase.

End of the game In order to avoid creating unrealistic inputs and consequently human intervention to eliminate unrealistic inputs, we put constraints on 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 exploit rewards and guide the search for mutations towards rewarded mutations. For this purpose, MCTS searches 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 MCTS is finished.

4.4 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 potentially makes the job of mutation selector harder. Therefore, 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.

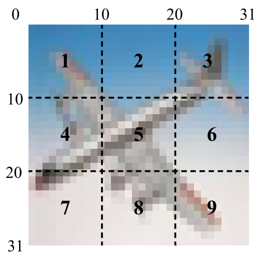


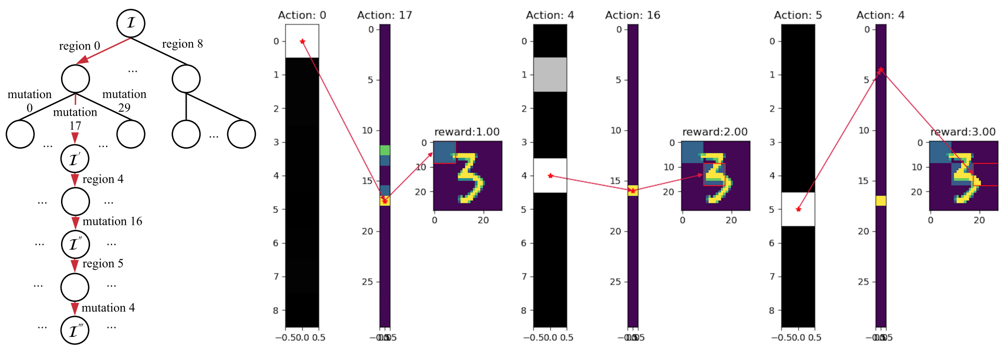
Figure 2: Regions

4.5 Algorithm

Algorithm 1 Algorithmic description of DeepSmartFuzzer

```
1:  procedure DeepSmartFuzzer(T, $tc_0$, $tc_1$, $tc_2$, $tc_3$)
2:      while not $tc_0$ do
3:          $\mathcal{I}$ = Input_Chooser(T)
4:          $R$ = MCTS_Node($\mathcal{I}$)
5:          best_cov, $\mathcal{I}^{best}$ = 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:                 $\mathcal{I}^{(n-1)}$ = get_batch_corresponding_to_node(C)
11:                 $r^{(n)}$, $m^{(n)}$ = MCTS_Simulation($C$)
12:                 $\mathcal{I}^{(n)}$ = Input_Mutator($\mathcal{I}^{(n-1)}$, $r^{(n)}$, $m^{(n)}$)
13:                 if not $tc_3$ then
14:                     cov_inc = Coverage(T $\cup$ $\mathcal{I}^{(n)}$) - Coverage(T)
15:                     if cov_inc $>$ best_cov then
16:                         best_cov, $\mathcal{I}^{best}$ = cov_inc, $\mathcal{I}^{(n)}$
17:                     MCTS_Backpropagation($C$, cov_inc)
18:             $R$ = select_child($R$)
19:         if best_cov $>$ 0 then
20:             T = T $\cup$ $\mathcal{I}^{best}$
21:     return T
```

We describe our method more formally 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 the 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 limits the search space by limiting the number of levels in the search tree that the MCTS can search. Next, the while loop in line 7 refers to iterating until a termination condition (tc_2) that determines the number of times the subtree of the root node R will be explored. In line 8, MCTS Selection, which is selection of a path from the root node R to a leaf node L using the potentials (calculated by using UCT) of the nodes, 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. In line 10, the mutations formed by the path from the root node of the game tree to the given node C are applied to the initial batch $\mathcal{I}$ and it results in $\mathcal{I}^{(n-1)}$. Basically, $\mathcal{I}^{(n-1)}$ is the mutated batch that is the result of MCTS Selection and Expansion. Then, 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). Our MCTS Simulation is different than the original MCTS Simulation. Instead of playing the game randomly until the end, our MCTS Simulation plays the game randomly until a complete action since our game formulation produces a reward for every complete action. 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) controls the rationality of the generated batch by limiting the 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 a candidate batch of test inputs (line 13-14). In line 15, coverage increase is calculated from the difference between the coverage of test set T together with the mutated batch $\mathcal{I}^{(n)}$ and the 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 to explore the root node until tc_2. When termination condition tc_2 is reached, it sets the best child of root R as the new root node (line 18). The best child is the node with the greatest value, which is the 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 child node (now called as the root node R). 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). Here, we add the complete batch to the test set in order to avoid the search for the effective inputs in the batch and thereby speed up the process. This concludes a complete iteration of MCTS on the batch $\mathcal{I}$ and the algorithm continues iterating with new batches until termination condition tc_0 is reached. When tc_0 is reached, the final test set which includes the mutated inputs found up to that point is returned (line 21).



