BET: Black-Box Efficient Testing for Convolutional Neural Networks

Jialai Wang[*]
Tsinghua University, BNRist
Beijing, China
wangjl19@mails.tsinghua.edu.cn

Han Qiu[*†]
Tsinghua University
Beijing, China
qiuhan@tsinghua.edu.cn

Yi Rong[‡]
Tsinghua University
Beijing, China
rongy19@mails.tsinghua.edu.cn

Hengkai Ye
Purdue University
West Lafayette, IN, USA
ye286@purdue.edu

Qi Li[*§]
Tsinghua University, BNRist
Beijing, China
qli01@tsinghua.edu.cn

Zongpeng Li[*]
Tsinghua University
Beijing, China
zongpeng@tsinghua.edu.cn

Chao Zhang[*†§]
Tsinghua University, BNRist
Beijing, China
chaoz@tsinghua.edu.cn

ABSTRACT

It is important to test convolutional neural networks (CNNs) to identify defects (e.g. error-inducing inputs) before deploying them in security-sensitive scenarios. Although existing white-box testing methods can effectively test CNN models with high neuron coverage, they are not applicable to privacy-sensitive scenarios where full knowledge of target CNN models is lacking. In this work, we propose a novel Black-box Efficient Testing (BET) method for CNN models. The core insight of BET is that CNNs are generally prone to be affected by *continuous perturbations*. Thus, by generating such continuous perturbations in a black-box manner, we design a tunable objective function to guide our testing process for thoroughly exploring defects in different decision boundaries of the target CNN models. We further design an efficiency-centric policy to find more error-inducing inputs within a fixed query budget. We conduct extensive evaluations with three well-known datasets and five popular CNN structures. The results show that BET significantly outperforms existing white-box and black-box testing methods considering the effective error-inducing inputs found in a fixed query/inference budget. We further show that the error-inducing inputs found by BET can be used to fine-tune the target model, improving its accuracy by up to 3%.

[*]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Corresponding author
[‡]School of Software, Tsinghua University
[§]Zhongguancun Lab

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534386

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Convolutional Neural Networks, Black-box Testing

ACM Reference Format:
Jialai Wang, Han Qiu, Yi Rong, Hengkai Ye, Qi Li, Zongpeng Li, and Chao Zhang. 2022. BET: Black-Box Efficient Testing for Convolutional Neural Networks. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534386

1 INTRODUCTION

Convolutional neural networks (CNNs) [11] are widely deployed in security-sensitive applications, such as autonomous driving [3], face recognition [29], blockchain forensics [17], and malware detection [34]. Recent studies [10, 27, 46] suggest that CNNs may have unexpected or incorrect prediction behaviors due to biased training data, overfitting, or modified model's inconsistency. To safeguard the accuracy of CNNs, various testing methods have been proposed [6, 7, 19, 28, 31, 49, 50]. They aim at identifying defects of CNN models such as finding adversarial inputs which induce errors. These error-inducing inputs can be used to fine-tune the model to improve its classification accuracy [22, 31].

Most existing testing methods work in white-box scenarios to explore the internal states of CNN models [7, 12, 19, 31]. Basically, they rely on either a predetermined [31] or an adaptive neuron-selection strategy [19] to achieve high neuron coverage and to identify error-inducing inputs. Although these methods can effectively identify defects for neural network models, they often require full inner knowledge of target models (i.e. structures and parameters). This is impractical in use cases such as differential testing [32, 46] that performs on multiple models and testers only have the inner knowledge of their own models but cannot access the other models. Moreover, white-box requirements for privacy-sensitive model testing scenarios are also hard to meet. For instance, model owners may allow a third-party to test their model but an "honest but curious" third-party [30] may cause model leakage.

Black-box testing methods are proposed to fill such gaps [28, 46]. A typical black-box testing method finds the defects or inconsistencies of models by continuously querying and updating samples according to the returned inference results. Without any inner knowledge of target models, black-box testing methods cannot measure the coverage but can only find error-inducing inputs with diversified labels. Since such approaches usually require a large number of queries, the main challenge of black-box testing methods is to find error-inducing inputs in a query-efficient manner.

In this work, we present BET, a novel *query-efficient black-box* testing method for CNN models. Based on analyzing the general characteristics of CNN models, we use local receptive fields of CNNs to guide the perturbation generation and craft test cases. We find that a CNN model's kernel can always be split into multiple *continuous zones* where the corresponding weights have the same sign (+/−). Applying continuous perturbations with matching shapes and signs to these zones can alter predictions to find error-inducing inputs. Thus, we craft test cases by adding such continuous perturbations directly to them (e.g. modifying pixel values of images) without requiring any inner knowledge of target models. Note that BET clearly departs from the white-box testing methods that first rely on gradient-based calculation and then transfer calculated decimals back to integers which may lead to ineffective samples (see inactivation issue in Section 4.3). We design a tunable objective function to guide the testing process to explore defects in more decision boundaries of CNNs, to diversify error-inducing inputs' labels. We also design an *efficiency-centric* policy to improve the query-efficiency of BET. Furthermore, the error-inducing inputs found by BET can be used to fine-tune the target model to improve the accuracy.

We conduct extensive evaluation of BET on three well-known datasets, i.e., CIFAR-10 [18], Tiny-ImageNet [1], and ImageNet [35], with five well-known CNN structures, i.e., VGG16 [37], VGG19 [37], ResNet18 [14], ResNet50 [14], and ResNet56 [15]. We compare our method BET with a highly-related black-box testing method [46] and four state-of-the-art white-box testing methods [7, 12, 19, 31]. Results show that our BET can acquire more error-inducing inputs with more diversified labels with a fixed query/inference budget. Moreover, these error-inducing inputs can improve target models' accuracy by $2 \sim 3\%$ with a few epochs of fine-tuning.

In summary, our contributions are as follows:

- We propose a novel black-box testing method BET for CNNs which exploits the insight that CNNs are prone to be affected by *continuous perturbations* related to local receptive fields.
- We design an *efficiency-centric policy* that enables BET to find error-inducing inputs in a query-efficient manner with a fixed query/inference budget.
- We conduct extensive evaluations to show that BET can outperform both black-box and white-box baseline testing methods in the efficiency of finding error-inducing inputs.
- We show that the error-inducing inputs found by BET can be used to fine-tune target models to improve accuracy by up to 3%.

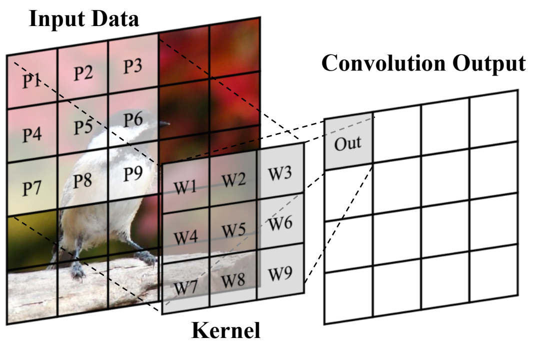

Figure 1: Simplified convolution operation.

2 BACKGROUND

2.1 The Workflow of CNNs

A convolutional neural network (CNN) is a widely-used class of deep neural network (DNN) which is also popular to be chosen as the testing target model in recent related works [7, 12, 19, 31, 45, 46]. CNNs are comprised of three types of layers: convolutional layers, pooling layers, and fully-connected layers. The convolutional layer is the core building block of CNNs and performs the convolution operation, which is a linear operation that involves the multiplication of a set of weights (called a kernel) with the input. Figure 1 illustrates a simplified convolution.

Given a kernel with n weights $W = \{w_1, w_2, ..., w_n\}$, an input data (image) x and one of its region with the same shape $P = \{p_1, p_2, ..., p_n\}$. The output of the convolution operation in a CNN model is defined as in Equation 1.

$$Out = W \cdot P = \sum_{i=1}^{n} w_i * p_i, \tag{1}$$

The kernel will be systematically applied to each region of the same shape as the kernel (even overlapping regions) of the input data (e.g. an image), from left to right and from top to bottom, to form a convolutional output data.

2.2 Related Work

Testing CNNs. The crux of CNN testing lies in the generation of test cases that explore incorrect behaviors of CNN models. The current testing techniques for finding CNNs' incorrect behaviors are analogous to coverage-guided fuzzing techniques [8, 48], which are used to maximize code coverage in finding software bugs. Similarly, existing CNN testing techniques define specific coverage metrics [16, 21, 31, 36, 38–40] called neuron coverage. They aim to fully explore the inner states of CNN models by maximizing the neuron coverage. These testing techniques adopt various algorithms to maximize neuron coverage and find incorrect behaviors of CNNs, i.e., error-inducing inputs. However, such metrics are not suitable for black-box testing techniques as neuron coverage needs to be acquired with full knowledge of target CNN models, e.g. models' parameters. Existing black-box testing techniques [28, 46] could only rely on CNNs' outputs as feedback to guide the testing process and find error-inducing inputs.

There are two common scenarios for CNN testing, i.e., *differential testing* (DT) [31, 46] and *single model testing* (ST) [12, 19]. DT finds error-inducing inputs which trigger behavior inconsistencies among multiple models that have similar functionality (e.g., models of self-driving cars by Google and Tesla) and can work without labeled data. For example, given two CNN models C_1, C_2 with similar functionality, an error-inducing input x_{err} is found if $C_1(x_{err}) \neq C_2(x_{err})$. ST only focuses on one target model C to find error-inducing inputs and requires labels for test data. For example, given a CNN model under test C, an original input x and the corresponding label l, an error-inducing input x_{err} is found if $C(x_{err}) \neq l$.

Recent testing methods can be generally classified into two categories, i.e., white-box testing [7, 12, 19, 31] and black-box testing [28, 46], according to the transparency degree of CNNs. Details are given as follows.

White-box testing methods. Pei et al. [31] focus on DT and propose the concept of neuron coverage to represent the internal states of target models. They use gradient ascent to find error-inducing inputs while improving neuron coverage for exploring various internal states of target models. Differently, Guo et al. [12] focus on ST and introduce the fuzzing idea [48] to test CNNs. The testing is guided by the neuron coverage proposed by DeepXplore [31], i.e., updating seed corpus and prioritizing test cases according to coverage guidelines. Xie et al. [45] propose DeepHunter, which generates perturbations by metamorphic mutations rather than computing gradients of optimization functions. Lee et al. [19] refine the white-box testing methods. They find that most white-box testing methods are limited to fixed neuron-selection strategies while using neuron coverage to guide testing. Correspondingly, they propose a dynamic neuron-selection strategy, which is dynamically adjusted through online learning. Wang et al. [43] focus on improving model robustness against adversarial attacks [41], thus compromising model accuracy, which is different from mainstream testing methods aiming to improve model accuracy.

Black-box testing methods. Odena et al. [28] propose the first black-box testing method for ST, TensorFuzz, which is based on a coverage-guided testing technique and directly adopts CNN's outputs as the coverage. They add random noise to test cases, which restricts their efficiency in finding effective error-inducing inputs. Xie et al. [46] propose a black-box method for DT, DiffChaser, which uses a genetic algorithm to find error-inducing inputs to test the inconsistency of a CNN model and its compressed version. These black-box methods require abundant queries to find error-inducing inputs and are not efficient.

Overall, existing white-box testing techniques mainly aim to achieve high coverage but require full knowledge of target CNN models, which limits their applications scenarios. Meanwhile, we note that existing black-box testing techniques aim to find error-inducing inputs by queries but are not highly efficient.

3 METHODOLOGY

3.1 Key Insight

A convolutional kernel can be split into multiple continuous zones where the corresponding weights have the same sign (+ or −).

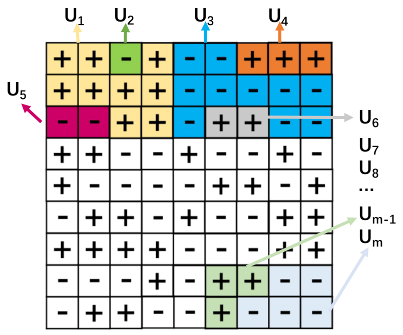

Figure 2: Continuous zones $\{U_1, U_2, ..., U_m\}$ of a kernel W.

Perturbations that have the same signs in each zone are called *continuous perturbations*.

After analyzing the workflow of CNNs, we find that they have a weakness regarding error-inducing inputs: *CNNs are prone to be affected by continuous perturbations*. Generally speaking, given an input x, continuous perturbations can maximize the output changes of the first convolutional layer that operates on x. The change on these output values will propagate to inner layers. Then, once the input of an inner layer has been significantly changed, such change propagates and may eventually cause the prediction result cross the decision boundary of CNNs to generate error-inducing inputs.

The analysis of this weakness is formulated as follows. Given a convolution kernel with n weights $W = \{w_1, w_2, ..., w_n\}$, an original input x and one of its region with the same shape $\hat{P} = \{\hat{p_1}, \hat{p_2}, ..., \hat{p_n}\}$, perturbations δ to apply, and the yielded test case $\hat{x} = x + \delta$. The output of the convolution operation is defined in Equation 2.

$$\hat{Out} = W \cdot \hat{P} = \sum_{i=1}^{n} (w_i \times \hat{p_i}), \tag{2}$$

Given Equation 1 and 2, and the fact that an error-inducing input needs to cross the decision boundary of its original prediction, we hope to maximize $Dif = |\hat{Out} - Out|$. Specifically, Dif represents changes that will propagate to inner layers behind the current convolutional layer. We can represent all following inner layers as a function F. In general, the bigger the Dif, the more likely $F(Out \pm Dif) \neq F(Out)$, i.e., the more likely to acquire error-inducing inputs. Dif can be computed as in Equation 3.

$$Dif = |W \cdot \hat{\delta}| = |\sum_{i=1}^{n} (w_i \times \hat{\delta}_i)|, \tag{3}$$

where $\hat{\delta} = \{\hat{\delta}_1, \hat{\delta}_2, ..., \hat{\delta}_n\}$ is the corresponding part of δ. The goal is maximizing Dif. In white-box settings, W is fixed and known by the tester, thus this optimization problem is straightforward.

In black-box scenarios, W is still fixed but unknown to us. However, we notice that $W = \{w_1, w_2, ..., w_n\}$ can be split into a batch of kernel zones $U = \{U_1, U_2, ..., U_m\}$, where

- For $\forall i \neq j, U_i \cap U_j = \varnothing$, and
- $W = U_1 \cup U_2, ..., \cup U_m (m \leq n)$.

Here U_i is a part of W, whose elements are adjacent and have the same sign, and $\sum_{i=1}^{m} |U_i| = n$. Figure 2 illustrates the split of an