(a) The game tree

(b) The selected mutations on a seed input

Figure 3: 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

Figure 3 illustrates the algorithm in action 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.

5 Experiments[2]

5.1 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 architecture [Wicker *et al.*, 2018].

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. DeepHunter 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 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. 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).

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 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 use the limitations that DeepHunter [Xie *et al.*, 2018] puts on the distance between mutated and seed inputs to avoid unrealistic mutated inputs.

5.2 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 in Table 1 and 3 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.

[2]Source code: https://github.com/hasanferit/DeepSmartFuzzer

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 1: 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 2: 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

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 3: 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 4: 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

Figure 4 shows examples of mutated MNIST and CIFAR inputs created by DeepSmartFuzzer.

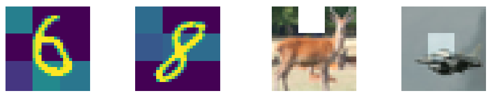


Figure 4: Example inputs generated by DeepSmartFuzzer

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 1 and 3 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 2 and 4 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 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.

In order to check the statistical significance of the results, we apply one-tailed t-test. We check two hypotheses which are whether DeepSmartFuzzer is better than Tensorfuzz and whether DeepSmartFuzzer is better than DeepHunter in terms of coverage increase. For the statistical comparisons, we use all 60 experiments (with different models and different coverage criteria) for each CGF method as observations. The significance threshold is set at .05. We find $P<.001$ for the former hypothesis and $P=.007$ for the latter hypothesis. Therefore, we accept the hypotheses.

Overall, we conclude that DeepSmartFuzzer provides a significant improvement over existing coverage-guided fuzzers for DNNs.

6 Conclusion & Future Work

In this study, we introduce a novel Coverage Guided Fuzzing (CGF) technique 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 previous CGFs for DNNs in terms of satisfying subject coverage criteria. Our results also show that MCTS methods can be promising for better DNN testing. We use naive coverage increase as reward. Therefore, experimentation with reward shaping and different reinforcement learning methods for this problem are left for future studies. Finally, we share the source code regarding to our experiments and implementation publicly in order to provide a base for future studies.

Acknowledgements

This research was supported in part by Semiconductor Research Corporation under task 2020-AH-2970.

References

[Ammann and Offutt, 2016] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

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

[Chaslot *et al.*, 2008] Guillaume M JB Chaslot, Mark HM Winands, H JAAP VAN DEN HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[Cireşan *et al.*, 2012] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3642–3649, 2012.

[Gerasimou *et al.*, 2020] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. Importance-driven deep learning system testing. In *International Conference on Software Engineering*, ICSE, 2020.

[Goodfellow *et al.*, 2015] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.

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

[Kim *et al.*, 2019] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41th International Conference on Software Engineering*, ICSE, 2019.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *17th European Conference on Machine Learning*, ECML'06, pages 282–293. Springer-Verlag, 2006.

[Krizhevsky and Hinton, 2009] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[LeCun *et al.*, 1998] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

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

[Madry *et al.*, 2017] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

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

[Papernot *et al.*, 2016] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, et al. The limitations of deep learning in adversarial settings. In *International Symposium on Security and Privacy (S&P)*, pages 372–387, 2016.

[Pei *et al.*, 2017] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Symposium on Operating Systems Principles (SOSP)*, pages 1–18, 2017.

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

[Sun *et al.*, 2018b] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Testing deep neural networks, 2018.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *International Conference on Neural Information Processing Systems*, pages 3104–3112, 2014.

[Szegedy *et al.*, 2013] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, et al. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[Wicker *et al.*, 2018] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 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)*, pages 408–426, 2018.

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