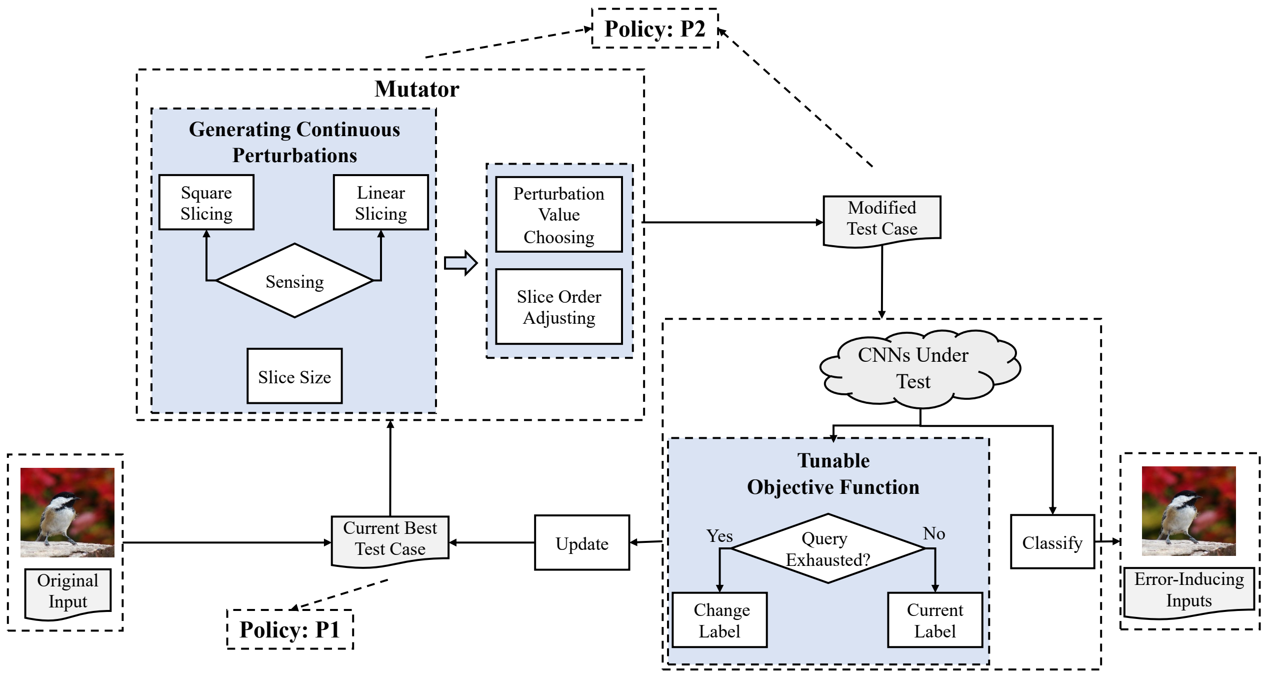

Figure 3: BET workflow.

example kernel. Then we can rewrite Dif in Equation 4.

$$Dif = |W \cdot \hat{\delta}| = |\sum_{i=1}^{m}(U_i \times \hat{\delta}_i)|, \quad (4)$$

where $\hat{\delta} = \{\hat{\delta}_1, \hat{\delta}_2, ..., \hat{\delta}_m\}$ is the corresponding part of the perturbation δ, different from the definition in Equation 3.

For simplicity, we fixate the absolute value of $\hat{\delta}_i$ as ϵ in BET, and try to find proper signs to maximize Dif. Note that, the perturbations can be divided into two general categories, i.e., discrete perturbations $\hat{\delta}_{dis}$ and continuous perturbations $\hat{\delta}_{con}$. (1) For discrete perturbations $\hat{\delta}_{dis}$, where the signs of $\hat{\delta}_{dis\_i} \in \hat{\delta}_{dis}$ are random, the corresponding Dif is represented as Dif_{dis}. (2) For continuous perturbations $\hat{\delta}_{con}$, the signs of $\hat{\delta}_{con\_i} \in \hat{\delta}_{con}$ are the same as U_i's. The corresponding Dif is represented as Dif_{con}.

Since all elements in U_i have the same sign, we can induce that $U_i \times \hat{\delta}_{con\_i}$ is positive and is larger than $|U_i \times \hat{\delta}_{dis\_i}|$, which makes $Dif_{dis} = |\sum_{i=1}^{m}(U_i \times \hat{\delta}_{dis\_i})| < |\sum_{i=1}^{m}(U_i \times \hat{\delta}_{con\_i})| = Dif_{con}$. In other words, if we elaborately craft continuous perturbations to match convolution kernels, the yielded Dif would be the largest and is more likely to affect the predictions of CNNs.

In summary, CNNs are prone to be affected by continuous perturbations. However, in black-box settings, it is impossible to perfectly match the unknown kernel to generate continuous perturbations. But still, we can approximately match convolution kernels which can still be effective to perform black-box testing.

3.2 Workflow of BET

The overall workflow of BET is shown in Figure 3. BET initializes the current best test case with the given original input. Then, BET iteratively searches error-inducing inputs based on the current best test case. Particularly, in each iteration, BET generates continuous perturbations and adds perturbations to the current best test case to get the modified test case. Then, the modified test case is sent to query target models to check whether an error-inducing input is acquired. The tunable objective function decides whether to update the current best test case with the modified test case. The whole process iterates until the query budget is exhausted.

The BET workflow consists of three main parts. (1) **Tunable objective function:** this objective function guides the testing process to explore different decision boundaries of CNNs. (2) **Efficiency-centric policy:** this policy improves our testing efficiency. It restricts only one output that will be derived after adding perturbations and chooses only the best one that will be kept for further testing. (3) **Mutator:** this module generates continuous perturbations with basic shapes according to the core insight. Particularly, in each iteration of the testing, after the mutator module generates continuous perturbations with basic shapes, the objective function decides whether to save current perturbations and dynamically drive the perturbations' shapes towards the underlying convolution kernels.

3.3 Tunable Objective Function

Our tunable objective function guides the testing process and thus is essential for BET to explore different decision boundaries and thoroughly test CNNs. Specifically, during the test process, perturbations with higher objective function scores will be saved for further optimization. If the objective function targets a specific label, the testing process will be led to enter the decision boundary of this specific label to explore corner cases, i.e., generate error-inducing inputs with the corresponding label. Besides, the objective function is tunable, i.e., its specific label will change dynamically (after a given query budget t). Thus, BET could explore different decision boundaries and perform thorough testing.

We propose tunable objective functions for different testing scenarios, i.e., the DT and the ST.

Tunable objective function for DT. For DT, we hope to identify inputs that can maximize the prediction difference between the target model with other models. So testers will prioritize exploiting error labels for DT. Given an input x, labels $L = \{l_1, l_2, ...l_n\}$ (including all classes except for x's original label), and query budgets $\{t_1, t_2, ...t_n\}$, the target CNN under test C, x's original label $l_0 = C(x)$, and other models $\hat{C} = \{\hat{C}_1, \hat{C}_2, ...\hat{C}_n\}$ for DT. The tunable objective function for DT is defined in Equation 5.

$$DOF(x) = C(x)[l_i] + \sum_{i=1}^{n} |C(x)[l_0] - \hat{C}_i(x)[l_0]|, \tag{5}$$

where the label l_i is selected from L in order ($l_i \in L$). Each label l_i will be used for t_i. In Equation 5, the first term $C(x)[l_i]$ means fully exploring the target model C, where l_i is dynamically adjusted for exploring different decision boundaries of the model. The second term $\sum_{i=1}^{n} |C(x)[l_0] - \hat{C}_i(x)[l_0]|$ aims to find disagreements between the target model C with a batch of other models $\hat{C} = \{\hat{C}_1, \hat{C}_2, ...\hat{C}_n\}$, as such disagreements indicate error-inducing inputs are found. In each iteration, when a test case is derived, it will be evaluated using this objective function, and saved for further exploration if the evaluation result is better.

Tunable objective function for ST. Given an input x and the test model C, the tunable objective function for ST is given in Equation 6.

$$SOF(x) = C(x)[l_i], \tag{6}$$

where l_i is the same as in Equation 5. In Equation 6, we only optimize $C(x)[l_i]$, as ST focuses on the target model C, and ground truth is provided to indicate whether error-inducing inputs are found

The order of labels and the query budget t are critical to the objective function. (1) For label orders, we prioritize labels that are prone to be vulnerable. Prioritizing error-prone areas could find more error-inducing inputs with limited queries and computing resources [47]. Particularly, we prioritize error-prone classes according to the output confidence of the target CNN (the higher the confidence score, the more priority). We also allow users to prioritize labels as they want to support ad-hoc testing cases. For example, given a face recognition model deployed in a company, it is more risky if an ordinary employee is misclassified as a manager rather than another ordinary employee. Thus, users may prioritize finding error-inducing inputs that would be misclassified as managers. (2) For the query budget t, by default we consider the global testing query budget and allocate it equally to each label. Similarly, we also allow users to allocate customized query budgets.

3.4 Efficiency-Centric Policy

The efficiency-centric policy consists of two rules **P1** and **P2** to achieve high efficiency of testing, i.e., acquiring more error-inducing inputs with more diversified labels with a fixed query budget.

P1: Only one current best test case will be kept by BET in each iteration. When a newly modified test case gets closer to the optimization target according to the objective function, then the current best test case will get updated.

P2: Only one modified test case will be derived in each iteration, i.e., by adding perturbations to a chosen area of the current best test case, to further restrain useless test cases from being generated and queried. The granularity of perturbations gets finer during testing, i.e., from coarse-grained to fine-grained, to make the optimization converge faster. Perturbation areas in multiple iterations are different (i.e., avoiding redundant test cases) and could jointly cover the whole test case (i.e., avoiding missing critical areas).

Our policy is derived from stochastic greedy [24] and the novelty of this policy mainly reflects in two points. (1) Existing solutions, in general, maintain a batch of test cases and derive a set of test cases in each iteration. BET only maintains the current best test case and derives only one modified test case in each iteration. This makes testing more efficient, as discussed in the following. (2) Stochastic greedy randomly selects perturbations from a corpus, but BET chooses perturbations from coarse-grained to fine-grained and avoids duplications of perturbation areas.

Efficiency Analysis. The efficiency-centric policy plays a key role in making testing query-efficient. Here, we explain our efficient policy and give the corresponding time complexity analysis.

Testing CNNs is an optimization problem since the goal is to maximize the objective function (details of the objective function are presented in Section 3.3) which can eventually find error-inducing inputs. However, maximizing the objective function is an NP-hard issue. Specifically, given a set of perturbations N with n elements and the objective function $F(N)$, it has to take $n!$ different cases into consideration to maximize $F(N)$. Thus, the time complexity of testing is $O(n!)$, which is unaffordable for testing CNNs.

Fortunately, in a testing task, we only need to find a feasible error-inducing input rather than finding the optimal one. As shown in previous works [2, 26], greedy algorithms are widely used to approximate the optimal solution. Specifically, for a testing task, the greedy algorithm needs to maximize $F(A)$, where F is a target function and A is the solution to the problem. At first $A = \varnothing$, in each iteration of the optimization process, the algorithm needs to (1) traverse $N \backslash A$, (2) select the best element $e \in N \backslash A$, (3) and then add e to A. After running the maximum number of iterations M, we could get the A, where $A \subseteq N$ and $|A| = M$. Thus, the time complexity reduces to $O(Mn)$ but is still high.

To further reduce the time complexity, we deploy the stochastic greedy [24] to achieve a time complexity of $O(n \log \frac{1}{\theta})$, where θ is the hyperparameter. For stochastic greedy policy, in each iteration of the optimization process, BET randomly samples $\frac{n}{M} \log \frac{1}{\theta}$ elements in $N \backslash A$ to choose the best one rather than traversing all elements in $N \backslash A$. It can approximate the optimal solution with a probability equals to $1 - \frac{1}{e} - \theta$ [24]. In our design, we adapt stochastic greedy to build a new efficiency-centric policy: (1) the policy restricts the sample size α from $\frac{n}{M} \log \frac{1}{\theta} \longrightarrow 1$, minimizing the time complexity in theory ($\alpha \propto \frac{1}{\theta}$). (2) Rather than randomly sampling elements in $N \backslash A$ as stochastic-greedy does, our policy avoids adding perturbations repeatedly (i.e., avoid redundant test cases) and samples different perturbations from coarse-grained to fine-grained to accelerate convergence.

3.5 Mutator

The key insight of our method is to craft test cases by adding continuous perturbations to utilize the weakness of CNNs. The mutator module of BET generates a set of continuous perturbations with basic shapes to craft test cases. During the testing, some qualified perturbations will be saved according to our tunable objective function, which will dynamically get updated and be approximated to the shape of the underlying convolution kernels.

Algorithm 1 depicts the workflow of the mutator module. To craft the perturbations, the mutator splits all pixels of the test case into multiple slices of the same size (i.e., having the same number of pixels). In each iteration, we only add perturbations (with the same size and shape) to one slice of the test case. All slices will be updated one by one in multiple iterations. In this way, no (potentially valuable) slices will be missed and no slices will be added with redundant perturbations. The search space of perturbations is thus greatly narrowed down which contributes to improving query efficiency in finding error-inducing inputs.

Algorithm 1: The Workflow of the Mutator

```
Input: initial test case;
Output: E: error-inducing input corpus; D: diversified label corpus ;
curBest = input;
size = SLICESIZE;
sCount = len(input) / size;
slices1 = Square-Slicing(input, size);
slices2 = Linear-Slicing(input, size);
square, _ = IterSlices(slices1, sCount, input);
linear, _ = IterSlices(slices2, sCount, input);
slices_ori = (U(square) > U(linear)) ? slices1 : slices2;
// choose a better slicing method according to U, which is the
   utility score of slicing methods
while not query budget exhausts do
    curBest = input;
    size = SLICESIZE;
    sCount = len(input) / size;
    slices = slices_ori;
    while size >= STEP do
        // split the test case into multiple slices of the
           same size
        slices = SplitEqually(slices, size);
        size = size / STEP;
        sCount = sCount * STEP;
        curBest, Flag = IterSlices(slices, sCount, curBest);
        if Flag == True then
            break;
Function IterSlices(slices, sCount, curBest):
    for i ∈ range(sCount) do
        modified-test-case = AddPerturbations(curBest, slices[i]);
        isErr, objScore, label = QueryCNNs(modified-test-case);
        if isErr then
            E = E ∪ modified-test-case ;
            D = D ∪ label ;
        if objScore >Tunable-Object-Func(curBest) then
            curBest = modified-test-case;
            Reorder(slices[i+1:]);
        if query budget for current label exhausts then
            Tunable-Object-Func changes label;
            return curBest, True;
    return curBest, False;
```

We give more details on how to slice the test cases and how to generate continuous perturbations as follows.

Slice size: While splitting the test case into slices, the mutator needs to specify slice size (the same as perturbation size). Particularly, the test case is firstly split into larger slices (e.g., of size 1024 on ImageNet in our experiments). So we can query fewer times and approach error-inducing input regions faster. Then, after larger slices are all tested, we continue to split the test case into smaller slices and update them one by one. In this way, we could zoom in on the test case and add perturbations to smaller areas to approach error-inducing inputs. Eventually, the slice size could be reduced to 1. For example, given a test case of 2048 bytes, we could split it into 2 slices of 1024 bytes, then 4 slices of 512 bytes, ..., and eventually 2048 slices of 1 byte. Note that, the slice size does not change along with the current best seed.

Slicing method: For simplicity, BET splits larger slices into smaller slices following the input order. We elaborate on how to design the shape of slices as follows. Ideally, as we have declared in Section 3.1, the shape of slices should match the shape of the convolution kernels' continuous zones (SCZ) to yield error-inducing inputs faster. But we do not know SCZ in black-box scenarios and cannot match them exactly. Instead, we can try to approximately match SCZ. We first formulate basic continuous shapes for perturbations, i.e. square and line. As the optimization process goes on, BET adds perturbations for smaller areas and saves qualified perturbations according to the objective function. In summary, BET changes the shape of perturbations dynamically to approximately match SCZ.

We provide two slicing methods for generating continuous perturbations, i.e., square slicing which divides input bytes into squares, and linear slicing which divides input bytes into lines. These two slicing methods formulate the basic shape of continuous perturbations with the following two advantages. First, they are building blocks for other shapes, useful for approximating SCZ. Second, perturbations with these two shapes are easy to cover all elements of inputs without overlapping.

For different inputs, BET needs to choose which slicing method to use, i.e., square slicing or linear slicing. It relies on a delicate sensing function to evaluate each slicing method's S' utility and therefore make the decision. For each slicing method S, we could get a set of largest slices (e.g., of size 1024). Starting from a given current best test case, we could add perturbations to these slices one by one, and generate a new current best test case, whose confidence score of the tunable objective function will increase (compared to the old best test case's) with Δ_{inc}, while its confidence score of the maximum component will decrease with Δ_{dec}. Intuitively, in a better optimization direction, most of the decrease Δ_{dec} should contribute to the increase Δ_{inc}. Therefore, we calculate a utility score of the slicing method S as $U(S) = \Delta_{inc}/\Delta_{dec}$.

In experimentation, BET first tries *square slicing* to get the largest slices and calculate its utility score $U(square)$. Then, BET uses *linear slicing* to get the largest slices and calculate its utility score $U(linear)$. If $U(square) > U(linear)$, BET chooses *square slicing* method, otherwise BET chooses *linear slicing* method. This special process is denoted as **sensing** which is useful for determining better slicing methods. Note that several queries will be made to the target CNNs in order to calculate the utility score of the slicing methods but the query cost is still affordable.

Slice order: The order of slices also matters. If perturbations are added to slices with crucial bytes earlier, we could get error-inducing inputs more efficiently. After adding perturbations to each slice, we could get an objective function score. If this score is higher than the score of the current best test case, we will update the current best test case according to this score. Then, bytes in this slice and neighbor slices are important. Therefore, we prioritize neighbor slices that have not been traversed to the head of the remaining slices to explore.

Perturbation values: Given the slice size, slicing method, and slice order, we could determine which slice should be added with perturbations in each iteration. The question to answer then is what perturbation values will be added to the input bytes of each slice. Note each byte has 256 kinds of perturbations. Thus, for input with N pixels, the search space of perturbations is 256^N which is too large to explore efficiently. Instead, we limit the perturbations to each byte to two choices $\{+\epsilon, -\epsilon\}$, where ϵ is a hyperparameter specific to application scenarios. Then, the search space of perturbations greatly reduces to 2^N.

Given an original input Eo, a current best test case Eb derived from it, and a slice to add perturbations, we examine each byte in the slice and add perturbations accordingly. Assuming the byte being examined is the i-th byte in the test case if $Eb[i] = Eo[i]$, we randomly add perturbation value of $+\epsilon$ or $-\epsilon$ to this byte. If $Eb[i] = Eo[i] - \epsilon$, we change this byte to $Eo[i] + \epsilon$. Otherwise, $Eb[i] = Eo[i] + \epsilon$ must hold and we change this byte to $Eo[i] - \epsilon$.

4 EVALUATION

In this section, we first evaluate the performance of BET on testing CNNs and then compare it with the state-of-the-art black-box and white-box testing methods on multiple CNNs and datasets in both DT and ST scenarios. Furthermore, we evaluate the accuracy improvement for target CNN models by fine-tuning with the error-inducing inputs generated by BET. All experiments are conducted on a machine with an Intel Xeon Gold 6154 CPU and four NVIDIA Tesla V100 GPUs.

4.1 Experimental Setup

Datasets and models. We conduct our experiments on the following three widely-used image datasets.

CIFAR-10 [18]: This dataset contains 50,000 training images and 10,000 testing images. Each image has a size of $32 \times 32 \times 3$ and belongs to one of 10 classes. We train target CNN models with two widely used structures, i.e., VGG16 [37] and ResNet18 [14]. For training these two models, we use SGD optimizer, and set learning rate: 0.1, decay rate: 1e-6, momentum: 0.9, batch size: 128, epoch: 80, loss: cross entropy.

Tiny-ImageNet [1]: This dataset is a simplified version of ImageNet consisting of color images with size 64×64×3 belonging to 200 classes. Each class has 500 training images and 50 validation images. We train target models with two widely used structures, i.e., VGG16 [37] and ResNet56 [14]. For training these two models, we use SGD optimizer, and set learning rate: 0.01, decay rate: 5e-4, momentum: 0.9, batch size: 128, epoch: 50, loss: cross entropy.

ImageNet [35]: This dataset contains about 1.2 million training images, 50,000 validation images, and 100,000 testing images from 1,000 classes. We consider each image is resized to $224 \times 224 \times 3$. We use two widely used pretrained models from keras, i.e., VGG19 [37] and ResNet50 [14] as our target CNNs.

In our experiments, we randomly selected 5000 images from CIFAR-10 test set, 5000 images from Tiny-ImageNet, and ImageNet validation sets, respectively, as our initial test cases. All these selected images are initially correctly classified by target models.

Hyperparameters. We use the widely used L_∞ metric [4] to restrict perturbations for error-inducing inputs on all compared methods. To set values of L_∞, we first run all compared methods under different L_∞ for multiple times, and then we choose values that make them perform relatively well, i.e., $L_\infty = 13/255$ on all datasets (all compared methods use the same setting). We set the fixed query budget to 30,000 for each image on all datasets. As for our tunable objective function, on CIFAR-10, we will consider all labels except for their original label, and allocate query budget to each label as we have mentioned in Section 3.3. On Tiny-ImageNet and ImageNet whose classes are 200 and 1,000 respectively, due to query limits, we only allocate query budget to 50 priority labels.

Metrics. We use the following commonly used metrics [19, 31, 46].

Err-Num. We compare the average number of error-inducing inputs per image (Err-Num). The higher the Err-Num is, the more error-inducing inputs are found to indicate a better testing result. It is meaningful for a testing method to generate as many error-inducing inputs as possible. These error-inducing inputs can be used to improve model accuracy, e.g. fine-tuning models with error-inducing inputs.

Label-Num. For each initial test case, BET may find multiple misclassified labels. Exploring different misclassified labels means exploring different decision boundaries of CNNs which means more defects in the testing result. We compare the average number of unique misclassified labels per image (Label-Num). The higher the Label-Num is, the label of error-inducing inputs is more diversified to indicate a better testing result.

SR. We compare the success rate (SR), which measures the percentage of original test cases that can find corresponding error-inducing inputs. The higher the SR is, the better the testing method is. This metric could reflect the effectiveness of testing methods.

4.2 Evaluation of Black-Box Differential Testing

Baselines. We compare BET with the highly-related black-box testing method DiffChaser [46]. We reproduce DiffChaser on the three datasets used in this paper. We overlook to compare with TensorFuzz in DT scenarios as it has been demonstrated less effective than DiffChaser for DT [46].

Constructing differential models. To get a convincing DT result, we directly use Tensorflow-Lite [5] (Tensorflow-Lite is among the most popular tools for CNN model migration to Android and iOS platforms) to construct quantized models for DT as DiffChaser does. Tensorflow-Lite provides two quantization options to quantize models, i.e., 16-bits and 8-bits quantization, which represent original models quantized from 32-bits to 16-bits and 8-bits respectively of floating precision. We apply these two options to all models and we get corresponding 8-bits and 16-bits quantized models as shown in Table 1. Then, we try to generate error-inducing inputs that

Table 1: Datasets, target CNNs, and quantized CNNs (quantization with 8-bits and 16-bits respectively).

Dataset	Model	8-bits	16-bits
CIFAR-10	VGG16	VGG16-q8	VGG16-q16
	ResNet18	ResNet18-q8	ResNet18-q16
Tiny-ImageNet	VGG16	VGG16-q8	VGG16-q16
	ResNet56	ResNet56-q8	ResNet56-q16
ImageNet	VGG19	VGG19-q8	VGG19-q16
	ResNet50	ResNet50-q8	ResNet50-q16

cause disagreements between the original model with its quantized models. *e.g.*, generating error-inducing inputs between ResNet50 with ResNet50-q8.

Results. Table 2, Table 3 and Table 4 show the results of DT on all datasets and models, respectively. Overall, under the same experimental conditions, BET significantly outperforms DiffChaser on all comparison metrics. Details are described as follows.

Result of Err-Num. Our method BET acquires more Err-Num than DiffChaser on all datasets and models. Particularly, on CIFAR-10, taking VGG16/VGG16-q8 as an example, for the 5,000 initial test cases, BET finds average more than 10,000 error-inducing inputs

Table 2: Evaluation results of Err-Num.

Dataset	Model	Err-Num	
		BET	DiffChaser
CIFAR-10	VGG16/VGG16-q8	**10296.2**	1779.4
	VGG16/VGG16-q16	**2475.8**	110.0
	ResNet18/ResNet18-q8	**7136.2**	6.7
	ResNet18/ResNet18-q16	**280.6**	0.2
Tiny-ImageNet	VGG16/VGG16-q8	**3153.5**	231.4
	VGG16/VGG16-q16	**142.8**	12.8
	ResNet56/ResNet56-q8	**860.2**	106.5
	ResNet56/ResNet56-q16	**88.4**	3.4
ImageNet	VGG19/VGG19-q8	**304.0**	15.6
	VGG19/VGG19-q16	**185.2**	0.8
	ResNet50/ResNet50-q8	**8611.4**	360.2
	ResNet50/ResNet50-q16	**409.8**	48.8

Table 3: Evaluation results of Label-Num.

Dataset	Model	Label-Num	
		BET	DiffChaser
CIFAR-10	VGG16/VGG16-q8	**5.8**	2.6
	VGG16/VGG16-q16	**4.6**	2.4
	ResNet18/ResNet18-q8	**6.2**	0.8
	ResNet18/ResNet18-q16	**2.0**	0.1
Tiny-ImageNet	VGG16/VGG16-q8	**11.4**	1.6
	VGG16/VGG16-q16	**2.2**	0.1
	ResNet56/ResNet56-q8	**10.2**	1.8
	ResNet56/ResNet56-q16	**3.8**	0.6
ImageNet	VGG19/VGG19-q8	**31.4**	3.3
	VGG19/VGG19-q16	**9.2**	0.7
	ResNet50/ResNet50-q8	**29.4**	4.8
	ResNet50/ResNet50-q16	**26.6**	0.6

Table 4: Evaluation results of SR.

Dataset	Model	SR (%)	
		BET	DiffChaser
CIFAR-10	VGG16/VGG16-q8	**100.0**	100.0
	VGG16/VGG16-q16	**100.0**	100.0
	ResNet18/ResNet18-q8	**100.0**	33.6
	ResNet18/ResNet18-q16	**96.5**	1.8
Tiny-ImageNet	VGG16/VGG16-q8	**100.0**	100.0
	VGG16/VGG16-q16	**100.0**	34.0
	ResNet56/ResNet56-q8	**100.0**	51.6
	ResNet56/ResNet56-q16	**100.0**	14.8
ImageNet	VGG19/VGG19-q8	**100.0**	43.5
	VGG19/VGG19-q16	**96.8**	11.4
	ResNet50/ResNet50-q8	**100.0**	72.4
	ResNet50/ResNet50-q16	**94.2**	33.2

with a fixed 30,000 query budget. This is 5.8 times more error-inducing inputs found than DiffChaser. On Tiny-ImageNet and ImageNet datasets, BET can also find significantly more error-inducing inputs on average than DiffChaser within 30,000 queries for each initial test case. This indicates BET could find more error-inducing inputs than DiffChaser in a query-efficient manner.

Result of Label-Num. Among all error-inducing inputs, we evaluate the Label-Num of BET and DiffChaser on all datasets and models. Particularly, BET can get significantly more Label-Num ($7 \sim 40\times$) than DiffChaser. Such results show BET could explore significantly more different decision boundaries of target models than DiffChaser. The more different decision boundaries explored by the testing method means the better this method is.

Result of SR. For all initial test cases, BET can find corresponding error-inducing inputs with high SR, which reflects the effectiveness of BET. However, we observe that DiffChaser could fail to achieve this. For instance, DiffChaser's SR are 33.6% and 1.8% on ResNet18/ResNet18-q8 and ResNet18/ResNet18-q16 respectively. This means BET is more effective than DiffChaser considering a higher success rate in finding error-inducing inputs.

4.3 BET for Single Model Testing

Baselines. We compared BET with four state-of-the-art white-box testing methods, i.e., ADAPT [19], two instances of DL-Fuzz [12], and DeepXplore [31]. For all these white-box methods, ADAPT has implemented them and released the source code. We directly use the source code and recommended parameters for a fair comparison. Note that, ADAPT offers two coverage strategies for them. Due to this, we run their code under these two coverage strategies respectively and choose the relatively better results for comparison.

Settings and metrics. Besides using the same testing settings and comparison metrics as Section 4.1, we also add one comparison metric (i.e., Inact-Rate) while comparing with white-box methods:

Inact-Rate. Many white-box testing techniques first preprocess initial test cases (e.g., pixel values of images are integers in [0,255]) to lift them to the continuous domain, such as decimals within [0,1] before generating error-inducing inputs.

However, after discretization, many error-inducing inputs would become inactive (i.e., can no longer mislead target CNNs) due to information loss. We use the *inactivation rate* (Inact-Rate) to reflect

Table 5: Evaluation of Err-Num in single model testing scenarios.

Dataset	Model	BET	ADAPT	DLFuzz-Best	DLFuzz-RR	DeepXplore
CIFAR-10	VGG16	**16264.4**	13131.4	12143.6	11427.8	10782.6
	ResNet18	**14272.2**	11681.3	11532.4	11738.8	8804.2
Tiny-ImageNet	VGG16	**8843.4**	6546.2	4896.8	4696.8	3542.3
	ResNet56	**9016.2**	8437.6	6867.8	5432.8	4982.8
ImageNet	VGG19	**7059.8**	3154.5	142.7	1031.7	208.6
	ResNet50	**7467.4**	5409.5	524.7	1199.0	2116.0

Table 6: Evaluation of Label-Num in single model testing scenarios.

Dataset	Model	BET	ADAPT	DLFuzz-Best	DLFuzz-RR	DeepXplore
CIFAR-10	VGG16	**9.0**	8.3	4.0	6.9	7.4
	ResNet18	**8.9**	8.9	2.6	7.9	8.1
Tiny-ImageNet	VGG16	**48.2**	41.8	3.4	16.8	12.7
	ResNet56	**49.6**	36.6	2.9	5.8	4.6
ImageNet	VGG19	**49.8**	27.1	1.3	7.5	2.6
	ResNet50	**34.7**	3.8	0.5	1.1	1.4

Table 7: Evaluation of SR (%) in single model testing scenarios.

Dataset	Model	BET	ADAPT	DLFuzz-Best	DLFuzz-RR	DeepXplore
CIFAR-10	VGG16	**100.0**	100.0	96.0	100.0	100.0
	ResNet18	**100.0**	100.0	100.0	100.0	100.0
Tiny-ImageNet	VGG16	**100.0**	93.2	89.3	91.8	90.1
	ResNet56	**100.0**	95.4	91.2	90.3	85.6
ImageNet	VGG19	**100.0**	100.0	52.4	92.4	40.2
	ResNet50	**100.0**	94.0	36.6	54.8	62.4

Table 8: Evaluation of Inact-Rate (%) in single model testing scenarios.

Dataset	Model	BET	ADAPT	DLFuzz-Best	DLFuzz-RR	DeepXplore
CIFAR-10	VGG16	**0.0**	48.4	42.1	49.8	57.9
	ResNet18	**0.0**	22.4	3.8	13.1	18.9
Tiny-ImageNet	VGG16	**0.0**	93.2	89.2	91.9	91.8
	ResNet56	**0.0**	91.7	84.2	88.8	94.5
ImageNet	VGG19	**0.0**	100.0	52.4	92.4	40.2
	ResNet50	**0.0**	94.0	36.6	54.8	62.4

this problem and compare the inactivation rate for all methods. Apparently, the lower the Inact-Rate is, the better the testing result is.

Results. Table 5, Table 6, Table 7, and Table 8 show the results of single model testing (ST) on all datasets and models, respectively. Overall, BET can acquire more error-inducing inputs with more diversified labels within a fixed query/inference budget. Details are described as follows.

Result of Err-Num. BET acquires more Err-Num than other methods on all datasets and models. Particularly, on CIFAR-10, taking VGG16 as an example, the Err-Num of BET is 3133.0 more than that of the best white-box testing method, i.e. ADAPT. On Tiny-ImageNet dataset, taking VGG19 as an example, the Err-Num of BET is 2297.2 more than that of ADAPT. On ImageNet, taking ResNet50 as an example, the Err-Num of BET are 2057.9 more than that of ADAPT. Such results indicate BET could generate more error-inducing inputs than white-box methods under the same query/inference budget indicating the query efficiency.

Result of Label-Num. BET acquires the most Label-Num than other white-box methods. Particularly, on CIFAR-10, taking VGG16 as an example, BET finds 1.1× Label-Num than ADAPT. On Tiny-ImageNet, taking ResNet56 as an example, BET finds 1.4× Label-Num than ADAPT. On ImageNet, taking ResNet50 as an example, BET finds 9.1× Label-Num than ADAPT. Such results show that BET is efficient for exploring different decision boundaries of CNNs.

Result of SR. For all initial test cases, BET can find corresponding error-inducing inputs on different datasets and models, which reflects the effectiveness of BET. We can observe that BET outperforms the baseline methods by achieving higher SR.

Result of Inact-Rate. The Inact-Rate of BET is 0.0%. The perturbations are directly added on pixel values as integers, BET does not suffer the inactivation issue. However, baseline methods, especially the ones that use gradient-based methods suffer from this problem. For instance, on ResNet50, the Inact-Rate of all these white-box methods Inact-Rate are 94.0%, 36.6%, 54.8%, and 62.4%, respectively. This is because their methods calculate error-inducing inputs in continuous domain and will suffer from invertible loss when transforming the perturbations back to pixel values as integers.

Analysis. We note that existing white-box testing methods aim to achieve higher neuron coverage. This metric is hard to experiment with in black-box settings since there is no inner knowledge of models for us to improve the neuron coverage or to measure it. Therefore, we choose other metrics such as the number and quality of error-inducing inputs found by BET and white-box testing methods as baselines. The results show that for ST scenario, BET can outperform the white-box testing methods even it is performed in black-box settings. There are two main possible reasons. (1) Achieving high neuron coverage by the baseline methods does not definitely mean they can find more error-inducing inputs. (2) Neuron coverage metrics may have bias as pointed out by [13, 20]. For instance, the neuron coverage metric in DeepXplore [31] promotes neurons whose values are below a threshold, in order to get higher values for better neuron coverage. It may fail to test inputs that will cause neurons with low values.

4.4 Improving CNN Accuracy by BET

In this section, we demonstrate error-inducing inputs generated by BET could improve the well-trained target CNN models' accuracy. We perform such experiments only on CIFAR-10 and Tiny-ImageNet datasets. First, we randomly select 500 error-inducing inputs generated by BET on CIFAR-10 and Tiny-ImageNet training sets, respectively. There are 50,000 and 100,000 images in the clean training set of CIFAR-10 and Tiny-ImageNet, respectively. We augment these two training sets by randomly mixing the 500 error-inducing inputs into them. We then fine-tune all target CNNs by 5 epochs and observe their accuracy improvement on test sets. Note that all error-inducing inputs here are generated from the training set which is different compared with the DeepXplore [31] which uses the test set to generate these error-inducing inputs.

Results are shown in Table 9. we observe that the error-inducing inputs generated by BET can improve target models' accuracy by 2 ∼ 3% with 5 epochs of fine-tuning. We also fine-tune these well-trained target models with only the training datasets under the same setting and the accuracy remains the same or even decreases.

We also follow DeepXplore [31] which uses the test set to generate 500 error-inducing inputs and mix them with training set to fine-tune the model. For Tiny-ImageNet and VGG16, we can improve the accuracy by 6%. However, using modified test samples to fine-tune the model may be unfair.

Table 9: Improving ACC with the error-inducing inputs of BET. Format: ACC of original CNN→ACC of fine-tuned CNN.

Dataset	Model	Top-1 Acc (%)	Top-5 Acc (%)
Fine-tune with the error-inducing inputs of BET			
CIFAR-10	VGG16	83.26→85.58	N/A
	ResNet18	85.34→87.03	N/A
Tiny-ImageNet	VGG16	51.37→54.41	75.10→75.74
	ResNet56	46.42→47.88	72.34→72.66
Fine-tune with the corresponding clean training set			
CIFAR-10	VGG16	83.26→83.43	N/A
	ResNet18	85.34→84.78	N/A
Tiny-ImageNet	VGG16	51.37→51.94	75.10→75.21
	ResNet56	46.42→45.48	72.34→72.38

4.5 Ablation Study on Efficiency-Centric Policy

We verify the function of the efficiency-centric policy (Section 3.4) in improving query efficiency through testing. Recall the process of our efficiency-centric policy, only one candidate is derived in each iteration. Consequently, we consider scenarios where multiple candidates are generated in each iteration by BET (disable the efficiency-centric policy) to compare. Particularly, in each iteration, we modify BET to generate a batch of candidates with four different candidate numbers, i.e., 5, 15, 20, and 25. Then, we rerun BET to evaluate the impact brought by the candidate numbers in one batch. The experiments are conducted in ST scenarios and use the same experiment settings as in Section 4.3.

Results are shown in Figure 4. On all datasets and models, our BET with an efficiency-centric policy enabled (candidate number is 1) performs better than all other compared candidate numbers. Therefore, we conclude that under the same query budget, our efficiency-centric policy makes sure BET could find error-inducing inputs in a query-efficient manner. Such experiment results also confirm our analysis of the efficiency-centric policy in Section 3.4.

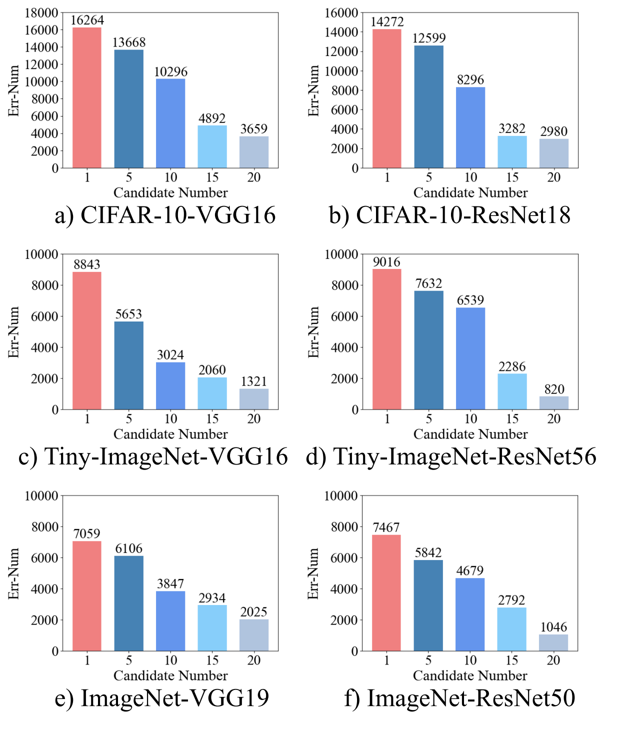

Figure 4: Impact of different candidate numbers. Candidate number equals 1 denote efficiency-centric policy enabled.

5 DISCUSSION

Comparison with adversarial attacks. Adversarial attacks [4, 23, 25, 33, 42] aim to generate human unnoticed perturbations on images as adversarial examples (AE) to effectively mislead CNNs.

There are similarities between the AEs and error-inducing inputs generated in CNN testing that can both mislead CNN classification [9]. But the metric and effect of adversarial attacks and CNN testing methods are quite different. First, an effective adversarial attack aims to successfully craft one sample falling to a specific or arbitrary classification result, but CNN testing methods aim to find more defects of one target model. For instance, white-box testing methods try to achieve a high neuron coverage and BET tries to explore different decision boundaries by diversifying the labels of error-inducing inputs found. Second, it is well-known that using AEs generated from the training set to fine-tune the model can increase the robustness [23] only to adversarial attacks but will harm the model accuracy [44]. This is also different with CNN testing methods since both DeepXplore [31] and BET have proved that fine-tuning the target model with error-inducing inputs can improve the accuracy.

Moreover, we do notice that using BET may also generate AEs by modifying its calculation goal and modification constraints. Since studying adversarial attacks in black-box settings is not within the scope of this paper, we leave such external research and experimentation as our first future work.

Novel metrics in black-box testing. White-box model testing methods focus on fully exploring various internal states of target models [19]. For example, under a white-box setting, the most common metric is to evaluate the neuron coverage to represent the internal states of CNNs. But this metric is not suitable for black-box testing methods since testers cannot try to improve the neuron coverage without any inner knowledge of the neurons. Thus, we need new metrics for black-box testing methods to understand the inner status of the target models during the test. We think this challenge may be solved through learning or knowledge extraction from the error-inducing inputs acquired during the test. Thus, we list our second future work, i.e., specifying a new metric to reflect the inner states of target models for black-box testing.

Potential improvement of BET in white-box. BET operates in a black-box setting by assuming there is no knowledge even on the kernel size and structures. As indicated in Section 3.1, we approximate the kernel information by analyzing the general weakness of common CNN models to generate error-inducing inputs. If deploying BET in a white-box setting, we can generate the perturbations exactly according to the CNN detailed structures like kernel size and structures. We believe that we can increase the error-inducing inputs found with a fixed inference budget in such a white-box setting. However, since BET is designed without any consideration of neuron coverage, we still need to improve our method to achieve such a goal. So our third future work is to extend the usage of BET in a white-box scenario to not only further improve efficiency but also achieve a high neuron coverage.

Applications of BET. The testing method of BET is not only applicable to image classification but also many other scenarios that could adopt CNN models. For instance, we have tested CNN models for audio recognition using BET as well. The evaluation results showed BET is effective too. Another scenario is blockchain-based money laundering forensics [17]. Adversaries could exploit the pseudo-anonymization feature of blockchains to hide their illegal transactions. Neural networks are a promising solution to identify such illegal activities in a large volume of transactions, however, they could be bypassed by adversaries. BET could be used to promote the robustness of such solutions and mitigate the bypass threats.

6 CONCLUSION

We propose a query-efficient black-box CNN testing method BET. By analyzing the general weakness of CNN models, we craft continuous perturbations based on a tunable objective function to find error-inducing inputs. Moreover, we establish an efficiency-centric policy that can help to find these label-diversified error-inducing inputs in a query-efficient manner. We conduct extensive experiments with three well-known datasets and five CNN structures to show that BET can outperform several state-of-the-art testing methods in two typical testing scenarios.

ACKNOWLEDGMENTS

This work was supported partially by National Key R&D Program of China (2021YFB2701000), National Natural Science Foundation of China under Grant 61972224, Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006, and Ant Group through CCF-Ant Innovative Research Program No. RF20210021. The work of Qi Li was supported in part by NSFC under Grant 62132011 and BNRist under Grant BNR2020RC01013.

REFERENCES

[1] [n.d.]. Tiny ImageNet. https://tiny-imagenet.herokuapp.com/.
[2] Francis R. Bach. 2013. Learning with Submodular Functions: A Convex Optimization Perspective. *Found. Trends Mach. Learn.* 6, 2-3 (2013), 145–373.
[3] 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).
[4] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *IEEE Symposium on Security and Privacy, S&P*.
[5] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. *Proceedings of Machine Learning and Systems, MLSys* (2021).
[6] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE*.
[7] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*.
[8] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE SP*.
[9] Justin Gilmer, Nicolas Ford, Nicholas Carlini, and Ekin Cubuk. 2019. Adversarial examples are a natural consequence of test error in noise. In *International Conference on Machine Learning*. PMLR, 2280–2289.
[10] Nathan A Greenblatt. 2016. Self-driving cars and the law. *IEEE spectrum* (2016).
[11] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. 2018. Recent advances in convolutional neural networks. *Pattern Recognition* 77 (2018), 354–377.
[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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*.
[13] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is neuron coverage a meaningful measure for testing deep neural networks?. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*.
[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*.

[15] Sanghyun Hong, Yiğitcan Kaya, Ionuţ-Vlad Modoranu, and Tudor Dumitraş. 2021. A Panda? No, It's a Sloth: Slowdown Attacks on Adaptive Multi-Exit Neural Network Inference. In *International Conference on Learning Representations, ICLR.*

[16] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *41st International Conference on Software Engineering, ICSE.*

[17] Kateryna Kolesnikova, Olga Mezentseva, and Tleuzhan Mukatayev. 2021. Analysis of Bitcoin Transactions to Detect Illegal Transactions Using Convolutional Neural Networks. In *2021 IEEE International Conference on Smart Information Systems and Technologies (SIST).* 1–6. https://doi.org/10.1109/SIST50301.2021.9465983

[18] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

[19] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *29th ACM International Symposium on Software Testing and Analysis, ISSTA.*

[20] Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. 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, ICSE (NIER).*

[21] 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 International Conference on Automated Software Engineering, ASE.*

[22] 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. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE.*

[23] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations, ICLR.*

[24] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. 2015. Lazier than lazy greedy. In *Proceedings of the AAAI Conference on Artificial Intelligence, AAAI.*

[25] Seungyong Moon, Gaon An, and Hyun Oh Song. 2019. Parsimonious Black-Box Adversarial Attacks via Efficient Combinatorial Optimization. In *Proceedings of the International Conference on Machine Learning, ICML.*

[26] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical programming* 14, 1 (1978), 265–294.

[27] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition, CVPR.*

[28] Augustus Odena, Catherine Olsson, David G. 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.*

[29] Omkar M. Parkhi, Andrea Vedaldi, and Andrew Zisserman. 2015. Deep Face Recognition. In *Proceedings of the British Machine Vision Conference, BMVC.*

[30] Andrew Paverd, Andrew Martin, and Ian Brown. 2014. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep* (2014).

[31] 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.*

[32] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP).* IEEE, 615–632.

[33] Han Qiu, Tian Dong, Tianwei Zhang, Jialiang Lu, Gerard Memmi, and Meikang Qiu. 2020. Adversarial attacks against network intrusion detection in IoT systems. *IEEE Internet of Things Journal* (2020).

[34] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. 2018. Malware Detection by Eating a Whole EXE. In *The Workshops of the 32th AAAI Conference on Artificial Intelligence, AAAI Workshops.*

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252.

[36] Jasmine Sekhon and Cody Fleming. 2019. Towards improved testing for deep learning. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER).*

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

[38] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2018. Testing deep neural networks. *arXiv preprint arXiv:1803.04792* (2018).

[39] 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.*

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

[41] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR.*

[42] Chun-Chen Tu, Pai-Shun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, and Shin-Ming Cheng. 2019. AutoZOOM: Autoencoder-Based Zeroth Order Optimization Method for Attacking Black-Box Neural Networks. In *the Thirty-Third AAAI Conference on Artificial Intelligence, AAAI.*

[43] Jingyi Wang, Jialuo Chen, Youcheng Sun, Xingjun Ma, Dongxia Wang, Jun Sun, and Peng Cheng. 2021. RobOT: Robustness-Oriented Testing for Deep Learning Systems. In *IEEE/ACM International Conference on Software Engineering, ICSE.*

[44] Dongxian Wu, Shu-Tao Xia, and Yisen Wang. 2020. Adversarial weight perturbation helps robust generalization. In *Annual Conference on Neural Information Processing Systems, NeurIPS.*

[45] 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.*

[46] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. DiffChaser: Detecting Disagreements for Deep Neural Networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI.*

[47] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS.*

[48] Michal Zalewski. 2017. American fuzzy lop. *http://lcamtuf. coredump. cx/afl* (2017).

[49] Lingfeng Zhang, Yueling Zhang, and Min Zhang. 2021. Efficient white-box fairness testing through gradient search. In *ACM SIGSOFT International Symposium on Software, ISSTA.*

[50] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. 2020. White-box fairness testing through adversarial sampling. In *International Conference on Software Engineering, ICSE.*